Workshop: Wildlife Data Analysis Using Program R

Robert C. Lonsinger

University of Wisconsin-Stevens Point
College of Natural Resources
Carnivore Ecology and Conservation Lab

Strengthening Natural Resources through Collaboration

78th Midwest Fish and Wildlife Conference
Milwaukee, Wisconsin | January 28-31, 2018

28 January 2018
Milwaukee, Wisconsin

TABLE OF CONTENTS

PREPARING FOR THE WORKSHOP

This workshop is intended to be in introduction to R and to provide participants with the ability to manage and prepare data for complex analyses in R. This necessitates that the workshop start with the most basic operations in R. Still, the workshop ramps up to complicated user defined functions, looping constructs, and simulation capabilities.

Workshop attendees can take several approaches to participating and benefiting from this workshop. One strategy (1) is to forego the computer and use this document to follow along and take notes as the instructor demonstrates the functionalities of R. This strategy has worked well for some, because all of the course materials (e.g., R code, explanations) are available to participants and can later be run and manipulated at an individual's preferred speed. Another strategy (2) is to bring a laptop and run the code that is being demonstrated on your own during the workshop, allowing yourself to become more comfortable with using R. This works well for those with some previous exposure to R and/or those comfortable with simultaneously following along and coding. A final strategy (3) would be to combine the first two: bring your laptop and run the code that is being demonstrated during the workshop initially, but to forego trying to run the more complex code that limits your ability to follow along. This strategy tends to be the most employed and allows participants to pay closer attention to the detailed code structuring and information provided by the instructor when working with the looping and simulation functionality.

If you intend to apply either of the latter two strategies (i.e., you want to have your laptop and the potential to run code), you will need to install the necessary programs, packages, and data prior to the start of the workshop, because the convention center will not be offering wireless internet. Specifically, you should take the following steps prior to arriving at the workshop:

1. Install R (*This step requires that you have internet access*)
    - Details for installing R can be found below on Page 3.
2. Load necessary packages and import data sets
    - After installing R, open R and type "getwd()" (without quotes) into the R console. R will return the identity of your working directory (example on Page 19 below).
    - From the folder that you downloaded (i.e., MW.Workshop), navigate to the DataFiles folder, select the 6 data files, and copy these directly into your working directory folder (i.e., the location identified by the getwd() function).
    - From the MW.Workshop folder, open the MW.Workshop.R file with a simple text editor (e.g., Notepad++ or Notepad).
    - In R, run the code at the beginning of the MW.Workshop.R file between "##Load Workshop files - Start" and "##Load Workshop files - End" by copying and pasting this code into the R Console. This code will attempt to load necessary packages and files to R. You may be prompted to select a CRAN mirror. If so, simply select a USA CRAN mirror close to you (e.g., KS)
      (*This step requires that you have internet access)*
    - Close R, but be sure to select "Yes" to save the workspace image when asked.

OBJECTIVES

To introduce participants to the functionality of R and to provide the knowledge necessary to effectively *begin* using R

To provide a foundation on syntax and R programming code that will allow users to *interpret* and therefore *utilize* the R language

To introduce participants to the benefits and structures of looping functionality an user defined functions, and give participants the opportunity to explore these capabilities within R

This course is not a statistical course or a complete review of R

INTRODUCTION & BACKGROUND

What is R?

- R is an open source software environment for *statistical computing and graphics*
- R is a fully functional programming language
- R is quickly becoming an industry standard
- R is a free alternative to costly data analysis software

Advantages of using R:

1. Cost
    - Downloaded FREE of charge
2. Convenience
    - Accessible through the Comprehensive R Archive Network (CRAN) along with thousands of data analysis/graphing packages
    - Scripts can be used to store detailed accounts of how you managed, modified, and analyzed you data.
3. Community
    - The R community is comprised of hundreds of thousands of users that are working on developing code, improving performance, and providing support
    - The R Core Development Team ensures packages meet documentation and quality standards
4. Capability
    - Comparable or superior to commercial data analysis/graphic packages
    - Great for analyzing data, conducting simulations, testing new algorithms, and plotting graphics
    - Users (including you) can modify code to meet your needs and can extend R's capabilities by contributing packages

Disadvantages of using R:

1. Programming can be intimidating
    - It is critical that users with no or limited programming experience start using R to accomplish simple data management and summary statistic procedures. This will allow users to begin using R without feeling like they are programming. They can then become comfortable with the syntax and eventually transition into more complex coding and programming as their needs and experience grows.
    - The only way to learn R, is to use R
2. Available code (online) may not be tested
    - Although the functionality downloaded through the CRAN has been tested, code available through one of the thousands of independent websites and/or forums may not always be tested
    - This does not mean you should not use these resources or this code, but rather, that you should be able to interpret the R language and ensure that code is performing the desired task before implementation
3. Packages may lack desired operations or may be overlapping
    - Packages are often designed by individuals or teams to meet their specific needs
    - Some packages have some of the same procedures or methods (though they may have been named differently)
    - Other desired operations may be lacking, or may be difficult to find
4. Technically, no technical support
    - Although R does not provide a technical support department, there are an unprecedented number of user based help pages, forums, and email list-serves
    - The R community provides all of the unofficial technical support that one may need

INSTALLING R

- Visit the official R Website (www.r-project.org) and click on the "download R" link
- Scroll through the available CRAN Mirror sites and select the site located closest to you
- Select the download that corresponds to your operating system (i.e., Linux, Mac, or Windows)
- Downloading for the first time: Select the "base" subdirectory
- Click on the "Download R" link, which indicates the version and the operating system
- When given the option, choose the "Run" option for the executable
- Walk through the Setup Wizard then click "Finish"– Recommend using all of the default settings

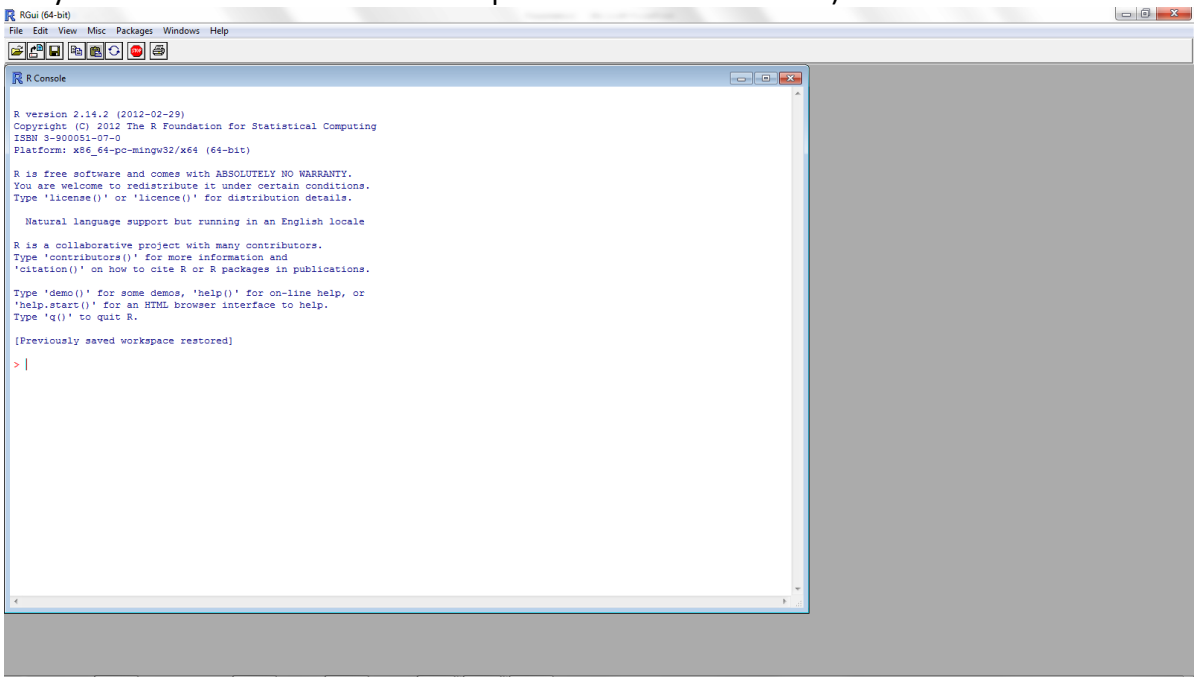
THE R GUI & CONSOLE

Navigate to your programs and click on R to launch the R GUI (Graphical User Interface)

- Alternatively, during set up you may have added a quick launch icon to your start menu or desktop

The R GUI:

- May have a slightly different appearance and/or layout on different platforms
- The standard R GUI is very basic, and has only limited functionality
- While this does not limit the capability of R, it can make producing complex algorithms or scripts cumbersome
- There are a few quick key icons for loading and saving you workspace, for copying and pasting, and stopping and printing
- More options can be found through the drop down tabs (Recommendation: before you start working in R, take a moment to go through the options available and familiarize yourself with the Console and drop down tab functionalities)



The R Console:

- Where you enter your commands and instruct R on what you would like to do
- The command line is indicated by a *prompt*, which looks like a greater than sign “>”
- Following an entry to the command line, press the enter button and R will perform some task and/or return some value or character
- If a command is incomplete when you press enter, the next line will begin with a continuation sign “+” rather than a “>”, indicating that you need to still complete the expression on the line(s) above

Alternative R GUIs:

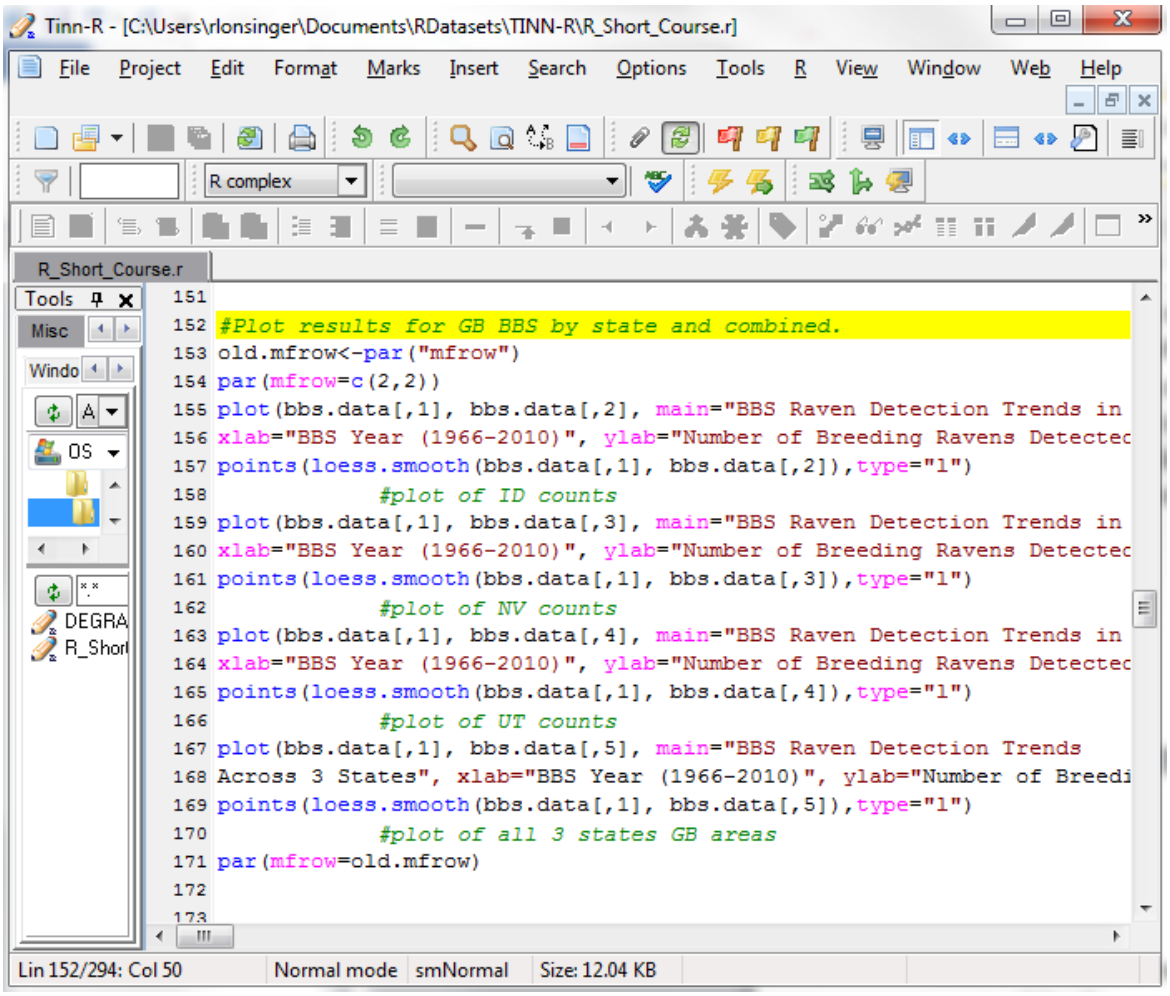
- There are a number of alternative R GUIs that have been developed and that can be downloaded free of charge
- These GUIs often offer improved code management by providing formatting options, coloring of code, and direct running of code in R
- They also allow you to SAVE your code so that you can rerun analyses, reuse you code, and recall how your analyses were conducted

A list of some alternative R GUIs:

RStudio
Tinn-R (pictured below)
R Commander
Rattle GUI
RKWard
RExcel
Eclipse with StatET
RapidMinor R extension

There are many more alternative GUIs. Many GUIs are platform specific, so explore which will work on your operating system

Each have different benefits, so it is important to find the GUI that meets your needs and style of use



R SYNTAX & COMMAND LINE ENTRIES

A few syntax items that are used frequently and that you should be able to recognize quickly:

<- Assignment notation; creates objects

- The left arrow (a combination of the less than and dash), is the assignment notation and is used to assign values or results to an object symbol

Comment notation

- The comment notation is incredibly valuable and I would suggest that as you learn R, that you notate your code with comments to help you keep track of what each line of code is doing
- The comment notation tells R that everything to the right of the "#" is a comment and does not have any inherent interpretation in R
- Can be used in data that is being imported to support metadata (e.g., comments on data collection procedures, challenges, or potential data complications)

c() Concatenate method joins values together into a vector

- The concatenate method is used very frequently and is used to create a vector, or string of values (or characters) that are joined together in order and that can be referenced by its location in the vector

: Series notation

- The series notation creates a sequential series from the first value to the second value in a step-wise fashion by increments of 1

Example:

```
> x<-c(1:10)     #store the vector 1, 2, …, 10 as x
> x
[1]  1  2  3  4  5  6  7  8  9 10
```

- We are calling a function, concatenate(), denoted by c(), and we are telling it to create a vector of 10 values numbered 1 to 10, and to assign these values to the object symbol x
- We are storing the resulting vector to x, so R does not print the results to the screen
- To see the results, we must call x
- Note that the "[1]" index indicates that the number immediately to the right of the index is the FIRST element of the vector x

Three Types of entries to the command line *prompt*:

1. Methods() – an object that does something in R
   - Often referred to as a *function*
   - Easily distinguishable because they are followed by ( )
   - Depending on the complexity, may or may not require arguments within the ( )

May have named arguments that can be passed to the function by placing them in the ( )

> *Function*(*argument1=value1, argument2=value2,…)*
   - Separate multiple arguments with commas
   - If arguments are in their defined order and all arguments are included, then the *argument names* can be omitted

Three Sources of Methods:
   - Many methods built into the base R download
   - Many more available via packages
   - Can create your own methods

*Note: It is critical to review the help page for methods with help(*method*) or ?*method* to obtain the description, arguments, usage, and examples

Frequently used help/search methods:

> help() or > ?
   - Allows you to View the html format for a specific function or method in R
   - Does not require internet access
   - Requires that you know that name of the function/method
   - Must pass the name of the function to help(*function*) or ?*function* as an argument

> help.search() or > ??
   - "Fuzzy search" allows you to search without knowing the exact name
   - Does not require internet access
   - Must pass some name or partial name, such as fisher for fisher.test, to help.search("fisher") or ??fisher as an argument
   - Note: help.search() requires " ", while ?? does not

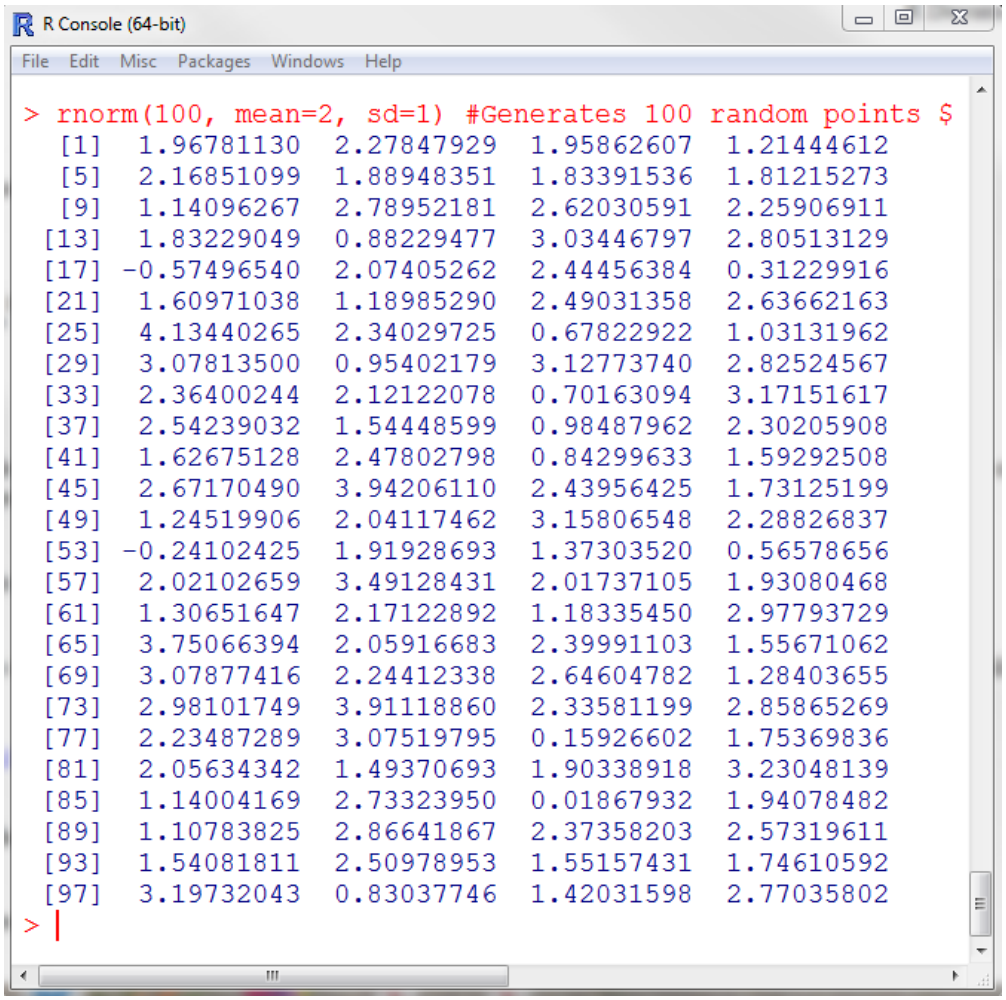
Example of arguments in their natural order with the rnorm() function:

rnorm() #Method that selects n random numbers from a normal distribution

> rnorm(n, mean=0, sd=1) #Usage outlined under help(rnorm)
   - Because the mean and sd are set to specific values, omission of these arguments will tell R to use the default values

```
> rnorm(100)                    #Requires n; defaults for mean and sd
> rnorm(100, mean=2, sd=1)      #Change mean and sd
> rnorm(100, 2, 1)              #same, with argument names omitted
```

Returns a vector of 100 values drawn at random from a normal distribution with a mean of 2 and standard deviation of 1

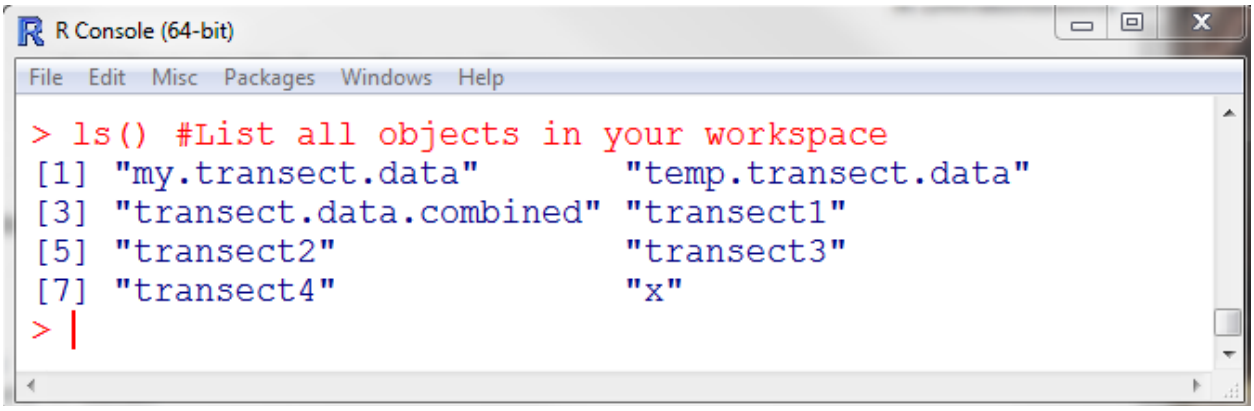


Example of a method and the use of arguments with the ls() function:

```
ls()              #Lists the objects in your workspace
```

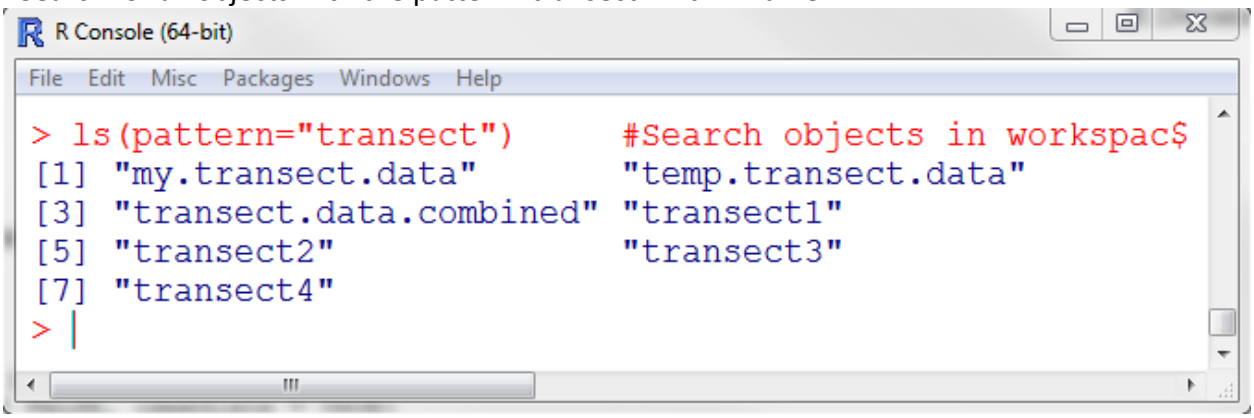
- Excludes objects starting with a "." unless the argument "all.names" is set to TRUE

```
> help(ls)        #First look at the usage as outlined with the help() method
```



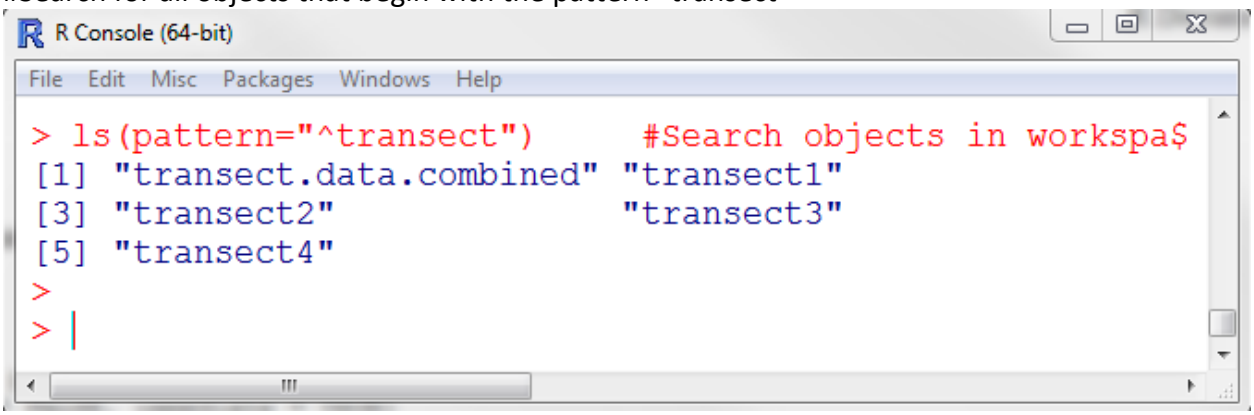
The argument “pattern” allows you to use ls() to effectively search your workspace:

#Search for all objects with the pattern "transect" within name



The argument “pattern” combined with a “grep” pattern (^) further refines the search”:

#Search for all objects that begin with the pattern "transect"



2. Variables – an object that stores data/information
   - Objects operated on by R

Naming standards (apply to methods as well):

- CaSe SeNsItIve

my.data1	These are all different variables
My.data1	because they each use a different
MY.DATA1	combination of upper and lower case

- May contain letters, numbers, periods, and underscores
  - Should always start with a letter
  - Because system variables start with a ".", these variables are not listed with ls()
  - Cannot have spaces; words are usually separated with a period

Variables can be used to store:

- Simple values and expressions

```
> x<-6              #read as "x gets 6"
> y<-4              #does not print by default, because value is being stored
> x+y               #prints by default to console (value not being stored to an object)
[1] 10

> z<-x+y            #substitution occurs during assignment
> z                 #Subsequently changing the value of x will not change z
[1] 10              #[1] indicates that the following value of 10 is the first element
```

- Results of a method

```
> Y<- c(1:3,5:8)*8  #Concatenate the series 1 to 3 and 5 to 8, then multiply each by 8
> Y                 #Note, capital Y is different from the lowercase y above
[1]  8 16 24 40 48 56 64

> Z<-sqrt(Y)        #Take the square root of the vector Y and store as Z
> Z                 #View the vector Z
[1] 2.828427 4.000000 4.898979 6.324555
[5] 6.928203 7.483315 8.000000
```

Common objects and their referencing indices:

- Vectors** v[i] #i = index of element I in vector v
- Matrices** m[r,c] #r,c = index of element in row r and column c of matrix m
- Arrays** a[r,c,m] #r,c,m = index of element in row r, column c, and matrix m of array a

**Note: Vectors are restricted to a single data type. Matrices are 2-dimensional extensions of vectors and arrays are 3-dimensional extensions of vectors. Consequently, both matrices and arrays are also restricted to a single data type.

- Data frames* `df[r,c]` #r,c = index of element in row r and column c of data frame df
  `df[[c]]` #c = index of column c (returns entire column)
  `df[[c]][r]` #c = index of column c, while r returns the element in row r of column c
- Lists* `L[[i]]` #i = index of element i in list L
  `L[[i]][g]` #g = index of element g in list element i (if L[[i]] returns a vector)
  `L[[i]][r,c]` #r,c = index of element in r and column c in list element i (if L[[i]] is a 2D element such as a matrix or data frame)

*Note: Data frames and lists are not restricted to a single data type (though any vectors, matrices, or arrays contained within them are). Ecologists commonly use data frames to store their data, since each column tends to represent a different variable which varies in type (e.g., numeric vs factor). Lists may contain vectors, matrices, arrays, data frames, or lists, and are therefore flexible at storing complex data sets of varying data types.

Common data types for a vector and examples of each:

- Character strings

```
> canids<-c("fox", "wolf", "jackal", "fox")
> canids
[1] "fox" "wolf" "jackal" "fox"
```

- Factor #Using the factor() function, the character vector above is converted

```
> canids<-factor(canids)
> canids                    #Now the output assigns each element to a factor level
[1] fox wolf jackal fox
Levels: fox jackal wolf
```

- Logical

```
> foxes<-canids=="fox"      #A test for which elements are fox returns a logical vector
> foxes                     #recall the original canid levels were: fox wolf jackal fox
[1] TRUE FALSE FALSE TRUE
```

- Numeric

```
> canids<-as.numeric(canids) #Converts factors to numeric based on the level
> canids                     #Recall the levels for canids were: fox jackal wolf
[1] 1 3 2 1
```

Avoid creating variables that conflict with the following:

- NA #Represents missing values (Not Available)
- NULL #Used as an argument in functions to indicate no value has #been assigned, or to initialize an empty variable
- NaN #Not a number–result from in a non-sensible computation
- Inf (-Inf) #Infinity and negative infinity
- TRUE (or T) #Logical
- FALSE (or F) #Logical

3. Operators – simple methods built into R

```
> c(1:12) %% 4 == 0                    #Logical test for multiples of 4
[1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

Table of common operators

Operator	Description	Operator	Description
+	Addition	&	AND (element wise)
-	Subtraction	&&	AND (programming control flow)
*	Multiplication	|	OR (element wise)
/	Division	||	OR (programming control flow)
<	Less than	^	Raised to the power of
<=	Less than or equal to	**	Raised to the power of
>	Greater than	:	Generate integer sequence
>=	Greater than or equal to	%o%	Outer product (matrix calculations)
==	Equal to	%*%	Matrix multiplication
!=	Not Equal to	~	Model relationship between variables
!	Logical negation	…	…

REFERENCING YOUR DATA

Referencing elements of a variable by location using [ ]:

```
> Z
[1] 2.828427 4.000000 4.898979 6.324555
[5] 6.928203 7.483315 8.000000

> Z[4]                    #reference an single element
[1] 6.324555

> Z[4:6]                  #reference a series of elements
[1] 6.324555 6.928203 7.483315

> Z[c(4,6)]               #c() for referencing >1 non-sequential element
[1] 6.324555 7.483315
```

Referencing elements of a variable by conditions:

```
> Z
[1] 2.828427 4.000000 4.898979 6.324555
[5] 6.928203 7.483315 8.000000

> Z[Z>4 & Z<7]                  #Select only those values of "z" between 4 and 7
[1] 4.898979 6.324555 6.928203

> Z[ceiling(Z)==7]              # Select all those values of "z" that round UP to (==) 7
[1] 6.324555 6.928203
```

Referencing elements of a 2-dimensional variable or data frame:

- Entering a small amount of data into your workspace:

```
> Animal.ID<-c("A1", "A2", "A3", "A4", "A5")
> total.mass<-c(3.3, 2.9, 2.8, 3.0, 3.4)
> left.ear<-c(110.2, 105.6, 97.5, 101.5, 107.3)
> hind.foot<-c(33.5, 33.9, 32.1, 32.8, 33.4)

> body.data<-data.frame(Animal.ID, total.mass, left.ear, hind.foot)
> body.data                     #View rabbit data, which is a data frame
```

	Animal.ID	total.mass	left.ear	hind.foot
1	A1	3.3	110.2	33.5
2	A2	2.9	105.6	33.9
3	A3	2.8	97.5	32.1
4	A4	3.0	101.5	32.8
5	A5	3.4	107.3	33.4

- Referencing elements by location using [ ] #[row, column]

```
> body.data[1,]        #Returns row 1, all columns b/c the column index is empty
  Animal.ID  total.mass  left.ear  hind.foot
1   A1          3.3        110.2      33.5

> body.data[,1]        #Returns column 1, all rows b/c the row index is empty
[1] A1 A2 A3 A4 A5
Levels: A1 A2 A3 A4 A5

> body.data[2,3]       #Returns the value in the cell in row 2 and column 3
[1] 105.6
```

- Referencing elements by name/location #$ notation references variables or #columns within a data frame

```
> body.data[,2]                #Returns the values in column 2 by indexing
[1] 3.3 2.9 2.8 3.0 3.4

> body.data$total.mass         #Returns the values in column 2 by $ notation (name)
[1] 3.3 2.9 2.8 3.0 3.4

> body.data[["total.mass"]]    #Note, can also reference a data frame column with [[ ]]
                               #[[ ]] should contain the column index or name (in " ")
[1] 3.3 2.9 2.8 3.0 3.4

> body.data[[2]][1]            #Can reference single elements by referencing the column
                               #with [[ ]], then the required element with [ ]
[1] 3.3
```

- Referencing elements by condition #Useful for selecting data from large #or complicated data sets

```
> body.data$total.mass[c(1,4:5)]                  #$ w/ specific locations
[1] 3.3 3.0 3.4

>body.data$total.mass[body.data$total.mass>=3]   #$ w/ conditional notation
[1] 3.3 3.0 3.4
```

Note: There are many different ways you could reference the same data. The approach you take will depend in part on your objective and experience. For example, all of the following will return the same values, but may be selected for use for different reasons:

```
> body.data[,2][body.data$total.mass>=3]
> body.data[body.data$total.mass>=3,2]
> body.data[[2]][body.data$total.mass>=3]
> body.data[["total.mass"]][body.data$total.mass>=3]
```

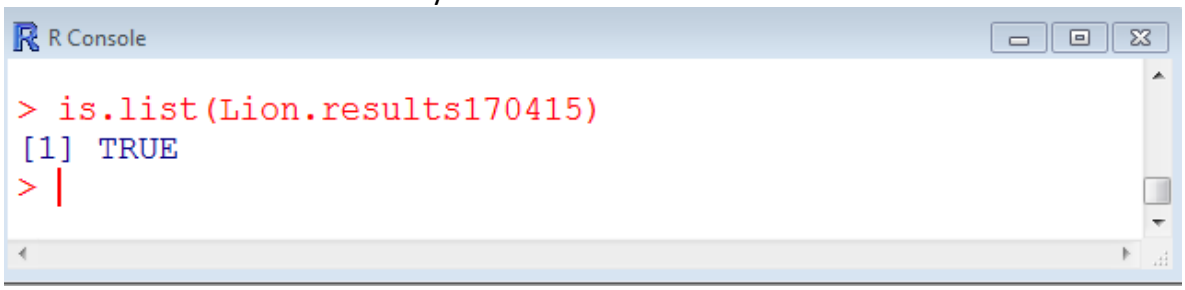
Referencing elements of a multi-dimensional variable:

- May have objects with >2 dimensions (e.g. arrays)
- Arrays can be thought of has having an Excel file with multiple worksheets, where each worksheet is a different matrix
- Referencing follows the same general trend except a layer index is added
    - Array1[row, column, layer]

Referencing elements of a list:

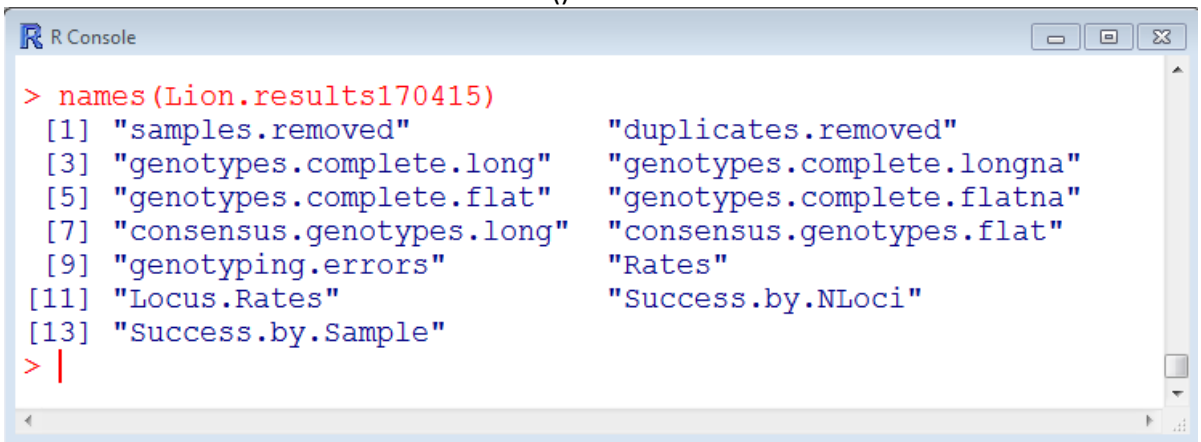
- Each element of a list can store a variable (e.g., a vector, array, data frame, list, etc.)
- [[ ]] is used to reference list elements
- $ notation may be used if the list is named
- Referencing of elements within a selected list element would be based on their data type (e.g., vector vs. data frame) and follow rule previously discussed.

Example: Review results from ConGenR for mountain lion data
Check that the results are actually in a list.



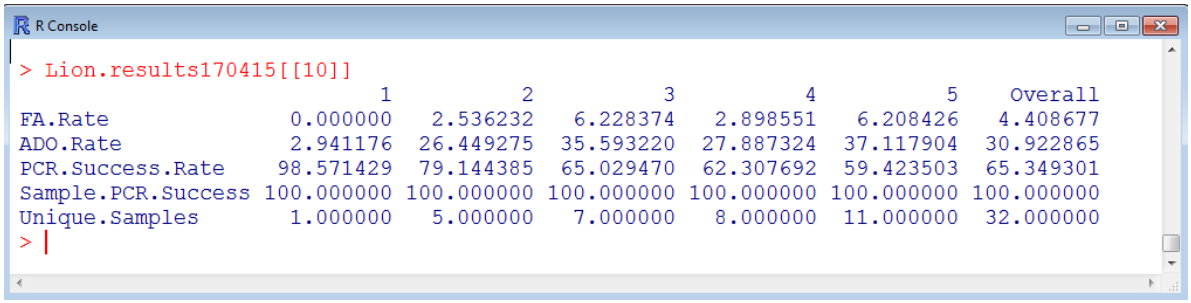
Note: is.list() is a function that returns a logical response. Similar functions exist to test for alternative structures such as is.data.frame(), is.vector(), is.array(), and so on.

View the names of the list with the names() function



Note: if the list elements were not named, the function would return the value “NULL”

View the results stored in the 10th element of the list



Note: Element 10 is named "Rates" and could have also been accessed with the following

```
> Lion.results170415$Rates
```

Furthermore, you can see that the list's 10th element is a data frame, therefore you could reference data within this following the rules for a data frame. For example, to reference the entire first column, you could use

```
> Lion.results170415$Rates[,1]
[1]   0.000000   2.941176  98.571429 100.000000   1.000000
```

THE R WORKSPACE

Your R workspace is the internal memory on your R (the R you have downloaded onto your computer). It is the memory space where all of your user created objects (i.e., any variables or methods that you have created and stored) are maintained.

Your R workspace does not contain system variables or methods, nor does it contain variables or methods derived from packages that have been installed.

Managing objects in your workspace:

```
ls()            #list of objects in workspace
                > ls(pattern="^RSC")  #Use the pattern argument and grep search to find all
                                      #objects in the workspace that start with "RSC"
                [1] "RSC1" "RSC2" "RSC3" "RSC4" "RSC5"

rm()            #removes/deletes objects
                > rm(RSC1)            #Remove/delete a single object by name
                > ls(pattern="^RSC")  #Search again and see that it has been removed
                [1] "RSC2" "RSC3" "RSC4" "RSC5"

                > rm(list=ls(pattern="^RSC")) #Remove/delete a group of objects by supplying
                                              #ls() with the pattern argument, as the "list"
                                              #argument to rm()
                > ls(pattern="^RSC")          #Confirm that all the RSC objects were removed
                character(0)
```

save.image() #saves current workspace/objects

- This can also be accomplished through the R GUI icons and file menu
- Default saves workspace to .RData file
- File argument can direct workspace to be saved to a different file
  > save.image(file="RSC.RData")

load() #loads a previous workspace

- This can also be accomplished through the R GUI icons and file menu
- Default loads .RData (auto loads at launch)
- File argument can identify workspace to be loaded
  > load(file="RSC.RData")

Redirecting R output from your workspace (not covered in the course, but important):

source(file="file.name") #Reads commands from file
sink(file="file.name") #Directs output to file
sink() #Resets sink to normal defaults

- Useful for simulations with lots of code/results

getwd() #identifies working directory
setwd() #sets your working directory

Entering data into your workspace:

If you are entering only a small amount of data, manual entry is a reasonable approach, where the first entry for each vector corresponds to the first row, the second entry corresponds to the second row, and so on…

```
> Animal.ID<-c("A1", "A2", "A3", "A4", "A5")
> total.mass<-c(3.3, 2.9, 2.8, 3.0, 3.4)
> left.ear<-c(110.2, 105.6, 97.5, 101.5, 107.3)
> hind.foot<-c(33.5, 33.9, 32.1, 32.8, 33.4)
```

data.frame() #Assembles vectors of equal length into a data frame
> body.data<-data.frame(Animal.ID, total.mass, left.ear, hind.foot)
> body.data #View data entered

	Animal.ID	total.mass	left.ear	hind.foot
1	A1	3.3	110.2	33.5
2	A2	2.9	105.6	33.9
3	A3	2.8	97.5	32.1
4	A4	3.0	101.5	32.8
5	A5	3.4	107.3	33.4

Importing data files into your workspace:

- More frequently, you will have complex data sets that you wish to read into R from another file
- Simplest importing format is a plain text file, such as .txt or .csv files
- Plain text files are the lowest common denominator of file formats and can be viewed in a wide range of editors (they are also less likely to be influenced or become unreadable by updates to text editors and/or R)
- Ecologists often store their data in Excel worksheets; these can be easily saved as a text (e.g., tab delimited) file through the save as function
- While you could import Excel files directly, converting to a plain text file requires you to clean up the file (e.g., removing entries or summary data outside of the primary data table)
- Note: R does not accept headers with spaces and thus will convert all spaces to a "."

```
read.table( )          #Reads in text data files
```

- Returns a data frame object
- Generic in that it can handle various delimiters (tab, comma, etc.)
- Includes headers without problem
- Allows for coding of missing values (R codes as NA)

```
read.csv( )            #Reads in .csv files
```

- Do not need to identify a delimiter
- In some regions, commas replace decimals and could be problematic

```
read.xls( )            #Reads in .xls files directly from Excel
```

- Do not need to identify a delimiter
- Data outside of the primary data table will influence importing (e.g., impacting the structure and data types of the data imported, introducing NAs)

Steps to efficiently importing your data files:

1. Identify your working directory (wd; data may or may not be in wd)

```
getwd()                #Note that the format separating directories and folders may vary by
                       #system, but using getwd() will show you how R needs the separators
                       #identified. For example, "/" vs. "\" vs. "//" and so on
                       #Also, note the MacOS do not have a C: drive, so you need to adjust
> getwd()
[1] "C:/Users/rlonsinger/Documents"          #indicates for my system, I should use "/"
```

2. Identify and store your data's path and file name

```
> data.file<-"C:/Users/rlonsinger/Documents/RDatasets/harvest.data.txt"
```

- Be sure to include the file type (e.g., .txt)
- The data path needs to be passed as a character string, so it should be in quotes (" ")
- The working directory can be abbreviated with ~, thus if your file is stored within your working directory (as it is here) we would shorten this to:
  `> data.file<-"~/RDatasets/harvest.data.txt"`

3. Use read.table() to import data file and store as an object in your workspace

```
> harvest.data<-read.table(file=data.file, header=TRUE, sep="\t", na.strings=-999)
```

- Identify if the file has headers, the delimiter (in this case, tab delimited), and how missing values are coded

Alternatively, if you imported the data earlier with the load.conference() function:

```
> harvest.data<-MW.data[[1]]
```

4. It is critical that you review that your data file was imported correctly
   - When importing data, R coerces data to either the numeric or factor data types. R does its best based on the structure of the data to infer the appropriate format, but often data are interpreted incorrectly
   - Typos or errors in data entry (e.g., extra decimal, letter within a number) can cause numeric variables to be imported as factors
   - Factor variables which researchers record as numeric characters may be incorrectly imported as numeric
   - Identifying numbers (e.g., hunt units, individual IDs) that should be considered as factors, are often coerced to numeric

```
> harvest.data[1:5,]   #Using the referencing approach already covered
> head(harvest.data)   #Using the head() to view only the first 6 rows as an alternative
```

	Unit	Species	Sex	Age	Weight	Method
1	6	1	1	juvenile	157	archery
2	55	2	1	yearling	230	muzzleloader
3	33	3	2	adult	223	rifle
4	27	1	2	juvenile	250	archery
5	26	3	1	adult	211	muzzleloader

5. It is important that you check the data structures as well (how the data is stored in R)
    - Common methods used to evaluate data include str(), attributes(), summary(), class()

```
> str(harvest.data)      #Method to view the data structure

'data.frame':   25 obs. of  6 variables:
 $ Unit:       int  6 55 33 27 26 15 39 15 61 60 ...
 $ Species:  int  1 2 3 1 3 2 2 1 1 3 ...
 $ Sex:      int  1 1 2 2 1 1 2 2 2 1 ...
 $ Age:      Factor w/ 3 levels "adult","juvenile",..: 2 3 1 2 1 3 2 2 2 2 ...
 $ Weight:   int  157 230 223 250 211 158 203 238 169 NA ...
 $ Method: Factor w/ 3 levels "archery","muzzleloader",..: 1 2 3 1 2 2 1 2 1...
```

- Note that two variables (columns), Species and Sex, are saved as integers. We may want these to be saved as categorical variables, or factors, for subsequent analyses and data management procedures

We could modify these two variables (or others) using the R GUI's Data editor (Edit > Data editor…), or the fix() function. While this approach may seem easier, it is limited (e.g., can only convert between numeric and character data types for columns) and does not document changes that have been made to the dataset (which is generally a bad practice).

- Alternatively, we can make these changes very simply with the following commands (take a moment to look up the different functions and understand what is occurring)

```
> harvest.data<-within(harvest.data, Species<-factor(Species))

#within() function indicates that you want to work within the harvest.data data set and
#therefore you do not need to specify harvest.data$Sex (you can just reference Sex)

#within the harvest.data, convert Species to a factor data type and store back to Species,
#then, store the data frame back to harvest.data

> harvest.data<-within(harvest.data, Sex<-factor(Sex, labels=c("M", "F")))

#Complete the same action for the variable Sex, but now also convert the factor level labels
#to "M" and "F", as opposed to 1 and 2 (how they were initially entered)

#to view additional arguments for these functions, check the help(within) and help(factor)
#support documentation
```

6. Re-check the data structures

- Both Species and Sex are now factors!

```
> str(harvest.data)
'data.frame':   25 obs. of  6 variables:
 $ Unit:       int  6 55 33 27 26 15 39 15 61 60 ...
 $ Species:  Factor w/ 4 levels "1","2","3","4": 1 2 3 1 3 2 2 1 1 3 ...
 $ Sex:      Factor w/ 2 levels "M","F": 1 1 2 2 1 1 2 2 2 1 ...
 $ Age:      Factor w/ 3 levels "adult","juvenile",..: 2 3 1 2 1 3 2 2 2 2 ...
 $ Weight:   int  157 230 223 250 211 158 203 238 169 NA ...
 $ Method: Factor w/ 3 levels "archery","muzzleloader",..: 1 2 3 1 2 2 1 2 1...
```

```
> View(harvest.data)          #View data in a spreadsheet format with scrolling capabilities
```

Data: harvest.data

	Unit	Species	Sex	Age	Weight	Method
1	6	1	M	juvenile	157	archery
2	55	2	M	yearling	230	muzzleloader
3	33	3	F	adult	223	rifle
4	27	1	F	juvenile	250	archery
5	26	3	M	adult	211	muzzleloader
6	15	2	M	yearling	158	muzzleloader
7	39	2	F	juvenile	203	archery
8	15	1	F	juvenile	238	muzzleloader
9	61	1	F	juvenile	169	archery
10	60	3	M	juvenile	NA	rifle
11	31	2	M	juvenile	224	muzzleloader
12	54	2	F	adult	222	archery
13	33	2	M	yearling	187	archery
14	27	4	F	juvenile	192	rifle
15	13	4	M	juvenile	175	muzzleloader
16	39	1	F	yearling	172	rifle
17	29	4	F	yearling	243	archery

Note: The importance of checking your data structures cannot be understated. Many statistical procedures determine what to do based on data types, or only operate with particular data types.

Example:

```
lm(j~x)          #lm() = linear model and models j as a function of x
                 #~ indicates a modeling relationship
```

- If x is numeric, performs a linear regression
- If x is a factor (categorical), performs and ANOVA

```
tapply(j, x, method)   #implicit loop that applies a method to a continuous variable (j) by levels
                       #of a grouping variable (x)
                       #If is.factor(x)==FALSE, tapply() will fail
```

Exporting data files from your workspace:

- Exporting a data file allows you to save a backup or work on the dataset in an alternative format (e.g. Excel)

```
write.table ()     #Exports data as text files
```

- Arguments are similar to read.table()
- Default for row.names=TRUE
- Default for quote=TRUE

Steps to efficiently exporting your data files:

1. Identify and your data path and file name as before (remember to include the file extension!)

```
> data.file<-"C:/Users/rlonsinger/Documents/RDatasets/harvest.data.modified.txt"
```

2. Use write.table() to export data file

```
> write.table(harvest.data, file=data.file, quote=FALSE, row.names=FALSE, sep="\t")
```

- First identify the *object* to export, identify the data path and file name as .txt file, and change necessary defaults

METADATA

Metadata includes all supporting information for a data set. Although there are ways to imbed some metadata into files within R, R is not efficient at storing metadata.

Examples of Metadata:

Variable names (column names)	Maps
Units of measurements	Known data issues, deficiencies, or errors
Data collection details	…

Tips for storing and tracking metadata in R:

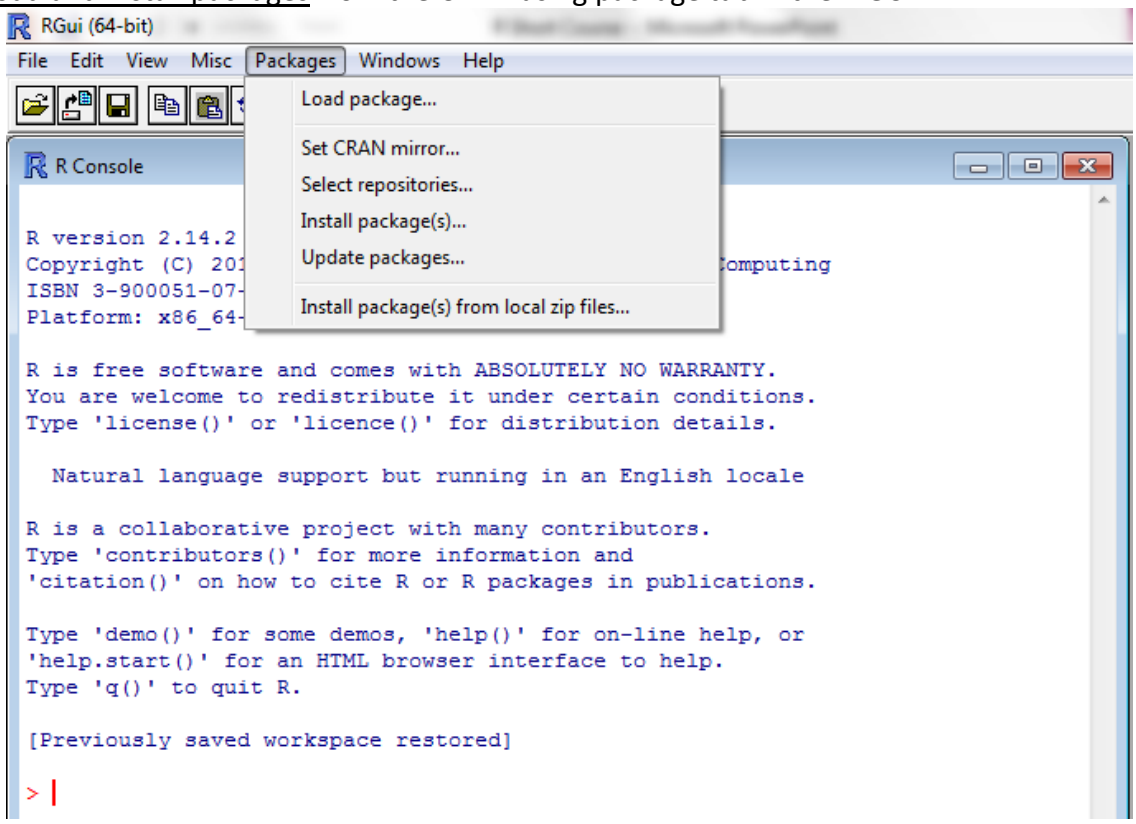
1. Utilize descriptive but manageable names
    - Columns, files, objects, etc.
2. Use the hashtag notation (#) to add context and annotation to files
    - Can be used in scripts, but also in imported data to tell R what not to read in as data
    - Can be used to reference locations of larger metadata that cannot be embedded (e.g. maps)
3. Store your scripts as essential metadata
    - These will provide a detailed record on how you read in, manipulated/modified, analyzed, and interpreted your data
    - These will allow you to verify steps, make changes, rerun analyses under different conditions, and/or run the same or similar analyses on new data sets

PACKAGES & SCRIPTS

Packages and scripts represent grouped objects (such as variables and/or methods) that can be downloaded as a single file

Packages:

Download and install packages from the CRAN using package tab in the R GUI



- Within the "Packages" tab, click "Set CRAN mirror…" and select the CRAN closest to you
- Return to the "Packages" tab, click "Install package(s)…" and select the package desired

A library is a set of packages

```
library()              #view all installed packages
```

- Similar to using ls() to view your objects, with library() you can view installed packages
- Only need to install a package once (unless requiring a new version or updated)
- Must load necessary packages during each session

```
library(package)       #load installed packages by passing the package to library()
> library(unmarked)
```

Scripts:

Download or copy scripts from repositories or websites

- Scripts are a collection of code that is not formally a package
- Often released prior to formal package development

Once downloaded, scripts are generally stored in the workspace as objects

```
ls()                   #view objects in the working directory
```

Note: you may still need to load a script after downloading or load supporting packages

- Loading scripts can often be accomplished by sourcing in the code (or simply copying and pasting it into the R console)

  Example:

```
> source("~/R_Datasets/ConGenR/ConGenR.r")      #Load ConGenR script
> load.ConGenR()
```

DATA MANAGEMENT

It is often desirable, or necessary, to manipulate and manage your data within R

- Adding data to current data frames
- Deleting data from data frames
- Sorting data
- Storing results of analyses back to a data frame
- Etc.

Common data management methods:

```
cbind()        #modify objects by adding, or binding, columns
rbind()        #modify objects by adding, or binding, rows
merge()        #combine two data sets by common fields
sort()         #sort a vector
order()        #determine the sort order of a vector
subset()       #subset selects a conditional portion of a data frame
```

Example: You acquired harvest data for another year, and want to include it in the analyses Import the new harvest data set, then view the names of the columns for each. Notice the new data set has a new column indicating the year of harvest.

If you imported the data earlier with the load.conference() function:
> harvest.data11<-MW.data[[2]]

```
> names(harvest.data)          #Use the names() to view the columns in the original data
[1] "Unit"    "Species" "Sex"     "Age"     "Weight"  "Method"

> names(harvest.data11)        #Use the names() to view the columns in the new data
[1] "Unit"    "Species" "Sex"     "Age"     "Weight"  "Method"  "Year"
```

- To combine the two data sets we must first add the harvest year to the original data set and then combine the two data sets.

Use the cbind() function to add the harvest year to the original data set from 2010
cbind(*df, new column*) #adds column to a data frame; additional arguments can be used

```
> harvest.data10<-cbind(harvest.data,Year=rep(2010,length(harvest.data[,1])))
        #Add a column named "Year" to harvest.data and save it as new data frame
        #To create the new column, repeat 2010 as many times as the data frame is long
```

> head(harvest.data10) #view the first 6 rows and see the year was successfully added

	Unit	Species	Sex	Age	Weight	Method	Year
1	6	1	M	juvenile	157	archery	2010
2	55	2	M	yearling	230	muzzleloader	2010
3	33	3	F	adult	223	rifle	2010
4	27	1	F	juvenile	250	archery	2010
5	26	3	M	adult	211	muzzleloader	2010
6	15	2	M	yearling	158	muzzleloader	2010

View the 2011 data structure. Recall that we had modified the Sex and Species variables in harvest.data10, so these are both factors now. We will need to do the same for harvest.data11

> head(harvest.data11)

	Unit	Species	Sex	Age	Weight	Method	Year
1	61	4	1	adult	190	archery	2011
2	42	4	1	yearling	179	rifle	2011
3	42	3	1	yearling	242	rifle	2011
4	21	3	1	juvenile	216	rifle	2011
5	49	4	1	adult	156	archery	2011
6	26	3	2	adult	199	muzzleloader	2011

```
> harvest.data11<-within(harvest.data11, Species<-factor(Species))
> harvest.data11<-within(harvest.data11, Sex<-factor(Sex, labels=c("M", "F")))
```

Use the rbind() function to combine the two data sets
rbind(*df, additional rows*) #adds rows to a data frame; additional arguments can be used

```
> harvest.data.combined<-rbind(harvest.data10, harvest.data11)  #2010 first, then 2011
```

```
> harvest.data.combined[c(1:3,48:50),]      #View first 3 and last 3 rows of combined data
   Unit Species Sex Age      Weight Method      Year
1  6    1       M   juvenile 157    archery     2010
2  55   2       M   yearling 230    muzzleloader 2010
3  33   3       F   adult    223    rifle      2010
48 45   4       M   adult    212    muzzleloader 2011
49 51   4       M   yearling 250    archery     2011
50 30   4       M   juvenile 201    muzzleloader 2011
```

Example: You are investigating mountain lion fitness and you have 100 collared animals

- These animals are captured annually to collect fitness measures
- You have 2 different data sets. The first is a dataset representing Individual IDs, their date of initial capture in 2008, and their weight at time of capture.
- The second data frame contains the date and weight of all 100 animals during their recapture in 2009.
- You want to combine these two data sets, but want each individual to be represented by only a single row.

Import the data for 2008 and the data for 2009. Store as mt.lion.data08 and mt.lion.data09, respectively. If you imported the data earlier with the load.conference() function:

```
> mt.lion.data08<-MW.data[[3]]; mt.lion.data08<-MW.data[[4]]
```

```
> mt.lion.data08[1:4,]      #view first 4 rows of 2008 capture data
   PUCO.IDs  Date.of.Capture1  Weight1
1  64F       19Nov2008         127.4343
2  56F       12Nov2008         133.9920
3  26M       16Dec2008         119.4594
4  59M       10Nov2008         111.4275
```

```
> mt.lion.data09[1:4,]      #view first 4 rows of 2009 capture data
   PUCO.IDs  Date.of.Capture2  Weight2
1  64F       2Dec2009          118.6665
2  5M        29Oct2009         114.5413
3  88F       22Nov2009         121.0169
4  42M       13Dec2009         115.2921
```

Combine the data sets based on a common field
merge(*df1, df2*) #combine by common fields, additional arguments can be used

> mt.lion.data0809<-merge(mt.lion.data08, mt.lion.data09)
#merge the two data sets, there is only one common field so you do not need to identify

>mt.lion.data0809[1:4,] #view first 4 rows of merged data

	PUCO.IDs	Date.of.Capture1	Weight1	Date.of.Capture2	Weight2
1	100F	10Nov2008	126.90221	15Dec2009	117.7457
2	10M	25Sep2008	92.83458	9Nov2009	118.3425
3	11M	18Nov2008	95.32404	5Dec2009	118.7353
4	12F	14Nov2008	114.31999	15Nov2009	119.7567

How could we reorder these columns?
How could we drop columns if there are some data that do not want to include?
> mt.lion.data0809[,-c(2,4)] # - indicates that columns 2 and 4 should be dropped
> mt.lion.data0809[,c(1,2,4,3,5)] #providing the column indices in the desired order reorders

Now: sort by the lion's weight at time of initial capture in decreasing order
sort(*vector*) #Sorts a vector but does <u>not</u> sort a data frame

> sort(mt.lion.data0809) #if we pass sort() a data frame, we get an error
Error in `[.data.frame`(x, …) : undefined columns…

> sort(mt.lion.data0809$Weight1, decreasing=TRUE)
[1] 135.78118 133.99202 133.78551 133.52239 133.49324 133.22172 132.53796
[8] 132.05205 131.48827 130.79640 130.61864 130.42739 129.59780 129.05274
[15] 129.02911 128.38725 128.08248 127.43430 127.43111 127.07528 127.03362
……
[99] 94.29917 92.83458

- If we pass sort() a column (vector) from a data frame, it returns only that vector sorted, not the data frame, which is what we desire

Finding the order of a vector can be used to identify the sorted location of each element
order(*vector*) #Determines the sort order and can be used to manipulate data frames

> order(mt.lion.data0809$Weight1)
[1] 2 31 54 3 83 75 30 99 76 89 …

- Orders elements in increasing order, thus element 2 is the smallest value
- Setting the argument decreasing=TRUE will change the direction of the order

> ord<-order(mt.lion.data0809$Weight1, decreasing=TRUE) #Save the decreasing order
>mt.lion.data0809<-mt.lion.data0809[ord,] #Apply the order as the row reference

> head(mt.lion.data.0809) #after sorting with order(), view the first 6 rows

	PUCO.IDs	Date.of.Capture1	Weight1	Date.of.Capture2	Weight2
9	17F	27Dec2008	135.7812	10Oct2009	118.0855
52	56F	12Nov2008	133.9920	15Sep2009	116.7297
28	34M	6Nov2008	133.7855	8Sep2009	118.3936
87	88F	14Nov2008	133.5224	22Nov2009	121.0169
59	62M	17Sep2008	133.4932	27Dec2009	117.0685
77	79F	5Sep2008	133.2217	27Oct2009	114.2977

Next: we want to select only Males (indicated by M in the PUCO.IDs)
We need to first split out the Sex from the PUCO.IDs

Use the 'stringr' package"

- Search your library to see if you already have it
- If not, install the 'stringr' package

```
> library(stringr)                                  #Load the stringr package

str_sub(vector, start char idx, end char idx)     #Select only the portion of each vector
                                                  #element from the indices of the start and
                                                  #end characters in the character string
                                                  #Note: a "-" character index starts counting
                                                  #from the right
> str_sub(mt.lion.data0809$PUCO.IDs, -1, -1)
[1] "F" "F" "M" "F" "M"…
> factor(str_sub(mt.lion.data0809$PUCO.IDs, -1, -1))
[1] F F M F M…
Levels: F M
```

We want to store sex as a factor back to the data frame as a new column

```
> mt.lion.data0809<-cbind(mt.lion.data0809, Sex= Insert.Code.Above.Here)
> mt.lion.data0809<-cbind(mt.lion.data0809, Sex=
                                    factor(str_sub(mt.lion.data0809$PUCO.IDs, -1, -1)))
```

Use subset() to select only males from the data set

subset(*df, condition*) #select a portion of a data set

```
> mt.lion.data0809M<-subset(mt.lion.data0809, mt.lion.data0809$Sex== "M")
```

DESCRIPTIVE STATISTICS

Common basic descriptive statistics:

```
median()    #returns the median value from a vector
mean()      #calculates the average value of a vector
min()       #returns the minimum value from a vector
max()       #returns the maximum value from a vector
range()     #returns both the min and max from a vector
sd()        #calculates the standard deviation of a vector
var()       #calculates the variance (sd^2) of a vector
sum()       #calculates the total sum for a given vector
```

Note: All of these functions work as you might expect, with the first argument being a vector for which you want the statistic

- NA values will result in a calculation of NA, unless the argument na.rm=TRUE

Example: R provides example data sets as part of the base package. The "trees" dataset includes the girth, height and volume for Black Cherry Trees

Take a look at the data structure using head()

```
> head(trees)
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
4  10.5     72   16.4
5  10.7     81   18.8
6  10.8     83   19.7
```

For summary statistics, should first check for missing values:

```
> is.na(trees)          #is.na() returns a logical response of TRUE or FALSE regarding if it is a NA
      Girth Height Volume
 [1,] FALSE  FALSE  FALSE      The output is truncated here, but in general it will return
 [2,] FALSE  FALSE  FALSE      a TRUE or FALSE for each cell in the data frame
 [3,] FALSE  FALSE  FALSE
  …                            This may be an inefficient way of determining if there are
[31,] FALSE  FALSE  FALSE      NA values for very large data frames

> sum(is.na(trees))     #sums up all of the cells returned by is.na() as TRUE
[1] 0                   #a more efficient way of assessing the number of NA values, in this case 0
```

Calculate the summary statistics for tree girths:

```
> median(trees$Girth); mean(trees$Girth)
[1] 12.9
[1] 13.24839
```

```
> sd(trees$Girth); var(trees$Girth)
[1] 3.138139
[1] 9.847914

> min(trees$Girth); max(trees$Girth)
[1] 8.3
[1] 20.6

> range(trees$Girth)
[1]  8.3  20.6
```

Again, these only work on vectors. summary() works on both vectors and data frames, but does not provide all of the summary statistics that may be desirable

```
summary()      #supplies descriptive statistics for vectors or data frames

> summary(trees$Girth)
   Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
   8.30    11.05     12.90     13.25     15.25     20.60

summary(trees)
     Girth           Height       Volume
 Min.   : 8.30   Min.   :63   Min.   :10.20
 1st Qu.:11.05   1st Qu.:72   1st Qu.:19.40
 Median :12.90   Median :76   Median :24.20
 Mean   :13.25   Mean   :76   Mean   :30.17
 3rd Qu.:15.25   3rd Qu.:80   3rd Qu.:37.30
 Max.   :20.60   Max.   :87   Max.   :77.00
```

<u>Dealing with NAs</u>:

NAs often do not respond to the same methods as other values

For example, suppose we had a vector x and wanted to remove all values that were == 0. This could be accomplished with

```
> x<-x[x!=0]          #Return and store the vector, excluding those elements that were 0
```

This will not work with NAs though. Instead, need to apply the is.na()

```
> x<-x[!is.na(x)]     #Remove NAs–compare to x[x!=NA]
```

Other common NA methods include:

```
> x[is.na(x)]<-0      #Replace NAs with 0
> x<-na.omit(x)       #drop NAs from vector x
```

IMPLICIT LOOPS

There is typically a need to *apply* descriptive statistics (or other methods) to multiple, specific variables or columns in a data frame

- This can be accomplished using implicit loops:

sapply() – returns a simple data type (i.e., vector)
tapply() – returns a table data type

Lets utilize a data set with body measurements for ringtails to explore each of these functions

```
> data.file<-"C:/Users/RLonsinger/Documents/R_Datasets/ringtail.data.txt"        #Identify file
> ringtail.data<-read.table(file=data.file,  header=TRUE, sep="\t")              #Import
```

Alternatively, if you imported the data earlier with the load.conference() function:

```
> ringtail.data<-MW.data[[6]]
```

```
> View(ringtail.data)            #Take a look at the data format
```

```
> length(ringtail.data[,1])      #determine how many observations (rows are in the data set
[1] 184
```

Note: There are many other ways to get the number of rows/observations - e.g., str(ringtail.data), attributes(ringtail.data), length(ringtail.data$ID), or nrow(ringtail.data)

```
> names(ringtail.data[5:10])   #identify columns that we want to apply the statistical methods to
[1] "ROSTRUM"   "LEFT.EAR"  "R.HIND.FT" "HEAD"     "BODY"     "TAIL"
```

sapply() applies a method to any number of columns and returns the results as a vector

sapply(*data frame, method,…*) #… can pass arguments to the method being applied

```
> sapply(ringtail.data[,5:10], mean, na.rm=TRUE)
 ROSTRUM  LEFT.EAR    R.HIND.FT   HEAD        BODY          TAIL
 3.100000   4.720139   6.519310   9.474766   31.412281     36.008904
```

You could pass user defined functions to sapply() as well, if you wanted to perform the same procedure over multiple columns in a data frame!

tapply() allows you to condition the descriptive statistics of one variable with another factor variable (i.e., one that is nominal or categorical) and returns a table

tapply(*df$continuous.variable, df$grouping.variable, method, …*)
…can pass additional arguments to the method

```
> tapply(ringtail.data$WEIGHT, ringtail.data$SEX, mean, na.rm=TRUE)
 F              M
 967.6471       1209.3776
```

PARENTHESES & BRACKETS

Parentheses:

- Parentheses contain an expression to evaluate and influence the order of operations

```
> 3 * 3 + 6          #Performs the multiplication first, then the addition
[1] 15

> 3 * (3 + 6)        #Parentheses alter the order of operations, performing
[1] 27               #operations inside the function first
                     #Interprets ( ) as the function ( )
```

Square brackets:

- Generally follows an object's symbol and contains indices for data within an object

[ ] Generally follows an object symbol – e.g., x[1]
Contains indices for data within an object

[[ ]] Contains indices for data within an object
More commonly used with lists, but may be used elsewhere

Note: Square brackets are commonly multi-dimensional and/or stacked

```
E.g., ListA[[1]][1,1]      #Return the element in row 1 and column 1 of element 1 in ListA
                           #Suggest that element 1 of ListA must be a 2D object

      ListA[[1]][,1][1]    #Selects the same data as above
                           #Selects the first element in column 1 of element 1 in ListA

      df[[1]]              #Returns the first column of data frame df as a vector
```

Brackets can be used to simple subsetting, sorting, and data management

```
> subset(mt.lion.data0809, mt.lion.data0809$Sex=="M")   #Subset and return only the males
> mt.lion.data0809[mt.lion.data0809$Sex=="M", ]         #Select males via a conditional index

> mt.lion.data0809[mt.lion.data0809$Sex=="M" & mt.lion.data0809$Weight2>125, ]
                                                  #Select only males >125 lbs in year 2 (2009)
```

Curly brackets:

- Evaluates a series of separate expressions and returns only the last expression
- Often used to group expressions for functions and looping structures

{ } Contents evaluated in the current working environment
f{ } Contents evaluated in a new environment

CONDITIONAL STATEMENTS

Two functions are commonly used for conditional statements:
if( ) #evaluates a logical condition for a single value (not a vector)
ifelse( ) #evaluates logical conditions for a vector

General structure for if( ):

if(*condition*) *true_expression* else *false_expression*

If multiple expressions are carried out under one or both conditions, structure is often:

```
if(condition){
  true_expressions
  } else  false_expression
```

Or:

```
if(condition){
  true_expressions
  } else {
  false_expressions
  }
```

```
> if(trees$Height[1]>trees$Girth[1]) "Correct"
[1] "Correct"

> if(trees$Height[1]>trees$Girth[1]) "Correct" else "Potential Issues"
[1] "Correct"
```

General structure for ifelse( ):

ifelse(*condition, true_expression, false_expression*)

```
> a <- c(6:-4)
> sqrt(a)          #Generates warning
[1] 2.45  2.24  2.00  1.73  1.41  1.00  0.00  NaN  NaN  NaN  NaN

> ifelse(a >=0, a, NA)                    #Change neg values to NA
[1]  6  5  4  3  2  1  0 NA NA NA NA

> sqrt(ifelse(a >= 0, a, NA))             #Avoids error, passes NA instead of NaN to sqrt( )
[1] 2.45  2.24  2.00  1.73  1.41  1.00  0.00  NA  NA  NA  NA
```

LOOPING LANGUAGE

Extremely valuable for processing large datasets, repeating processes, and simulations

Primary looping constructs:

repeat{*expression*} #Repeat an expression – repeat followed by { } rather than ( )

```
> i <- 10                                          #Initialize a vector i with the value
> repeat{                                          #print(i) = print i on each loop
      if(i > 100) break else {print(i); i <- i + 10}   #Without conditions → infinite loop
      }                                            #the keyword "break" stops looping
```

while(*condition*) *expression* #Repeat an expression while a condition is TRUE

```
> i <- 10
> while(i <=100) {                  #if conditions are never met, will be an infinite loop
      print(i); i <- i + 10}        #Can use the keyword "break" to stop looping
      }
```

for(*var* in *list*) *expression* #Iterate through elements of a vector or list and evaluate an #expression on each element

```
> b <- seq(10, 100, 10)             #Vector of 10-100, by 10, with the sequence
> for(i in b) print(i)              #Print i, which indicates element i

> for(i in seq(10,100,10)) print(i) #Or could imbed seq()  into the for()
```

Note: The function may not operate as desired if the vector/list is not numeric and/or the *expression* involves complex methods. On the fly conversion to numeric (e.g., with as.numeric()) may influence the desired order of operations (i.e., the order of the elements over which are looped).

Consequently, it may be better practice to loop over elements in a vector/list sequentially by employing a combination of ":" and the length() function.

```
> b <- seq(10, 100, 10)             #Vector of 10-100, by 10, with the sequence
> for(i in 1:length(b)) print(b[i]) #Print element i of vector b on each loop
```

Commonly used functions to help control loops/output:

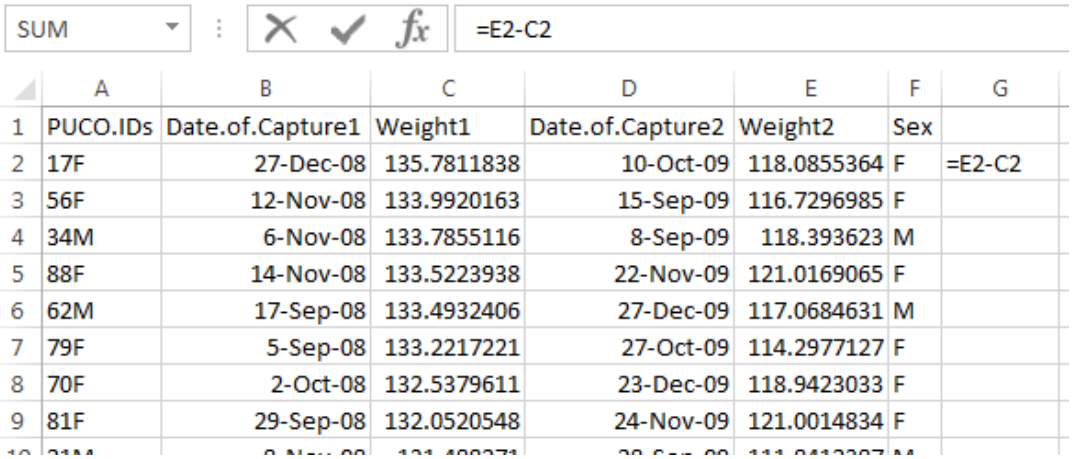
```
if( )       #Conditional test for single value
ifelse( )   #Conditional test of a vector
length( )   #Return the length of a vector
sum( )      #Return the total cumulative sum of arguments
c( )        #Concatenate or string together values
append( )   #Appends values to a vector at a specified location
```

Looping Example with mountain lion dataset (Dataset = mt.lion.data0809):

Steps to consider:

1. Calculate the change in body weight for each animal from 2008 and 2009
2. Store this new metric as a column in the data frame
3. Select only those animals that have lost weight

In Excel, we might accomplish this by selecting an empty column, calculating Weight2 – Weight1 and copying the formula down through the data set, then labeling the column in some way. To select those that have lost weight we may then sort by change in weight.



	A	B	C	D	E	F	G
1	PUCO.IDs	Date.of.Capture1	Weight1	Date.of.Capture2	Weight2	Sex	
2	17F	27-Dec-08	135.7811838	10-Oct-09	118.0855364	F	=E2-C2
3	56F	12-Nov-08	133.9920163	15-Sep-09	116.7296985	F	
4	34M	6-Nov-08	133.7855116	8-Sep-09	118.393623	M	
5	88F	14-Nov-08	133.5223938	22-Nov-09	121.0169065	F	
6	62M	17-Sep-08	133.4932406	27-Dec-09	117.0684631	M	
7	79F	5-Sep-08	133.2217221	27-Oct-09	114.2977127	F	
8	70F	2-Oct-08	132.5379611	23-Dec-09	118.9423033	F	
9	81F	29-Sep-08	132.0520548	24-Nov-09	121.0014834	F	

Translate this process into an R loop:

- Calculate the change in body weight for each animal from 2008 and 2009

  ```
  > Delta.weight <- NULL     #Create and empty vector to store new data
  ```

  To calculate a weight change for the first row of the dataset in R, simply:

  ```
  > mt.lion.data0809$Weight2[i]-mt.lion.data0809$Weight1[i]              #if i = 1
  ```

  Store value to the end of the new data vector

  ```
  > Delta.weight<-append(Delta.weight, __insert calculation above for row i____)
  ```

  Pull this all together and use a for loop to do this for each row

  ```
  > Delta.weight <- NULL     #Create and empty vector to store new data
  > for(i in length(mt.lion.data0809$PUCO.IDs)){
      Delta.weight<-append(Delta.weight,
                           mt.lion.data0809$Weight2[i]-mt.lion.data0809$Weight1[i])
      }
  ```

- Store this new metric as a column in the data frame

  ```
  > mt.lion.data0809<-cbind(mt.lion.data0809, Delta.weight)
  ```

- Select only those animals that have lost weight

  ```
  > mt.lion.data0809[mt.lion.data0809$Delta.weight<0, ]          #Or use subset()
  ```

USER DEFINED FUNCTIONS

We often develop code that we want to reuse and/or incorporate into more complicated scripts, simulations, and/or packages

- User defined functions provide a way to encapsulate a set of expressions and quickly recall and use them
- Operations an variables operated on within a function are done so in a separate *environment*

function (*arguments*) *body* #Create a function that takes *arguments* and executes the #expressions in the *body*

- arguments are symbol names which may or may not have default values
- body is a collection of expressions to be executed when the function is called
- { } can be used as a wrapper for long body expressions
- By default, user defined functions return the result of the last evaluated expression

Example: methods for determining the mean and standard deviation are incorporated in the base R download (i.e., mean() and sd(), respectively). Suppose you regularly need to calculate the standard error with the following general formula

Standard Error = Standard Deviation/ Square root of N

where N is the number of observations in the data set

This could be calculated in R for a vector "x" with:

```
> sd(x)/sqrt(length(x))
```

If the data contains NAs, then the calculation returns NA unless you set na.rm=TRUE:

```
> sd(x, na.rm=TRUE)/sqrt(length(x))
```

Still, if NAs do exist, the above will calculate the standard deviation based on the number of non-NA values, while the square root of N (i.e. , sqrt(length(x))) will be calculated based on the total number of elements in the vector (including NAs). The following can correct this:

```
> sd(x, na.rm=TRUE)/sqrt(length(x) - sum(is.na(x)))
```

This can be stored as a user defined function with function():

```
> SE <- function(x){
        sd(x, na.rm=TRUE)/sqrt(length(x) - sum(is.na(x)))
        }
```

And used to quickly assess the standard error of a continuous variable:

```
> SE(x=ringtail.data$WEIGHT)
[1] 16.50207
```

To return more than just the last expression evaluated, employ the return() function
return(*arguments*) #contained within a function, tells the function what to return

Re-write SE function to return not only the standard error, but also the mean and a count of any missing values removed during the calculation:

```
> SE.list <- function (x){
    SE<-sd(x, na.rm=TRUE)/sqrt(length(x)-sum(is.na(x)))      #Same as before
    return(list(Mean = mean(x,na.rm=TRUE),                   #Now return a named
                SE = SE,                                     #list with mean, SE,
                NA.removed = sum(is.na(x))))                 #and number of NAs
    }                                                        #removed
```

SIMULATIONS

Scripts that repeat some modeling process many times over

- Usually thousands of times, under variable conditions, and/or until some conditions is met
- Generally incorporates some form of stochasticity
- Records the outcome of each replicate
- Often interested in summary statistics and/or distribution of the outcomes
- May consider different conditions to draw inferences on the influence of different parameter values

The conditions varied will depend on the research question(s)

- Sample size
- Statistic used
- Parameter estimates (e.g., those with uncertainty)
- Variance, error, etc.
- Distributions

Simulations rely heavily on looping constructs, with loops being nested for different parameters

- Each loop tends to iterate over different conditions for a different parameter

Example: Use population estimates over 39 years for a brown bear population to conduct a count-based population viability analysis (PVA). A simple (perhaps the simplest) count-based PVA calculates a mean population growth rate (and associated standard deviation) from the data provided, then uses this summary data to project the population forward some period of time while incorporating stochasticity. It repeats this process many times, then summarizes the results by determining the proportion of projections that were below some threshold.

Steps to consider:

1. Import the PopEstimates.txt data set
2. Calculate population growth rate (lambda) for each time step (year)
    - Where lambda for a given time step is N_{t+1}/N_t (← this will require a loop)

3. Calculate the mean and SD of lambdas
    - mean(*vector*)
    - sd(*vector*)
4. Simulate population viability for 100 years
    - Determine a starting value for each simulation (Nhat=last value of PopEstimates)
    - Assume Lambda~Normal(mean, SD)
    - Randomly select lambda for each year via rnorm( )
5. Conduct 100 iterations
6. Summarize the result as Pr(Extinction)

Next: Consider how you might modify/add to this in order to evaluate sensitivity to variation (standard deviation) in lambda

- What additional loops would be required?
- What levels of SD should you consider?
- How can you store and summarize the output

Next: Convert this simulation code into a user defined function

- Which variables should be included as arguments?
- Which of these arguments are required entries and which should have default values?
- What control flow (e.g., conditional statements) would help make the function more user friendly?
- How should the output be returned to the user?

Finally: Use this new user defined function within a short script/loop to evaluate the influence of both variation in SD and also initial population size. View the final results as a data frame.

Note: One possible solution is provided in the supporting MW.Workshop.r file

GRAPHICS

Objectives: (1) provide examples of some of R's graphing capabilities and (2) review some of the commonly used arguments

R has the ability to produce visually appealing graphics

- Very easy to plot basic graphics for exploratory purposes
- Publication quality graphics can require quite a bit more use of settings (arguments)
- Recommend creating/retaining code for quality graphics
- Recommended packages: graphics and lattice

Common plotting methods (review the usage via the help() for details on arguments):

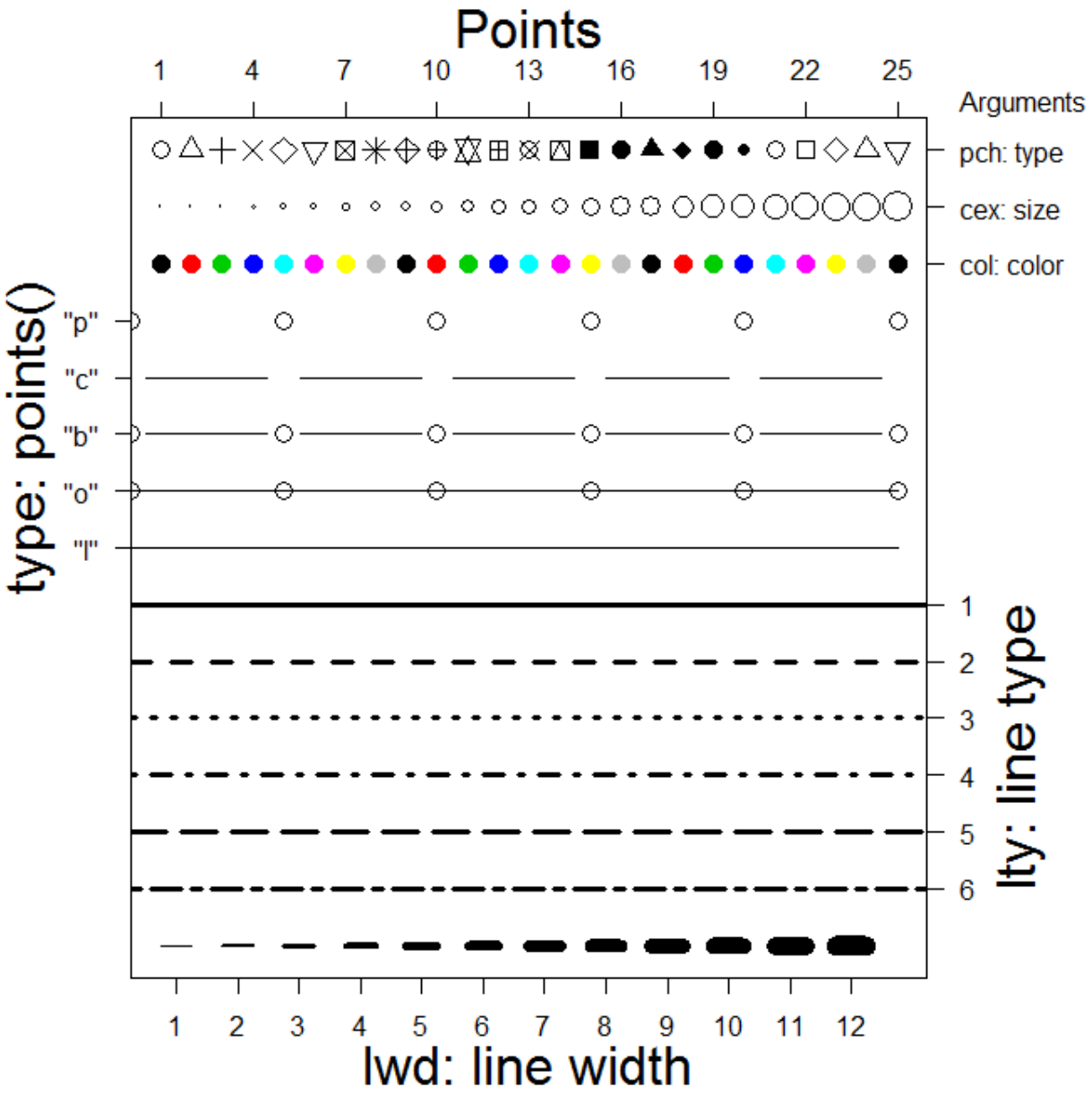
plot()	#Scatterplots	stripchart()	#Strip Charts
hist()	#Histograms	boxplot()	#Box Plots
pie()	#Pie Charts	barplot()	#Bar Plots

Common methods for modifying graphics:

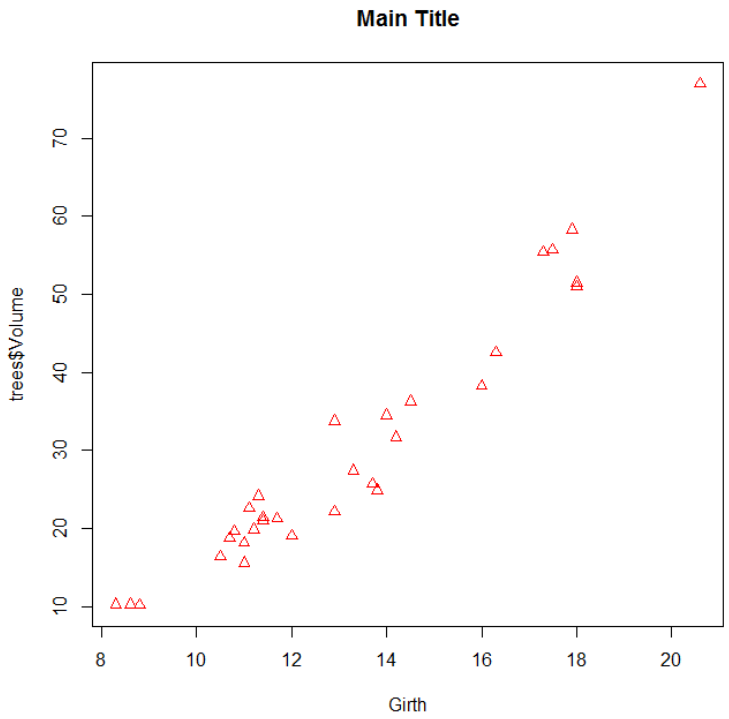
par()	#set graphical parameters
lines()	#add points joined with line segments to a plot
points()	#add a sequence of points to a plot
abline()	#add straight lines to a plot
axis()	#adds an axis to a plot
mtext()	#adds text to one of the four margins
text()	#adds text at specified location in a plot
legend()	#adds a legend to the plot

Common arguments to graphing and plotting methods:

Argument	Description	Argument	Description
main	Title for plot	col	Plot character color *
sub	Subtitle for plot	font	Plot character font *
xlab	Title for x-axis	cex	Plot character size *
ylab	Title for y-axis	pch	Plot character
lab	Vector of labels for axes	mfrow	Array plot arrangement by row
lty	Line type	mfcol	Array plot arrangement by column
lwd	Line width	mar	Margin dimensions by lines
xlim	Dimensions of x-axis	ylim	Dimensions of y-axis
*May be followed by .axis, .main, .sub, .lab to modify the axes, titles, and labels			

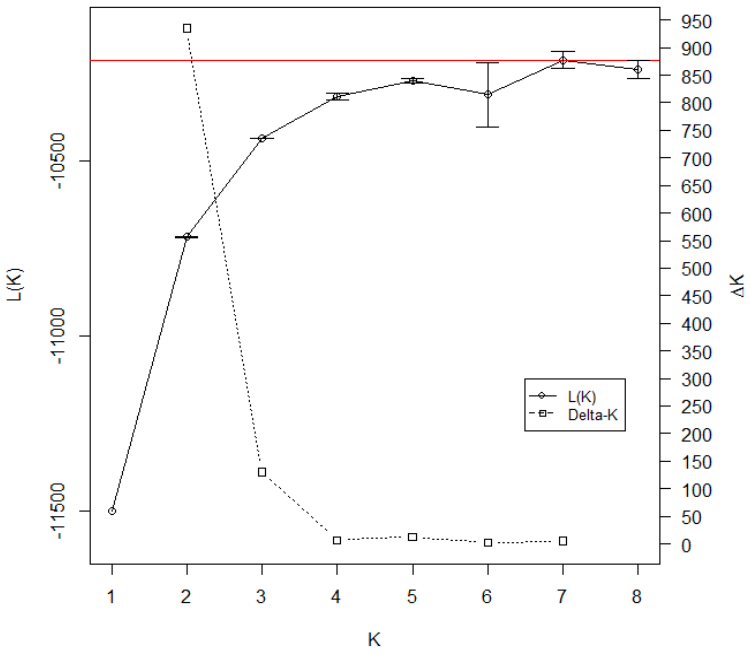

Points
1
4
7
10
13
16
19
22
25
Arguments
pch: type
cex: size
col: color
type: points()
"p"
"c"
"b"
"o"
"l"
lty: line type
1
2
3
4
5
6
1
2
3
4
5
6
7
8
9
10
11
12
lwd: line width

> plot(trees$Girth, trees$Volume, col="red", main= "Main Title", xlab="Girth", pch=2)

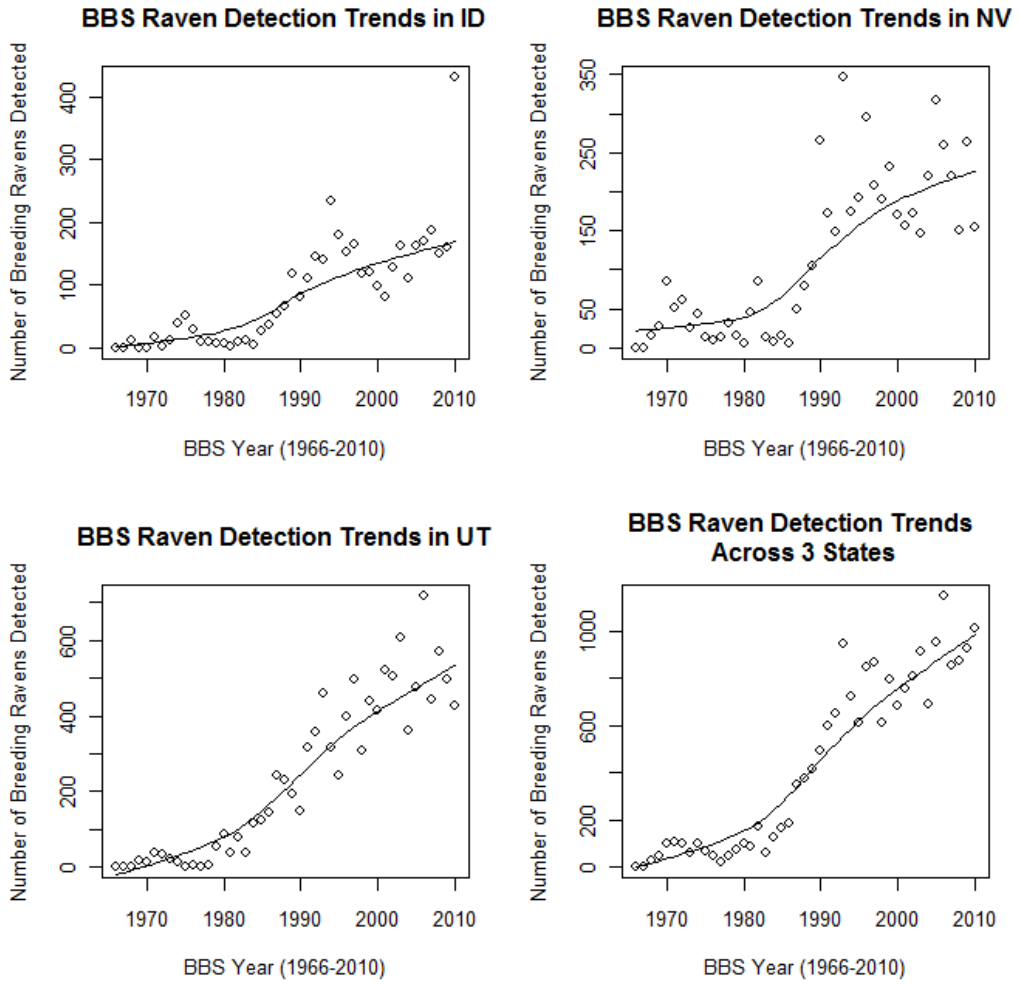


Examples of plots created in R include the following manipulations or changes:

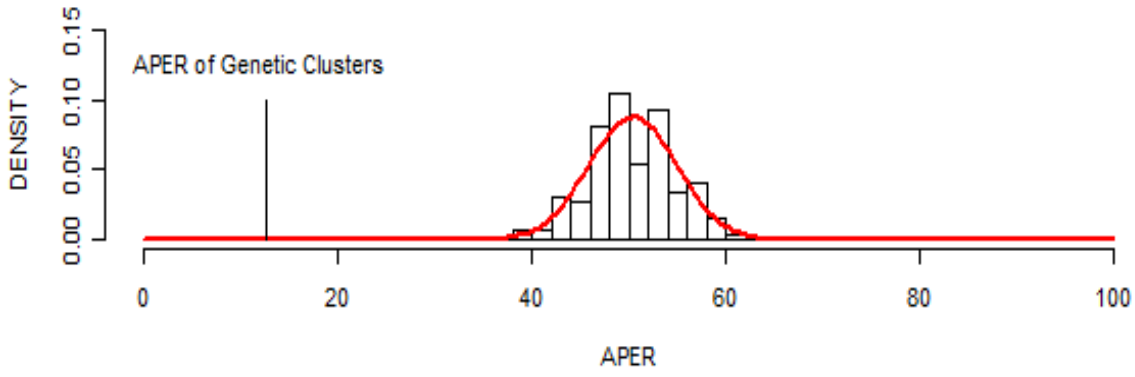
- Add multiple data series to the same plot with two y axes
- Add standard error bars, a asymptote line, and a legend



Plot multiple graphics together, add fitted lines, and modify x and y labels



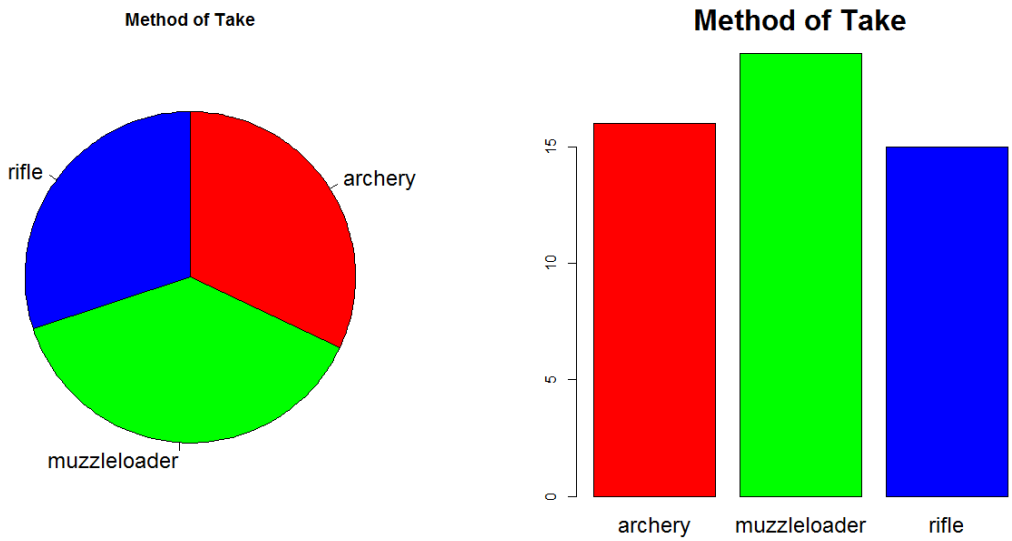
> hist(mc.rate.ra.lda.s2, freq=FALSE, ylab="DENSITY", xlab= "APER", xlim=c(0,100), ylim=c(0,0.15))



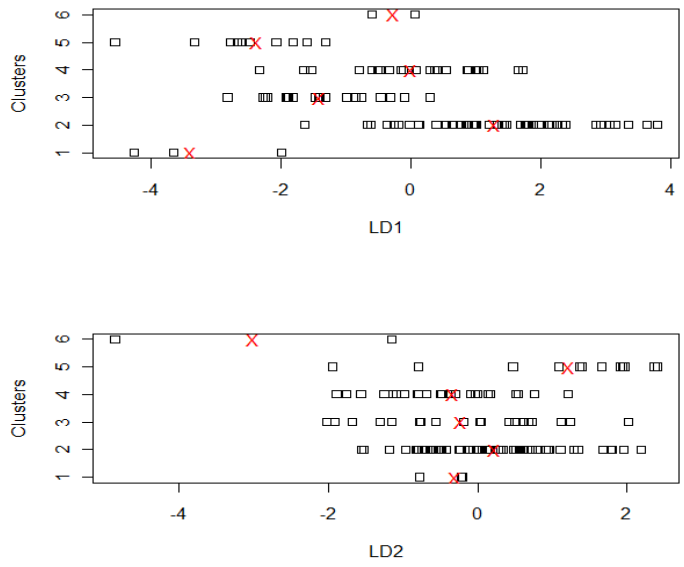
- Normal distribution added with lines(), vertical line and labels added with lines() and text()

> pie(harvest.data.plot, main="Method of Take",labels=levels(harvest.data.combined$Method), cex=1.5,clockwise=TRUE, col=rainbow(length(harvest.data.plot)))

> barplot(harvest.data.plot, main="Method of Take", names.arg= levels(harvest.data.combined$Method), cex.main=2, cex.names=1.5, col=rainbow(length (harvest.data.plot)))



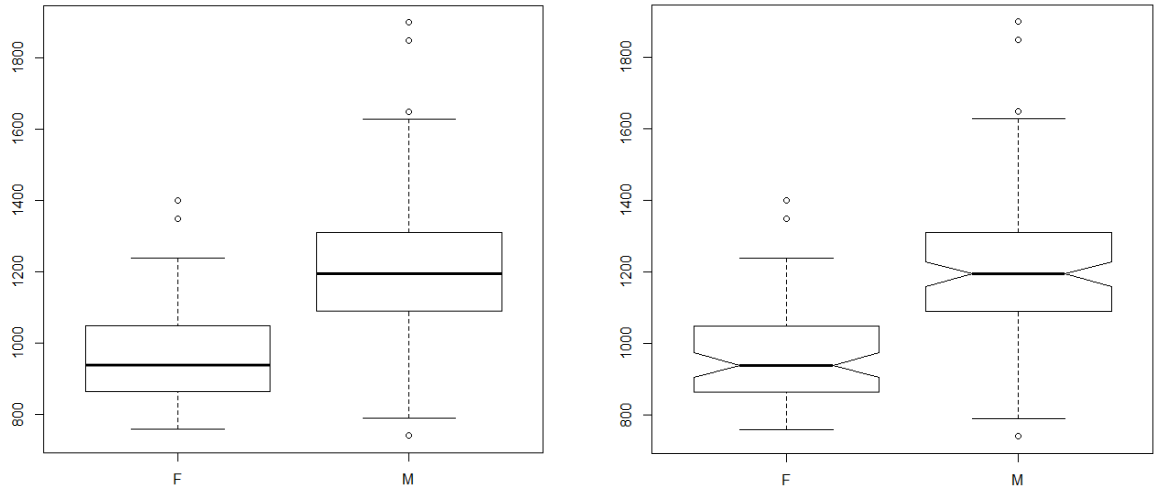
> par(mfrow=c(2,1)) #change parameters to plot two plots in a format of 2 rows and 1 column
> stripchart(ld$LD1~ ld$PopAssigment, xlab="LD1", ylab="Clusters")



- Means were added as red “X”s using points()
- Second plot code is excluded, but would be comparable

> boxplot(WEIGHT~SEX, data=ringtail.data) # "~" indicates a modeling relationship
with numeric ~ grouping

> boxplot(WEIGHT~SEX, data=ringtail.data, notch=TRUE)



> stripchart(WEIGHT~SEX, data=ringtail.data, method="jitter", col="red")
> boxplot(WEIGHT~SEX, data=ringtail.data, horizontal= TRUE, add=TRUE, boxwex=.5, outline=FALSE) #add=TRUE plots the second plot over the first