**Intro to R**

***BASICS***

Arithmetic:

+ \*

^ - /

%% (modulo)

* shows the remainder of the division of the number on the left by the number on the right.
  + 5%%3
* 2

Data Types:

Numerics: Decimal values (4.5) Integers: integers (4)(also considered numerics)

Logical: TRUE FALSE (Boolean values) Characters: text/string (put in quotations” “)

*Checking data types*

* **class()** will return the type of data that the variable is

***VECTORS***

Vector: One dimensional array that can store numeric, character, or logical data.

*Creating vectors*

* will create a vector that stores elements separated by a comma (combine function)
  + numeric\_vector <- c(1, 5, 10)
  + character\_vector <- c(“a”, “b”, “c”)
  + boolean\_vector <- c(TRUE, FALSE, TRUE)

*Naming vector elements*

* **names()** is used to name the elements within a vector, with the vector variable in the parenthesis
* names(numeric\_vector) <- c(“small”, “med”, “large”)

numeric\_vector

* small med large

1 5 10

*Summing elements in a vector*

* **sum()** will sum all of the elements in a vector
* sum(numeric\_vector)
  + - 16

*Comparing Vectors*

* You want to know if Wednesday’s earnings were greater than Friday’s for each earning and for the day as a whole:
* wed\_vector <- c(50, 100, 25)

fri\_vector <- c(900, -830, 25)

wed\_vector > fri\_vector

* FALSE,TRUE,FALSE

wedtotal\_vector <- sum(wed\_vector)

fritotal\_vector <- sum(fri\_vector)

wedtotal\_vector > fritotal\_vector

* TRUE

*Selecting an element from a vector*

* **vectorname[element position]**
  + You can select an element in a vector by putting the position of the element in brackets. (the first element of the vector is position 1)

*Selecting multiple elements from a vector*

* **vectorname[ c(#,#,#...)]**
  + states[ c(3, 5, 8)] will select the 3rd, 5th, and 8th elements from the states vector
* **vectorname[#:#]**
  + states[8:15] will select elements 8 through 15

*Selecting elements by name*

* **vectorname[“elementname”]**
  + states[“southern”] will select all of the elements under the name “southern”

Comparison operations:

< > == (equal to)

<= >= != (not equal to)

*Selecting elements by operations*

* wed\_vector > 55 will check which elements in wed\_vector are greater than 55
  + - wed\_vector > 55
    - FALSE, TRUE, FALSE

OR

* If you want to know the values of wed\_vector elements that are greater than 55
  + value\_vector <- wed\_vector > 55
    - wed\_vector[value\_vector]
    - 10

***MATRICIES***

Matrix: Collection of elements of the same data type, arranged into columns and rows (2-D)

*Creating a matrix*

* **matrixname(A1,A2,A3)**
  + A1 = the collection of elements that will be arranged into columns and rows
    - 1:9 will arrange c(1,2,3,4,5,6,7,8,9) into a 2-D figure
  + A2 = indicates whether the matrix is filled by row *(byrow = TRUE)* or by column *(byrow = FALSE)*
  + A3= determines the number of rows that the matrix will have *(nrow = #)*
    - matrix(1:9, byrow = TRUE, nrow = 3)
* *1 2 3*

*4 5 6*

*7 8 9*

*\*If the data length is not a multiple of the number of rows, the data will begin to repeat itself*

* matrix(1:9, byrow = TRUE, nrow = 2)
  + 1 2 3 4 5

6 7 8 9 1

*Summing rows/columns*

* **rowSum(matrix) & colSum(matrix)** will sum the values in your rows or columns of a matrix

*Adding rows & columns to a matrix*

* **rbind( , ) & cbind( , )** will add a row or column to the matrix
  + to add a row to the bottom of the matrix
    - rbind(matrix, new\_row\_vector)
  + to add a row to the top of the matrix
    - rbind(new\_row\_vector, matrix)
  + same format with columns
* locations\_totaled\_mat <- cbind(earnings\_matrix, location\_total\_earnings)

locations\_totaled\_mat

* inside outside locations\_totaled\_earnings

coach 50 100 150

sales 100 75 175

taxi 10 30 40

*Selecting an from a matrix*

* **matrix[#,#]**
  + Because they are 2-D, the 1st argument determines the row/s and the second argument determines the column/s
    - locations\_totaled\_matrix[2,2]
      * 75

*Selecting multiple elements*

* **[#:#, #:#] or [#:#, #] or [#, ] or [ ,#:#]**
  + Leaving an argument blank will select elements in every row or column(whichever is left blank)

*Arithmetic with matricies*

* + - \* / work in an elecment wise fashion
  + - 2 \* my\_matrix will multiply every element in my\_matrix by 2

*+ - \* / matricies together*

* matrix\_1 \* matrix\_2 will result in the elements from matrix 1 being multiplied by the element in the same position from matrix 2. The result will be a new matrix.

***FACTORS***

Factor: a data type that stores categorical variables. (unlike continuous variables)

* Categorical variable: limited to a set number of categories (gender = M or F)

*Creating Factors*

1. First create a vector where the elements are observations belonging to a set number of categories
   * + Size\_vector <- c(“small”, ”med”, “large”, “small”)
2. Then **factor()** will encode the vector as a factor
   * factor\_size\_vector <- factor(size\_vector)

factor\_size\_vector

* small med large small

levels: small < med < large

Factor Levels: The possible observation categories

Nominal categorical variable: a categorical variable without implied order

* + - animals = cat, dog, mouse (levels: cat dog mouse)

Ordinal Categorical Variable: a categorical variable with implied order

* + - temp = low, med, high (levels: low < med < high)

*Naming Factor Levels*

* **levels(factor\_vector) <- c(“name1”, “name2”…)** will assign names to factor levels
* Check the levels with levels(factor\_gender\_vector)
* If the result is F M then name them based on position Otherwise it might name Female M and Male F.

*Summarizing a factor*

* **summary(factor\_vector)** will give you an overview of the contents in your factor
  + - gender\_vector <- c(“M”, “F”, “M”)

factor\_gender\_vector <- factor(gender\_vector)

levels(factor\_gender\_vector) <- c(“Female”, “Male”)

summary(factor\_gender\_vector)

* Female Male
  + - * 1. 2

*Creating Ordered Factors*

* **factor(vector, ordered = TRUE, levels =(“lev1”, “lev2”, “lev3”…)**
  + The **ordered** argument tells R that we want the factors to be ordered
  + The **levels** argument tells R the order of the levels. (lesser levels are in lower positions, greater levels are in higher positions)
    - cash\_vector <- c(“little”, “lot”, “some”, “little”)

factor\_cash\_vector <- factor(cash\_vector, ordered = TRUE, levels = c(“little”, “some”, “lot”))

factor\_cash\_vector

* little lot some little

levels: little < some < lot

summary(factor\_cash\_vector)

* little some lot

2 1 1

*Comparing Ordered Factors*

* You can select elements from a factor using **[ ]** then store them as variables and compare the two variables
* day3 <- factor\_cash\_vector[3]

day4 <- factor\_cash\_vector[4]

day3 > day4

* TRUE

***DATA FRAMES***

* Matricies can only store elements of the same data type, but sometimes you need to organize data of different types (numeric, logical, character). That is what data frames are for.
* Data frames store the variables of a data set as columns, and the observations as rows.
  + - mtfood
* sugar fat carbs prot

rice 4 2 10 5

bagel 7 5 29 3

pop 45 10 10 0

egg 2 4 2 10

milk 2 8 1 9

*Understanding structure and main elements of your data set*

* **head(df)** prints the first observations of the df with the variables
  + - head(mtfood)
* sugar fat carbs prot

rice 4 2 10 5

bagel 7 5 29 3

* **tail(df)** prints the last observations of the df with the variables
* **str(mtfood)** prints the structure of the data frame, which includes: # of observations, # of variables, variable names, 1st observations, data type of each variable.
  + - str(mtfood)
* ‘data.frame’: 5 obs. of 4 variables:

sugar : num 4 7…

fat : num 2 5…

carbs : num 10 29…

prot : num 5 3…

*Creating a data frame*

* First, create vectors relating to the variables you want to have in your data frame
  + - name <- c(“rice”, “bagel”, “pop”, “egg”, “milk”)

type <- c(“side”, “side”, “drink”, “main”, “drink”)

sugars <- c( 4, 7, 45, 2, 2)

healthy <- c(TRUE, FALSE, FALSE, TRUE, TRUE)

* Then, use **data.frame(vectors)** to create a data frame with the vectors that you made, in the order that you want to see them
* The vector names will become the variable names
  + - food\_df <- data.frame(name, type, sugars, healthy)

food\_df

* name type sugars healthy

rice side 4 TRUE

bagel side 7 FALSE

pop drink 45 FALSE

egg main 2 TRUE

milk drink 2 TRUE

**Intermediate R**

***RELATIONAL OPERATORS***

Relational Operators

* See how one object relates to another
* Used to compare logical, string, and numeric data
* == equal to
* != not equal to
* <= less than or equal to
* >= greater than or equal to
  + - “hello” != “goodbye”
* TRUE

*Comparing different data types*

* R uses alphabetical order when comparing string, so “A” < “F” is TRUE
* TRUE corresponds to 1 and FALSE corresponds to 0 so FALSE < TRUE is TRUE

*Comparing vectors*

* You want to know which days your linkedin page had more than 10 views

linkedin <- c(16, 9, 13, 5, 2, 17, 14)

linkedin > 10

* TRUE

*Comparing vectors to vectors*

* + - linkedin <- c( 5, 15, 7, 10)

facebook <- c(8,4, 10, 15)

* When comparing these vectors, R will compare in a position-wise manner
  + - linkedin > facebook
* FALSE TRUE FALSE FALSE

*Summing results*

* **sum(vec1 > vec2)**
* sum(facebook > linkedin)
  + - 3 (because facebook was greater than linkedin 3 times.

***LOGICAL OPERATORS***

Operators: And, Or, Not

* **& (and)** takes 2 logical values (or results of comparisons) and returns TRUE only if BOTH logical values are TRUE
  + - x <- 3

x > 1 & x > 5

* FALSE
* **| (or)** is the same as and (&), except only one of the statements being compared must be true in order for the comparison to be true.
* **! (not)** operator negates the logical value/comparison that it acts on
  + - x <- 3

!(x < 5)

* FALSE

*Comparing logical operators*

* + - c(TRUE, FALSE, FALSE, FALSE) & c(TRUE, TRUE, FALSE)
* TRUE FALSE FALSE

\*using && or || will examine only the first element of each vector

***CONDITIONAL STATEMENTS***

*If Statements*

* **if(condition){**

**expression**

**}**

* The if statement requires that a condition be met.
* If the condition evaluates to TRUE: the code associated with the condition (aka the expression) is executed.
* If the condition evaluates to FALSE: nothing happens
  + - X <- -3

if(x < 0){

print(“x is a negative number”

}

* “x is a negative number”

*Else Statements*

* Else statements do not require a condition, but they must follow an if statement, another else statement, or an if else statement.
* The code associated with an else statement ( the expression) is executed if the condition of the if statement is not met( in a down-the-line manner).
* **if(condition){**

**expression1**

**} else{**

**expression2}**

\*If an else statement ever follows an if or else statement, it must be written on the same line as the previous closing squiggly bracket “{“.

* x <- 5

if( x < 0 ){

print(“x is a negative number”)

} else { print(“x is a positive number or zero”)

}

* “x is a positive number or zero”

*Else if statements*

* Else if statements must come between the if and else statements.
* Else if statements require a condition to be met and have an expression (code)
* **if(condition){**

**expression1**

**} else if(condition2){**

**expression2**

**}**

* x <- 5

if(x < 0){

print(“x is a negative number”)

} else if (x == 0){

print(“x is zero”)

} else { print(“x is a positive number”)

}

* “x is a positive number”
* As soon as one of the conditions is met, the expression is executed and R stops going down the line, even if there are other TRUE conditions in the code.

***LOOPS***

*While loop*

* Like if statements, it will execute the expression code if the condition is TRUE.
* But, it will continue to execute the code over and over, as long as the condition holds to be true.
* **while(condition){**

**expression**

**}**

* ctr <- 1

while(ctr <= 7){

print(paste(“ctr is set to”, ctr))

ctr <- ctr + 1}

* ctr is set to 1

ctr is set to 2

ctr is set to 3

ctr is set to 4

ctr is set to 5

ctr is set to 6

ctr is set to 7

***\*Closer look\****

This allows this to work.

It tells R to add the value of ctr to the end of “ctr is set to” after every run

print(paste(“ctr is set to”, ctr))

ctr <- ctr + 1

This tells R to increase the value of ctr by 1 after every run.

* As soon as ctr = 8, the condition is not met and nothing further is printed.
* Always make sure there is something in the code to stop the loop, otherwise it will go on forever.
* If statements cannot start on the same line as a “}”.

*Break Statement*

* A break statement will end the while loop.
* **while(condition1){**

**if(condition to break){**

**break**

**}**

**expression1**

* The break statement goes between the while loop’s condition and the expression code.
  + - ctr <- 1

while(ctr <= 7){

if(ctr %% 3 = 0){

break

}

print(paste(“ctr is set to”, ctr))

ctr <- ctr + 1

}

* ctr is set to 1

ctr is set to 2

ctr is set to 3

*Adding if/else statements to while loops*

* simply add the code
* **while(condition1){**

The whole body of the while loop is within a set of { }, even the if/else statements

**expression1**

**if(condition2){**

**expression2**

**} else {**

**expression3**

**}**

*For loop*

This is read as:

“For each variable (var) in sequence (seq), execute expression (expr)”

* **for(var in seq) {**

**expr**

**}**

* cities <- c(“NY”, “Paris”, “London”, “Tokyo”, “Rio”)

for(city in cities){

print(city)

}

* NY

Paris

London

Tokyo

Rio

* + Process:

1. At the start of the loop, R will recognize the sequence as a vector containing 5 elements.
2. R will then store the 1st element of the sequence (cities) into the variable we created (city).
3. R will then execute the expression and print “NY”. (end of 1st run)
4. R will then store the next element of the sequence (“Paris”) as city, and print it.
5. It will repeat this until all elements are printed.

*Break statements in For loops*

* Same as in while loops
* For(city in cities){

if(nchar(city) == 6){

break

}

print(city)}

* NY

Paris

*Next Statement*

* Same as break statement but just skips the rest of the code on the runs that satisfy the “next condition”.
  + **for(var in seq){**

**if(next condition){**

**next**

**}**

**expr**

**}**

You can now select which variables within the sequence that you want to run through the loop.

*For Loop Version 2*

* **for(var in 1:length (seq)){**

**print(seq[var])**

**}**

* for(a in 1:length(cities)){

print(paste(cities[a],”is on position”, i, “in the cities vector”))

} CHECK THIS CODE

* “NY is on position 1 in the cities vector”

“Paris is on position 2 in the cities vector”

“London is on position 3 in the cities vector”

“Tokyo is on position 4 in the cities vector”

“Rio is on position 5 in the cities vector”

*Nested loop*

* A loop inside a loop
* **for(var1 in seq1){**

**for(var2 in seq2){**

**expr**

**}**

**}**

* ttt
* o na x

na o x

x na x

for(i in 1:nrow(ttt)){

for(j in 1:col(ttt)){

print(paste(“on row”, i, “and column”, j, “ the board contains”, ttt[i,j]))

}

}

* “on row 1 and column 1 the board contains o”

…

…

…

*Using loops, breaks, next, and conditional statements together*

* linkedin <- c(5, 10, 1, 2, 8)

for( li in linkedin){

if( li > 5){

print(“You’re cool”)

You can add if statements anywhere in here, but they cannot be on the same line as a closing bracket }

} else { print(“Be more visible”)

}

if(li > 8){

print(“I’m done”)

break

} else if ( li < 2){

print (“Loser”)

You can add commands within the body too, as long as they don’t follow a closing bracket }

next

}

print(li)

}

***FUNCTIONS***

Function

Input Output

*Function Documentation*

* **help(function) or ?(function)**
  + will take you to a page with info on the function
* **args(function)**
  + will provide information about the function’s arguments so you don’t have to read through the whole help page to find it.

*Functions’ Arguments*

* + Most functions have arguments.
  + Some of the arguments are required arguments and must be inputted
  + Optional arguments are arguments that do not need to be inputted by the user
  + If the user does not input the optional arguments, there is a default setting that they will resort to.
  + But, if you want to change one of the optional arguments, to manipulate the function, you must specify it in the function’s arguments.

Ex: sd() has one required and one optional argument.

* **sd(x, na.rm = FALSE)**

Required Optional

* The na.rm argument specifies wether or not to remove missing values when computing the standard deviation. It is set to FALSE by default.
* If you have missing values the function will not work so you must change it to:

**sd(x, TRUE)**

*Writing Functions*

* **my\_fun <- function(arg1, arg2){**

**body**

**}**

* This will create a function called my\_fun()
* It uses arg1 and arg2 as arguments
* It performs the code in the body on the arguments and generates an output
  + triple <- function(x){

x \* 3

}

*Return Statement*

* **return()**
* The return command will return whatever is in the parentheses after the function processes the input if you put the return command at the end of the body

*Manipulating your Function’s Arguments*

* math <- function(a, b){

a \* b + a / b

}

* This will not work if there is only 1 value inputted.
* To stop this from returning an error you can make the second argument optional and set its default value to 1. ( simply write b = 1 for the second argument)
* Now math(4) will return 8 because 4 \* 1 + 4 / 1 = 8
* This function will still not work if the second argument is set to 0 because #/0 = infinity.
* To prevent infinity from being returned, we can use an if statement that tells R to return 0 if the second argument is 0.
* math <- function(a, b = 1){

if(b == 0){

return(0)

}

a \* b + a / b}

Example:

pow\_two <- function(x, print\_info = TRUE){

y <- x^2

if(print\_info == TRUE){

print(paste(x, “to the power of 2 is”, y))

return(y)

} else{ return(y)

}}

*Creating Functions Without Arguments*

* It’s simple
* **name <- function(){**

**body/command**

**}**

* hello <- function(){

print(“Hi there!”)

}

hello()

* “Hi there!”

***PACKAGES***

***lapply/sapply/vapply***

* The for loop allows us to run a function over elements of data structures.
* applys also allow us to run functions over all elements in a list or vector, but it is much easier.

*lapply()*

* **lapply(x, fun)**
* This applies the function(fun) to all elements of data structure (x), and returns the result as a list.
  + - prices <- c(1, 2, 3)

lapply(prices, triple)

* 3

6

9

*Changing arguments of functions in lapply()*

* If you want to change an optional argument in a function then use it in lapply(). Just define it right after the function in lapply().
  + - multiply <- function(x, factor){

x \* factor

}

lapply(prices, multiply, factor = 4)

* + - 4

8

12

*Anonymous functions in lapply()*

* lapply(prices, function(x, factor){

x \* factor}, factor = 5)

* + - 5

10

15

* lapply() returns objects in a list because lists are required to store heterogeneous content. That way lapply() can return objects of different classes.

*sapply()*

* aka simplify apply
* sapply() uses lapply() to apply the desired fuction to all elements of x BUT it also converts the list that lapply() generates into an array if possible. It also uses its USE.NAMES optional argument to name the objects (which can be set to false but is TRUE by default).
* **sapply(x, fun) OR sapply(x, fun, USE.NAMES = FALSE)**
* If simplification to an array is not possible, sapply() will return objects the same way that lapply() would.

*vapply()*

* Uses lapply() then tries to simplify the results BUT you must specify what type the return value will be, AND if the results cannot be simplified, it will throw an error.
  + - (temp is a list with 3 temperature recordings from 5 days)

minmax <- function(x){

c(min = min(x), max = max(x)}

sapply(temp, minmax)

* + - [1] [2] [3]

min 3 6 2

max 7 30 55

* In this example, lapply() put the results into an appropriate array with appropriate names.

***USEFUL FUNCTIONS***

Math Related

* **abs()**
* gives the absolute value of numerical elements
* **round()**
* rounds numerical elements
* **sum()**
* calculates the sum of the inputted array
* **mean()**
* calculates the mean
* **seq(x, y, by = z)**
* generates a sequence starting at x going to y in increments of z
* **rep(x, times = y)**
* replicates input x, y times
* **rep(x, each = y)**
* instead of repeating the entire input, it repeats each element of the input
* **sort(x)**
* sorts the input vector in ascending order
* **sort(x, decreasing = TRUE)**
* sorts in decreasing order

Data Structure Related

* **is.\*()**
* used to check the type of data structure
  + - is.list(linkedin)
    - FALSE
* is.vec(linkedin)
  + - TRUE
* **as.\*()**
* used to convert vectors to lists
  + - as.list(linkedin)

is.list(linkedin)

* + - TRUE
* **unlist()**
* Turns list into a non-list(vector)
* Also tries to give appropriate names
* **append()**
* Allows you to add elements to a vector or list in a readable way
* Merges lists or vectors
* **rev()**
* Reverses elements in a data structure

Regular Expressions

* Used to see if certain patters exist
* Replace certain patterns
* Extract certain patterns from string
* **grepl(pattern = “a”, x = animals)**
  + - TRUE FALSE TRUE
* This will check to see which elements have an “a” and which do not
* “^a” will check to see if there are elements that start with a
* “a$” will check to see if there are elements that end with a
* **grep(pattern = “ “, x = array)**
* grep() does the same thing as grepl() BUT it returns the results as the orientation numbers of the elements that had the pattern
  + - grep(pattern = “a”, x = animals)
    - 13
* **sub(pattern = “ “, “rep”, x = array)**
* sub() finds the desired **pattern** in **array** and replaces it with the **rep** string.
* But, it only replaces it after the first time it finds it.
* **gsub(pattern = “ “, “rep”, x = array)**
* Does the same thing as sub() but it replaces the pattern everywhere that it shows up

*Tools when using regular expressions.*

**|**

* You want to replace every a or I in the animals vector with o:

gsub(pattern = “a|i”, “o”, x = animals)

* + - “cot” “moose” “impolo”

**.\*** Matches any character zero or more times

**\\** escapes things

* @.\*\\.edu$ means you’re looking for -------@------.edu

$ tells R that this whole pattern is at the end of a word

.\* covers whatever is between @ and .edu

[\\s](file://s) matches a space

[0-9] matches the numbers 0 through 9 at least once

( [0-9]+ ) Does something but I’m not sure what

***DATES AND TIMES***

*Asking current date*

This is not simply string. It is a certain type of object: A date object

* **Sys.Date()**
  + - “2015 – 05 – 07”

*Asking current time*

* **Sys.time()**
  + - “2015 – 05 – 07 10:34:52 CEST”

*Creating date objects in standard format*

* **as.Date(“date”)**
* as.Date() will turn whatever date that is in the quotes into a date object, as long as it fits the format %Y - %m - %d

*Creating date objects not in standard format*

* **as.Date(“date”, format = “ “)**
* You can format the date almost any way you want using these:

%Y = 4 digit year %y = 2 year digit

%m = 2 digit month %d = 2 digit day of the month

%A = day of the week %a = abbreviated day of the week

%B = month %b = abbreviated month

*Creating time objects*

* **As.POSIXct(“time“)**
* As.POSIXct() will turn whatever is in quotes into a time
* You can also change the format of the time object using these:

%H = hours as a decimal (00 – 23) %I = hours as a decimal (01 – 12)

%M = minutes as a decimal %S = seconds as a decimal

%P = AM/PM indicator

%T = shorthand notation for %H %M %S format

*Date and time arithmetic*

* “date” + 1 will increase the day by 1
* “time” + 1 will increase the seconds by 1, but if the difference is too large it will increase the number of days
* **unclass(“time”) OR unclass(“date”)**
* unclass() will show the numerical representation of the number of days or seconds that your date or time is away from Jan 1, 1970 at midnight.

**Importing Data in R**

***IMPORTING FLAT FILES***

* Getting data from a different source into R

**Utils package**

*Flat Files*

* Flat files are simple text files that display data
* states.csv
  + - state, cap, pop, area

SD, Pierre, .85, 771

NY, Albany, 19, 545

csv stands for comma separated values

This is what it looks like in flat file form

* + - State cap pop area

1 SD Pierre .85 771

2 NY Albany 19 545

This is what it looks like in R

*Getting from flat file form to R*

* **read.csv(“csv file path”, stringsAsFactors = FALSE)**
* If the file is in your working directory just use the name of the file (states.csv).
* If it is located somewhere else you must build a file path using the **file.path** function.
* The stringsAsFactors argument stores strings as categorical variables when it is true, but it is set to FALSE by default.

*Creating a file path*

* + - If states.csv file is located in the “data sets” folder of the home directory, Use: path <- file.path(“~”, “datasets”, “states.csv”)

path

* + - “~/datasets/states.csv”

Then: read.csv(path, stringsAsFactors = FALSE)

*Calling str() on imported flat files*

* **str(read.csv(“file\_path\_name”))**

*Tab-delimited file data*

* Fields are separated by tabs
* **read.delim(“file\_path\_name”, stringsAsFactors = FALSE)**
  + - states.txt
    - state cap pop area

SD Pierre .853 771

NY Albany 19 545

Tab-delimited form

*Exotic file format data*

* + - states2.txt
    - state/cap/pop/area

SD/Pierre/.85/771

NY/Albany/19/545

Exotic form

* read.delim() will not work here because they aren’t separated by tabs
* **read.table(“file\_path\_name”, header = FALSE, sep “ “, stringsAsFactors = FALSE)**
* header tells R to make the 1st row of text into headers.
* sep tells R what is used to separate fields and it is set to space by default.
* colClasses() is an argument that designates the type of data of the resulting df. If one of the columns is set to “NULL” it will be skipped and not shown in the df. Ex: colClasses = c(“type1”, “type2”, “NULL”)

**readr package**

* **read\_csv()**
* readr’s equivalent of utils read.csv() except instead of printing a data frame it prints a *tibble:* a supercharged version of a data frame that can do more things.
* In readr, strings are NOT imported as factors by default
* **read\_tsv()**
* readr’s equivalent of utils read.delim().
* tsv = tab separated value
* **read\_delim()**
* readr’s equivalent of utils read.table().
* args: read\_delim(“file”, delim = “/”)
  + delim argument specifies how the fields will be separated.

|  |  |
| --- | --- |
| **Utils** | **readr** |
| read.table() | read\_delim() |
| read.csv() | read\_csv() |
| read.delim() | read\_tsv() |

*Setting column names in readr*

* Readr functions set the 1st row of text to the column names using the col\_names argument.
* You can also set column names using col\_names = c(“name1”, “name2”, ….)

*Setting column types in readr*

* col\_types is readr’s equivalent of utils colClasses argument.
* If you don’t specify it, R will guess the classes based on the first 30 observations

*Class type shorthand:*

c = character d = double(numeric) i = integer l = logical - = skip

ex: (col\_types = “ccdi”)

*skip and n\_max arguments*

* read\_delim(“states2.txt”, delim = “/”, skip = 2, n\_max = 3)
* This will skip 2 observations then print 3
* You must manually specify column names so that they don’t get skipped if they’re in the 1st row.
* skip argument designates how many lines of the flat file data will be skipped before R begins printing data.
* n\_max designates how many lines of data you will skip.

**data.table package**

* Its key metric is speed.
* Used for data manipulation in R.
* Its main function for data importation is **fread().**
* When given a data file without 1st line column names it creates its own column names.
* When given a data file WITH first line names, it uses those names.
* It infers the column types and field seperators.
* It does all of this without specifying arguments.
* Think of it as an improved version of read.table().

A few of its arguments:

**drop()** will remove selected variables

**select()** will keep selected variables

* + - Potatoes <- fread(“potatoes.csv”, select = c(6,8)) will select variables 6 and 8 of the potatoes file.

***IMPORTING EXCEL DATA***

**readxl package**

2 main functions:

* **excel\_sheets(file\_path)**
* Lists the different data sheet names in your excel file
* **read\_excel(file\_path)**
* Imports the sheet’s data into R
  + - excel data:

|  |  |
| --- | --- |
| Cap | Pop |
| NY | 160 |
| Ber | 301 |
| Mad | 168 |

|  |  |
| --- | --- |
| Cap | Pop |
| NY | 300 |
| Ber | 310 |
| Mad | 400 |

year\_1990 year\_2000

excel\_sheets(“cities.xlsx”)

* + - “year\_1990” “year\_2000”

read\_excel(“cities.xlsx”)

* + - Cap pop

NY 160

Ber 301

Mad 168

* You can specify what sheet to import with the sheet argument, either by name or position
  + - read\_excel(“cities.xlsx”, sheet = 2)

OR

read\_excel(“cities.xlsx”, sheet = “year\_2000”)

* + - cap pop

NY 300

Ber 310

Mad 400

* col\_names argument designates whether or not the 1st line of data contains column names(TRUE) or if R will assign names itself(FALSE). You can also set col\_names to a character vector containing the column names.
* col\_types­ argument determines what the class type the data will be. Setting it to NULL will have R guess the data types. You can also set it to a vector containing the column\_types: num, char, date, blank(removes the column)
* skip argument: specifies the number of columns that will be skipped before R starts importing data. If the column names are in the 1st row and you skip it, you must manually put in the column names with a character vector.

*Displaying multiple sheets*

* use **lapply()** in conjunction with **excel\_sheets()**, **read\_excel(),** and the path argument.
  + - lapply(excel\_sheets(“cities.xlsx”), read\_excel, path = “cities.xlsx”)

Function

x

\*read\_excel also uses the col\_names and col\_types arguments\*

**gdata package**

* tools for performing data manipulation
* read.xls() allows you to import excel data, but you can only handle xls format unless you install a driver to handle the newer xlsx format.
* There is no function in gdata to list the different sheets like excel\_sheets().
* Some people use gdata instead of readxl because readxl is still under development.

How it works

read.csv()

Perl

XLS CSV R data frame

*Using read.xls()*

* Basically the same as read\_excel() except it imports .xls files
* Because read.xls() uses Perl to convert an Excel file to a .csv then uses read.csv to import the data into R, all of the options you can specify in read.csv() can also be specified in read.xls().

Example:

You have a dataset urbanpop.xls with 3 sheets and the column names are in the first row. You want to bind all 3 sheets together into one data frame.

path <- “urbanpop.xls”

sheet1 <- read.xls(path, sheet = 1)

sheet2 <- read.xls(path, sheet = 2)

This will remove the 1st columns of these sheets. We did this because the column names are in the first row of all 3 sheets, so if we left them in, the first columns of the 2nd and 3rd sheets would be duplicates.

sheet3 <- read.xls(path, sheet = 3)

all\_data <- cbind(sheet1, sheet2[-1], sheet3[-1])

**XLConnect package**

* A bridge between excel and R
* Can do almost anything that you can do in excel but through R, so you can do it in a reproducible way.
* Works with xls and xlsx files
* **install.packages(“XLConnect”)** to install the package
* You might have to install Oracle’s Java Devel. Kit first

*Loading a workbook into R*

* **loadWorkbook(“workbook name”)**

*Loading sheet names*

* **getSheets(“workbook name”)**
* same as excel\_sheets() function

*Getting data from sheets*

* **readWorksheet(“workbook name”, sheet = name or position)**

*Specifying rows and columns*

Example: You want to produce the values in the 3rd and 4th row of the 2nd column.

readWorksheet(“name”, sheet = name/position, startRow = 3, endRow = 4, startCol = 2, header = FALSE)

Don’t need to specify an endCol when there are only 2 columns.

*Adding new sheets to a workbook*

* Start with your workbook
  + - **book <- loadWorkbook(“file-name”)**
* then, use **createSheet(book, name = “new sheet name”)** to create the new sheet
* now, actually fill the new sheet with data using **writeWorksheet(book, new sheet file, sheet = “new\_sheet”)**

The name of the sheet that you want to add your new data to.

The name of the file that you’re getting your new data from.

* In order to see these changes in Excel, you must save the workbook to a file.
* **saveWorkbook(book, file = “new file name.xlsx or .xls”)** will save the workbook to a file.

So you don’t overwrite the file you started with.

*Re-naming sheets*

* **renameSheet(workbook,”old name”, “new name”)**
* You must do this for each sheet you want to re-name.
* Must save file to see the results.

*Removing sheets*

* **removeSheet(workbook, sheet = “name or position”)**
* Again, you must save the file to see the results.

***IMPORTING DATA FROM DATABASES***

*Connecting to a database*

* Depending on the database you want to connect to, you will have to use different R packages
* For MySQL databases, use the RMySQL package
* For PostgresSQL, use the RPostgresSQL package
* For Oracle, use ROracle
* Conventions for manipulating database data are specified in the DBI package
* **install.packages(“package name”)** will install the package
* **dbConnect(RMySQL::MySQL( ), dbname = “name”, host = “courses.example.exex.com”, port = 3306, user = “username”, password = “pass”)**
* This MySQL function from the RMySQL package simply constructs the driver that dbConnect can use.
* You must specify the database name, where it is hosted, through which port you want to connect, and the credentials to authenticate yourself.

*Listing database tables*

* **dbListTables(connection object)**
* dbListTables() will list the names of all of the tables in the database

*Reading table data*

* **dbReadTable(connection object, “table name”)**
* The first argument specifies the connection to use.
* The second argument is the name of the table data that you want to import.
* The result is a data frame with the desired table’s info.

*Disconnecting the database*

* It is polite to disconnect the database when you’re done
* **dbDisconnect(connection object)** will disconnect the database

Example:

Import the information from all of the tables in the dab connection.

Names\_tables <- dbListTables(dab)

tables <- lapply(names\_tables, dbReadTable, conn = dab)

tables

* + - All of the table info will be returned
* The conn argument specifies the connection.

**SQL Queries from Inside R**

* If a database table is very large and you don’t want to import all of it, you can specify what to import.
* Relational databases typically use SQL as a language for querying.
* You can write SQL queries to retrieve data based on specific criteria.
* You can set these queries TRUE to a function specified by the DBI package, and implement it by an R package that depends on the database you’re using.
* The following cases involve writing SQL queries.

2 ways to get your information (examples)

|  |  |  |
| --- | --- | --- |
| id | Name | started\_at |
| 1 | Tom | 2009 – 05 – 17 |
| 4 | Frank | 2012 – 07 – 06 |
| 6 | Julie | 2013 – 01 – 01 |
| 7 | Heather | 2014 – 11 – 23 |
| 9 | John | 2014 – 11 - 23 |

First, load the DBI package and create a connection to the database “con”. Then, get the names of the employees who started after September, 1 2012.

1. Read the entire database table, then subset it in R

* employees <- dbReadTable(con, “employees”)

subset(employees, subset = started\_at > ”2012 – 09 – 01”, select = name)

* + - Name

Julie

If you don’t specify the select argument R will keep all the columns

Heather

John

* This code imports the entire table, then does subsetting in R to get the names from the employees table where started\_at > 2012 – 09 – 01.

1. Load the DBI package and create a connection to the database con.

* dbGetQuery(con, “SELECT name FROM employees WHERE started\_at > \”2012 – 09 – 01\””)
  + - Name

This second argument is an SQL query in the form of a character string.

Julie

Heather

John

If you write SELECT \* FROM… it will keep all columns

If you use an equals sign, use = not ==

* This code sends an SQL query to the database, where the query is then run on the database side and then only the results are imported to R. This way more efficient, especially when dealing with large sets of data.

*Selecting multiple variables in a query*

Example

There is a table in a database, named “people”, that has info on people and you want to find the age and country (both of which are variables) of married males, provided there is a married column that is equal to 1 when they are married:

* dbGetQuery(name\_, “SELECT age, country FROM people WHERE gender = \”male\” AND married = 1”)

If this 1 was a character value (not a numerical) you would need to use \”1\”

*SQL functions in the WHERE clause of an SQL query*

Example

CHAR\_LENGTH is a dedicated SQL function

* dbGetQuery(con, “SELECT id, name FROM users WHERE CHAR\_LENGTH(name) , 5”)
  + - id name

2 mike

3 thea

6 kate

*Inner Join clause*

* Using the **INNER JOIN** clause on an SQL query allows you to search for variables from different tables.
* (“SELECT name, post FROM users INNER JOIN tweats on users.id = user\_id WHERE date >\” 2015 – 09 – 19\””)

Not sure about the format of the code for this.

* This will select the name variable from the users table and the post variable from the tweats table, where date .2015 – 09 – 19.

***Importing data from the Web***

HTTP: HyperText Transfer Protocol

* System of rules for how data should be exchanged between computers.
* “Language of the Web”
* When you browse a webpage, your computer( the client) is actually sending an HTTP Request to the server, which sends back data representing the webpage (response).

*Reading Files from the Web*

* **a.read.command(“URL”)**
* The URL will start with [http://](NULL)
* The command is one of the commands that allows you to view files. (from readr: read\_csv, read\_tsv, read.csv, …)
* The argument is just the file URL
* R sees that it is a URL and does an HTTP GET request, then the server responds with a file that R can read in.

HTTPS: HyperText Transfer Protocol Secure

* Some websites only accept secure connections.
* You can only download their files with the https prefix.
* To get these files, do the same thing as with HTTP but use the HTTPS prefix instead.

*Downloading files from the web*

* **download.file(URL, destination\_path)**
* download.file() performs a GET request to the URL that you specified, then stores the contents of the response to the location that you specified.
* The second argument is where you want to import the file to. It stores the downloaded file to the location of that argument and stores it as that name.
* This will import the file into the destination you described in your second argument, which is coded like this:

**destination\_path <- file.path(“destination”, “name”)**

What you want to name the downloaded file

Where you want to put the downloaded file

* After calling the download.file() command, use a read function on the destination path, to import the data from the local file (if the file you’re downloading is an excel file, use a function from the readxl package).

*Creating file paths*

* **file.path(“destination”,”name”)**
* The file.path() command will create a file path.
* The first argument tells R the destination of the file.
* The second argument tells R the name of the file.

RData

* RData is a file type that is a very efficient form to store R data in.

*Loading an RData file*

* **load(“path”)** will load the RData file
* load() uses only one argument, which is the path to the file (not the URL).

**httr package**

* The httr package has a convenient **GET()** function which executes a GET request
* The result is a response
* **Variable <- GET(“file’sURL”)** will get the file stored in the URL
* **Content(variable, as = “raw”)** will extract the content of the URL File
* 1st argument = GET variable
* 2nd argument = how you want to retrieve the content (raw object: “raw”, character vector: “text”, R object) using **as.**

***APIs and JSON***

JSON

* A format to represent data
* Simple, concise, well structured
* Human readable
* Easy to interpret and generate for computers
* This makes it perfect to communicate with *Web APIs*

API = Application Programming Interfaces

* It’s a set of routines/protocols for building software components
* It is a way in which different software components interact

Web API

* An interface that gets data and processed information from a server, or that adds data to a server through the HTTP methods that were described previously.

Example: Twitter has an API. After authentication, you can request specific URLs from twitter, allowing you te get the tweets for a particular person. Twitter does all of the work for you on their servers and gives you the information you need. You can also use the API to comment on peoples tweets. There are many applications.

Example: There is an API called the OMDb API that can give information on any movie. If you visit the webpage(URL) specific to the movie Rain Man, you will be taken to a JSON that has everything you need to know about the movie. You can convert the JSSON to an R data structure by using the **jsonlite package.**

**jsonlite package**

* The jsonlite package converts data between JSONs and R data objects.

*Converting JSONs to R data*

* **fromJSON(“JSON or URL”)**
  + calling fromJSON() on the URL or the JSON will convert the JSON into a named R data list.

JSON object

* An unordered collection of name:value pairs
* The name elements can only be string, but the value elements can be string, number, Boolean, null, another JSON object, or JSON array.
  + - {“id”:1, “name”:”Frank”, “age”:23, “married”:false} *JSON*
* If you wrap it in quotes, you can turn it into an R string.
  + - ‘{“id”:1, “name”:”Frank”, “age”:23, “married”:false}’ *R data*

JSON array

* Ordered sequence of zero or more values

\*Use [ ] instead of ( ) when listing in JSON arrays and objects\*

* + - [4, 7, 4, 6, 4, 5] *JSON*
* Calling **fromJSON()** on it will turn it into a character vector
  + - fromJSON(‘[4, 7, 4, 6, 4, 5]’) *R data*

*Other jsonlite functions:*

* **toJSON()** converts R data back to JSON
* **prettify() and minify()** change how JSONS are displayed

JSON array/object examples

Json1 <- ‘[[1,2]], [3,4]]’

fromJSON(json1)

* + - 1 2

3 4

Json2 <- ‘[{“a”:1, “b”:2}, {“a”:3, “b”: 4}, {“a”:5, “b”:6}]’

fromJSON(json2)

* + - a b

1 2

3 4

5 6

***Importing Data from Statistical Software Packages***

Most common statistical software packages

SAS STATA SPSS

Statistical Analysis Software Statistics and Data Statistic Package for Social Sciences

Business analytics Economists Social Sciences

Biostatistics

Medical Sciences

Packages that import data

Haven Foreign

Newer Supports more data formats

Consistent, easy, fast

**Haven package**

*Importing SAS data using Haven*

* **read\_sas(“file path”)**
* Deals with SAS, STATA, SPSS and data files
* Simply pass the path to the data file and an R data frame results
  + - Import the data file ontime.sas7bdat into R

read\_sas(“ontime.sas7bdat”)

*Importing STATA files using Haven*

* **read\_stata(“file path”) OR read\_dta(“file path”)** will import STATA 13 and STATA 14 files.
* If one of your columns comes out as numbers when it is supposed to be characters, you can change the data type to factors or string using **as\_factor(path$colname) OR as.character(path$colname)**

**Cleaning Data in R**

***CLEANING DATA***

* Very important
* Collect, clean, analyze, report
* Most time consuming part of data science

3 Steps to cleaning data:

1. Exploring Raw Data
2. Tidying Data
3. Preparing Data for Analysis
4. Putting it all together

**Exploring Raw Data**

1. Understand the structure of your data
2. Look at your data
3. Visualize your data

*Understanding the structure of your data*

* **class()** will provide the class of the data.
* **dim()** will provide the dimensions of the dataset (#(columns), #(rows)).
* **names()** will provide the column names.
* **str()** will provide the # of rows (observations), # of columns (variables), column names, data types, and a preview of the data.
* **glimpse()** is the equivalent of str(), but it is from the ***dplyr package.***
* It attempts to preview as much of the data set as will fit on your screen
* **summary()** gives a 5 number summary for each column

*Looking at your data*

* **head()** shows the first 6 rows by default.
* The argument (n = #) specifies how many rows will be displayed.
* **tail()** does the same thing as head() except it shows the last rows.

*Visualizing your data*

* **hist(name$column name)** will make a histogram of all of the observations in that column.
* **plot(name$col1name, name$col2name)** will produce a scatterplot showing the relationship between 2 variables.

***TIDYING DATA***

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Age | Eye color | Height |
| Jake | 20 | B | 6 |
| Bob | 30 | R | 7 |
| Tim | 40 | G | 8 |

* Columns are called variables/attributes.
* Each individual color/age/name/height is called a value of the color/age/name/ height variable.
* All observations describe characteristics of a person, therefore the table only has one type of observational unit***.***
* If some rows described things other than characteristics of people, then the table would contain more than one type of observational unit.
  + Tidy data will have one type of observational unit per table.

*When column headers are values instead of variable names*

Wide datasets: Datasets that have more columns than rows

Long datasets: When there are more rows than columns

**Tidyr package**

* A package that is designed to help you apply the principles of tidy data.
* Here, we focus on a subset of functions that will help with common tidy data tasks.

*Gathering columns into key-value pairs*

* **gather(df name, key, value, ….)**
* gather() makes wide datasets long
* key = name of new key column
* value = name of new value column
* … = names of columns to gather or not
  + To gather, put the new name of the column.
  + To ignore (not gather) put -col name.
    - wide\_df
    - col A B C

x 1 2 3

y 4 5 6

gather(wide\_df, my\_key, my\_val, -col)

* + - col my\_key my\_val

x A 1

y A 4

x B 2

y B 5

x C 3

y C 6

*Spreading key-value pairs into columns*

* **spread(df name, key, val)**

Names of current columns that you want to spread

* Spread() makes long datasets wide
  + - long\_df
    - col my\_key my\_val

x A 1

y A 4

x B 2

y B 5

x C 3

y C 6

spread(long\_df, my\_key, my\_val)

* + - col A B C

x 1 2 3

y 4 5 6

*Separating data from a single column into multiple columns*

* **separate(dataset, col, into = c(“col1”, “col2”, …)**

Character vector of the new column names

Name of the column to separate

* treatments
  + - patient year\_mo

x 2010 – 10

y 2010 – 08

z 2010 – 05

* separate(tratments, year – mo, c(“year”, “mo”)
  + - * + patient year mo

x 2010 10

y 2010 08

z 2008 05

sep argument:

* sep = “ “
* whatever is in quotations should be what separates the 2 variables in the original column.

*Joining 2 separate columns into 1*

* **unite(dataset, name of new col, cols to unite)**
* with unite() the sep argument determines what will separate the 2 variables in the single column.
  + - unite(treatments,year – mo, year, mo)

*Data Type conversions*

Types of R data

* Character
* Numeric: NaN = not a number, Inf = infinity
* Integers: always include L after the number ( 1123L or 4L)
* Factors: created with factor()
* Logical: TRUE, FALSE, NA = used for missing values
* Date – Time
* **as.character()** will turn whatever is in parenthesis into a character string
* **as.numeric()** will turn whatever is in parenthesis into its numeric equivalent

**Lubridate package**

* + - ymd(“2015 – 08 – 25”)
* 2015 – 08 – 25
  + - mdy(“August 25, 2015”)
* 2015 – 08 – 25
  + - hms(“13:33:09”)
* 13H 33M 9S
  + - ymd\_hms(“2015/08/25 13:33:09)
* 2015 – 08 – 25 13:33:09
* All of these functions’ names represent the format of the input.
* They give the result in a standard format.
* They can also be applied to whole columns:
  + - dmy\_hms(dataset name$column name)

*String Manipulation*

**Stringr package**

Key functions of stringr:

* **str\_trim(“ string ”)**
* str\_trim() will trim all exvess white space at the beginning or end of character strings.
  + - str\_trim(“ this is a test “)
* “this is a test”
* **str\_pad(“string”, width = #, side = “left/right/both”, pad = “ “)**
* str\_pad() allows you to add characters or white space to either side of the character string.
* Its arguments determine how long the whole character string should be, which side to pad, and what the padding should contain.
* **str\_detect(data set, “string”)**
* str\_detect allows you to search for specific character strings within a dataset.
* The result is logical values
* friends <- c(“bob”, “tim”, “joe”)

str\_detect(friends, “bob”)

* TRUE FALSE FALSE
* **str\_replace(dataset, “string1”, “string2”)**

What we’re replacing it with

String being replaced

* str\_replace() allows you to replace specific character strings with other character strings

***MISSING AND SPECIAL VALUES***

Missing Data

* Data may be missing for many reasons
* It is represented by NA in R, but may appear in many other forms such as: #N/A(imported excel data), . (SPSS or SAS), or empty string.

Special Values

* Inf = Infinity
* NaN = Not a number

*Finding Missing Values*

* **is.na(df)** will check for NA’s reporting TRUE everywhere NA was found and FALSE everywhere it wasn’t.
* df <- data.frame(A = c(1, NA, 8, NA), B = c(3, NA, 88, 23), C = c(2, 45, 3, 1))

is.na(df)

* A B C

FALSE FALSE FALSE

TRUE TRUE FALSE

FALSE FALSE FALSE

TRUE FALSE FALSE

* you can sum the result to find out how many NA’s there are because TRUE = 1

**OR**

* **any(is.na(df))** will tell you if there are ANY NA’s in your dataset, and it will report either TRUE or FALSE
* **summary(df)** provides the number of NA’s and which column they are in.

*Dealing with missing values*

* **complete.cases(df)** will return a vector with 1 element for each row of the data.
* It returns TRUE if there are no missing data in the row (complete case) and FALSE otherwise.
* This can then be used to create a data frame with only the rows that have no missing values: df[complete.cases(df), ]
* A B C

1 2 3

8 88 3

**OR**

* **na.omit(df)** will create a new data frame without rows containing missing values. (This command performs both of the previous operations in one)
* social\_df$status[social\_df$status == “ “] <- NA
* Apparently this will replace any empty strings with <NA>, in the status column of the social\_df data frame

Outliers

* May be extreme measurements, variability in measurements, or a mistake
* May be discarded or kept

Obvious errors

* Values so extreme that they cannot be plausible
* Generally should be removed or replaced

*Finding Outliers and Errors*

* **hist(df$col, breaks = #)** will create a histogram.
* The breaks argument controls the number of buckets that the data are broken up into.
* **boxplot(dataset)** creates a boxplot for every column.
* **boxplot(dataset$col)** creates a boxplot for a single column.
* **Hist(df$col, right = FALSE)**
* When the right argument is set to FALSE, it will force values of zero to be bucketed to the right of the zero.