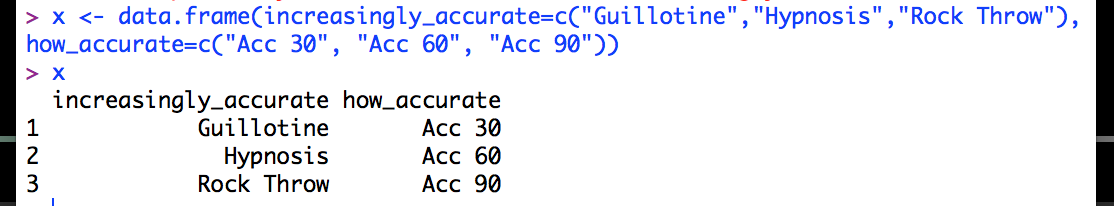
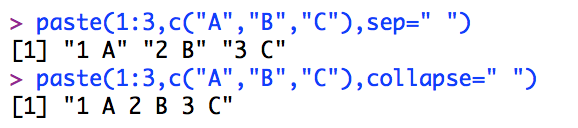
Coursera R

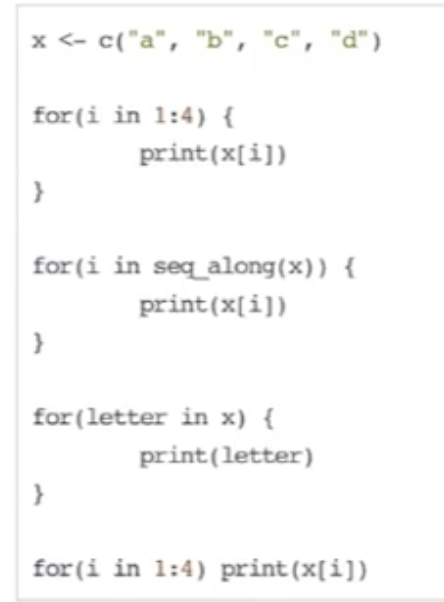
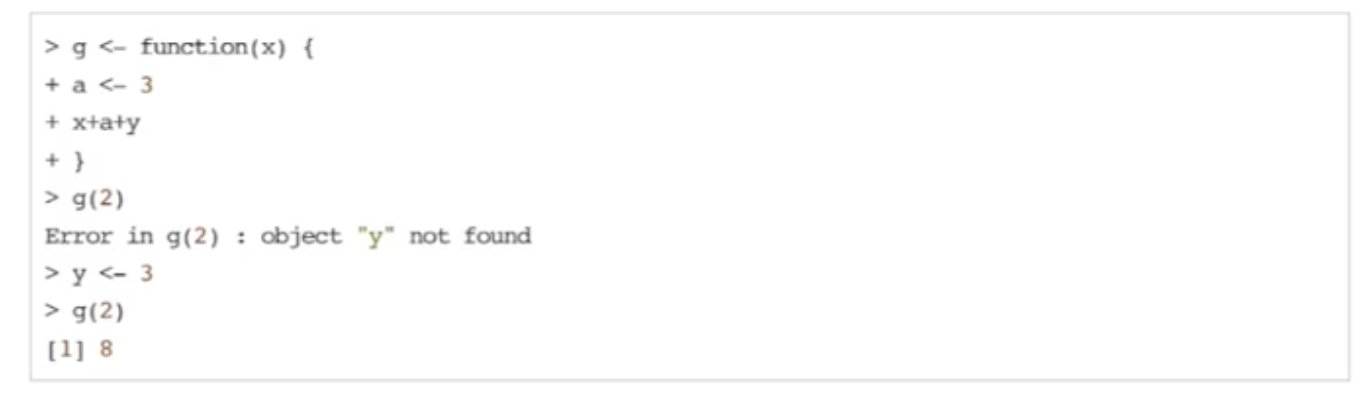
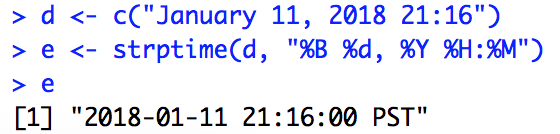
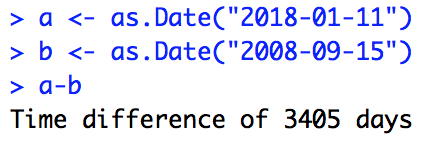
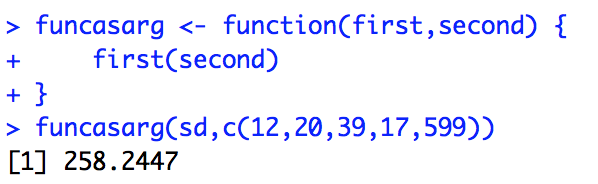
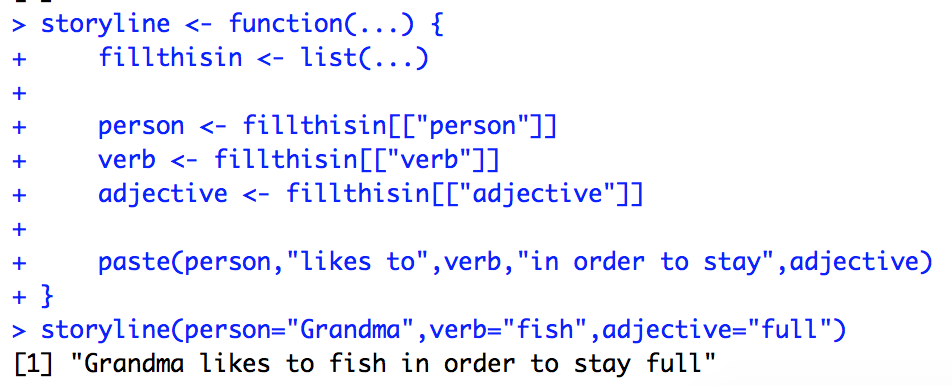
Found on my own:

* option-command-L to clear console
* stack() combines multiple vectors of a data frame; undo w/ unstack

Week 1

* getwd() in R to check working directory
* dir() to list all files in working directory
* ls() lists objects in workspace
* myfunction <- function(x) means myfunction takes input x
* source(“myfunction.R”)
* myfunction(x:y) = x through y, integer sequence
* <- to assign a value to a variable
* ## to insert comments
* x <- 1: x = auto-printing, print(x) = explicit printing
* Number in []: nth element of vector
* Atomic (object) classes: characters, real numbers, integers, complex numbers, logical (TRUE/FALSE or T/F)
* vector = most basic object; basically an array; can contain multiple objects but only of one type (only characters, only integers, etc.) except for lists; create w/ vector(class, length)
* vector 1-dimensional, matrix 2-dimensional
* L suffix makes object integer
* Inf = infinity
* NaN = not a number
* Attributes = attached to object such as names, dimensions, class, length, user-defined/metadata
* c() to concatenate
* Numeric vector default value 0
* Coercion: different object classes mixed in vector 🡪 program makes them all same class
* Explicit coercion: as.numeric(x) makes all objects numbers, as.logical(x) makes T/F, etc.
* NA = nonsensical coercion
* list()
* Matrix = atomic vector w/ dimension attribute; filled by column
* dimension attribute: # of rows, # of columns
* dim(m) <- c(x,y) assigning dimension attribute to vector
* attributes()
* matrix(x:y, nrow= , ncol= ) nrow & ncol can be left blank?
* Factor = integer vector where each integer has a label (low = 1, medium = 2, high = 3)
* x <- factor(c(“a”,”b”,”c”,…))
* table(x) gives how many of each value in factor
* unclass(x) gives factor representation as integer vector
* levels = c(“x”,”y”): level x comes before y
* NA & NaN = missing values
* is.na(x) checks if each element of x = NA
* is.nan(x) checks if each element of x = NaN
* NA values have classes (character, integer, etc.)
* NaN is an NA value but NA is not an NaN value
* Data frames – elements all have to be same length (each column has to have same # of values)
* Pokemon example of data frame: 
* row.names
* read.table(), read.csv() or data.frame() to create data frame
* data.matrix() to convert frame into matrix
* names(x) lists names of each element (column)
* dimnames(x) <- list(c(“a”, “b”), c(“c”, “d”)) – a & b are row names, c & d are column names
* Reading data: read.table, read.csv, readLines, source, dget, load, unserialize
* Writing data: write.table, write.csv, writeLines, dump, dput, save, serialize
* read.table() arguments: file name, header, sep (separating columns), colClasses, nrows, comment.char (default #), skip (how many lines to skip from beginning), stringsAsFactors(default TRUE)
* data <- read.table(“blahblahblah.txt”) skips lines starting w/ a number, finds # of rows, finds type of variable for each column
* read.csv (comma separated values) for Excel files, etc.
* ? or help() to get help w/ functions; e.g. help(read.table) or ?read.table opens help page for read.table
* Larger data sets: comment.char=””, specify colClasses, set nrows
* Example: tabAll <- read.table(“datatable.txt”, colClasses=classes)
* Calculating memory requirements to store data frames: # of rows \* # of columns \* bytes per object (depends on class)
* dput(x) converts object into text form; save to memory w/ dput(x, file=”x.R”); convert object back into original form w/ new.x <- dget(“x.R”)
* dump writes multiple objects; dump(c(“x”,”y”), file=”xandy.R”), source(“xandy.R”)
* str(file) output: description = file name, open r = read only, w = writing, a = appending, rb/wb/ab = read/write/append in binary on Windows)
* Using connection to read part of a file: con <- file(“x.txt”), x <- readLines(con, 3) to read first 3 lines of file
* Connection to read first lines of webpage: con <- url(http://www.website.com), x <- readLines(con,3) to read first 3 lines
* Subsetting = only extract necessary variables from (often large) data sets
* [ returns object(s) of same class as original, [[ extracts single element from list or data frame of any class, $ extracts elements by name
* x[] = subset of x
* x <- c(1:9): numeric index – x[8] returns 8th object (8), logical index – x[x >= 3] returns objects greater than or equal to 3
* y <- x > #, y, x[y]
* x <- list(a=…,b=…,c=…): x[2] gives list,x[[2]] gives just sequence, x$b gives element assoc. w/ b, x[[“b”]] same as last one, x[“b”], list of objects w/ element b, $b, x[c(1,3)]
* Subsetting netted elements (list within list): x[[c(# element of outer list, # element of inner list)] or x[[# element of outer list][# element of inner list]]
* Subsetting matrices: x[# element in row, # element in column]
* drop=FALSE to return matrix instead of vector
* Partial matching: x <- list(numbers=1:100), x$n or x[[“n”, exact=FALSE]] give sequence 1-100
* Removing NA values: y <- is.na(x), x[!y]
* complete.cases(x, y): which elements aren’t missing from both x and y
* variable[y, ][#:#, ] to get only rows w/o NAs
* Vectorized operations: x <- 1:10, y <- 12:21, math operations on nth elements of both vectors
* Vectorized matrix operations: x <
* rep(a,b) replicates a b times
* %\*% for matrix multiplication
* If arithmetic operations performed on vectors of different lengths, shorter vector’s elements get cycled through; example: c(1,2,3,4)+c(0,10) does (1+0)+(2+10)+(3+0)+(4+10)
* ls() to get objects in workspace
* list.files() or dir() to get files in working directory
* dir.create(“nameofdirectory”)
* setwd(“nameofdirectory”)
* file.create(“nameoffile.extension”)
* file.exists
* file.infofil
* file.rename
* file.copy
* file.path to make platform-independent pathname
* Making nested/recursive directories with dir.create requires argument recursive = TRUE
* Nested directory: dir.create(file.path(“1st branch”,”2nd branch”), recursive=TRUE)
* file.remove
* seq(): by, length
* 1:length(sequencevariable), seq(along.with = sequencevariable or seq\_along(my\_seq) create vector of same length as sequencevariable (count number of element)
* rep(c(#1,#2,#3),each=10) gives vector w/ each number repeated x10
* Join elements of character vector with paste(vectorname, collapse = “ “)
* Join multiple single-element character vectors with paste(“1st word”,”2nd word”,sep=” “)
* 
* Random sample of 100 subjects from 2 groups, 1000 available subjects each: x <- rep(1:3,1000), y <- rep(4:6,1000), z <- sample(c(x,y),100)
* x[1:10] gives index vector w/ first 10 elements of vector x (subset 1st 10 values)
* y[…] gives index vector w/ all elements of y fulfilling conditions inside brackets
* x[-#] gives index vector w/ #th element removed
* Add names to vector elements: names(vectorname) <- c(…)
* identical(1st,2nd) checks whether 2 vectors are identical
* class(variable) to get variable class
* colnames(), rownames()

Week 2

* if, else (optional), else if
*  - all same output
* if only 1 expression inside for loop, can omit curly braces
* seq\_len() takes integer inside & creates sequence 1:integer
* 10x expressed as 1e-x
* next to skip iterations of a loop
* x[,i]: subsetting by columns
* x = mydata is an example of naming an argument (previously unspecified value)
* Order of operations for argument: check for exact match, partial match, positional match
* If a function takes 2 arguments & only 1st argument is used in body, 2nd argument not used & function runs fine
* … = variable number of arguments, usually passed on to other functions; useful if you only want to change 1 argument inside function body & rest preserved w/ …
* … also used for generic functions & when you don’t know how many arguments there’ll be
* No partial matching after …; must name arguments explicitly
* Variable types: function arguments & things that aren’t
* Global environment = user’s workspace
* Lexical scoping: free variables’ values searched for in environment where function was defined
* Environment = (symbol, value); example (y, data\_frame\_y)
* Closure = function + environment
* Searching for free variable value starts from bottom (where function was defined) up (through global & empty environments), through successive parent environments
* ls(environment(variablename)) to find variable’s environment
* get(“argument”, environment(variable)) to get argument’s value
* Free variables not defined inside function or an argument of function
* y defined in defining (not calling) environment, g in global environment
* Free variable: define first, value can then be used in function w/o putting it as an argument
* Dates -> class Dates
* Times -> classes POSIXct (just date & time as a number) or POSIXlt (list of seconds, minutes, hours, etc. making up date & time)
* weekdays (day of week), months, quarters (functions)
* Sys.time() to get current time
* strptime() converts dates in character string format to time objects
* 
* 
* & means only true if left & right side of expression both true
* | means only false if both sides false
* TRUE & c(TRUE,FALSE,TRUE): TRUE on left recycled through each element right-hand vector
* && only evaluates first element of right side of expression
* AND evaluated before OR
* isTRUE(expression), isFALSE
* xor() means if one inside argument TRUE & other FALSE, return value TRUE otherwise FALSE
* sample(#) yields # random numbers
* which() returns which indices inside expression are TRUE (e.g. returned values 3,6 means 3rd & 6th in expression are TRUE)
* any() returns TRUE if at least one element TRUE, all() returns TRUE if all elements TRUE
* Expressions w/ multiple &, &&, |, ||, etc.: & before |, evaluated left to right
* Default arguments in functions can still be changed (put in yourself when calling function)
* %% to get remainder
* Explicitly assigning values to arguments 🡪 doesn’t matter what order you put them in
* Anonymous function example: evaluate(function(x){x+1},6) 🡪 6+1 = 7
* x[length(x)] to get last element of vector
*  <- example of passing functions as arguments to other functions
*  : example of unpacking arguments from …
* Binary operators in form %variablename%; used as Arg1 %variablename% Arg2 instead of usual variablename(Arg1,Arg2)
* str(unclass()) gives more compact view
* timevariable$sec gives just seconds
* difftime(arg1,arg2,units=’minutes/hours/etc.’) to find difference in times

Week 3

* Looping commands: lapply (loop over list & evaluate function on each element), sapply (simplify lapply), apply (evaluate function over array margins), tapply (evaluate over vector subsets), mapply (multivariate mapply, evaluate in parallel over arguments)
* split
* x <-list(a=1.5,b=rnorm(10)): lapply(x, mean) takes mean of both a & b
* rnorm(#observations,mean,sd)
* runif: generate uniform random variables
* Given matrices: lapply(x, function(funcname) funcname[,1]) gives 1st element of each row of each matrix
* sapply results vector if all elements length 1, matrix if all elements same length >1, list otherwise
* apply: margin of 2 collapses rows & applies function to columns, margin of 1 collapses columns & applies function to rows
* apply(x,1,quantile,probs=c(.25,.75)) to get 25th & 75th percentiles of rows; apply(x,2,…) would get percentiles of columns
* array(rnorm(2\*2\*10),c(2,2,10)) = 3-dimensional array
* apply(a,c(1,2),mean) or rowMeans(a,dims=2) collapses 3rd dimension of 3D array, take average of 3rd dimension
* mapply has to take at least as many arguments as # of lists
* mapply(rep, 1:4, 4:1) does same thing as list(rep(1,4),rep(2,3),rep(3,2),rep(4,1))
* mapply can vectorize function that doesn’t allow for vector arguments; e.g. mapply(funcname,n1:n2,mean1:mean2, sd) instead of list(funcname(n#,mean#,sd)) multiple times
* gl(# of levels, # times repeat)
* tapply: evaluate small pieces of vector, put them back together again
* tapply can take group means & find group ranges
* split takes vector & returns list grouped by factors
* lapply(split(x,f),mean) takes means of each factor level
* split(): drop=TRUE removes empty levels
* invisible(): result doesn’t get printed
* Debugging: traceback (print out function history must call immediately after error occurs), debug (start debug mode), browser (suspend function execution & start debug mode), trace (put debugging code into function), recover (stop function @ error & print out function history)
* recover: 1 calls 2, 2 calls 3, etc.
* head() gives first 6 lines
* convert data frame into vector: lapply(variablename,class) then as.[integer/character/etc.](variablename); alternatively, sapply(variablename,class)
* range(x) finds minimum & maximum values of x
* unique() returns vector w/ duplicate values removed
* sapply(variablename, length) determines how many categories a variable has
* vapply like sapply except you specify what format the result should be in (error if what you specify is different from the result)
* table(flags$landmass): how many of each flag in each landmass category
* tapply(flags$animate,flags$landmass,mean): proportion of flags per landmass w/ animate
* lapply/sapply/vapply to loop a function over many columns (e.g. flag color sums or getting the 2nd element of each flag column), tapply for finding statistics per other statistics (e.g. population summary for each country w/ red in their flag)
* sapply(#,x,simplify=F) gives same result as lapply
* unlist(lapply()) gives vector
* Use lapply to get lists, sapply to get vectors