R\_handy\_commands

Howie Rosen

3/8/2017

# CHEAT SHEETS

Master list of all cheat sheets you could ever want:

<https://www.rstudio.com/resources/cheatsheets/>

The most useful to start with is probably

<https://www.rstudio.com/wp-content/uploads/2016/10/r-cheat-sheet-3.pdf>

# Created sample datasets

Created small sample dataset, saved in file named test.frame.csv

It looks like this (commands below read it in and display the data):

test.frame.local <- read.csv("test.frame.csv")  
test.frame.local

## PIDN age sex dx visit volume1 volume2  
## 1 1111 55 m CON 1 100 15  
## 2 1111 56 m CON 2 90 11  
## 3 1111 57 m CON 3 80 10  
## 4 1111 58 m CON 4 70 9  
## 5 2222 60 f BV 1 90 18  
## 6 2222 61 f BV 2 85 19  
## 7 2222 62 f BV 3 75 17  
## 8 3333 66 m CON 1 80 22  
## 9 3333 67 m CON 2 77 20  
## 10 3333 68 m CON 3 75 20  
## 11 4444 56 f BV 1 70 19  
## 12 4444 57 f BV 2 65 14  
## 13 5555 62 m BV 1 70 15  
## 14 5555 63 m BV 2 65 12  
## 15 5555 64 m BV 3 60 11  
## 16 5555 65 m BV 4 60 10  
## 17 6666 58 f CON 1 60 18  
## 18 6666 59 f CON 2 55 17

Also creating sample objects

one numeric vector called test.numbers

test.numbers <- c(2, 10, 30, 70, 90, 110)  
test.numbers

## [1] 2 10 30 70 90 110

and the other with strings, a character vector called test.char

test.char <- c("howie", "yann", "gabe", "adam")  
test.char

## [1] "howie" "yann" "gabe" "adam"

It looks like this (commands below read it in and display the data):

# GENERAL R FUNCTIONS

## Open, Close and Directories

### Quit R

quit()

### Install Packages

install.packages("package\_name")

If you want to use the package, you have to give the following command: Library(“package\_name”)

DON’T FORGET to put “ “ around package name, or it will give you error messages

### Update all installed packages

update.packages()

### Getting Help

help(function)

e.g. help(plot)

OR

?function

e.g. ?plot

OR

Apropos(“function”)

Lists all functions with string “function” in the name

OR

Example(function)

Gets example of using the function

### See current working directory

getwd()

### List directory contents

dir("directory\_path")

dir() ; this will list contents of current directory

list.files() ; this will do the same thing as dir

### Make new directory

dir.create(“directory\_name”)

Will create new directory in the current working directory

### Remove a directory

unlink(“directory\_name”, recursive = TRUE)

This will remove a directory and everything in it

DANGEROUS, this won’t ask you any questions, it’ll just delete it

### Change Working directory

setwd(“directory\_path” )

The path has to be in “quotes”. You can use “..” to indicate one directory above

### Run command current line

cmd-return

### List the objects in the current workspace

ls( )  
  
OR  
  
Objects( )

ls lists all the objects in the directory, as opposed to list.files or dir, which only list files and other directories

### Display recent commands

history( )

This will display last 25 commands

OR

history( max.show = inf)

This will display all previous commands

### Run a specific script from a file

source(“myfilename”)

If no filename or path is specified, will look in current directory

## Working with data files

### Import a data file, like a .csv file

mydata <- read.table( “path\mydata.csv” , header = TRUE, sep = “,” , row.names =“id”)

OR

mydata <- read.csv(“path.csv”)

This is if you know the file is a csv file

You can read in other types of files if you indicate a different separator, for instance, for a tab separated file you would do:

mydata <- read.table( “path.tsv” , header = TRUE, sep = “”, row.names=“id”)

DON’T FORGET to use the header and sep arguments and to include the .csv or it won’t read the file in correctly

### Import a .csv file, if you know it’s a .csv file

mydata <- read.csv( “path\mydata.csv” , header = TRUE, row.names = “id”)

Here, you don’t have to specify the separator, cause it assumes comma separated

### Export to a .csv file

write.table ( mydata, “path\mydata.csv”, sep = “ ,“ )

OR

To write to a tab separated file

write.table ( mydata, “path.csv”, sep = “ “ )

**BE VERY CAREFUL!**

**DON’T FORGET** to use the header and sep arguments and to include the .csv or it won’t write the file correctly and might overwrite your old file and mess it up.

**BEST NEVER TO WRITE FROM R TO A FILE THAT ALREADY EXISTS AS A CSV FILE**

### Import a stata file

library(foreign)  
mydata <- read.dta( “mydata.dta” )

### Export to a stata file

library(foreign)  
write.dta(mydata, “path/mydata.dta” )

### Make a new file

file.create(“filename”)

e.g.

file.create(“myfile.R”)

this will create a file that R will recognize as an R file

### Check whether a file exists

file.exists(“filename”)

Useful for programming to make sure a file exists before the program uses it

### Rename a file

file.rename (“old\_name”, “new\_name”)

### Get info on a file

file.info(“filename”)

## Saving and Loading Output

### Save and Load dataset

save(object1\_tobesaved, object2\_tobesaved,..., file = ‘filename’)  
  
load( ‘filename’)

e,g.

save(staff.data, file=’staffdata.RData’

load(‘staffdata.RData’)

In this case, we are saving an object that happens to be a data frame and loading it. You can also save other objects such as lists or vectors, all in the same file, you just separate each object from the next using a comma

### Attaching data frame to avoid having to preface each query with the data frame name

Attach(frame\_name)  
detach(frame\_name)

e.g.

attach(staff.data)

staff.data[age > 30, ]

|  |  |  |
| --- | --- | --- |
| name | height | age |
| 1 howie | short | 51 |
| 2 yann | tall | 40 |

e.g.

height

[1] short tall tall

Levels: short tall

### Save all objects in current working directory

save.image()

This saves all objects in the current working directory.

These objects will automatically be loaded when you open R in this directory

Saves objects in a file called .RData

### Saving your recent commands

savehistory ( file = “myfilename”)

The default is to save in a file called .Rhistory

### Loading your recent commands

loadhistory ( file = “myfilename”)

the default is to save in a file called .Rhistory

### Direct output to a specific location/file

sink(“myfilename”, append=FALSE, split=FALSE)  
  
sink() ; This latter command returns output to terminal

### Direct graphic output to a location/file

jpg(“myfilename.jpg”)  
  
This will output the graph to a jpg file  
  
dev.off( )  
  
The tells R to redirect output back to terminal

Instead of jpg, can also say:

pdf (“myfile.pdf”)

bmp (“myfile.bmp”)

png (“myfile.png”)

postscript (“myfile.ps”)

## Writing Functions

### Format of a function

Function\_name <- function(arguments) {  
 Operations  
 return output  
}

Arguments are usually values that are given to the function, which could be numbers, variables with numbers in them, and paramters

### Saving output of a function

Output\_object <- function\_name(arguments)

Key is that you have to remember that the objects you make in a function, or modifications to objects that you make, only exist within the function unless you save the output of the function as an objects

e.g.

frame.with.new.column <- function\_that\_adds\_column(frame.without.new.column)

### Saving a function in R studio

Write it in the top left window  
Check source on save  
Hit save

### To write a function that will take a column name as an argument

Use double square brackets [[ ]] to refer to the column in the function, instead of the $  
  
Have to pass the column name as argument in quotes when you call the function

e.g.

function\_name <- function(data.frame.name, “column\_name”) {

result\_variable <- function(data.frame.name [[column\_name]][element\_number]

}

e.g.

newmean <- mean(test.frame.local[["age"]])  
newmean

## [1] 60.77778

NOTE: you have to remember to put the name of the column in quotes: " "

OR, here it is using a function that takes the column name as an argument:

makemean <- function(frame.name, column) {

average <- mean(data.frame[[column]])

}

if you call this function using the variable age, this function will take the mean of the column labeled age and store it in the object called average.

Like this: makemean(frame.name, “age”)

OR

e.g.

getsecondvalue <- function(frame.name, variable){

second <- frame.name[[variable]][2]

}

if you call this function using the variable age, this function will get the second observation in the column labeled age and store it in the object called second. Like this: getsecondvalue(frame.name, “age”)

### Control functions

If-else  
if (cond) expression  
if (cond) expression1 else expresssion2  
  
OR  
  
for (var in seq) expresssion  
  
OR  
  
While (cond) expression  
  
OR  
  
ifelse (test, yes, no)

### Logical Operations

& (and, specifies conjunction, e.g. if number meets condition 1 and condition 2)  
  
| (or, specifies dysjunction, e.g. if number meets condition 1 or condition 2)  
  
== (for identifying values for classifying, this is like saying “is equal to”, and evaluates whether both sides of the operator are the same)  
  
= (for equations)  
  
!= (is not equal to)  
  
&& and || will only evaluate the first expression on the one side of the operand:  
  
e.g.  
  
TRUE & c(TRUE, FALSE, FALSE)  
  
TRUE FALSE FALSE  
  
TRUE && c(TRUE, FALSE, FALSE)  
  
TRUE

# GENERATING DATA

## Making Objects and Vectors

### Storing an object

Object\_name <- data to be stored

e.g. x <- 100

This will store the number 100 in the object named x

Typing the name of the object will show the contents

e.g.

x

[1] 100

Object names can contain numbers, letters and periods

Object names are case sensitive

Each object is by definition a vector

Objects can have numbers, logical values (e.g. true false), characters, complex numbers

Character data must be surrounded by quotes

### Remove object from memory

rm(object\_name)  
  
rm(list = object1, object2, ...)  
  
latter removes multiple objects

### Remove ALL objects from memory

rm(list = ls())

### Combine arguments into a vector

c(element1, element2, element3, …)

e.g. x <- c(1, 9, 3, 5, 9)

x

[1] 1 9 3 5 9

This puts all those numbers in the object x. Typing the name of the object will tell R to list the elements in the vector, starting with the first one, which is what the [1] means

**Need commas** between elements going into the object

All elements in the vector must have the same mode

Vector can include NA, for missing data

### Cut a continuous variable into a factor variable

equal.count(dataset\_name$variable, number=#, overlap = #/#)

e.g.

agegroups <- equal.count(my.data$age, number =4)

This command will take the age variable and cut it into 4 equally sized groups and save the group designation as a nominal factor variable with 4 levels called agegroups

### Paste together or concatenate strings

paste(“string1”, “string2”, “string3”, …, sep = “ “, collapse = NULL)

sep tells the function what character to use to separate the strings

### Place value in a particular element location in a vector

Object\_name[element\_number] <- value

e.g. x[7] <- “great”

This will put the word “great” in the 7th element in the vector called x

### Making factors

factor(object\_name)

e.g. x <- c(“m”, “m”, “m”, “f”, “f”)

x <- factor(x)

x

[1] m m m f f

level: f m

you can do this more efficiently like this:

x <- factor(c(“m”, “m”, “m”, “f”, “f”)

if your vector only happens to have one value, but you want to preserve the ability to have multiple levels, you have to specify that

x <- factor(c(“m”, “m”, “m”), levels = c(“f”, “m”))

x

[1] m m m

levels: f m

### Generating names for elements

Names(object\_name) <- (“element\_name1” “element\_name2, “element\_name3”, …)

e.g.

x <- c(15 13)

names(x) <- c(“eli”, “jonah”)

Can also use this to rename all the variables in a dataset, but you must enter names for every variable

### Generating labels for values in a vector

mydata$v1 <- factor(mydata$v1,levels = c(1, 2, 3), labels = c( “young”, “middle-age”, “old” )

OR if data are ordinal instead of factor variable

mydatav1,levels = c(1, 2, 3), labels = c( “young”, “middle-age”, “old” )

### Vector operations

operation(object\_name)

e.g.

v

[1] 3 5 10

If the object v has 3 elements, the command below will multiply each element by 2, and the new values will be placed in object x

x <- 2\*(v)

x

[1] 6 10 20

Doing operations where one vector is combined with another will result in the operation being done for the corresponding element location in the two vectors

e.g.

v

[1] 3 5 10

y

[1] 5 10 10

z <- v+y

z

[1] 8 15 20

When doing operations with 2 vectors, if they are not the same length, R “recycles” elements in the shorter vector staring with the first element

### Generate numbers from a normal distribution

rnorm(number\_values, mean = #, sd = #)

e.g.

rnorm(100, mean = 50, sd = 10)

This command will create a vector with 100 random numbers from a normal distribution with a mean of 50 and standard deviation of 10

### Recode data into a factor variable

Datafilename$newfactorvariablename[existingcontinuousvariablename criterion] <- “factorlevelname”

This example will make a new factor variable agecat that will make levels corresponding to three age ranges

attach(mydata)

mydata$agecat[age > 75] <- "Elder"

mydata$agecat[age > 45 & age <= 75] <- "Middle Aged"

mydata$agecat[age <= 45] <- "Young"

detach(mydata)

### Centering a variable

scale(variable\_name, center = TRUE, scale = FALSE)

### Generating z-scores

scale(variable\_name, center = TRUE, scale = TRUE)

### Changing a string to a date

object\_name <- as.date(date\_variable, format)

e.g.

strDates <- c("01/05/1965", "08/16/1975")

dates <- as.Date(strDates, "%m/%d/%Y")

ALSO

If you want to calculate time between datesXXXXX

Dates are represented as the number of days since 1970-01-01, with negative values for earlier dates

If string is indicated by capital letter, it indicates longer/unabbreviated version, for instance %Y is YYYY, %y is yy. %A is full weekday name (like Monday) and %a is abbreviated weekday name, like Mon.

### Generate a sequence

seq()

This command can be used in several ways:

Seq( start\_number, end\_number, length = number)

e.g. seq( 0, 10, length =3)

[1] 0 5 10

this generates a sequence of numbers between 1 and 10 such that the length of the resulting vector is 3

OR

Seq( start\_number, end\_number, by = number)

e.g. seq( 0, 10, by =2)

[1] 0 2 4 6 8 10

this generates a sequence of numbers between 1 and 10 counting by 2

OR

Seq(starting\_number, ending\_number, interval)

e.g. seq( 0, 2, 0.5)

[1] 0.0 0.5 1.0 1.5 2.0

### Generate a repetitive sequence

rep( number, repeats)

e.g. rep( 10, 5)

[1] 10 10 10 10 10

you can also do it with sequences

e.g. rep( 1:3, 3)

[1] 1 2 3 1 2 3 1 2 3

OR you can have each number in the sequence repeated

e.g. rep( 1:3, each = 3)

[1] 1 1 1 2 2 2 3 3 3

### Generate repetitive values within factors

gl( number of levels, repeats in each level)

e.g. gl( 3, 4)

[1] 1 1 1 1 2 2 2 2 3 3 3 3

levels: 1 2 3

another example with character data

e.g. gl( 2 , 3, labels = c(“m”, “f”))

[1] m m m f f f

levels: m f

### Generate random numbers

rfunction( number\_of\_data, parameters)

e.g. rnorm(5)

[1] 1.2241399 0.8170304 -0.2244951 -0.3439073 1.2656202

this generates 5 random numbers from a normal distribution with a mean of 0 and sd of 1. You can change the parameters of the distribution you draw from

e.g. rnorm(5, mean = 10, sd = 3)

[1] 14.642944 7.436970 12.740121 9.238453 4.280766

### Grab a random sampling of data from a vector

Sample(“variable\_name”, number)

e.g. sample(my.data$height, 4)

this command will take 4 random values from the variable height You specify the number of elements you want to grab

OR

e.g.

Sample(1:6, 4, replace = TRUE)

This will grab 4 random values from the sequence 1:6. The replace=TRUE argument says you can grab the same observation/number more than once

OR

e.g.

sample(my.data$height)

This command will randomly rearrange all the values in the variable height. It will “sample” all the values because a number to sample was not specified

OR

e.g.

Sample(c(0, 1), 100, prob = c(0.3, 0.7), replace = TRUE)

This command will grab 100 values from the vector containing 0 and 1, with a probability of grabbing 0 being 0.3 and the probability of grabbing 1 being 0.7. In this command, you had to say replace = TRUE because you don’t have 100 observations in the vector, so you have to replace the value each time you grab it

## Making Matrices and Data Frames

### Separate vector into matrix

dim(object\_name) <- (nu\_of\_rows, nu\_of\_columns)

e.g.

x <- c(1, 2, 3, 5, 10, 14, 19, 22)

dim(x) <- c(2, 4)

[1] 1 3 10 19

[2] 2 5 14 22

### Create a matrix

Object\_name <- matrix (c(element1, element2, …), nu\_rows, nu\_columns)

e.g.

x <- matrix (c(1, 2, 3, 5, 10, 14, 19, 22), 2 ,4)

x

[1] 1 3 10 19

[2] 2 5 14 22

OR

Object\_name <- matrix (c(element1, element2, …), no\_rows, no\_columns, byrow = T)

e.g.

x <- matrix (c(1, 2, 3, 5, 10, 14, 19, 22), 2, 5, byrow = T))

x

[1] 1 2 3 5

[2] 10 14 19 22

If you make a matrix with the standard command, R will fill the column with consecutive values and go to the next column. If you want to fill a whole, then move to the next, use the byrow = TRUE parameter.

### Generating names for columns and rows in a matrix

colnames(matrix\_name) <- c(“column\_name1” “column\_name2, “column\_name3”, …)  
  
 rownames(matrix\_name) <- (“row\_name1” “row\_name2, “row\_name3”, …)

e.g.

x <- matrix (c(1, 2, 3, 5, 10, 14, 19, 22), 2, 5, byrow = T))

colnames(x) <- c(“visit1”, “visit2”, “visit3”, “visit4”)

rownames(x) <- c(“wait1”, “wait2”)

x

waits visit1 visit2 visit3 visit4

wait1 1 2 3 5

wait2 10 14 19 22

You can also save the column names or row names in variables and use them

e.g.

cnames <- c(“visit1”, “visit2”, “visit3”, “visit4”)

colnames(x) <- cnames

### Change name of a column in a dataframe

colnames(data.frame)[col\_number] <- “new\_name”

### Join two vectors/matrices by columns

cbind(object\_name, object\_name)

e.g.

x

[1] 1 2 3 5

[2] 10 14 19 22

y

[1] 3 3

[2] 4 5

cbind(x[ ,4], y[ ,1])

*spacer* [ , 1] [ ,2]

[1, ] 5 3

[2, ] 22 4

### Join two vectors/matrices by rows

rbind(object\_name1[position], object\_name2[position])

e.g.

x

[1] 1 2 3 5

[2] 10 14 19 22

y

[1] 3 3 4 9

[2] 4 5 10 20

rbind(x[ 1, ], y[2, ])

*spacer* [ , 1] [ ,2] [ ,3] [ ,4] [1, ] 1 2 3 5 [2, ] 4 5 10 20

### Creating a data frame

Dateframe\_name <- data.frame(object1\_name = c(value1, value2, …), object2\_name = c(value1, value2, …), object3\_name = c(value1, value2,…)

e.g.

staff.data <- data.frame(name = c(“Howie”, “yann”, “gabe”), height = c(“short”, “tall”, “tall”), age = c(51 40 24))

*sp* name height age

1 howie short 51

2 yann tall 40

3 gabe tall 24

If already created the name, height and age variables, can just do this: Staff.data <- c(name, height, age)

### Adding columns to a data frame

Frame\_name$new\_variable\_name <- c(value1, value2, …)

e.g.

staff.data$howmuchhair <- c(“not much”, “a lot”, “a lot”)

staff.data

*sp* name height age howmuchhair

1 howie short 51 not much

2 yann tall 40 a lot

3 gabe tall 24 a lot

### Adding columns to a data frame other ways

data$size <- c("small", "large", "medium")   
 data[["size"]] <- c("small", "large", "medium") data[,"size"] <- c("small", "large", "medium")   
 data$size <- 0

### Move a column from one location to the beginning of a dataset

example  
 col\_idx <- grep("string", names(dataset))  
 dataset <- dataset[, c(col\_idx, (1:ncol(dataset))[-col\_idx])]  
 names(dataset)

Here, you are using grep to find the location of the columns with names containing the “string” you have specified, you save those locations to an objects, then you rewrite the dataset using data from all rows of the dataset in the columns saved in col\_idx first, and then all other columns in the dataset excluding those at the positions saved in col\_idx.

### Sorting data

Newdata <- my.data [order (mpg), ]

Above command will sort all the data in my.data by the variable mpg and save the sorted data to a new variable, “newdata”

Newdata <- my.data [order (mpg, cyl), ]

Above command will sort all the data in my.data by the variable mpg and then the variable cyl and save the sorted data to a new variable, “newdata”

Newdata <- my.data [order (mpg, - cyl), ]

Above command will sort all the data in my.data by the variable mpg in ascending order and then the variable cyl in descending order and save the sorted data to a new variable, “newdata”

### Remove rows from a data frame

Dataset\_name <- dataset\_name [ -c(row#, row#...), ]

### Creating a list

List\_name <- list(object1\_name = value, object2\_name = value, object3\_name = value, …)

e.g.

howie.lst <- list(stud.id = 3344, stud.name = “Howie”, stud.scores = c(99, 73, 95, 85)

howie.lst

$stud.id

[1] 3344

$stud.name

[1] “Howie”

$stud.scores

[1] 99 73 95 85

A list contains several objects. The objects don’t have to have the same length or mode

### Naming objects in a list

Names(List\_name) <- c(“name1”, “name2”, “name3”, etc.)

e.g.

names(howie.lst <- c(“id”, “name”, “scores”)

Howie.lst

$id

[1] 3344

$name

[1] “Howie”

$scores

[1] 99 73 95 85

### Extending a list (adding new objects)

List\_name$object\_name <- c (value1, value2, etc.)

e.g.

Howie.lst$parents <- (“dora”, “seymour”)

Howie.lst

$id

[1] 3344

$name

[1] “Howie”

$scores

[1] 99 73 95 85

$parents

[1] “dora” “Seymour”

### Concatenating lists

New\_list <- c(list1, list2)

e.g.

list2 <- c(age = 51, sex = “man”)

new.lst <- c(Howie.lst, list2)

new.lst

$id

[1] 3344

$name

[1] “Howie”

$scores

[1] 99 73 95 85

$parents

[1] “dora” “Seymour”

$age

[1] 51

$sex

[1] “man”

### Unflatten a list

unlist(listname)

All list elements will become elements in a single vector, and be forced to have the same mode (e.g. character, numeric…), and each element will have a name

# EXAMINING DATA

## Examining Objects, Vectors

### Reading an object

View(object\_name)

### Find length of an objects

length(object\_name)

e.g.

x

[1] 1 9 3 5 9

length(x)

[1] 5

### See the type of an object

Class(object\_name)

Above command will return data.frame if it’s a dataframe, numeric if it’s numeric, etc…

As compared with mode, class seems more useful to tell you what properties an object has, what you can do to it and what you can't

### Find type of data in objects

mode(object\_name)

If you want to know if a vector is character vs integer vs float, use mode

e.g.

x

[1] 1 9 3 5 9

mode(x)

[1] “numeric”

### See the dimensions of an object

dim(object\_name)

remember can also use dim to seaprate a vector into a matrix (see generating data)

### Evaluate whether two arguments are identical

identical(argument1, argument2)

e.g.

identical(7, 7)

TRUE

This can help you decide whether the contents of two objects are the same

### Evaluate two arguments and return TRUE if they are not both true or both FALSE

xor(argument1, argument2)

This function will return TRUE if one argument is FALSE and one argument is TRUE. If both are TRUE, or both are FALSE, the function will return FALSE e.g.

xor(7>7, 3==3) TRUE

### Find the value for an element in a vector

Object\_name[element\_number]

e.g. x[7]

This will show the value of the 7th element in the vector called x

e.g.

x[length(x)]

This will find the value of the last element in a vector, regardless of how long it is

### List levels in a factor variable

levels(dataset\_name$variable\_name)

### Look at factor level as its integer

as.integer(dataset\_name$variable\_name)

### Look for a particular string in a dataset

grep(“string”, object)

e.g.

myname <- grep(“Howie”, namelist)

This would find the position of the name howie in a column of names and assign it to the variable myname The variable myname would not have the string “Howie” in it, just the row number or column number where it is in the larger dataset

### Get descriptors for a variable

mean(dataset\_name$variable, na.rm = TRUE)  
 median(dataset\_name$variable, na.rm = TRUE)  
 min(dataset\_name$variable, na.rm = TRUE)  
 max(dataset\_name$variable, na.rm = TRUE)  
 sd(dataset\_name$variable, na.rm = TRUE)

Using na.rm = T means you should leave NA observations out of the equation

## Examining Matrices, Dataframes

### Counting columns and rows in data frame

ncol(frame\_name)  
 nrow(frame\_name)

### Read beginning and ends of datasets

Head(dataset\_name)  
 Tail(dataset\_name)

OR

Head(dataset\_name, #rows)

Tail(dataset\_name, #rows)

To specify the number of rows to be shown (instead of the default 6)

### List names of variables in dataset

names(dataset\_name)

### List variables in dataset with info about each

str(dataset\_name)

Give a bit about type of each variable, and initial few values

### Summarize properties of variables in dataset

summary(dataset\_name)

For numeric variables, will give mean, median, max, min and 1st and 3rd quartiles

For character variables, will give counts for each level

### lappy: Apply functions to multiple elements in an object

lapply(dataset\_name, function, na.rm = TRUE)

NOTE: lapply always returns a list, which will have a bunch of elements, each of which corresponds to the element that was evaluated in the original dataset

One major value of using lapply is to do a function iteratively without having to write a loop code

e.g.

lapply(test.frame.local, mean)

## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:  
## returning NA  
  
## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:  
## returning NA

## $PIDN  
## [1] 3641.611  
##   
## $age  
## [1] 60.77778  
##   
## $sex  
## [1] NA  
##   
## $dx  
## [1] NA  
##   
## $visit  
## [1] 2.111111  
##   
## $volume1  
## [1] 73.72222  
##   
## $volume2  
## [1] 15.38889

This command will try to take the mean of every column in the dataset. Note it gives warnings for the character variables, cause you can't get a mean.

NOTE that apply wants to work on each element in a vector, so if you give it only one column out of the data frame, it will take the mean for every value in the column separately, like this:

lapply(test.frame.local$age, mean)

## [[1]]  
## [1] 55  
##   
## [[2]]  
## [1] 56  
##   
## [[3]]  
## [1] 57  
##   
## [[4]]  
## [1] 58  
##   
## [[5]]  
## [1] 60  
##   
## [[6]]  
## [1] 61  
##   
## [[7]]  
## [1] 62  
##   
## [[8]]  
## [1] 66  
##   
## [[9]]  
## [1] 67  
##   
## [[10]]  
## [1] 68  
##   
## [[11]]  
## [1] 56  
##   
## [[12]]  
## [1] 57  
##   
## [[13]]  
## [1] 62  
##   
## [[14]]  
## [1] 63  
##   
## [[15]]  
## [1] 64  
##   
## [[16]]  
## [1] 65  
##   
## [[17]]  
## [1] 58  
##   
## [[18]]  
## [1] 59

BUT, you can subset a couple of columns and do what you want, like this:

lapply(test.frame.local[c("age", "volume1")], mean)

## $age  
## [1] 60.77778  
##   
## $volume1  
## [1] 73.72222

Other functions include sd, var, min, max, median, range, and quantile, or any other function

### sapply: Apply functions and get the result formatted properly

sapply(dataset\_name, function, na.rm = TRUE)

NOTE: sapply is like lapply but better because it will return simplify the data into the simplest data structre possible, like a simple vector or matrix if it can

If using sapply results in a bunch of elements of length 1, it will save them in a vector

If using sapply results in a bunch of vectors that all have multiple elements and have the same length, it will save them in a matrix.

If using sapply results in a bunch of vectors that all have multiple elements that are not the same length, it will save the results as a list, which is the same thing that lapply will do

e.g.

means <- sapply(my.data, mean, na.rm = TRUE)

This command will try to take the mean of every column in the dataset my.data, and save those means in a vector. Because the output of mean is numbers, the vector will be numeric

e.g.

class\_list <- sapply(my.data, class)

This command will assess the class of each column in the my.data dataset and pass the class values to a vector called class\_list. Because the output of class is a string, the vector will be a character vector

sapply will also make a matrix if the data it brings back is a series of vectors

e.g.

maxmin <- sapply(my.data, range)

This will take the minimum and maximum of the numbers in each column of my.data and save it in a matrix that would be 2 rows by however many columns are in the dataset

### vapply: Specify the type of output provided by apply

vapply(dataset\_name, function, vector\_type)

e.g.

vapply(my.data, class, character(1))

This command will determine the class of each object in my.data and the output is specified as a character vector of length 1.

Using this function does the same thing as sapply, but it is ‘safer’ than sapply because it will give an error if the output is not in the form you expect. Thus, vaapply is good to use in scripts where you want error messages if things don't go well. In contrast, sapply will output the result in whatever format it needs to, and you won’t know if it’s not what you expect

### tapply: Get statistics for an object in a list split according to another factor vector

tapply(variable\_to\_be\_assessed, factor\_variable, function)

e.g.

tapply(my.datadiagnosis, mean)

this command with provide the means for the variable age at each level of diagnosis

### Extract value of an element from a list

List\_name[[list\_item\_number]]

e.g.

howie.lst[[3]]

$stud.scores [1] 99 73 95 85

OR

List\_name$object\_name

e.g.

howie.lst$stud.scores

[1] 99 73 95 85

If you just use single brackets, you will just create a new list based on the object you have queried (in this case, the third object). If you want the value of an element, you have to use the double brackets

### Make tables for factor data

table(Object\_name)

e.g.

table(test.frame.local$sex)

##   
## f m   
## 7 11

you can also use table to cross-tabulate multiple factors for the same cases. For instance if you also have a second factor with age categories for the same people references in x, you can show sex and age in a table

e.g.

table(test.frame.local$sex, test.frame.local$dx)

##   
## BV CON  
## f 5 2  
## m 4 7

### Getting frequencies from cross-tabulated data

First put the table in an object

prop.table(table\_object\_name, dimension\_number)

e.g.

# first put the data into a table  
sex\_table <- table(test.frame.local$sex, test.frame.local$dx)  
# then run the command to make proportions, use the second argument to say which dimension of the table to use, the first or the second  
prop.table(sex\_table, 1)

##   
## BV CON  
## f 0.7142857 0.2857143  
## m 0.3636364 0.6363636

This set of commands gives you the proportions of other levels within each level of the first dimension in the table, which is sex, and says that 70% of the females are BV and 30% are CON, and 36% of the males are BV and 64% are CON

OR

e.g.

# first put the data into a table  
sex\_table <- table(test.frame.local$sex, test.frame.local$dx)  
# then run the command to make proportions  
prop.table(sex\_table, 2)

##   
## BV CON  
## f 0.5555556 0.2222222  
## m 0.4444444 0.7777778

This gives you the proportions of other levels within each level of the second dimension in the table, which is dx, and says that 55% the BVs are female adn 44% are male, and 22% of the CONs are female and 78% are male

OR

e.g.

# first put the data into a table  
sex\_table <- table(test.frame.local$sex, test.frame.local$dx)  
# then run the command to make proportions  
prop.table(sex\_table)

##   
## BV CON  
## f 0.2777778 0.1111111  
## m 0.2222222 0.3888889

This gives you the proportions with respect to all categories, and says that 27% of all the people are BV females, 11% are CON females, 22% are BV males and 39% are CON males

OR

You can run the prop.table command with the table command:

e.g.

prop.table(table(test.frame.local$sex, test.frame.local$dx))

##   
## BV CON  
## f 0.2777778 0.1111111  
## m 0.2222222 0.3888889

### Getting percentages from cross-tabulated data

prop.table(table\_object\_name, dimension\_number)\*100

e.g.

prop.table(table(test.frame.local$sex, test.frame.local$dx))\*100

##   
## BV CON  
## f 27.77778 11.11111  
## m 22.22222 38.88889

### Getting marginal counts from cross-tabulated data

Margin.table(table\_object\_name, dimension\_number)

margin.table(table(test.frame.local$sex, test.frame.local$dx), 1)

##   
## f m   
## 7 11

OR

e.g.

margin.table(table(test.frame.local$sex, test.frame.local$dx), 2)

##   
## BV CON   
## 9 9

## Subsetting various ways

### Get value at a specific position in a vector

Object\_name[position]

e.g.

test.char[2]

## [1] "yann"

### Get values at multiple positions in a vector

Object\_name[c(position, position, …)]

e.g.

test.char[c(2,4)]

## [1] "yann" "adam"

You have to combine the positions into a vector or it will think you are looking at a matrix and get confused

OR

Object\_name[c(sequence)]

e.g.

test.char[c(2:4)]

## [1] "yann" "gabe" "adam"

OR, designate a position to ignore

Object\_name[-position\_to\_ignore]

test.char[-4]

## [1] "howie" "yann" "gabe"

OR, designate multiple positions to ignore

Object\_name[-c(position, position, …)]

test.char[-c(2,4)]

## [1] "howie" "gabe"

this last command uses the negative sign to tell R to give the elements in the object, ignoring the specified positions

### Get values at specific position in a matrix

Object\_name[row\_position, column\_position]

e.g.

test.frame.local[2,4]

## [1] CON  
## Levels: BV CON

OR

To list all the values in a particular variable in a data frame:

frame\_name$object\_name  
   
 OR  
   
 frame\_name[["object\_name"]]

When using the double square brackets, you have to put the name in quotes, when using the $ you don't

e.g.

test.frame.local$dx

## [1] CON CON CON CON BV BV BV CON CON CON BV BV BV BV BV BV CON  
## [18] CON  
## Levels: BV CON

OR

test.frame.local[["dx"]]

## [1] CON CON CON CON BV BV BV CON CON CON BV BV BV BV BV BV CON  
## [18] CON  
## Levels: BV CON

### Get a whole row

Object\_name[rownumber, ]

e.g.

test.frame.local[3,]

## PIDN age sex dx visit volume1 volume2  
## 3 1111 57 m CON 3 80 10

OR, get multiple rows

e.g.

test.frame.local[2:3, ]

## PIDN age sex dx visit volume1 volume2  
## 2 1111 56 m CON 2 90 11  
## 3 1111 57 m CON 3 80 10

### Get a whole column

Object\_name[ , columnnumber]  
 Object\_name[ , "columnname"]

e.g.

test.frame.local[,2]

## [1] 55 56 57 58 60 61 62 66 67 68 56 57 62 63 64 65 58 59

OR, get multiple columns

e.g.

test.frame.local[,2:4]

## age sex dx  
## 1 55 m CON  
## 2 56 m CON  
## 3 57 m CON  
## 4 58 m CON  
## 5 60 f BV  
## 6 61 f BV  
## 7 62 f BV  
## 8 66 m CON  
## 9 67 m CON  
## 10 68 m CON  
## 11 56 f BV  
## 12 57 f BV  
## 13 62 m BV  
## 14 63 m BV  
## 15 64 m BV  
## 16 65 m BV  
## 17 58 f CON  
## 18 59 f CON

e.g.

test.frame.local[,"age"]

## [1] 55 56 57 58 60 61 62 66 67 68 56 57 62 63 64 65 58 59

### Get values from a matrix and keep the result as a matrix instead of a vector

Object\_name[row\_position, column\_position, drop = FALSE]

If you do this command and do not include the drop =FALSE parameter, R would just make a vector, not a matrix because the result would have just one row of values

### Identifying elements that meet specific criteria

Object\_name[test]

e.g.

test.numbers

## [1] 2 10 30 70 90 110

test.numbers[test.numbers>10]

## [1] 30 70 90 110

this command gave us the elements in object test.numbers that was greater than 3

### Querying data frame using conditions

Frame\_name[frame\_name$variable\_name condition]

e.g.

To get all columns where the row value for age is more than 60

test.frame.local[test.frame.local$age < 60, ]

## PIDN age sex dx visit volume1 volume2  
## 1 1111 55 m CON 1 100 15  
## 2 1111 56 m CON 2 90 11  
## 3 1111 57 m CON 3 80 10  
## 4 1111 58 m CON 4 70 9  
## 11 4444 56 f BV 1 70 19  
## 12 4444 57 f BV 2 65 14  
## 17 6666 58 f CON 1 60 18  
## 18 6666 59 f CON 2 55 17

OR

e.g.

To get all columns where the row value for dx is BV

test.frame.local[test.frame.local$dx == "BV", ]

## PIDN age sex dx visit volume1 volume2  
## 5 2222 60 f BV 1 90 18  
## 6 2222 61 f BV 2 85 19  
## 7 2222 62 f BV 3 75 17  
## 11 4444 56 f BV 1 70 19  
## 12 4444 57 f BV 2 65 14  
## 13 5555 62 m BV 1 70 15  
## 14 5555 63 m BV 2 65 12  
## 15 5555 64 m BV 3 60 11  
## 16 5555 65 m BV 4 60 10

OR

e.g.

OR

e.g.

To get only the age for those where the dx is BV

test.frame.local[test.frame.local$dx == "BV", "age"]

## [1] 60 61 62 56 57 62 63 64 65

To get only the age and volume1 value for those where the dx is BV

test.frame.local[test.frame.local$dx == "BV", c("age", "volume1")]

## age volume1  
## 5 60 90  
## 6 61 85  
## 7 62 75  
## 11 56 70  
## 12 57 65  
## 13 62 70  
## 14 63 65  
## 15 64 60  
## 16 65 60

### Probe elements in a vector to get the **positions** of those meeting certain criteria

which(object\_name criterion)

e.g.

test.numbers

## [1] 2 10 30 70 90 110

which(test.numbers > 10)

## [1] 3 4 5 6

which(test.char == "gabe")

## [1] 3

### See if any elements in a dataset meet specific criteria

any(object\_name criterion)

e.g.

x <- c(1, 3, 8, 10)

any( x > 7)

TRUE

### See if all elements in a dataset meet specific criteria

all(object\_name criterion)

e.g.

x <- c(1, 3, 8, 10)

all( x > 7)

FALSE

### Subsetting one or more columns based on the name

Object\_name[“element\_name”]

If you want to get the values of a whole column based on the column name, you can do this

test.frame.local[c("PIDN","age")]

## PIDN age  
## 1 1111 55  
## 2 1111 56  
## 3 1111 57  
## 4 1111 58  
## 5 2222 60  
## 6 2222 61  
## 7 2222 62  
## 8 3333 66  
## 9 3333 67  
## 10 3333 68  
## 11 4444 56  
## 12 4444 57  
## 13 5555 62  
## 14 5555 63  
## 15 5555 64  
## 16 5555 65  
## 17 6666 58  
## 18 6666 59

### Subsetting rows in a data frame

subset(frame\_name, condition)  
   
 OR, to combine multiple conditions, using |, which is "or":  
   
 Subset(frame\_name, variable1==condition | variable1==condition)

e.g

subset(test.frame.local, test.frame.local$PIDN==4444 | test.frame.local$PIDN==6666)

## PIDN age sex dx visit volume1 volume2  
## 11 4444 56 f BV 1 70 19  
## 12 4444 57 f BV 2 65 14  
## 17 6666 58 f CON 1 60 18  
## 18 6666 59 f CON 2 55 17

### By function: Apply a function to some data according to levels of another variable

by(data, factor\_variable, function)

e.g.

by(test.frame.local$age, test.frame.local$dx, mean)

## test.frame.local$dx: BV  
## [1] 61.11111  
## --------------------------------------------------------   
## test.frame.local$dx: CON  
## [1] 60.44444

### With function: Apply a function to some data specifying the data that goes into the function (or limiting the input)

with(data, expresssion)

e.g.

with(test.frame.local[test.frame.local$dx=="BV", ], mean(test.frame.local$age))

## [1] 60.77778

this command with provide the means for the variable age for those with a diagnosis of FTD

the "with" command sets the environment in which to do the operation. Here, we are telling it what data frame to use, then within the square brackets we are specifying the dataset to create in which to run the analysis, meaning a new dataset made up of all columns for the rows where the dx variable has a value of BV

### Aggregating data

Newobjects <- aggregate(data=data.frame, by = list (factor\_var1, factor\_var2…, FUN = function, na.rm = TRUE)  
  
OR  
  
 Newobject <- aggregate(variables\_to\_aggregate~variables\_for\_grouping, data=data.frame, FUN=function, na.rm=TRUE)

e.g.

means <- aggregate (size.data, by = list(agegroup, sex), FUN = mean, na.rm = TRUE)

This command will aggregate all the numeric data in the dataset called size.data and produce means for each numeric variable grouped according to the agegroup and sex variable (4 means per numeric variable)

OR, example with more concise code

e.g.

agg\_gap\_country <- aggregate(.~country, data=gapM, FUN=mean)

Here, you are getting the mean of all the variables in the dataset gapM, divided according to country. The . means all, the ~ means by.

agg\_gap\_country <- aggregate(pop~country, data=gapM, FUN=mean)

Here, you are doing the same thing, but only for the variable pop

### Find observations with missing data in a variable of interest

is.na(dataset\_name$variable)

or can find lines where the variable of interest is not missing

!is.na(dataset\_name$variable)

### Omit observations with missing data

na.omit(dataset\_name$variable)

### Identify rows with NA values in ANY of the variables in the row

complete.cases(dataset\_name)

e.g.

my.data[complete.cases(my.data), ]

This command will extract the rows that have complete data in the my.data dataset

e.g.

my.data[ ! complete.cases(my.data), ]

This command will extract the rows that DO NOT have complete data in the my.data dataset

The complete cases command returns a vector with one line for each line in the data frame, and the value TRUE for those lines that are complete