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 dataset

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

# 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”)

he 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.

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

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

}

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

## Generating 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

## 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”