# R for Biologist - An Introduction to R (Beginner)

## What is R

R is a language and environment for statistical computing and graphics. It provides a wide variety of statistical and graphical techniques (linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, ...) and graphical techniques, and is highly extensible. It is a GNU project (Free and Open Source) which is similar to the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. R was created by Ross Ihaka and Robert Gentleman[4] at the University of Auckland, New Zealand, and now, R is developed by the R Development Core Team, of which Chambers is a member. R is named partly after the first names of the first two R authors (Robert Gentleman and Ross Ihaka), and partly as a play on the name of S. R can be considered as a different implementation of S. There are some important differences, but much code written for S runs unaltered under R.

Some of R's strengths: \* The ease with which well-designed publication-quality plots can be produced, including mathematical symbols and formulae where needed. Great care has been taken over the defaults for the minor design choices in graphics, but the user retains full control. \* It compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS. \* R can be extended (easily) via packages. \* R has its own LaTeX-like documentation format, which is used to supply comprehensive documentation, both on-line in a number of formats and in hardcopy. \* Its FREE! \* It has a vast community both in academia and in business.

### The R environment

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes

* an effective data handling and storage facility,
* a suite of operators for calculations on arrays, in particular matrices,
* a large, coherent, integrated collection of intermediate tools for data analysis,
* graphical facilities for data analysis and display either on-screen or on hardcopy, and
* a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

The term "environment" is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of very specific and inflexible tools, as is frequently the case with other data analysis software.

R, like S, is designed around a true computer language, and it allows users to add additional functionality by defining new functions. Much of the system is itself written in the R dialect of S, which makes it easy for users to follow the algorithmic choices made. For computationally-intensive tasks, C, C++ and Fortran code can be linked and called at run time. Advanced users can write C code to manipulate R objects directly.

Many users think of R as a statistics system. The R group, prefers to think of it of an environment within which statistical techniques are implemented.

## The R Homepage

The R homepage has a wealth of information on it,

[R-project.org](http://r-project.org/)

On the homepage you can: \* Learn more about R \* Download R \* Get Documentation (official and user supplied) \* Get access to CRAN 'Comprehensive R archival network'

## RStudio

Relatively new project that is the BEST integrated developement environment I have ever used.

[RStudio](http://rstudio.org/)

RStudio has many features: \* syntax highlighting \* code completion \* smart indentation \* "Projects" \* workspace browser and data viewer \* imbedded plots \* Sweave authoring and knitr with one click pdf or html \* runs on all platforms and over the web

# Topics covered in this introduction to R

1. Basic data types in R
2. Importing and exporting data in R
3. Basic statistics in R
4. Simple data visulization in R
5. Installing packages in R
6. lapply(), sapply(), tapply()

# Topic 1. Basic data types in R

# Simple variables: variables that have a numeric value, a character value (such as a string), or a logical value (True or False)

Examples of numeric values.

# assign number 150 to variable a.  
a <- 150  
a

## [1] 150

# assign a number in scientific format to variable b.  
b <- 3e-2  
b

## [1] 0.03

Examples of character values.

# assign a string "Professor" to variable title  
title <- "Professor"  
title

## [1] "Professor"

# assign a string "Hello World" to variable hello  
hello <- "Hello World"  
hello

## [1] "Hello World"

Examples of logical values.

# assign logical value "TRUE" to variable is\_female  
is\_female <- TRUE  
is\_female

## [1] TRUE

# assign logical value "FALSE" to variable is\_male  
is\_male <- FALSE  
is\_male

## [1] FALSE

To find out the type of variable.

class(is\_female)

## [1] "logical"

# To check whether the variable is a specific type  
is.numeric(hello)

## [1] FALSE

is.numeric(a)

## [1] TRUE

is.character(hello)

## [1] TRUE

The rule to convert a logical variable to numeric: TRUE > 1, FALSE > 0

as.numeric(is\_female)

## [1] 1

as.numeric(is\_male)

## [1] 0

R does not know how to convert a numeric variable to a character variable.

b

## [1] 0.03

as.character(b)

## [1] "0.03"

# Vectors: a vector is a combination of multiple values(numeric, character or logical) in the same object. A vector is created using the function c() (for concatenate).

friend\_ages <- c(21, 27, 26, 32)  
friend\_ages

## [1] 21 27 26 32

friend\_names <- c("Mina", "Ella", "Anna", "Cora")  
friend\_names

## [1] "Mina" "Ella" "Anna" "Cora"

One can give names to the elements of a vector.

# assign names to a vector by specifying them  
names(friend\_ages) <- c("Mina", "Ella", "Anna", "Carla")  
friend\_ages

## Mina Ella Anna Carla   
## 21 27 26 32

# assign names to a vector using another vector  
names(friend\_ages) <- friend\_names  
friend\_ages

## Mina Ella Anna Cora   
## 21 27 26 32

Or One may create a vector with named elements from scratch.

friend\_ages <- c(Mina=21, Ella=27, Anna=26, Cora=32)  
friend\_ages

## Mina Ella Anna Cora   
## 21 27 26 32

To find out the length of a vector:

length(friend\_ages)

## [1] 4

To access elements of a vector: by index, or by name if it is a named vector.

friend\_ages[2]

## Ella   
## 27

friend\_ages["Ella"]

## Ella   
## 27

friend\_ages[c(1,3)]

## Mina Anna   
## 21 26

friend\_ages[c("Mina", "Anna")]

## Mina Anna   
## 21 26

# selecting elements of a vector by excluding some of them.  
friend\_ages[-3]

## Mina Ella Cora   
## 21 27 32

To select a subset of a vector can be done by logical vector.

my\_friends <- c("Mina", "Ella", "Anna", "Cora")  
my\_friends

## [1] "Mina" "Ella" "Anna" "Cora"

has\_child <- c("TRUE", "TRUE", "FALSE", "TRUE")  
has\_child

## [1] "TRUE" "TRUE" "FALSE" "TRUE"

my\_friends[has\_child == "TRUE"]

## [1] "Mina" "Ella" "Cora"

\*\*\* NOTE: a vector can only hold elements of the same type.

# Matrices: A matrix is like and Excel sheet containing multiple rows and columns. It is used to combine vectors of the same type.

col1 <- c(1,3,8,9)  
col2 <- c(2,18,27,10)  
col3 <- c(8,37,267,19)  
  
my\_matrix <- cbind(col1, col2, col3)  
my\_matrix

## col1 col2 col3  
## [1,] 1 2 8  
## [2,] 3 18 37  
## [3,] 8 27 267  
## [4,] 9 10 19

rownames(my\_matrix) <- c("row1", "row2", "row3", "row4")  
my\_matrix

## col1 col2 col3  
## row1 1 2 8  
## row2 3 18 37  
## row3 8 27 267  
## row4 9 10 19

t(my\_matrix)

## row1 row2 row3 row4  
## col1 1 3 8 9  
## col2 2 18 27 10  
## col3 8 37 267 19

To find out the dimension of a matrix:

ncol(my\_matrix)

## [1] 3

nrow(my\_matrix)

## [1] 4

dim(my\_matrix)

## [1] 4 3

Accessing elements of a matrix is done in similar ways to accessing elements of a vector.

my\_matrix[1,3]

## [1] 8

my\_matrix["row1", "col3"]

## [1] 8

my\_matrix[1,]

## col1 col2 col3   
## 1 2 8

my\_matrix[,3]

## row1 row2 row3 row4   
## 8 37 267 19

my\_matrix[col3 > 20,]

## col1 col2 col3  
## row2 3 18 37  
## row3 8 27 267

Calculations with matrices.

my\_matrix \* 3

## col1 col2 col3  
## row1 3 6 24  
## row2 9 54 111  
## row3 24 81 801  
## row4 27 30 57

log10(my\_matrix)

## col1 col2 col3  
## row1 0.0000000 0.301030 0.903090  
## row2 0.4771213 1.255273 1.568202  
## row3 0.9030900 1.431364 2.426511  
## row4 0.9542425 1.000000 1.278754

Total of each row.

rowSums(my\_matrix)

## row1 row2 row3 row4   
## 11 58 302 38

Total of each column.

colSums(my\_matrix)

## col1 col2 col3   
## 21 57 331

It is also possible to use the function apply() to apply any statistical functions to rows/columns of matrices. The advantage of using apply() is that it can take a function created by user.

The simplified format of apply() is as following:

apply(X, MARGIN, FUN)

X: data matrix MARGIN: possible values are 1 (for rows) and 2 (for columns) FUN: the function to apply on rows/columns

To calculate the mean of each row.

apply(my\_matrix, 1, mean)

## row1 row2 row3 row4   
## 3.666667 19.333333 100.666667 12.666667

To calculate the median of each row

apply(my\_matrix, 1, median)

## row1 row2 row3 row4   
## 2 18 27 10

# Factors: a factor represents categorical or groups in data. The function factor() can be used to create a factor variable.

friend\_groups <- factor(c(1,2,1,2))  
friend\_groups

## [1] 1 2 1 2  
## Levels: 1 2

In R, categories are called factor levels. The function levels() can be used to access the factor levels.

levels(friend\_groups)

## [1] "1" "2"

Change the factor levels.

levels(friend\_groups) <- c("best\_friend", "not\_best\_friend")  
friend\_groups

## [1] best\_friend not\_best\_friend best\_friend not\_best\_friend  
## Levels: best\_friend not\_best\_friend

Change the order of levels.

levels(friend\_groups) <- c("not\_best\_friend", "best\_friend")  
friend\_groups

## [1] not\_best\_friend best\_friend not\_best\_friend best\_friend   
## Levels: not\_best\_friend best\_friend

By default, the order of factor levels is taken in the order of numeric or alphabetic.

friend\_groups <- factor(c("not\_best\_friend", "best\_friend", "not\_best\_friend", "best\_friend"))  
friend\_groups

## [1] not\_best\_friend best\_friend not\_best\_friend best\_friend   
## Levels: best\_friend not\_best\_friend

The factor levels can be specified when creating the factor, if the order does not follow the default rule.

friend\_groups <- factor(c("not\_best\_friend", "best\_friend", "not\_best\_friend", "best\_friend"), levels=c("not\_best\_friend", "best\_friend"))  
friend\_groups

## [1] not\_best\_friend best\_friend not\_best\_friend best\_friend   
## Levels: not\_best\_friend best\_friend

If you want to know the number of individuals at each levels, there are two functions.

summary(friend\_groups)

## not\_best\_friend best\_friend   
## 2 2

table(friend\_groups)

## friend\_groups  
## not\_best\_friend best\_friend   
## 2 2

# Data frames: a data frame is like a matrix but can have columns with different types (numeric, character, logical).

A data frame can be created using the function data.frame().

# creating a data frame using previously defined vectors  
friends <- data.frame(name=friend\_names, age=friend\_ages, child=has\_child)  
friends

## name age child  
## Mina Mina 21 TRUE  
## Ella Ella 27 TRUE  
## Anna Anna 26 FALSE  
## Cora Cora 32 TRUE

To check whether a data is a data frame, use the function is.data.frame().

is.data.frame(friends)

## [1] TRUE

is.data.frame(my\_matrix)

## [1] FALSE

One can convert a object to a data frame using the function as.data.frame().

class(my\_matrix)

## [1] "matrix"

my\_data <- as.data.frame(my\_matrix)  
class(my\_data)

## [1] "data.frame"

A data frame can be transposed in the similar way as a matrix.

my\_data

## col1 col2 col3  
## row1 1 2 8  
## row2 3 18 37  
## row3 8 27 267  
## row4 9 10 19

t(my\_data)

## row1 row2 row3 row4  
## col1 1 3 8 9  
## col2 2 18 27 10  
## col3 8 37 267 19

To obtain a subset of a data frame can be done in similar ways as we have discussed: by index, by row/column names, or by logical vaalues.

friends["Mina",]

## name age child  
## Mina Mina 21 TRUE

# The columns of a data frame can be referred to by the names of the columns  
friends

## name age child  
## Mina Mina 21 TRUE  
## Ella Ella 27 TRUE  
## Anna Anna 26 FALSE  
## Cora Cora 32 TRUE

friends$age

## [1] 21 27 26 32

friends[friends$age > 26,]

## name age child  
## Ella Ella 27 TRUE  
## Cora Cora 32 TRUE

friends[friends$child == "TRUE",]

## name age child  
## Mina Mina 21 TRUE  
## Ella Ella 27 TRUE  
## Cora Cora 32 TRUE

Function subset() can also be used to get a subset of a data frame.

# select friends that are older than 26  
subset(friends, age > 26)

## name age child  
## Ella Ella 27 TRUE  
## Cora Cora 32 TRUE

# select the information of the ages of friends  
subset(friends, select=age)

## age  
## Mina 21  
## Ella 27  
## Anna 26  
## Cora 32

A data frame can be extended.

# add a column that has the information on the marrital status of friends  
friends$married <- c("YES", "YES", "NO", "YES")  
friends

## name age child married  
## Mina Mina 21 TRUE YES  
## Ella Ella 27 TRUE YES  
## Anna Anna 26 FALSE NO  
## Cora Cora 32 TRUE YES

A data frame can also be extended using the functions cbind() and rbind().

# add a column that has the information on the salaries of friends  
cbind(friends, salary=c(4000, 8000, 2000, 6000))

## name age child married salary  
## Mina Mina 21 TRUE YES 4000  
## Ella Ella 27 TRUE YES 8000  
## Anna Anna 26 FALSE NO 2000  
## Cora Cora 32 TRUE YES 6000

# Lists: a list is an ordered collection of objects, which can be any type of R objects (vectors, matrices, data frames).

A list can be created using the function list().

my\_list <- list(mother="Sophia", father="John", sisters=c("Anna", "Emma"), sister\_age=c(5, 10))  
my\_list

## $mother  
## [1] "Sophia"  
##   
## $father  
## [1] "John"  
##   
## $sisters  
## [1] "Anna" "Emma"  
##   
## $sister\_age  
## [1] 5 10

# names of elements in the list  
names(my\_list)

## [1] "mother" "father" "sisters" "sister\_age"

# number of elements in the list  
length(my\_list)

## [1] 4

To access elements of a list can be done using its name or index.

my\_list$mother

## [1] "Sophia"

my\_list[["mother"]]

## [1] "Sophia"

my\_list[[1]]

## [1] "Sophia"

my\_list[[3]]

## [1] "Anna" "Emma"

my\_list[[3]][2]

## [1] "Emma"

# Topic 2. Importing and exporting data in R

R base function read.table() is a general funciton that can be used to read a file in table format. The data will be imported as a data frame.

data <- read.table(file="raw\_counts.txt", sep="\t", header=T, stringsAsFactors=F)

Take a look at the beginning part of the data frame.

head(data)

## C61 C62 C63 C64 C91 C92 C93 C94 I561 I562 I563 I564 I591  
## AT1G01010 322 346 256 396 372 506 361 342 638 488 440 479 770  
## AT1G01020 149 87 162 144 189 169 147 108 163 141 119 147 182  
## AT1G01030 15 32 35 22 24 33 21 35 18 8 54 35 23  
## AT1G01040 687 469 568 651 885 978 794 862 799 769 725 715 811  
## AT1G01046 1 1 5 4 5 3 0 2 4 3 1 0 2  
## AT1G01050 1447 1032 1083 1204 1413 1484 1138 938 1247 1516 984 1044 1374  
## I592 I593 I594 I861 I862 I863 I864 I891 I892 I893 I894  
## AT1G01010 430 656 467 143 453 429 206 567 458 520 474  
## AT1G01020 156 153 177 43 144 114 50 161 195 157 144  
## AT1G01030 8 16 24 42 17 22 39 26 28 39 30  
## AT1G01040 567 831 694 345 575 605 404 735 651 725 591  
## AT1G01046 8 8 1 0 4 0 3 5 7 0 5  
## AT1G01050 1355 1437 1577 412 1338 1051 621 1434 1552 1248 1186

Depending on the format of the file, several variants of read.table() are available to make reading a file easier.

read.csv(): for reading "comma separated value" files (.csv).

read.csv2(): variant used in countries that use a comma "," as decimal point and a semicolon ";" as field separators.

read.delim(): for reading "tab separated value" files (".txt"). By default, point(".") is used as decimal point.

read.delim2(): for reading "tab separated value" files (".txt"). By default, comma (",") is used as decimal point.

data2 <- read.csv(file="raw\_counts.csv", stringsAsFactors=F)  
head(data2)

## C61 C62 C63 C64 C91 C92 C93 C94 I561 I562 I563 I564 I591  
## AT1G01010 322 346 256 396 372 506 361 342 638 488 440 479 770  
## AT1G01020 149 87 162 144 189 169 147 108 163 141 119 147 182  
## AT1G01030 15 32 35 22 24 33 21 35 18 8 54 35 23  
## AT1G01040 687 469 568 651 885 978 794 862 799 769 725 715 811  
## AT1G01046 1 1 5 4 5 3 0 2 4 3 1 0 2  
## AT1G01050 1447 1032 1083 1204 1413 1484 1138 938 1247 1516 984 1044 1374  
## I592 I593 I594 I861 I862 I863 I864 I891 I892 I893 I894  
## AT1G01010 430 656 467 143 453 429 206 567 458 520 474  
## AT1G01020 156 153 177 43 144 114 50 161 195 157 144  
## AT1G01030 8 16 24 42 17 22 39 26 28 39 30  
## AT1G01040 567 831 694 345 575 605 404 735 651 725 591  
## AT1G01046 8 8 1 0 4 0 3 5 7 0 5  
## AT1G01050 1355 1437 1577 412 1338 1051 621 1434 1552 1248 1186

R base function write.table() can be used to export a data frame or matrix to a file.

write.table(data2[1:20,], file="output.txt", sep="\t", quote=F, row.names=T, col.names=T)

It is also possible to export data to a csv file.

write.csv()

write.csv2()

# Topic 3. Basic statistics in R

|  |  |
| --- | --- |
| Description | R\_function |
| Mean | mean() |
| Standard deviation | sd() |
| Variance | var() |
| Minimum | min() |
| Maximum | max() |
| Median | median() |
| Range of values: minimum and maximum | range() |
| Sample quantiles | quantile() |
| Generic function | summary() |
| Interquartile range | IQR() |

Calculate the mean expression for each sample.

apply(data, 2, mean)

## C61 C62 C63 C64 C91 C92 C93 C94   
## 391.9998 336.4872 333.7007 380.6545 364.6587 407.0191 361.3672 314.1931   
## I561 I562 I563 I564 I591 I592 I593 I594   
## 398.8421 380.4970 382.0019 378.7685 387.7994 349.4061 400.9421 385.1493   
## I861 I862 I863 I864 I891 I892 I893 I894   
## 219.8517 379.0522 341.6387 271.0391 395.3089 426.0254 350.8965 358.8508

Calculate the range of expression for each sample.

apply(data, 2, range)

## C61 C62 C63 C64 C91 C92 C93 C94 I561 I562 I563  
## [1,] 0 0 0 0 0 0 0 0 0 0 0  
## [2,] 81764 89072 43781 64539 51516 68279 64407 53799 116414 90133 69623  
## I564 I591 I592 I593 I594 I861 I862 I863 I864 I891 I892  
## [1,] 0 0 0 0 0 0 0 0 0 0 0  
## [2,] 76426 111873 73071 114566 89630 69853 122114 98449 51835 102672 80998  
## I893 I894  
## [1,] 0 0  
## [2,] 116025 89270

Calculate the quantiles of each samples.

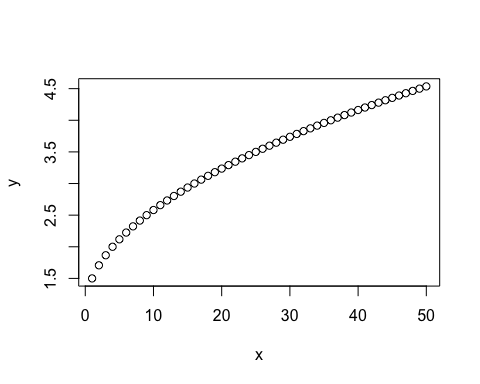
apply(data, 2, quantile)

## C61 C62 C63 C64 C91 C92 C93 C94 I561 I562 I563  
## 0% 0 0 0 0 0 0 0 0 0 0 0  
## 25% 0 0 0 0 0 0 0 0 0 0 0  
## 50% 43 38 45 47 48 45 47 39 41 45 47  
## 75% 330 270 294 331 326 344 311 266 327 333 314  
## 100% 81764 89072 43781 64539 51516 68279 64407 53799 116414 90133 69623  
## I564 I591 I592 I593 I594 I861 I862 I863 I864 I891 I892  
## 0% 0 0 0 0 0 0 0 0 0 0 0  
## 25% 0 0 0 0 0 0 0 0 0 0 0  
## 50% 45 48 41 45 43 21 49 33 31 46 49  
## 75% 316 330 298 338 333 149 327 274 211 333 354  
## 100% 76426 111873 73071 114566 89630 69853 122114 98449 51835 102672 80998  
## I893 I894  
## 0% 0.00 0  
## 25% 0.00 0  
## 50% 44.00 41  
## 75% 300.75 304  
## 100% 116025.00 89270

# Topic 4. Simple data visulization in R

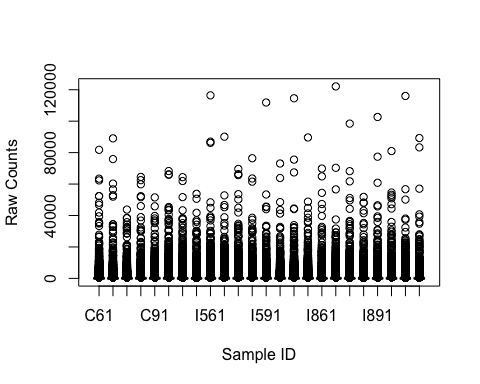
Scatter plot can be produced using the function plot().

x <- c(1:50)  
y <- 1 + sqrt(x)/2  
plot(x,y)

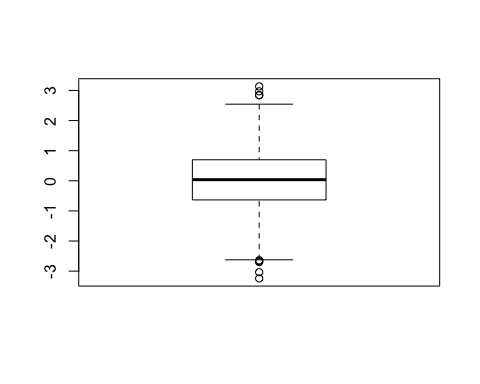


Boxplot() can be used to summarize expression data.

boxplot(data, xlab="Sample ID", ylab="Raw Counts")



x <- rnorm(1000)  
boxplot(x)



# Topic 5. Installing packages in R

There two ways to install bioconductor packages in R: biocLite(), install.packages()

source("http://bioconductor.org/biocLite.R")

## Bioconductor version 3.2 (BiocInstaller 1.20.3), ?biocLite for help

## A new version of Bioconductor is available after installing the most  
## recent version of R; see http://bioconductor.org/install

## install core packages  
#biocLite()  
## install specific packages  
#biocLite("edgeR")  
#biocLite(c("WGCNA", "gplots"))

install.packages("ggplot2", repos="http://cran.us.r-project.org")

##   
## The downloaded binary packages are in  
## /var/folders/hf/rgnmc5jj57qcf06q6f\_pmf1w0000gn/T//Rtmpp9isHd/downloaded\_packages

# biocLite() is the recommended way to install Bioconductor packages.

* Bioconductor has a repository and release schedule that differ from R (Bioconductor has a ‘devel’ branch to which new packages and updates are introduced, and a stable ‘release’ branch emitted once every 6 months to which bug fixes but not new features are introduced). This mismatch causes that the version detected by install.packages() is sometimes not the most recent 'release'.
* A consequence of the distince 'devel' branch is that install.packages() sometimes points only to the 'release' repository, while users might want to have access to the leading-edge features in the develop version.
* An indirect consequence of Bioconductor's structured release is that packages generally have more extensive dependences with one another.

# To update the installed Bioconductor packages.

#biocLite("BiocUpgrade")

# Topic 6. lapply(), sapply()

# lapply() is to apply a given function to every element of a list and obtain a list as results.

# The difference between lapply() and apply() is that lapply() can be applied on objects like dataframes, lists or vectors. Function apply() only works on an array of dimension 2 or a matrix.

# To check the syntax of using lapply():

#?lapply()  
  
data <- as.data.frame(matrix(rnorm(49), ncol=7), stringsAsFactors=F)  
dim(data)

## [1] 7 7

lapply(1:dim(data)[1], function(x){sum(data[x,])})

## [[1]]  
## [1] -6.069094  
##   
## [[2]]  
## [1] -0.06897243  
##   
## [[3]]  
## [1] 1.404895  
##   
## [[4]]  
## [1] 1.488403  
##   
## [[5]]  
## [1] -4.42584  
##   
## [[6]]  
## [1] -0.1820905  
##   
## [[7]]  
## [1] -7.085642

apply(data, MARGIN=1, sum)

## [1] -6.06909422 -0.06897243 1.40489535 1.48840298 -4.42583988 -0.18209054  
## [7] -7.08564205

lapply(1:dim(data)[1], function(x){log10(sum(data[x,]))})

## Warning in FUN(X[[i]], ...): NaNs produced  
  
## Warning in FUN(X[[i]], ...): NaNs produced  
  
## Warning in FUN(X[[i]], ...): NaNs produced  
  
## Warning in FUN(X[[i]], ...): NaNs produced  
  
## Warning in FUN(X[[i]], ...): NaNs produced

## [[1]]  
## [1] NaN  
##   
## [[2]]  
## [1] NaN  
##   
## [[3]]  
## [1] 0.147644  
##   
## [[4]]  
## [1] 0.1727205  
##   
## [[5]]  
## [1] NaN  
##   
## [[6]]  
## [1] NaN  
##   
## [[7]]  
## [1] NaN

# The function sapply() works like function lapply(), but tries to simplify the output to the most elementary data structure that is possible. As a matter of fact, sapply() is a "wrapper" function for lapply(). By default, it returns a vector.

# To check the syntax of using sapply():  
#?sapply()  
  
sapply(1:dim(data)[1], function(x){log10(sum(data[x,]))})

## Warning in FUN(X[[i]], ...): NaNs produced  
  
## Warning in FUN(X[[i]], ...): NaNs produced  
  
## Warning in FUN(X[[i]], ...): NaNs produced  
  
## Warning in FUN(X[[i]], ...): NaNs produced  
  
## Warning in FUN(X[[i]], ...): NaNs produced

## [1] NaN NaN 0.1476440 0.1727205 NaN NaN NaN

# If the "simplify" parameter is turned off, sapply() will produced exactly the same results as lapply(), in the form of a list. By default, "simplify" is turned on.

sapply(1:dim(data)[1], function(x){log10(sum(data[x,]))}, simplify=FALSE)

## Warning in FUN(X[[i]], ...): NaNs produced  
  
## Warning in FUN(X[[i]], ...): NaNs produced  
  
## Warning in FUN(X[[i]], ...): NaNs produced  
  
## Warning in FUN(X[[i]], ...): NaNs produced  
  
## Warning in FUN(X[[i]], ...): NaNs produced

## [[1]]  
## [1] NaN  
##   
## [[2]]  
## [1] NaN  
##   
## [[3]]  
## [1] 0.147644  
##   
## [[4]]  
## [1] 0.1727205  
##   
## [[5]]  
## [1] NaN  
##   
## [[6]]  
## [1] NaN  
##   
## [[7]]  
## [1] NaN