Eleventh R Practice exercise using sapply and split and also use of ellipsis to pass in additional arguments

Alan E. Berger Feb 11, 2022; edit June 20 to emphasize requirement on first argument of FUN in sapply

available at https://github.com/AlanBerger/Practice-programming-exercises-for-R

Introduction

This article will discuss **sapply** at length, and then practice using **split** and **sapply**. sapply is a version of **lapply** that when possible will give its output as a vector (or matrix when that is appropriate) instead of a list (lapply will produce a list; a list that is "naturally a vector" can be converted to the vector using the **unlist** function; using sapply avoids having to do that extra step).

Much of what is described below for sapply is also applicable to lapply.

We will go over some simple examples, and then use the iris data set to demonstrate sapply and split, and the ellipsis (. . .) functionality available in sapply.

Note if you want to copy lines from this for use in R, it is best to copy from the .Rmd file, since sometimes text in the pdf file can contain formatting characters that R does not accept. To download an Rmd file, open the Rmd file to display it in GitHub and then toward the upper right of the resulting window there will be a "raw box" (to the left of "Blame"), on which one should be able to do (for Windows): right click, "save as" to download the .Rmd file as a text file.

```
# The help for sapply indicates a "standard" call to sapply will
# have the form:

# sapply(X, FUN, ...)

# Here X is a vector or data.frame or list
# FUN is the name of a function that is available in the current R session
# (which could be a user defined function, or a function that
# "comes with R", or that has been "loaded" (via library)from a package),
# OR is an anonymous function defined within the call to sapply.
# The function specified by FUN will be applied to each entry of X if X
# is a vector; to each column of X (taken as a vector) if X is
# a data frame, to each entry of X if X is a list (vectors and data.frames
# are special types of list). When possible the result from sapply will
# be a vector (or matrix if that is appropriate). Otherwise sapply will
# return a list (just as lapply would).
```

```r
# The optional ellipsis argument can be used to pass in additional
# arguments to the function specified in the FUN argument of sapply
# as illustrated below.

# a simple case: square each element of a vector (yes we could do this
# by x^2, but here we demonstrate use of sapply and an anonymous function):

v <- 1:5
sapply(X = v, FUN = function(y) y^2)
```

```
## [1]  1  4  9 16 25
```

```r
# ********  IT IS BEST to use the form: argument.name = user.chosen.value
# ********  when calling one of the apply family of functions. In this call
# ********  to sapply, the name of the first argument of sapply is X and
# ********  the value chosen for X is v; and the second argument of sapply
# ********  is FUN, and the value chosen for FUN is the
# ********  anonymous function, function(y) y^2

# Note y in the definition of this function is a "formal argument",
# also called a "dummy argument", meaning any variable name permissible
# in R could be used, for example

# sapply(X = v,
# FUN = function(user.selected.name) user.selected.name^2)

# One could also optionally include curly brackets to delineate the
# function:

# sapply(X = v, FUN = function(y) {y^2})

# sapply, in effect, successively calls the function the user has
# provided for FUN, here function(y) {y^2}
# with each entry of v as its argument,
# and "collects" the results in a vector.

# My point of view is it is fine to use an anonymous function if it
# is short (will fit in a line or so), but otherwise it makes the code
# much easier to proofread and so avoid/detect bugs if one codes the
# function as a stand-alone function, call it, for example,
# user.function and uses its name in the FUN argument of sapply via
#    FUN = user.function
# Here user function could also be any function that is available in the
# current R session
```

```
# demonstrate use of the ellipsis argument of sapply to pass in an
# additional argument of FUN


sapply(X = v, FUN = function(x, power) x^power, power = 3)
```

```
## [1]   1   8  27  64 125
```

```
# Here each entry of v is passed into the function as the value for x,
# and the additional argument, power, of the function is supplied via
# the third (ellipsis) argument of sapply.

# One could also rely on lexical scoping to "find" the value of power,
# but that is not good programming practice when it can be avoided,
# since the value where R "finds" it might not be what was wanted
# (or it might have gotten changed from what you thought it would be).

power <- 4
sapply(X = v, FUN = function(x) x^power)  # this worked but is not best practice
```

```
## [1]   1  16  81 256 625
```

```
# Here is an example with two additional arguments supplied using
# the ellipsis functionality of sapply, and where sapply returns a matrix

v <- 1:5
sapply(X = v, FUN = function(x, power1, power2) {c(x^power1, x^power2)},
       power1 = 2, power2 = 3)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    9   16   25
## [2,]    1    8   27   64  125
```

The first argument of the function in sapply MUST BE for the entries of X

Suppose one is calling sapply via, for example,

result <- sapply(X = v, FUN = user.function, other.arg1 = a1, other.arg2 = a2)

where v is a vector or data frame or list, so sapply will successively call user.function with each entry, refer to it as E, of v (each column if v is a data frame). **It is important to note that**:

sapply will successively call user.function with E **as the first argument of user.function**

So user.function (if programmed by us) should be programmed that way, and if user.function is a built in function that comes with the base R installation, or has been "loaded" from a package, its first argument should be for entries of v. (Technically, this is not absolutely required, but not conforming with this "first argument condition" can lead to mysterious errors and is best avoided.) In the conceptual example above, other.arg1 and other.arg2 are other arguments of user.function and their values have been set to a1 and a2, respectively, in this call. Arguments passed into user.function via the ellipsis functionality of sapply must be called in the

argument.name = chosen.value

format.

This "first argument condition" is pretty natural, in that for many R functions, the first argument (or first couple of arguments) are what the function "acts on" and following arguments (which have default values) govern options on how that is done. For example the mean function takes the mean of a vector, and its other options modulate how that is done, and read.table reads in a suitable file as a data frame, and it has quite a few options on how that is done, and the plot function can produce a scatterplot of y vs. x with a multitude of options allowing for fine detailed control of the form of the plot.

Now use the iris data set to illustrate use of sapply

From the R help on the iris data set (? iris): "This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are Iris ***setosa***, ***versicolor***, and ***virginica***"

```
# from the R help on the iris data set:
# iris is a data frame with 150 cases (rows) and 5 variables (columns)
# named Sepal.Length, Sepal.Width, Petal.Length, Petal.Width, and Species,
# respectively, for 50 flowers from each of 3 species of iris.
# The species are Iris setosa, versicolor, and virginica.

# There are equal numbers of the three types of flowers, and they are
# grouped together in the data frame, but neither "equal numbers" nor
# "grouped together" are necessary for using split together with sapply
# as will be demonstrated below.

data(iris)  # make the iris data set available to this R session
iris.df <- iris  # a copy, emphasizing it is a data frame

# take a look at it
# head(iris.df)
```

```
# tail(iris.df)

# place an NA into the Sepal.Width column, to later demonstrate
# how to use the ellipsis (...) functionality for sapply
# to pass in one or more additional optional arguments into the function
# being used (for example na.rm = TRUE, trim = 0.11, for the mean function)

iris.df[2,2] <- NA
# check that was done
head(iris.df)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9          NA          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
# An aside: Explanation of what the optional argument trim in
# the mean function does:

# we will use this information to do a "by hand" check of
# the output obtained below from using sapply with the mean function
# on the iris.df data frame

# If V is a numeric vector and one does
# mean(V, na.rm = TRUE, trim = w)
# what will happen is that the mean function will, in effect,
# first produce the vector Vn which is V but with any and all NAs removed
# from V.
# If L is the length of Vn (here assume it is > 0), then let K = w * L
# (w was what the argument trim was set to; the default for trim is 0).
# Let Ki be the integer part of K, e.g., if K was 2.3 then Ki is 2
# if K was 0.4 then Ki is 0
# Then mean(V, na.rm = TRUE, trim = w) is evaluated as follows:
# Let Vn.sorted be the result of doing sort(Vn)
# Let Vn.sorted.trimmed be the result of:
#     removing the first Ki entries of Vn.sorted AND
#     removing the last Ki entries of Vn.sorted
# (assume that Vn.sorted.trimmed still has at least 1 element left in it)

# mean(V, na.rm = TRUE, trim = w) is then equal to mean(Vn.sorted.trimmed)
```

```r
# first use sapply on the 4 data columns of iris.df,
# using na.rm = TRUE and trim = 0.11
# (which will trim 16 entries from each end of the vector whose mean
# is being calculated). The sapply command below will successively pass
# each column of iris.df[, 1:4] AS A VECTOR into the mean function

sapply(X = iris.df[, 1:4], FUN = mean, na.rm = TRUE, trim = 0.11)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.807627     3.043590     3.762712     1.183898
```

```r
# Note that sapply "captured" the names of the columns of iris.df[, 1:4]

# since the lengths of the 4 columns after removing NAs
# were 150, 149, 150, 150;
# setting trim equal to 0.11 in effect removed 16 elements from
# each end of each column AFTER
# any NAs were removed and the column was sorted;
# then the mean was calculated

# check the result from sapply above "by hand"

x <- sort(na.omit(iris.df[[1]]))
mean(x[17:(length(x) - 16)])
```

```
## [1] 5.807627
```

```r
x <- sort(na.omit(iris.df[[2]]))
mean(x[17:(length(x) - 16)])
```

```
## [1] 3.04359
```

```r
# these and also checking columns 3 and 4 are OK

# Note
# sapply(X = iris.df[, 1:4], FUN = mean, na.rm = TRUE, trim = 0.11)
# worked to get the optional arguments
# na.rm = TRUE, trim = 0.11
# into the mean function (FUN = mean) since the mean function
# is "part of" R and it has na.rm and trim as optional arguments
```

```
# When using an anonymous function in sapply, AND using the optional
# ellipsis (...) argument of sapply to pass in additional arguments to
# the function specified by FUN, one needs to either have those additional
# arguments be declared arguments of the function (as with the power
# argument(s) in the examples above), OR to have an ellipsis in the
# argument list of the function specified by FUN, AND used appropriately
# within the anonymous function. Again, the first argument
# of the function FUN in sapply should be for
# what will be passed from each entry of the first argument X of sapply.
# For example in the call to sapply below,
# x will successively have the value of each of the first 4 columns
# of iris.df

sapply(X = iris.df[, 1:4],
       FUN = function(x, ...) {mean(x, ...)},
       na.rm = TRUE, trim = 0.11)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.807627     3.043590     3.762712     1.183898
```

```
# In effect, in the above call to sapply,
#     the ellipsis ...   was "set to":   na.rm = TRUE, trim = 0.11
# and that got "passed through" via the ... in the argument list of
# the anonymous function AND within the anonymous function itself
# (here "into" the ... in the argument list of the mean function)
#    * * *  It was necessary to have ... included in the argument list of
#    * * *  the mean function within this anonymous function

# Here is what happens if the ellipsis ... is left out of the
# argument list of mean in the anonymous function
# (it runs, but na.rm = TRUE, trim = 0.11 are not used,
# so the result is not what we wanted):
sapply(X = iris.df[, 1:4],
       FUN = function(x, ...) {mean(x)},
       na.rm = TRUE, trim = 0.11)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333           NA     3.758000     1.199333
```

```
# The result is indeed the same as if na.rm = TRUE and trim = 0.11 were
# NOT invoked when calling the mean function:
sapply(X = iris.df[, 1:4], FUN = mean)
```

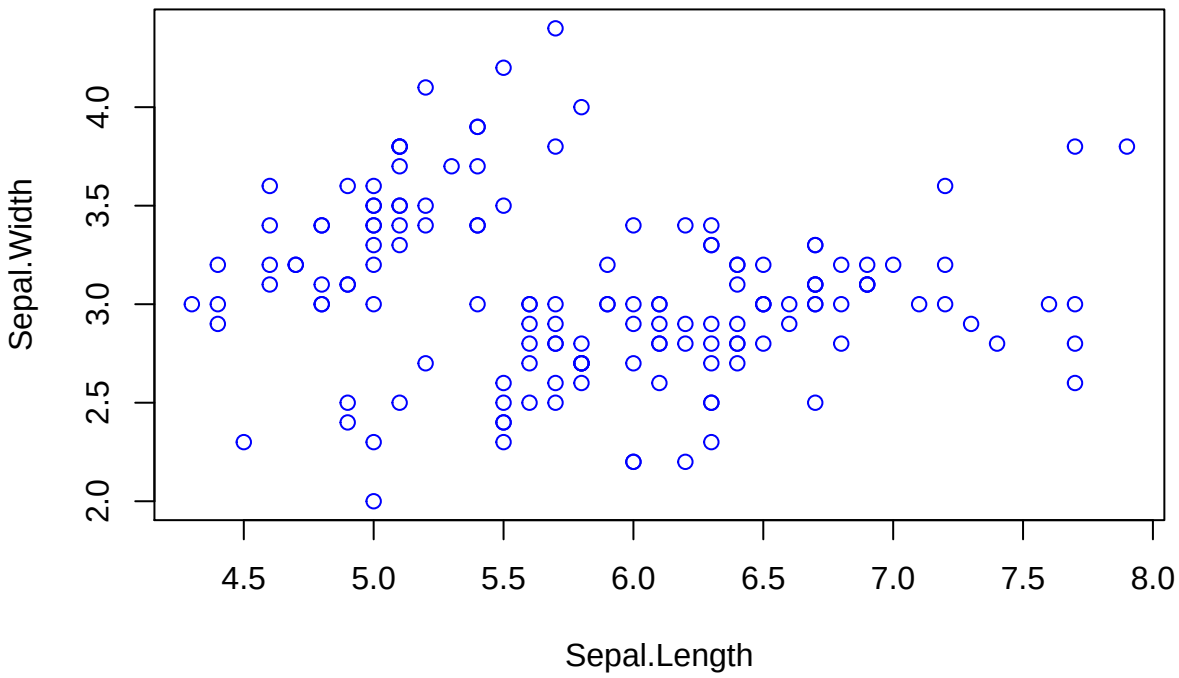
```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333           NA     3.758000     1.199333
```

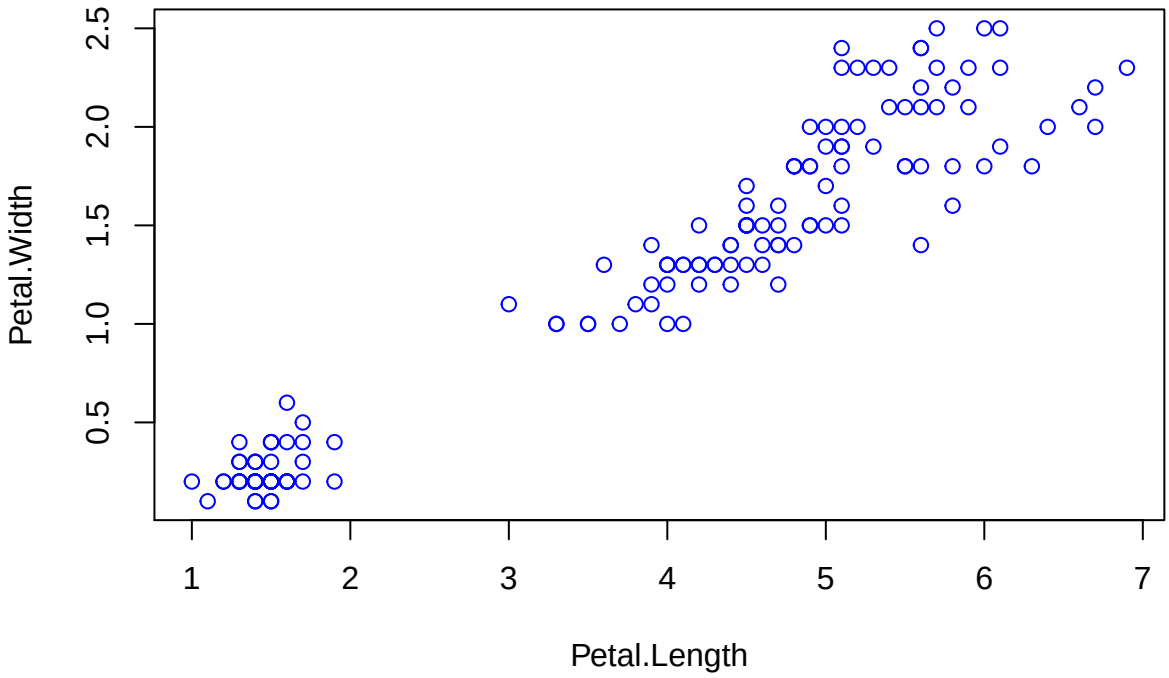
```
# Note this also works correctly:
sapply(X = iris.df[, 1:4],
 FUN = function(x, na.rm.value, trim.value)
                      {mean(x, na.rm = na.rm.value, trim = trim.value)},
               na.rm.value = TRUE, trim.value = 0.11)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.807627     3.043590     3.762712     1.183898
```

```
#  ************  note various plotting functions, e.g., plot(x,y, ...)
# have ... in their argument list,
# so various of the many optional arguments for them can be passed in
# by sapply.
# For example, the first two arguments of plot, x and y, can be vectors
# for making a scatterplot.
# If X = data.frame.list is a list of data frames whose first two columns
# are numeric values (our example will be constructed from iris.df),
# and  plotting_function  is a function that
# will make a scatterplot from data in the first two columns of a data
# frame, then, one could, conceptually, do for example:

# sapply(X = data.frame.list, FUN = plotting_function,
#        type = "p", pch = 1, col = "blue")
#
# Here plotting_function
# should extract the x and y vectors from the first and second columns
# of the data frame it is called with, and then call the plot function via
# plot(x, y, ...)
# In this setup, plotting_function must have an ellipsis in its argument
# list. Then sapply would pass through, via the ellipsis functionality,
# the various plotting arguments that were specified
# (there are many many optional arguments for plot);
# the ones given here "say": do a point plot with circle symbols,
# and have the symbol color be blue

plotting_function <- function(df, ...) {
# here df is a data frame that will be passed in by sapply
# scatter plot column 2 of df vs. column 1 of df
   x <- df[[1]]
   y <- df[[2]]
```

```
# get x and y axis labels from column names of df
  xlabel <- colnames(df)[1]
  ylabel <- colnames(df)[2]

# other arguments for the plot function are passed into plot from
# the call to sapply via the ellipsis

   plot(x, y, xlab = xlabel, ylab = ylabel, ...)
}


# Demonstrate this:
# Construct a list of two data frames, each with two columns,
# from iris.df

data.frame.list = list(iris.df[, c(1,2)], iris.df[, c(3,4)])


# invoke sapply, creating 2 plots, while using the ellipsis functionality
# to pass plotting arguments into the plot function

# one could do more informative plotting with this data,
# but the point here is to illustrate
# use of the ellipsis functionality in sapply

sapply(X = data.frame.list, FUN = plotting_function,
   type = "p", pch = 1, col = "blue")
```


Sepal.Width
2.0
2.5
3.0
3.5
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
Sepal.Length




```
## [[1]]
## NULL
##
## [[2]]
## NULL
```

```
# this produced the plots along with a "pro forma" "NULL" from each
# call to the plot function
# to suppress this useless text, one could use the invisible function:

# invisible(sapply(X = data.frame.list, FUN = plotting_function,
#   type = "p", pch = 1, col = "blue"))
```

A note when using sapply and passing in arguments using its ellipsis capability

When using the ellipsis functionality in sapply to pass arguments, the R help on sapply recommends specifying the X and FUN arguments explicitly, (not just by position), for example

sapply(X = some.data.frame, FUN = function.that.does.plots, type = "p", pch = 1, col = "green")

The split function.

The **split** function can be used to "split up" a data frame df into a list of data frames that are subsets of df, call it df.split.list, based on a character or factor column of df. For each of the distinct entries E in the character (or factor) column of df being used to "do the split", there will be a data frame in df.split.list which is the subset of df containing all the rows of df for which the entry in the column being used to do the split equals E, and the name of that entry in df.split.list will be the character string E.

The practice exercise

The practice exercise is: given the name of one of the 4 numeric data columns in iris.df, compute the mean of the entries in that column that correspond to each of the 3 iris flower types (so one will compute 3 means; one mean for all the
entries in that column whose iris type is setosa, one mean for the versicolor flowers and one mean for virginica flowers). Do this by using split on iris.df to produce a list of 3 data frames, one for each of the 3 types of iris flower. Then write a function that given one of these data frames, and the name of one of the 4 numeric data columns in iris.df, will compute the mean for the specified column. Use this function, and the list of 3 data frames produced by split, in sapply to get the result.

Try doing this before looking at one possible solution in the R session given below.

```
# make this session self contained
data(iris)  # make the iris data set available to this R session
iris.df <- iris  # a copy, emphasizing it is a data frame

head(iris.df)  # look at it
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
# split iris.df by the 3 iris flower types (species)
df.split.list <- split(iris.df, iris.df$Species)
```

```r
get.mean.of.specified.column <- function(df, column.name) {
# df will be one of the 3 data frames in df.split.list
# column.name will be one of the quantities measured for each flower:
# (Sepal.Length, Sepal.Width, Petal.Length, Petal.Width)

mean.of.given.column <- mean(df[[column.name]])
# there are no NAs in iris.df so here don't need to use na.rm = TRUE
# (in an example above I placed an NA in iris.df to test ways of passing
#  na.rm = TRUE into the mean function)

return(mean.of.given.column)
}

# Use sapply for the various columns
# (One could also use sapply (with a modified function) to return a matrix
# containing the means for the 3 flower species for each of the 4
# measured quantities. The modified function would return a vector of the
# means for the 4 quantities for any of the 3 data frames in df.split.list)

sapply(X = df.split.list, FUN = get.mean.of.specified.column,
       column.name = "Sepal.Length")
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

```r
sapply(X = df.split.list, FUN = get.mean.of.specified.column,
       column.name = "Sepal.Width")
```

```
##     setosa versicolor  virginica
##      3.428      2.770      2.974
```

```r
sapply(X = df.split.list, FUN = get.mean.of.specified.column,
       column.name = "Petal.Length")
```

```
##     setosa versicolor  virginica
##      1.462      4.260      5.552
```

```r
sapply(X = df.split.list, FUN = get.mean.of.specified.column,
       column.name = "Petal.Width")
```

```
##     setosa versicolor  virginica
##      0.246      1.326      2.026
```

```
# Now program the function to be used in sapply to return the
# vector of all 4 means. Then sapply will return a matrix

# Here is a solution

get.all.4.means.in.df <- function(df) {
# df will be one of the 3 data frames in df.split.list
# Its 4 numeric column names are
#     Sepal.Length, Sepal.Width, Petal.Length, Petal.Width

# use sapply within this function to get the 4 means
the.4.means <- sapply(df[, 1:4], mean)
# this will also "capture" the column names

# there are no NAs in iris.df so here don't need to use na.rm = TRUE
# (in an example above I placed an NA in iris.df to test ways of passing
#  na.rm = TRUE into the mean function)

return(the.4.means)
}

# use sapply and this function to get the matrix of means
result <- sapply(X = df.split.list, FUN = get.all.4.means.in.df)
result
```

```
##              setosa versicolor virginica
## Sepal.Length  5.006      5.936     6.588
## Sepal.Width   3.428      2.770     2.974
## Petal.Length  1.462      4.260     5.552
## Petal.Width   0.246      1.326     2.026
```

```
# notice sapply conveniently captured the row and column names for
# the matrix from the data frames it operated on

rownames(result)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
```

```
# And in particular:

colnames(result)
```

```
## [1] "setosa"     "versicolor" "virginica"
```

```r
# will be the same as

names(df.split.list)
```

```
## [1] "setosa"     "versicolor" "virginica"
```

```r
# if one prefers one can transpose the matrix
print(t(result))
```

```
##              Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa              5.006       3.428        1.462       0.246
## versicolor          5.936       2.770        4.260       1.326
## virginica           6.588       2.974        5.552       2.026
```

Hope this discussion has been helpful.

= =