FACTORS and DATAFRAMES and Functions

```
# Create two vectors of different lengths.
vector3 <- c(9,1,0)
vector4 <- c(6,0,11,3,14,1,2,6,9)
array2 <- array(c(vector3,vector4),dim = c(3,3,2))
print(array2)
```

```
     [,1] [,2] [,3]
[1,]    9    6    3
[2,]    1    0   14
[3,]    0   11    1

, , 2

     [,1] [,2] [,3]
[1,]    2    9    6
[2,]    6    1    0
[3,]    9    0   11

>
```

```
# Create two vectors of different lengths.
vector3 <- c(9,1,-2)
vector4 <- c(6,-1,11,3,14,1,2,6,9)
array2 <- array(c(vector3,vector4),dim =
c(3,3,2))
print(array2)
```

```
, , 1

     [,1] [,2] [,3]
[1,]    9    6    3
[2,]    1   -1   14
[3,]   -2   11    1

, , 2

     [,1] [,2] [,3]
[1,]    2    9    6
[2,]    6    1   -1
[3,]    9   -2   11

, , 3

     [,1] [,2] [,3]
[1,]    3    2    9
[2,]   14    6    1
[3,]    1    9   -2
```

```
# Create two vectors of different lengths.
vector3 <- c(9,1,-2)
vector4 <- c(6,-1,11,3,14,1,2,6,9)
array2 <- array(c(vector3,vector4),dim = c(3,3,2))
print(array2)
# create matrices from these arrays.
matrix1 <- array1[,,2]
matrix2 <- array2[,,2]
```

FACTORS

Factor is a data structure used for fields that takes only predefined, finite number of values (categorical data for encoding). For example: a data field such as marital status may contain only values from single, married, separated, divorced, or widowed.

```
x
[1] single married married single
Levels: married single
```

Factor x has four elements and two levels

FACTORS

The primary advantage of a factor object is efficiency in data storage. An integer requires less memory to store than a character. Such efficiency was highly desirable when many computers had much more limited resources than current machines

FACTORS

To check if a variable is a factor or not using class() function

Levels of factor can be checked using the levels() function

```
class(x)
[1] "factor"
> levels(x)
[1] "married" "single"
```

FACTORS

```
x <- factor(c("single", "married", "married", "single"));
x
 [1] single married married single
Levels: married single
x <- factor(c("single", "married", "married", "single"), levels =
c("single", "married", "divorced"));
 >x
 [1] single married married single
Levels: single married divorced
```

FACTORS

Levels are stored in a character vector and the individual elements are stored as indices

```
x <- factor(c("single","married","married","single"))
str(x)
Factor w/ 2 levels "married","single": 2 1 1 2
```

How to Access Components of FACTORS

Accessing components of a factor is similar to that of vectors

```
>x
[1] single married married single
Levels: married single
>x[3]                        # access 3rd element
[1] married
Levels: married single
> x[c(2, 4)]           # access 2nd and 4th element
[1] married single
Levels: married single
x[-1]        # access all but  not 1st element
[1] married married single
Levels: married single
> x[c(TRUE, FALSE, FALSE, TRUE)] # using logical vector
[1] single single
Levels: married single
```

How to Modify a FACTOR?

```
x
[1] single married married single
 Levels: single married divorced

x[3] <- "widowed" # cannot assign values outside levels
 Warning message:
In `[<-.factor`(`*tmp*`, 3, value = "widowed") :
invalid factor level, NA generated
>single     married <NA> single
Levels:married  single
```

How to Modify a FACTOR?

To add the value to the level first

```
>levels(x) <- c(levels(x), "widowed") # add new level
x[3] <- "widowed"
x
[1] single married widowed single
Levels: single married widowed
```

How to Modify a FACTOR?

To add the value to the level first

Use relevel function

```
g<-relevel(f, "n") # moves n to be the first level
levels(g)
# [1] "n" "c" "W"
```

FACTORS

```
# Create a vector as input.
data <-
c("East","West","East","North","North","East","West","West","West",
"East","North")

print(data)
print(is.factor(data))    # FALSE

# Apply the factor function.
factor_data <- factor(data)
print(is.factor(factor_data))   # TRUE
```

Generating Factor Levels

We can generate factor levels by using the **gl()** function. It takes two integers as input which indicates how many levels and how many times each level.

```
gl(n, k, labels)
```

the parameters used −

- **n** is a integer giving the number of levels.
- **k** is a integer giving the number of replications.
- **labels** is a vector of labels for the resulting factor levels.

Generating Factor Levels

```
v <- gl(3, 4, labels = c("Tampa", "Seattle","Boston"))
print(v)
```

```
[1] Tampa Tampa Tampa Tampa Seattle Seattle Seattle Seattle Boston
[10] Boston Boston Boston
Levels: Tampa Seattle Boston
```

Changing the Order of Levels FACTORS or Redefine the factor

The order of the levels in a factor can be changed by applying the factor function again with new order of the levels.

```
data <- c("East","West","East","North","North","East","West",
"West","West","East","North")
 # Create the factors
 factor_data <- factor(data)
print(factor_data)
 # Apply the factor function with required order of the level.
new_order_data <- factor(factor_data,levels = c("East","West","North"))
print(new_order_data)
```

```
[1] East West East North North East West West West East Nor
Levels: East North West
[1] East West East North North East West West West East Nor
Levels: East West North
```

Factors:Labels

When the input levels are different than the desired output levels, we use the labels parameter which causes the levels parameter to become a "filter" for acceptable input values, but leaves the final values of "levels" for the factor vector as the argument to labels:

```
fm <- factor(LETTERS[1:6], levels = LETTERS[1:4],  # only 'A'-'D' as inpu
             labels = letters[1:4])          # but assigned to 'a'-'d'
 fm
```

Ordered Factors

ordered factors are different from factors, the first one are used represent *ordinal data*, and the second one to work with *nomi data*. At first, it does not make sense to change the or of levels for ordered factors, but we can change its labels.

```
z <- factor(LETTERS[6:1])
print(z)
is.ordered(z)
```

```
[1] F E D C B A
Levels: A B C D E F
>  is.ordered(z)
[1] FALSE
```

```
z <- factor(LETTERS[6:1], ordered=TRU
is.ordered(z)


#TRUE
```

Factors

```
## Now, labels maybe duplicated:
## factor() with duplicated labels allowing to "merge levels"
x <- c("Man", "Male", "Man", "Lady", "Female")
## Map from 4 different values to only two levels:
 (xf <- factor(x, levels = c("Male", "Man" , "Lady", "Female"),
                labels = c("Male", "Male", "Female", "Female")))
#> [1] Male Male Male Female Female
#> Levels: Male Female
```

Factors(Battery Example)

```
set.seed(18)
 ii <- sample(1:4, 20, replace=T)
 ii
 fii <- factor(ii, levels=1:4) # it is necessary to indicate the
numeric levels
 fii
 levels(fii) <- c("empty", "low", "normal", "full")
 fii
```

DATAFRAMES

Data frame is a two dimensional data structure in R. It is a special case of a list which has each component of equal length.
Each component form the column and contents of the component form the rows.

How to create a DATAFRAME

```
# Create the data frame.
  emp.data <- data.frame(
  emp_id = c (1:5),
  emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
  salary = c(623.3,515.2,611.0,729.0,843.25)
)
# Print the data frame.
print(emp.data)
```

```
> print(emp.data)
  emp_id emp_name salary
1      1     Rick 623.30
2      2      Dan 515.20
3      3 Michelle 611.00
4      4     Ryan 729.00
5      5     Gary 843.25
>
```

Functions of DATAFRAMES

```
SN  Age  Name
1   1    21  John
2   2    15  Dora
```

```
>names(x)
[1] "SN" "Age" "Name"
>ncol(x)
[1] 3
>nrow(x)
[1] 2
> length(x) # returns length of the list, same as ncol()
 [1] 3
```

DATAFRAMES

```
> typeof(emp.data)
[1] "list"
> class(emp.data)
[1] "data.frame"
>
```

How to create a DATAFRAME

```
> x <- data.frame("SN" = 1:2, "Age" = c(21,15), "Name" = c("John","Dora"))
> str(x)     # structure of x
'data.frame':    2 obs. of  3 variables:
$ SN   : int  1 2
$ Age  : num  21 15
$ Name: Factor w/ 2 levels "Dora","John": 2 1
```

The Third column , Name is type factor, instead of a character vector,

By default , data,frame() function converts character vector into factor.
To supress this behaviour, we can pass the argument
StringAsFactors=FALSE

How to create a DATAFRAME

```
x <- data.frame("SN" = 1:2, "Age" = c(21,15), "Name"
= c("John", "Dora"), stringsAsFactors = FALSE)

str(x) # now the third column is a character

 vector 'data.frame': 2 obs. of 3 variables:
 $ SN : int 1 2
$ Age : num 21 15
$ Name: chr "John" "Dora"
```

How to create a DATAFRAME

```
#
# Creating a dataframe
df = data.frame(
  "Name" = c("Amiya", "Raj", "Asish"),
  "Language" = c("R", "Python", "Java"),
  "Age" = c(22, 25, 45)
)
print(df)
 # Accessing first and second column
cat("Accessing first and second column\n")
print(df[, 1:2])
```

```
   Name Language Age
1 Amiya        R  22
2   Raj   Python  25
3 Asish     Java  45

Accessing first and second column
   Name Language
1 Amiya        R
2   Raj   Python
3 Asish     Java
```

How to access components of DATAFRAME

Components of data frame can be accessed like a list or like matrix

We can use either [, [[or $ operator to access columns of data frame

```
Accessing Like List
x["Name"]
 Name
1 John
2 Dora
> x$Name
 [1] "John" "Dora"
> x[["Name"]]
[1] "John" "Dora“
 > x[[3]]
[1] "John" "Dora"
```

Accessing with [[or $ is similar . It differs for [in that , indexing with [will return us a data frame but the other two will reduce into a vector

How to access components of DATAFRAME

Accessing Like Matrix

We will use the `trees` dataset which contains `Girth`, `Height` and `Volume` for Black Cherry Trees.

A data frame can be examined using functions like `str()` and `head()`.

```
> str(trees)
'data.frame':    31 obs. of 3 variables:
$ Girth : num   8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
$ Height: num   70 65 63 72 81 83 66 75 80 75 ...
$ Volume: num   10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
> head(trees,n=3)
Girth  Height  Volume
1    8.3       70    10.3
2    8.6       65    10.3
3    8.8       63    10.2
```

How to access components of DATAFRAME

Accessing Like Matrix

Now we proceed to access the data frame like a matrix.

```
> trees[2:3,]     # select 2nd and 3rd row
Girth Height Volume
2    8.6       65    10.3
3    8.8       63    10.2
> trees[trees$Height > 82,]     # selects rows with Height greater than 82
Girth Height Volume
6    10.8       83    19.7
17   12.9       85    33.8
18   13.3       86    27.4
31   20.6       87    77.0
> trees[10:12,2]
[1] 75 79 76
```

We can see in the last case that the returned type is a vector since we extracted data from a single column.

How to access components of DATAFRAME

Accessing Like Matrix

```
> trees[10:12,2, drop = FALSE]
Height
10       75
11       79
12       76
```

How to Modify a Data Frame in R

```
>x
SN Age Name
1 1 21 John
2 2 15 Dora
x[1,"Age"] <- 20;    # through reassignment variable
x
  SN Age Name
 1 1 20 John
 2 2 15 Dora
```


```
x[[3]][3] = 30
```

emp.data[[3]] = 34

Adding Components Data Frame in R (cbind and rbind)

```
rbind(x,list(1,16,"Paul"))
 SN Age Name
1 1 20 John
2 2 15 Dora
3 1 16 Paul
```

```
newDf = rbind(df,
data.frame(Name = “Geeta
Kasana",Language = "C",
 Age = 23                    )
cat("After Added a row\n")
print(newDf)
```

```
cbind(x,State=c("NY","FL"))
  SN Age Name State
1 1 20    John NY
2 2 15    Dora FL
```

```
newDf = cbind(df, Rank=c(3
5, 1))
 cat("After Added a
column\n")
print(newDf)
```

Deleting Component

Data frame columns can be deleted using NULL

```
x$State <- NULL
x
  SN Age Name
1  1  20 John
2  2  15 Dora
```

Data frames rows can be deleted through reassignments

```
x <- x[-1,]
>x
 SN Age Name
2 2 15 Dora
```

Data Frame using subset

```
# Creating a dataframe
df = data.frame(
  "Name" = c("Amiya", "Raj", "Asish"),
  "Language" = c("R", "Python", "Java"),
  "Age" = c(22, 25, 45)
)
print(df)
 # Selecting the subset of the data frame
# where Name is equal to Amiya oR age is greater than 30
newDf = subset(df, Name =="Amiya"|Age>30)

cat("After Selecting the subset of the data frame\n")
print(newDf)
```

```
> print(df)
   Name Language Age
1 Amiya        R  22
2   Raj   Python  25
3 Asish     Java  45
> print(newDf)
   Name Language Age
1 Amiya        R  22
3 Asish     Java  45
>
```

Adding Components Data Frame in R

Since data frames are implemented as list , we can add new columns through simple list like assignments

```
x
SN Age Name
1 1 20 John
2 2 15 Dora
 > x$State <- c("NY","FL");
x SN Age Name State
1  1 20  John  NY
2  2 15   Dora FL
```

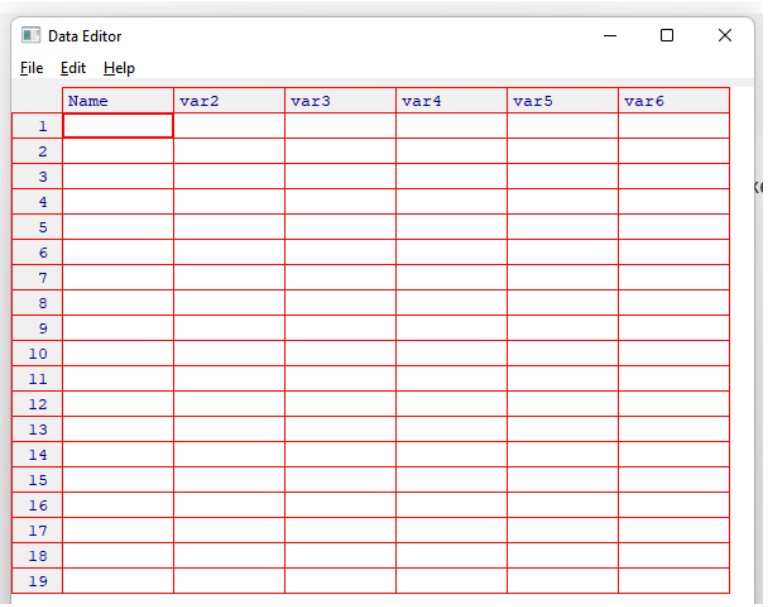
Editing dataframes using the edit() command:

STEP 1: *myTable = data.frame()*

STEP 2: *myTable = edit(myTable)*

STEP 3: *Enter the entries*

Step 4: *myTable*



FORMATS

```
# Format treats everything as a string.
result <- format(6)
print(result)
```

```
# The minimum number of digits to the right of the decimal point.
result <- format(23.47, nsmall = 5)
print(result)
```

```
# Numbers are padded with blank in the beginning for width.
result <- format(13, width = 3)
print(result)
```

```
# Display numbers in scientific notation.
result <- format(c(6, 13.14521), scientific = TRUE)
print(result)
```

FORMATS

```
# Total number of digits displayed. Last digit rounded off.
result <- format(23.123456789, digits = 9)
print(result)
```

```
[1] "    23"
```

```
# Left justify strings.
result <- format("Hello", width = 8, justify = "l")
print(result)
```

```
> print(result)
[1] "Hello   "
```

```
#Counting the number of Characters
result <- nchar("Count the number of characters")
print(result)          #30
```

FUNCTIONS

```
# Create a function to print squares of numbers in sequence.
new.function <- function(a) {
   for(i in 1:a) {
      b <- i^2
      print(b)
   }
}

# Call the function new.function supplying 6 as an argument.
new.function(6)
```

```
> new.function(6)
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

FUNCTIONS

```
# Create a function with arguments.
new.function <- function(a,b,c) {
  result <- a * b + c
  print(result)
  }

# Call the function by position of arguments.
new.function(5,2,3)

# Call the function by names of the arguments.
new.function(b = 5, a = 2, c = 3)
```

```
> new.function(5,2,3)
[1] 13
```

FUNCTIONS

```
# Create a function with default arguments.
new.function <- function(a = 3, b = 6) {
   result <- a * b
   print(result)
}

# Call the function without giving any argument.
new.function()

# Call the function with giving new values of the argument.
new.function(9,5)
```

```
18
45
```

FUNCTIONS

```
# Create a function with default arguments.
# Without using NEW keyword.
myFirstFunction <- function(a = 3, b = 6) {
  result <- a * b
  print(result)
}

# Call the function without giving any argument.
myFirstFunction()

# Call the function with giving new values of the argument.
myFirstFunction(9,5)
```

```
18
45
```

FUNCTIONS(LAZY EVALUATION IN R)

```
f <- function(a, b=c)
{
  c = mean(1:3);
  a*b
}
print(f(3))
```

6


Thank
You