UNIT II

DATA FRAMES

Introduction

- A data frame is like a matrix, with a two-dimensional rows-and columns structure.

- However, it differs from a matrix in that each column may have a different mode.

- For instance, one column may consist of numbers, and another column might have character strings.

- Just as lists are the heterogeneous analogs of vectors in one dimension, data frames are the heterogeneous analogs of matrices for two-dimensional data.

Create Data Frames

```
# 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),

   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
  )
# Print the data frame.
print(emp.data)
```

```
> kids <- c("Jack","Jill")
> ages <- c(12,10)
> d <- data.frame(kids,ages,stringsAsFactors=FALSE)
> d  # matrix-like viewpoint
  kids ages
1  Jack   12
2  Jill   10
```

If the named argument stringsAsFactors is not specified, then by default, stringsAsFactors will be TRUE. (You can also use options() to arrange the opposite default.) This means that if we create a data frame from a character vector—in this case, kids—R will convert that vector to a *factor*. Because our work with character data will typically be with vectors rather than factors,

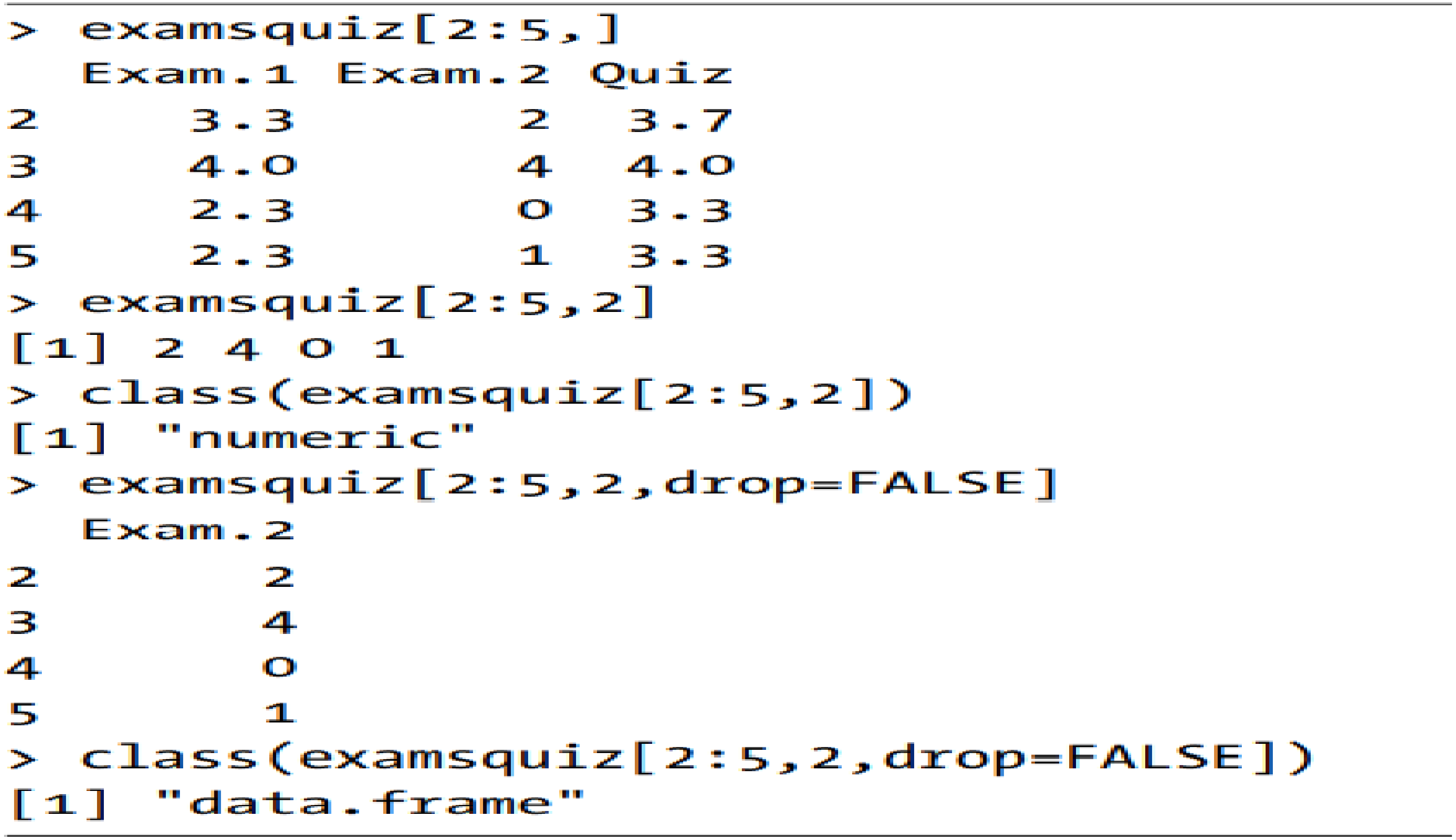
Extract Data from Data Frame

```
# 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),

   start_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-
11",
      "2015-03-27")),
   )
# Extract Specific columns.
result <- data.frame(emp.data$emp_name,emp.data$salary)
print(result)
```

Other Matrix-Like Operations

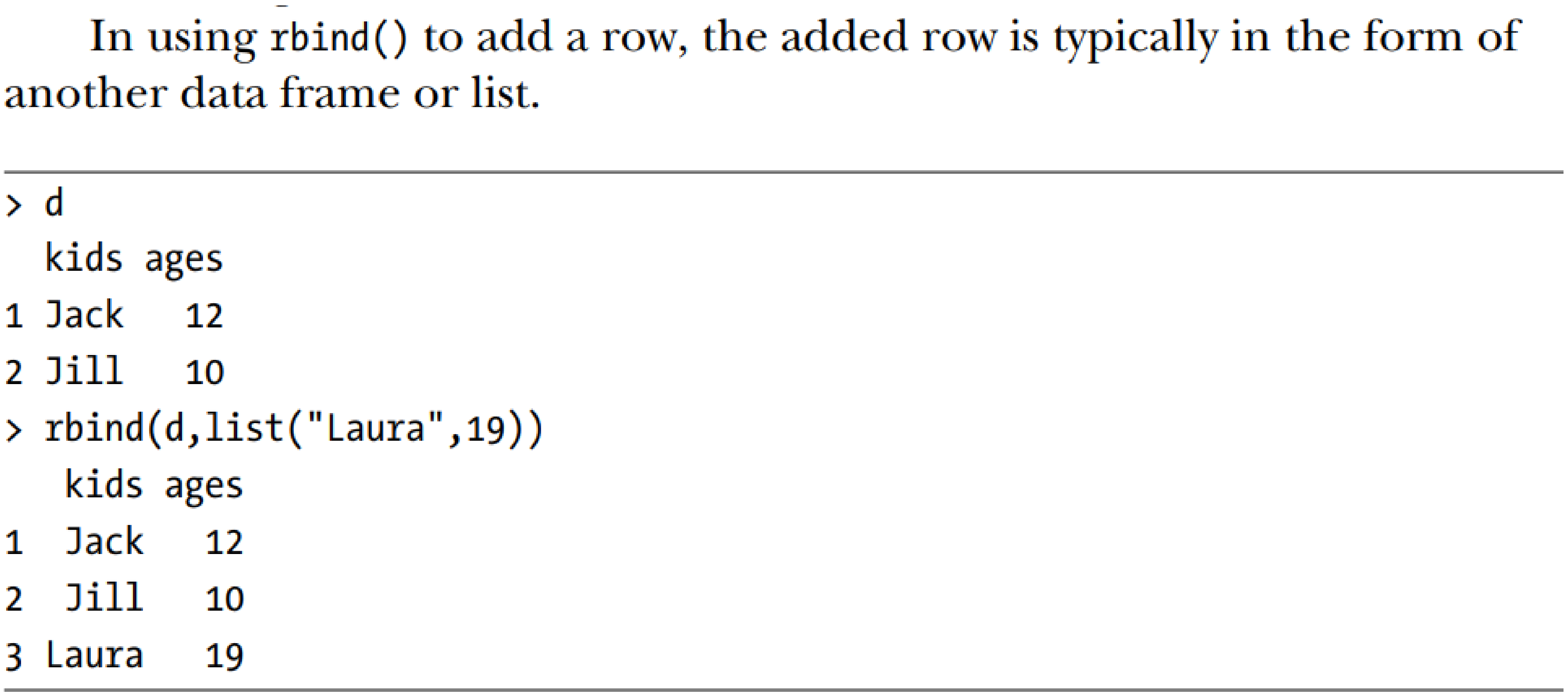
Extracting Subdata Frames

we can extract subdata frames by rows or columns.



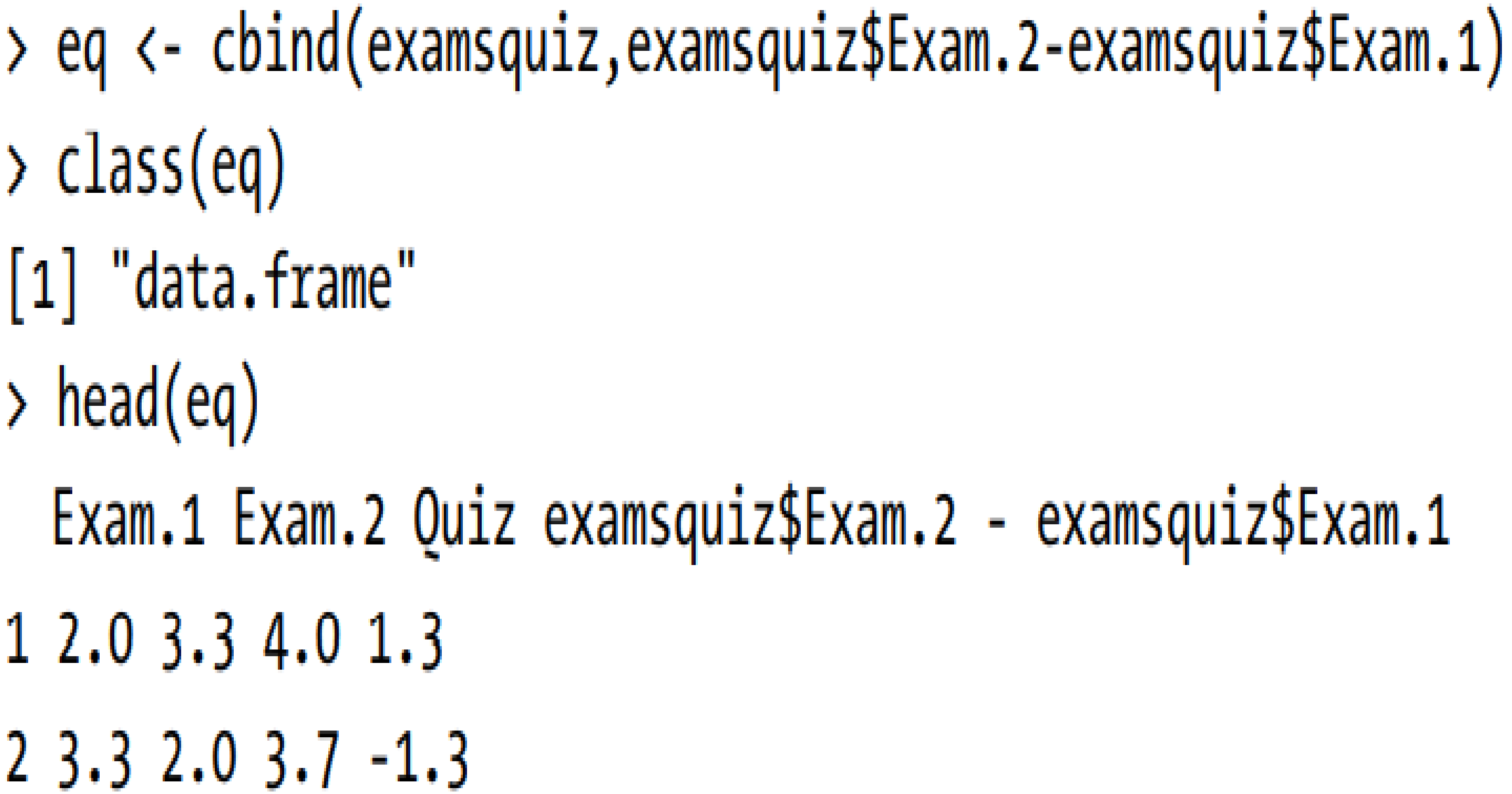
```
> examsquiz[2:5,]
  Exam.1 Exam.2 Quiz
2    3.3      2  3.7
3    4.0      4  4.0
4    2.3      0  3.3
5    2.3      1  3.3
> examsquiz[2:5,2]
[1] 2 4 0 1
> class(examsquiz[2:5,2])
[1] "numeric"
> examsquiz[2:5,2,drop=FALSE]
  Exam.2
2      2
3      4
4      0
5      1
> class(examsquiz[2:5,2,drop=FALSE])
[1] "data.frame"
```

Using the rbind() and cbind() Functions and Alternatives

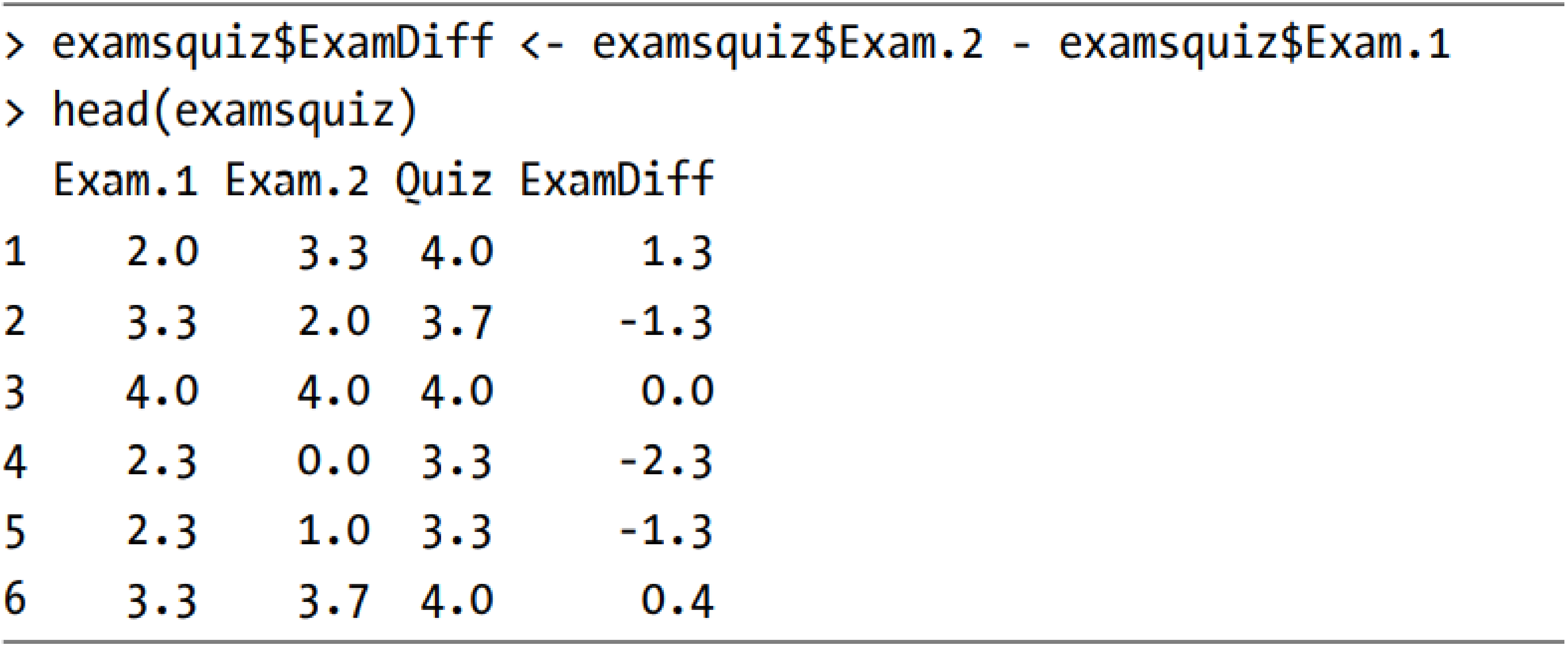


In using rbind() to add a row, the added row is typically in the form of another data frame or list.

```
> d
  kids ages
1 Jack   12
2 Jill   10
> rbind(d,list("Laura",19))
   kids ages
1  Jack   12
2  Jill   10
3 Laura   19
```



```
> eq <- cbind(examsquiz,examsquiz$Exam.2-examsquiz$Exam.1)
> class(eq)
[1] "data.frame"
> head(eq)
  Exam.1 Exam.2 Quiz examsquiz$Exam.2 - examsquiz$Exam.1
1 2.0 3.3 4.0 1.3
2 3.3 2.0 3.7 -1.3
```



```
> examsquiz$ExamDiff <- examsquiz$Exam.2 - examsquiz$Exam.1
> head(examsquiz)
  Exam.1 Exam.2 Quiz ExamDiff
1    2.0    3.3  4.0      1.3
2    3.3    2.0  3.7     -1.3
3    4.0    4.0  4.0      0.0
4    2.3    0.0  3.3     -2.3
5    2.3    1.0  3.3     -1.3
6    3.3    3.7  4.0      0.4
```

Merging Data Frames

- In the relational database world, one of the most important operations is that of a join, in which two tables can be combined according to the values of a common variable.

- In R, two data frames can be similarly combined using the merge() function.

- The simplest form is as follows:

```
merge(x,y)
```

```
> d1
     kids states
1    Jack     CA
2    Jill     MA
3 Jillian     MA
4    John     HI
> d2
  ages    kids
1   10    Jill
2    7 Lillian
3   12    Jack
> d <- merge(d1,d2)
> d
  kids states ages
1 Jack     CA   12
2 Jill     MA   10
```

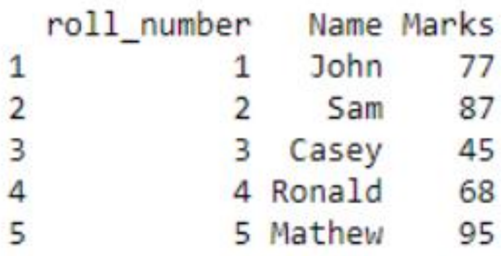
Here, the two data frames have the variable kids in common. R found the rows in which this variable had the same value of kids in both data frames (the ones for Jack and Jill). It then created a data frame with corresponding rows and with columns taken from data frames (kids, states, and ages).

The merge() function has named arguments by.x and by.y, which handle cases in which variables have similar information but different names in the two data frames. Here's an example:

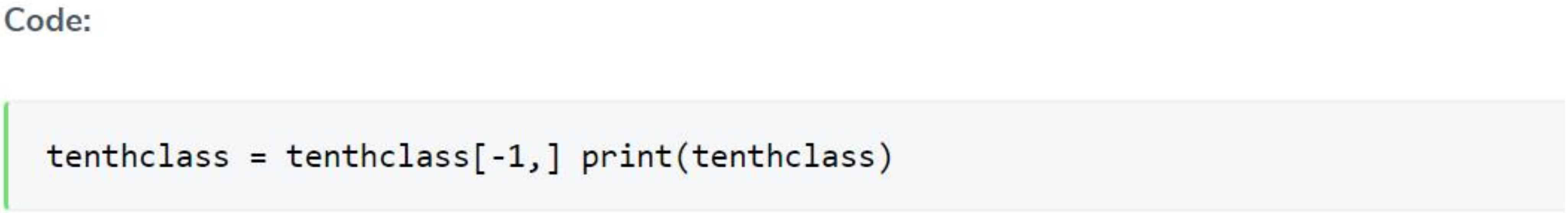
```
> d3
  ages    pals
1   12    Jack
2   10    Jill
3    7 Lillian
> merge(d1,d3,by.x="kids",by.y="pals")
  kids states ages
1 Jack     CA   12
2 Jill     MA   10
```

Even though our variable was called kids in one data frame and pals in the other, it was meant to store the same information, and thus the merge made sense.

Delete Row

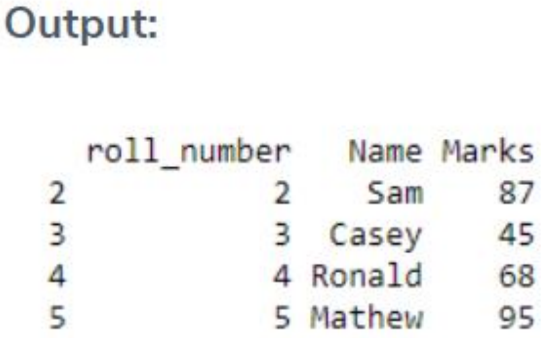


```
  roll_number   Name Marks
1           1   John    77
2           2    Sam    87
3           3  Casey    45
4           4 Ronald    68
5           5 Mathew    95
```



Code:

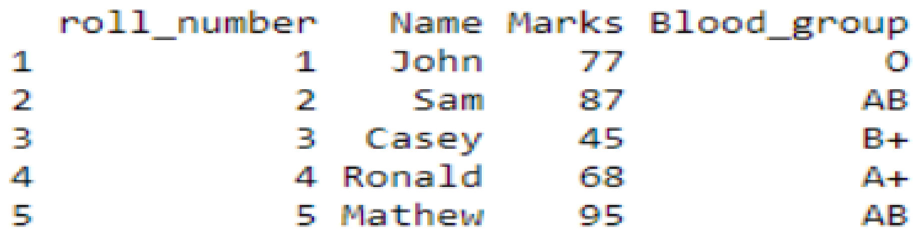
```
tenthclass = tenthclass[-1,] print(tenthclass)
```



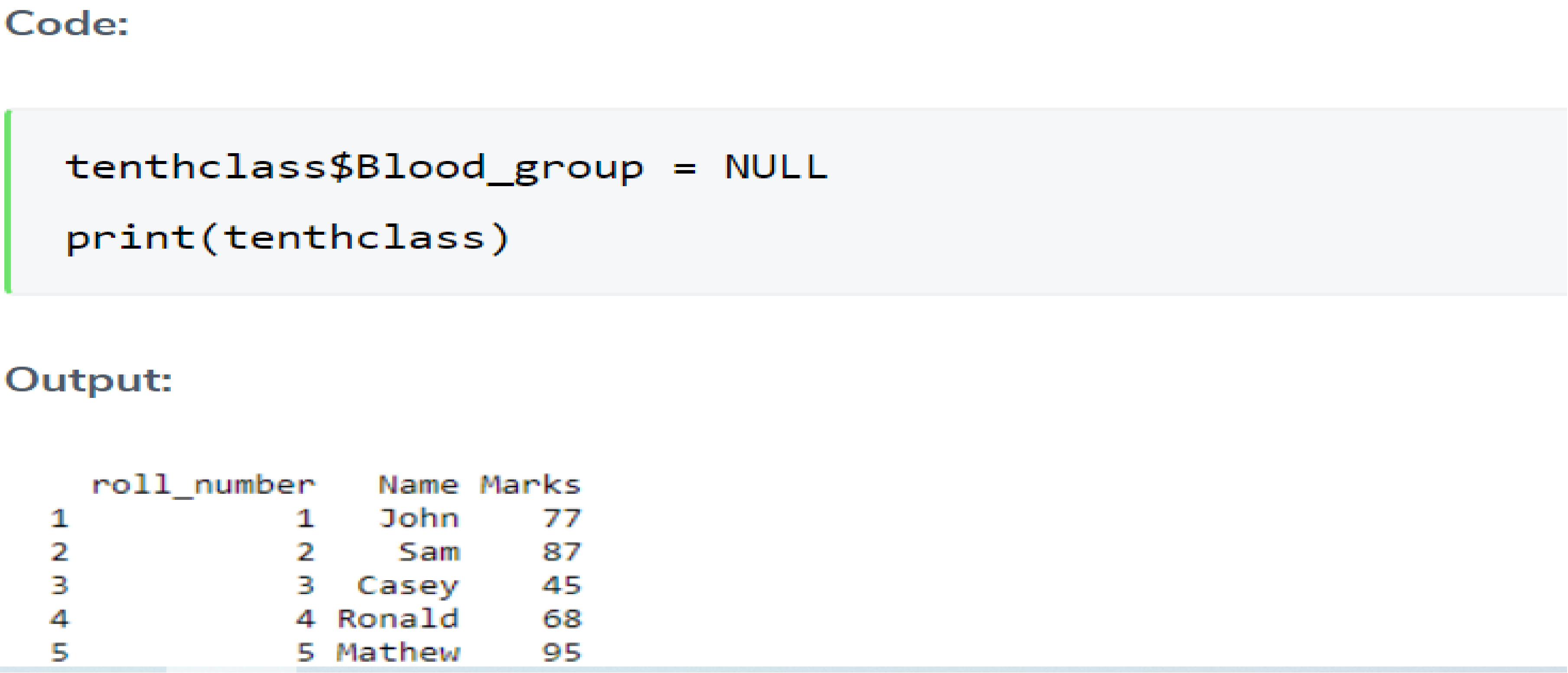
Output:

```
  roll_number   Name Marks
2           2    Sam    87
3           3  Casey    45
4           4 Ronald    68
5           5 Mathew    95
```

Delete Column



```
  roll_number   Name Marks Blood_group
1           1   John    77           O
2           2    Sam    87          AB
3           3  Casey    45          B+
4           4 Ronald    68          A+
5           5 Mathew    95          AB
```

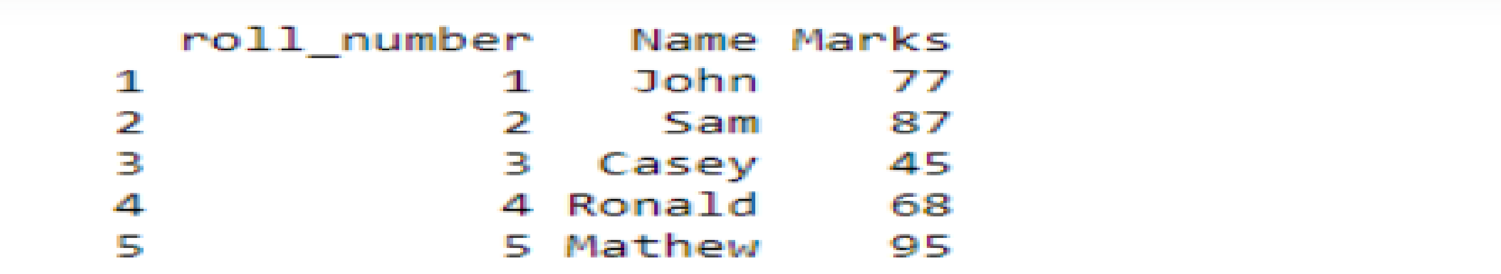


Code:

```
tenthclass$Blood_group = NULL
print(tenthclass)
```

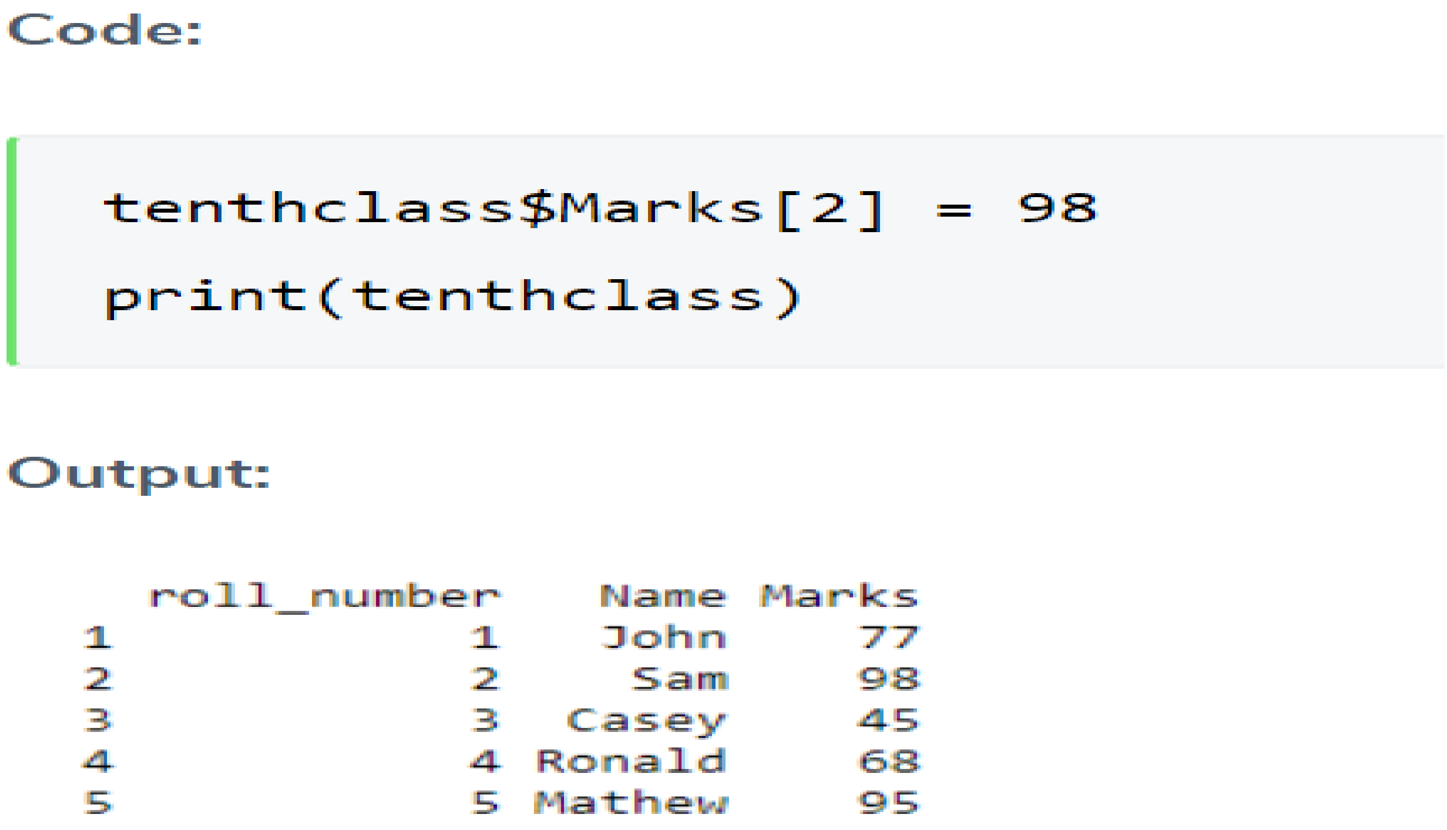
Output:

```
  roll_number   Name Marks
1           1   John    77
2           2    Sam    87
3           3  Casey    45
4           4 Ronald    68
5           5 Mathew    95
```



```
  roll_number   Name Marks
1           1   John    77
2           2    Sam    87
3           3  Casey    45
4           4 Ronald    68
5           5 Mathew    95
```

Update Data in Data Frame



Code:

```
tenthclass$Marks[2] = 98
print(tenthclass)
```

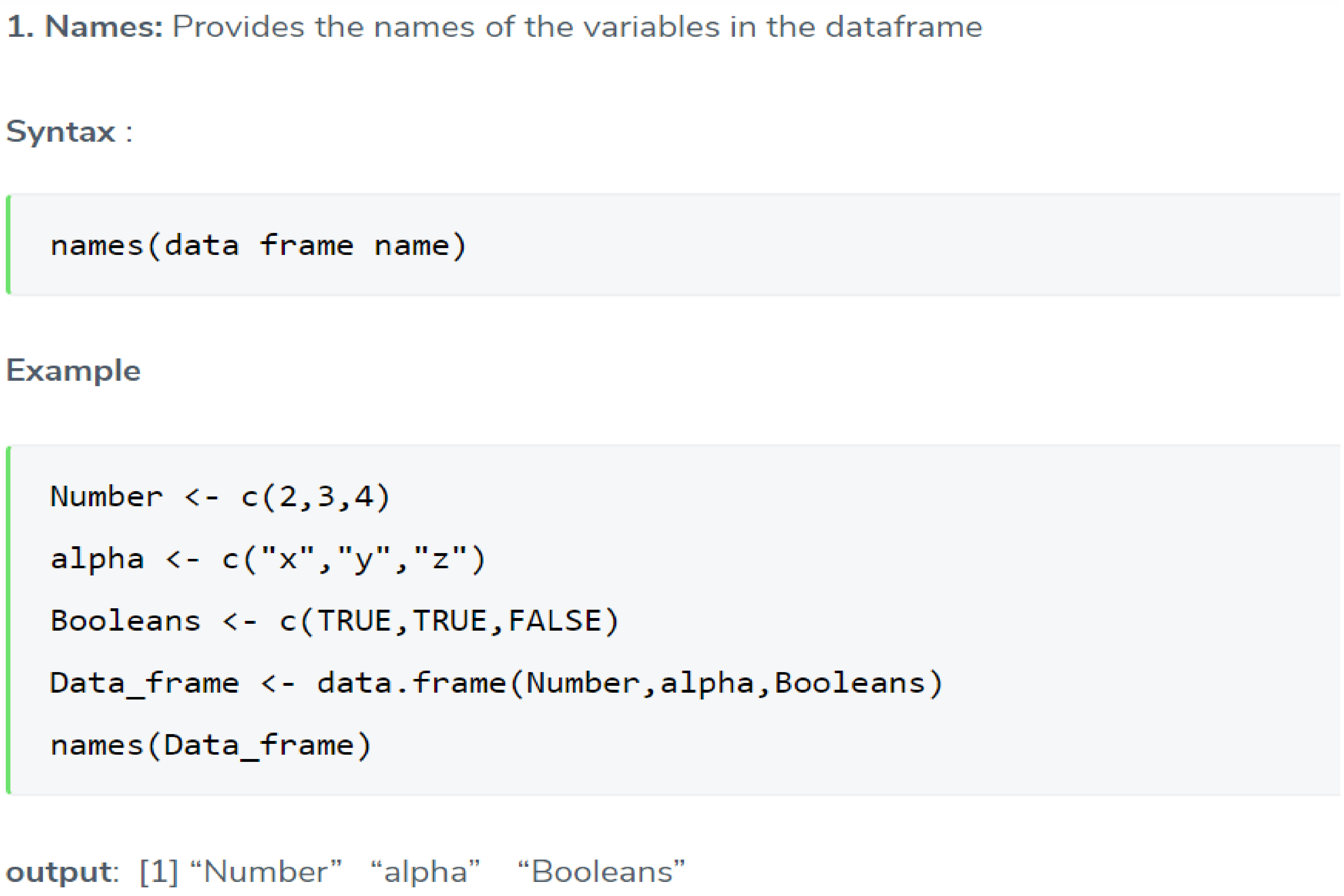
Output:

```
  roll_number   Name Marks
1           1   John    77
2           2    Sam    98
3           3  Casey    45
4           4 Ronald    68
5           5 Mathew    95
```

Applying Functions to Data Frames

```
> 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
```

Inspecting Data Frames



1. Names: Provides the names of the variables in the dataframe

Syntax :

```
names(data frame name)
```

Example

```
Number <- c(2,3,4)

alpha <- c("x","y","z")

Booleans <- c(TRUE,TRUE,FALSE)

Data_frame <- data.frame(Number,alpha,Booleans)

names(Data_frame)
```

output: [1] “Number” “alpha” “Booleans”

2. Summary: Provides the statistics of the data frame.

Syntax:

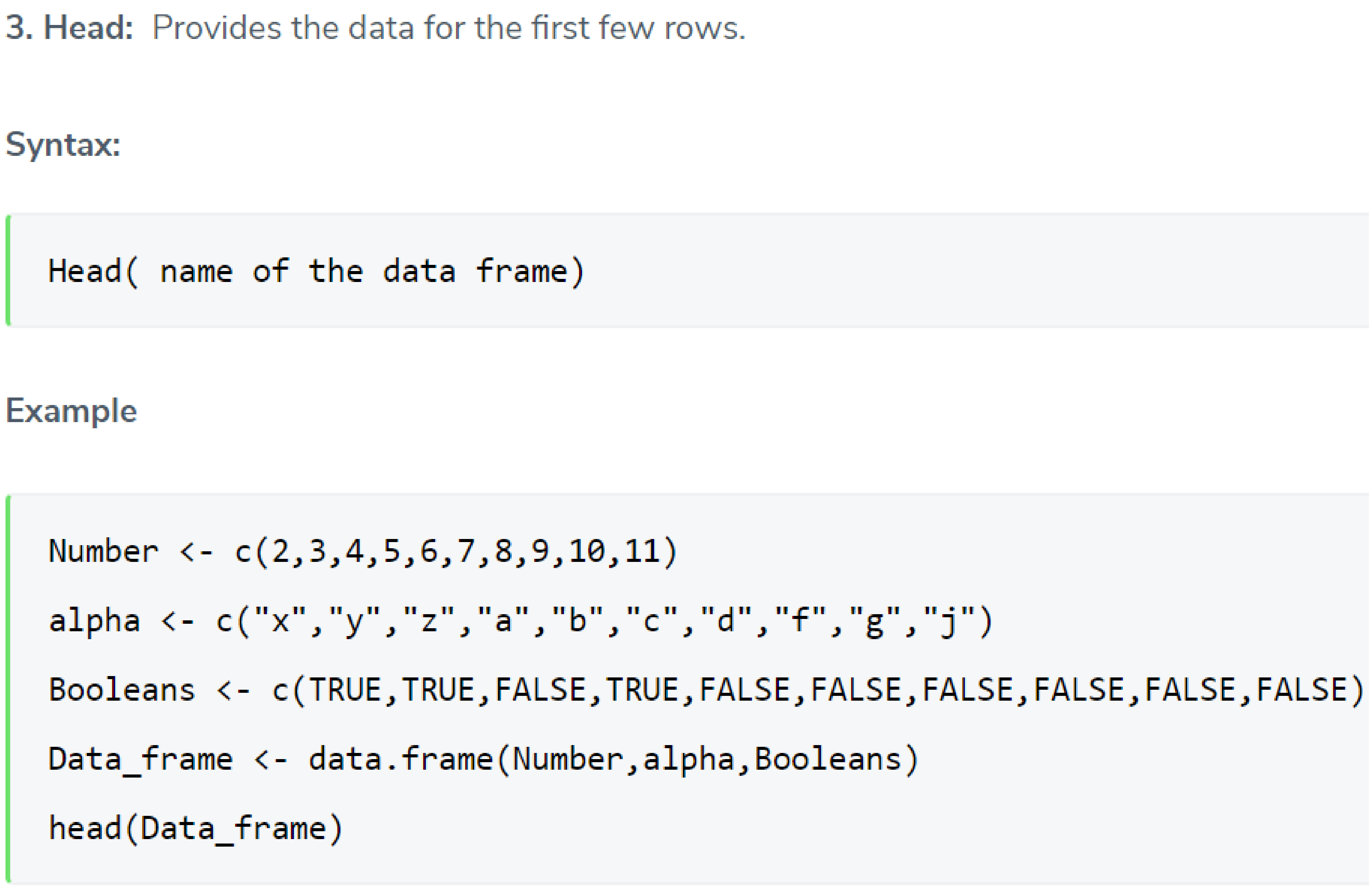
```
summary(data frame name)
```

Example

```
Number <- c(2,3,4)

alpha <- c("x","y","z")

Booleans <- c(TRUE,TRUE,FALSE)

Data_frame <- data.frame(Number,alpha,Booleans)

summary(Data_frame)
```

Output:

```
Number alpha Booleans

Min. :2.0 x:1 Mode :logical

1st Qu.:2.5 y:1 FALSE:1

Median :3.0 z:1 TRUE :2

Mean :3.0 NA’s :0

3rd Qu.:3.5

Max. :4.0
```



3. Head: Provides the data for the first few rows.

Syntax:

```
Head( name of the data frame)
```

Example

```
Number <- c(2,3,4,5,6,7,8,9,10,11)

alpha <- c("x","y","z","a","b","c","d","f","g","j")

Booleans <- c(TRUE,TRUE,FALSE,TRUE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE)

Data_frame <- data.frame(Number,alpha,Booleans)

head(Data_frame)
```

Output:

```
Number alpha Booleans
1 2 x TRUE
2 3 y TRUE
3 4 z FALSE
4 5 a TRUE
5 6 b FALSE
6 7 c FALSE
```

4. Tail: Prints the last few rows in the data frame.

Syntax:

```
tail( name of the data frame)
```

```
Number <- c(2,3,4,5,6,7,8,9,10,11)

alpha <- c("x","y","z","a","b","c","d","f","g","j")

Booleans <- c(TRUE,TRUE,FALSE,TRUE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE)

Data_frame <- data.frame(Number,alpha,Booleans)

tail(Data_frame)
```

Output:

```
Number alpha Booleans
5 6 b FALSE
6 7 c FALSE
7 8 d FALSE
8 9 f FALSE
9 10 g FALSE
10 11 j FALSE
```

FACTORS AND TABLES

- Factors are the data objects which are used to categorize the data and store it as levels.

- They can store both strings and integers.

- They are useful in the columns which have a limited number of unique values.

- Like "Male, "Female" and True, False etc.

- They are useful in data analysis for statistical modeling.

Attributes of Factors in R Language

- **x:** It is the vector that needs to be converted into a factor.
- **Levels:** It is a set of distinct values which are given to the input vector x.
- **Labels:** It is a character vector corresponding to the number of labels.
- **Exclude:** This will mention all the values you want to exclude.
- **Ordered:** This logical attribute decides whether the levels are ordered.

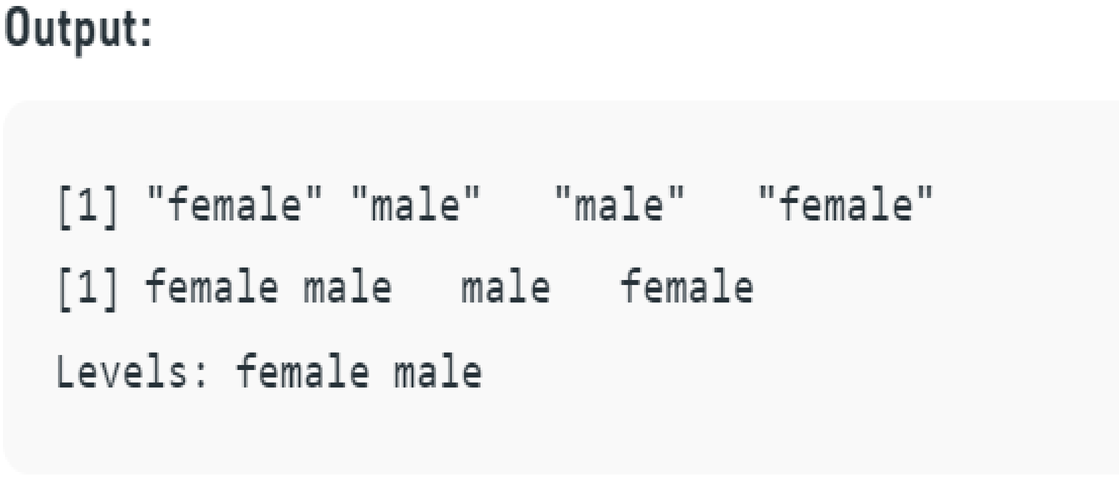
Creating a Factor in R Programming Language

The command used to create or modify a factor in R language is – **factor()** with a vector as input.
The two steps to creating a factor are:

- Creating a vector
- Converting the vector created into a factor using function factor()

Examples: Create a factor gender with levels female, male and transgender.

```
# Creating a vector
x < -c("female", "male", "male", "female")
print(x)
# Converting the vector x into a factor
# named gender
gender < -factor(x)
print(gender)
```



Output:

```
[1] "female" "male"   "male"   "female"
[1] female male   male   female
Levels: female male
```

Checking for a Factor in R

- The function **is.factor()** is used to check whether the variable is a factor and returns “TRUE” if it is a factor.
- **Example** :

```
gender <- factor(c("female", "male", "male", "female"));
print(is.factor(gender))
```

Output:

```
[1] TRUE
```

- Function **class()** is also used to check whether the variable is a factor and if true returns “factor”.

```
gender <- factor(c("female", "male", "male", "female"));
class(gender)
```

Output:

```
[1] "factor"
```

Accessing elements of a Factor in R

- Like we access elements of a vector, the same way we access the elements of a factor. If gender is a factor then gender[i] would mean accessing ith element in the factor.
- **Example:**

```
gender <- factor(c("female", "male", "male", "female"));

gender[3]
```

O/P

```
[1] male

Levels: female male
```

More than one element can be accessed at a time.

Example :

```
gender <- factor(c("female", "male", "male", "female"));
gender[c(2, 4)]
```

Output:

```
[1] male   female
Levels: female male
```

Example :

```
gender <- factor(c("female", "male", "male", "female" ));
gender[-3]
```

Output:

```
[1] female male   female
Levels: female male
```

Modification of a Factor in R

- After a factor is formed, its components can be modified but the new values which need to be assigned must be at the predefined level.

Example :

```
gender <- factor(c("female", "male", "male", "female" ));
gender[2]<-"female"
gender
```

Output:

```
[1] female female male   female
Levels: female male
```

- For selecting all the elements of the factor gender except ith element, gender[-i] should be used. So if you want to modify a factor and add value out of predefines levels, then first modify levels.

Example :

```
gender <- factor(c("female", "male", "male", "female" ));


# add new level
levels(gender) <- c(levels(gender), "other")
gender[3] <- "other"
Gender
```

Output:

```
[1] female male   other  female
Levels: female male other
```

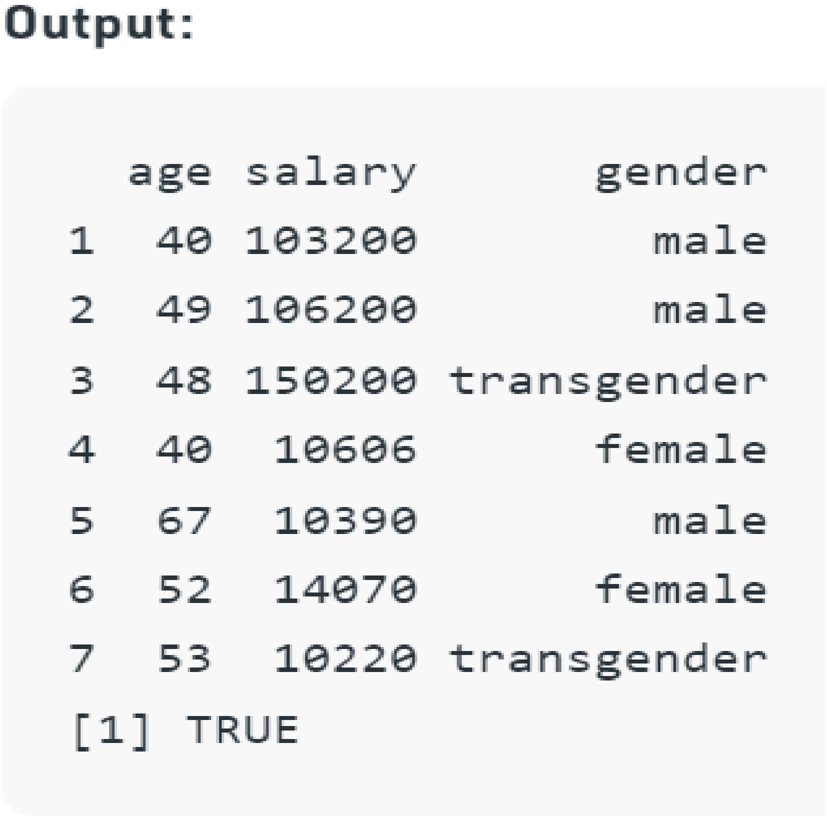
Factors in Data Frame

In R language when we create a data frame, its column is categorical data and hence a factor is automatically created on it.

We can create a data frame and check if its column is a factor.

Example:

```
age <- c(40, 49, 48, 40, 67, 52, 53)
salary <- c(103200, 106200, 150200,
        10606, 10390, 14070, 10220)
gender <- c("male", "male", "transgender",
        "female", "male", "female", "transgender")
employee<- data.frame(age, salary, gender)
print(employee)
print(is.factor(employee$gender))
```

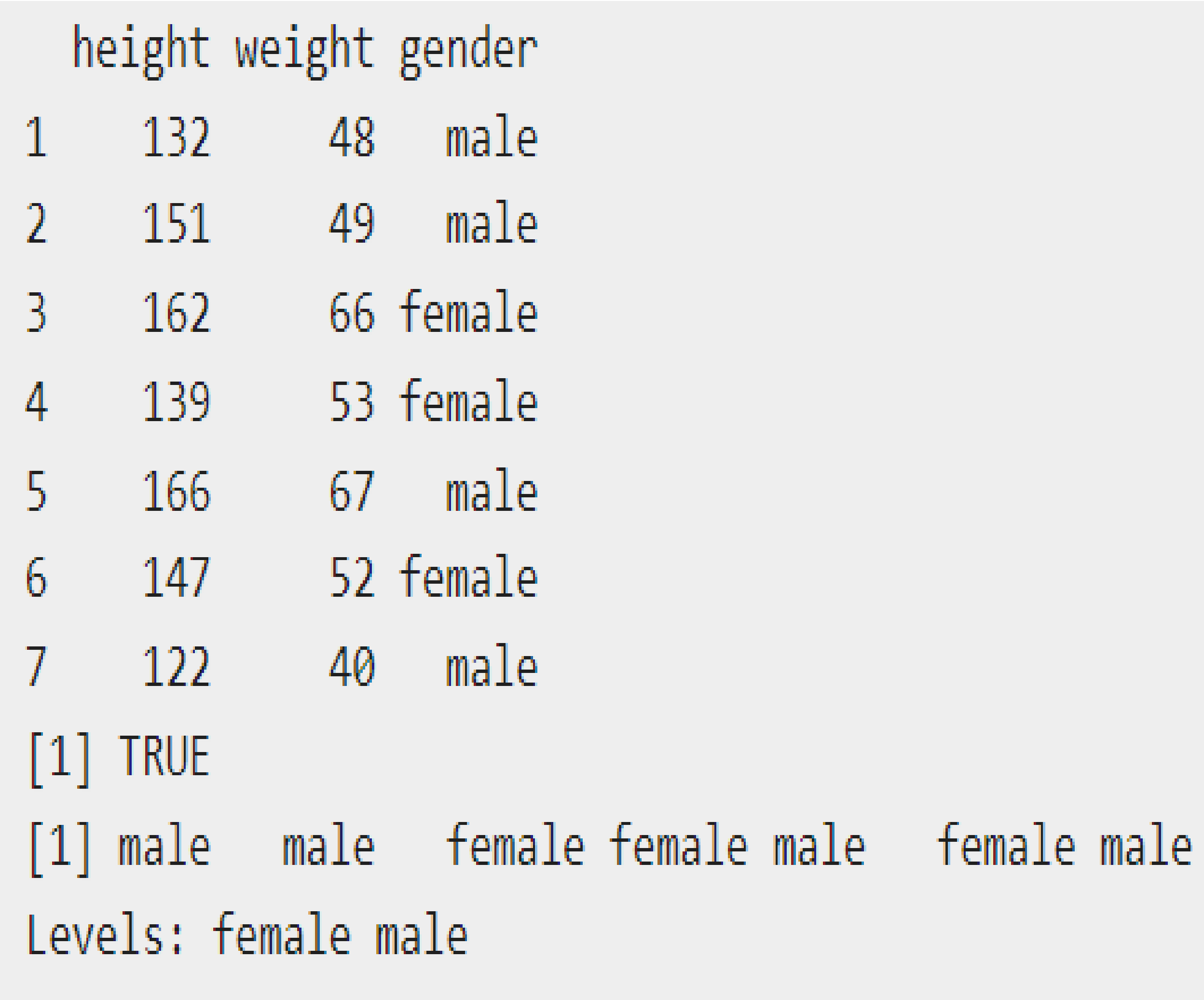


Output:

```
  age salary      gender
1  40 103200        male
2  49 106200        male
3  48 150200 transgender
4  40  10606      female
5  67  10390        male
6  52  14070      female
7  53  10220 transgender
[1] TRUE
```

Example:

```
# Create the vectors for data frame.
height <- c(132,151,162,139,166,147,122)
weight <- c(48,49,66,53,67,52,40)
gender <- c("male","male","female","female","male","female","male")

# Create the data frame.
input_data <- data.frame(height,weight,gender)
print(input_data)

# Test if the gender column is a factor.
print(is.factor(input_data$gender))
```



```
  height weight gender
1    132     48   male
2    151     49   male
3    162     66 female
4    139     53 female
5    166     67   male
6    147     52 female
7    122     40   male
[1] TRUE
[1] male   male   female female male   female male
Levels: female male
```

Changing the Order of Levels

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

Example :

```
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)
```

Output

```
[1] East  West  East  North North East  West  West  West  East  North
Levels: East North West
 [1] East  West  East  North North East  West  West  West  East  North
Levels: East West North
```

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.

Syntax

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

Following is the description of 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.

Example:

```
v <- gl(3, 4, labels = c("A", "B","C"))
print(v)
```

Common Functions Used with Factors

- tapply() function
- split() function
- by() function

➢ tapply() function

- **tapply()** computes a measure (mean, median, min, max, etc..) or a function for each factor variable in a vector.

- It is a very useful function that lets you create a subset of a vector and then apply some functions to each of the subset.

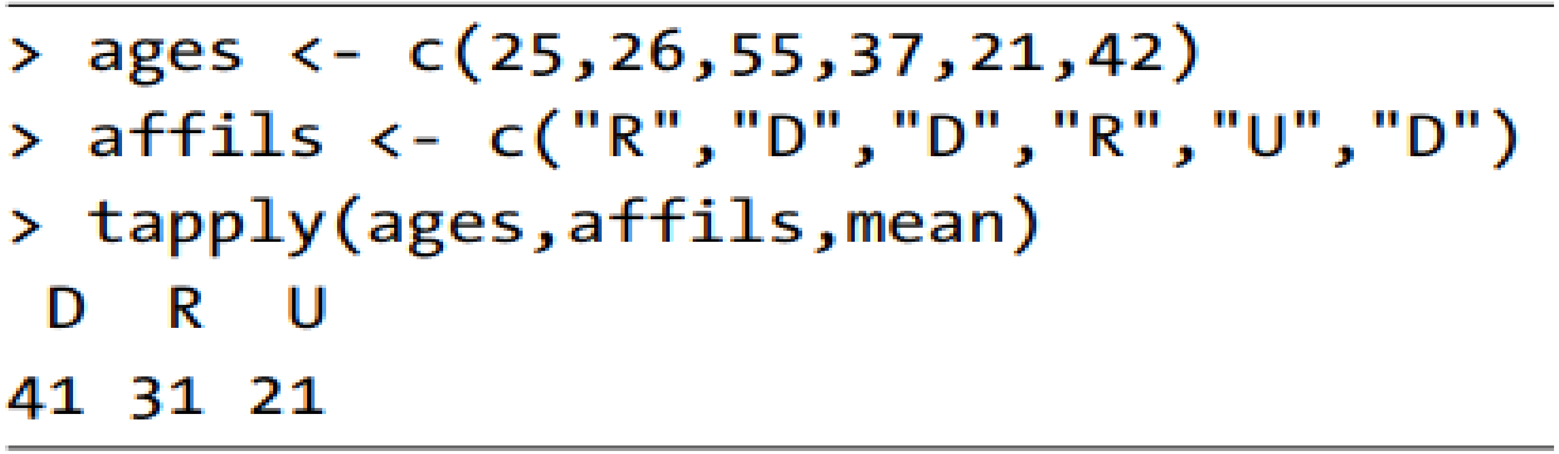
- **Syntax**: tapply(x,f,g)

Where,

x is a vector,

f is a factor or list of factors, and

g is a function



```
> ages <- c(25,26,55,37,21,42)
> affils <- c("R","D","D","R","U","D")
> tapply(ages,affils,mean)
 D  R  U
41 31 21
```

- The function tapply() treated the vector ("R","D","D","R","U","D") as a factor with levels "D", "R", and "U".
- It noted that "D" occurred in indices 2, 3 and 6; "R" occurred in indices 1 and 4; and "U" occurred in index 5.
- let's refer to the three index vectors (2,3,6), (1,4), and (5) as x, y, and z, respectively.
- Then tapply() computed mean(u[x]), mean(u[y]), and mean(u[z]) and returned those means in a three-element vector.
- And that vector's element names are "D", "R", and "U", reflecting the factor levels that were used by tapply().

➢ The split() Function

- In contrast to tapply(), which splits a vector into groups and then applies a specified function on each group.

- The basic form is split(x,f), with x and f playing roles similar to those in the call tapply(x,f,g); that is, x being a vector or data frame and f being a factor or a list of factors.

- The action is to split x into groups, which are returned in a list.

- Note that x is allowed to be a data frame with split() but not with tapply().

```
> d
  gender age income over25
1      M  47  55000      1
2      M  59  88000      1
3      F  21  32450      0
4      M  32  76500      1
5      F  33 123000      1
6      F  24  45650      0
> split(d$income,list(d$gender,d$over25))
$F.0
[1] 32450 45650

$M.0
numeric(0)

$F.1
[1] 123000

$M.1
[1] 55000 88000 76500
```

By() function

- The by() function in R is an easy function that allows us to group data within a data set, and perform mathematical functions on it.
- It takes a vector, a string, a matrix, or a data frame as input and computes that data based on the mentioned functions.
- The by() function takes the data as input and computes that based on a given function.

Syntax:

by(x,indices,FUN)

Where,

X = The input data frame.

Indices = It is the list of variables or the factors.

FUN = The function which needs to be applied for the variables/factors.

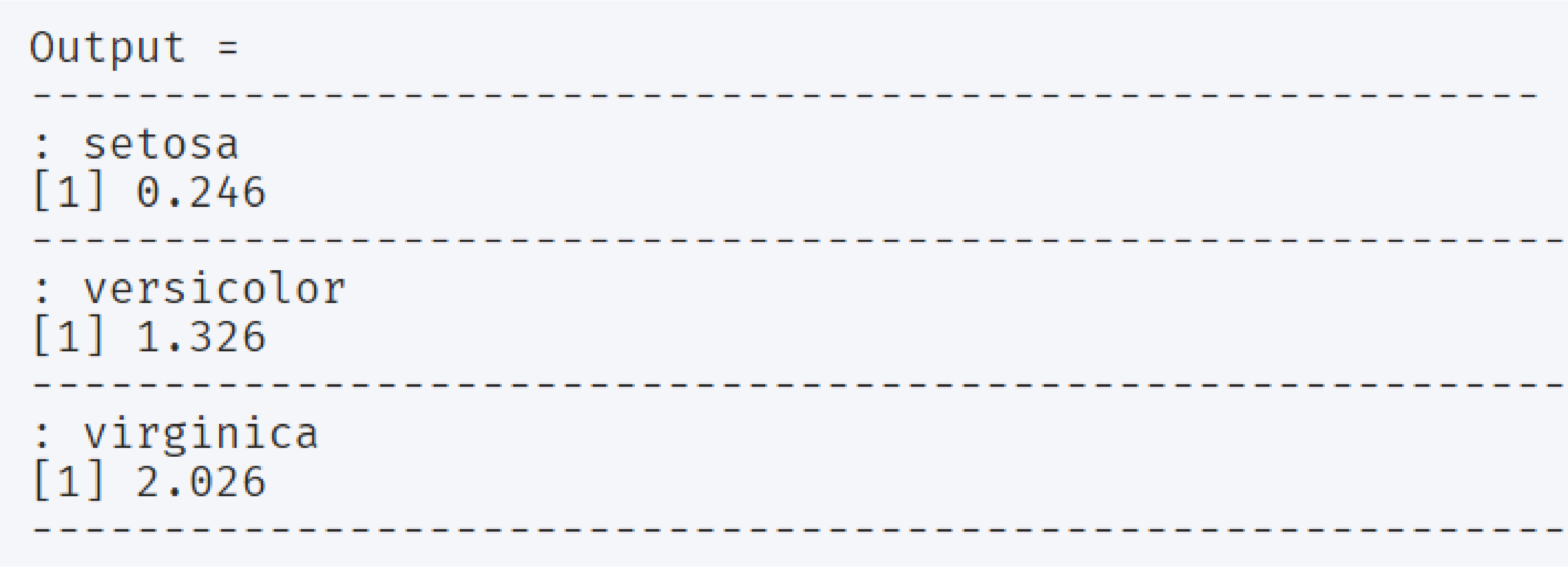
Exampls:

#importing data and assigning to variable df

df<-iris

#computes the mean for species categories in terms of petal.width

by(df$Petal.Width,list(df$Species),mean)



```
Output =
---------------------------------------------------------
: setosa
[1] 0.246
----------------------------------------------------------
: versicolor
[1] 1.326
----------------------------------------------------------
: virginica
[1] 2.026
----------------------------------------------------------
```