Unit 4 - Data Structures

Contents

1. Variables and Assignment

Variables

- A variable is an identifier (name) that points to a memory location in RAM that stores a value that can change when the program is run.

Assignment

- Putting a value into a variable is known as assignment.
- To set up a vector named x, say

```
> x <- c(10.4,4.5,3.7,7.6,121.0)
```

- This is an assignment statement using the function **c**() which in this context can take an arbitrary number of arguments and whose value is a vector got by concatenating its arguments end to end.
- A number occurring by itself in an expression is taken as a vector of length one.
- Notice that the assignment operator ('<-') which consists of two characters '<' ("less than") and '-' ("minus") occurring strictly side-by-side and it "points" to the object receiving the value of the expression. In most contexts the "=" operator can be used as an alternative.
- Assignment can also be made using the function **assign**(). An equivalent way of making the same assignment as above is with

```
> assign("x",c(10.4,4.5,3.7,7.6,121.0))
```

- Assignments can also be made in the other direction, using the change in the assignment operator. The same assignment could be made using

```
> c(10.4,4.5,3.7,7.6,121.0) -> x
```

- R uses to work with temporary variables in the functions, with only a return(). If you assign a variable in the Global environment or when you are using functions more than scripts, you can use
  - ➢ super-assignment operator: <<-
  - ➢ assign function: assign("b",value,envir=globalenv())

2. Data Types

- We can see how R stores and organizes data for the following types:
  1) Variable Types
  2) Tables

1.1. Variable Data Types

- There are several basic R data types that are of frequent occurrence in routine R calculations. Some of them are given below:

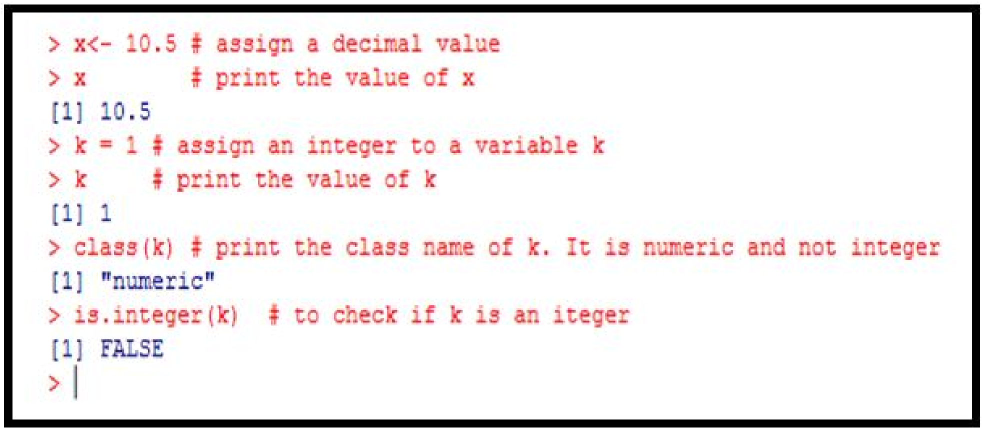
  a. Numeric
  b. Integer
  c. Complex

d. Logical
e. Character

- We will try to understand them better by direct experimentation with the R code.

a. Numeric

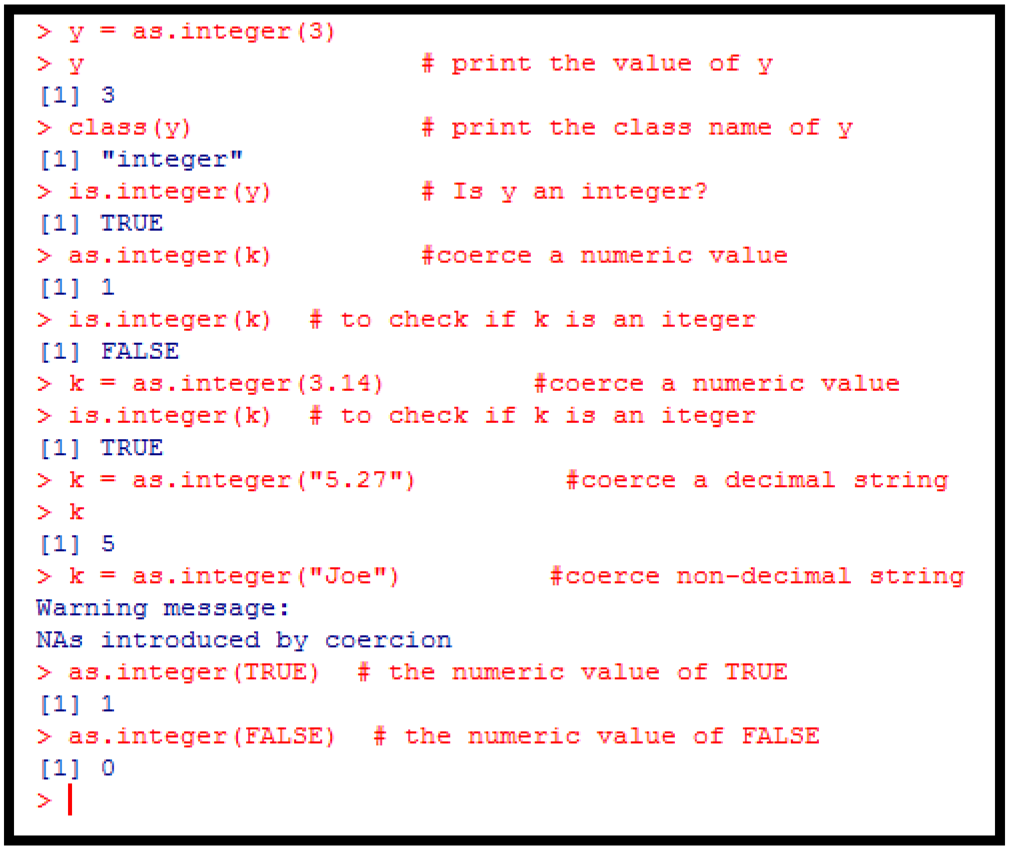
- Decimal values are called numerics in R. It is the default computational data type. If we assign a decimal value to a variable x as follows, x will be of numeric type.



```
> x<- 10.5 # assign a decimal value
> x        # print the value of x
[1] 10.5
> k = 1 # assign an integer to a variable k
> k     # print the value of k
[1] 1
> class(k) # print the class name of k. It is numeric and not integer
[1] "numeric"
> is.integer(k)  # to check if k is an iteger
[1] FALSE
> |
```

b. Integer

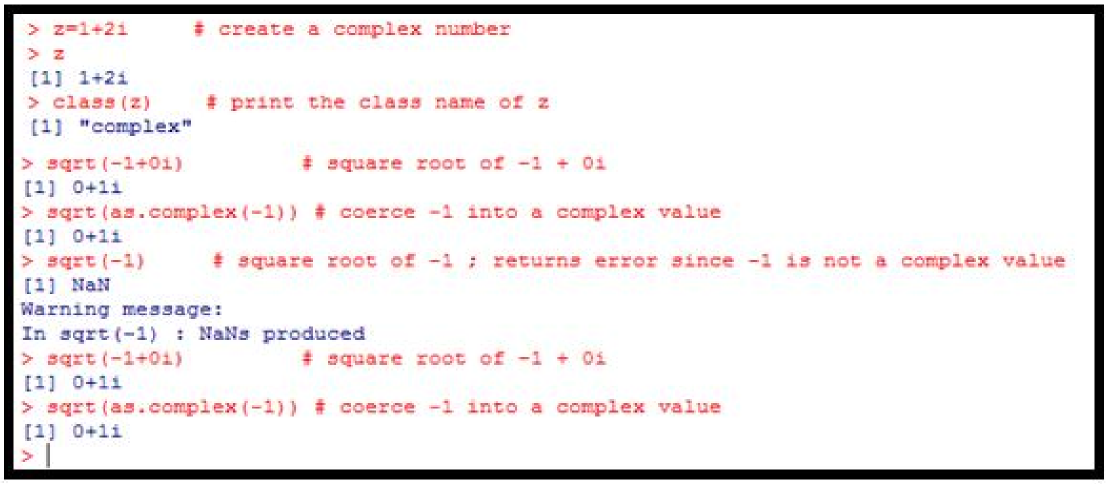
- In order to create an integer variable in R, we invoke the as.integer function.
- We can be assured that y is indeed an integer by applying the is.integer function.



```
> y = as.integer(3)
> y                        # print the value of y
[1] 3
> class(y)                 # print the class name of y
[1] "integer"
> is.integer(y)            # Is y an integer?
[1] TRUE
> as.integer(k)            #coerce a numeric value
[1] 1
> is.integer(k)   # to check if k is an iteger
[1] FALSE
> k = as.integer(3.14)            #coerce a numeric value
> is.integer(k)   # to check if k is an iteger
[1] TRUE
> k = as.integer("5.27")           #coerce a decimal string
> k
[1] 5
> k = as.integer("Joe")           #coerce non-decimal string
Warning message:
NAs introduced by coercion
> as.integer(TRUE)   # the numeric value of TRUE
[1] 1
> as.integer(FALSE)   # the numeric value of FALSE
[1] 0
> |
```

c. Complex

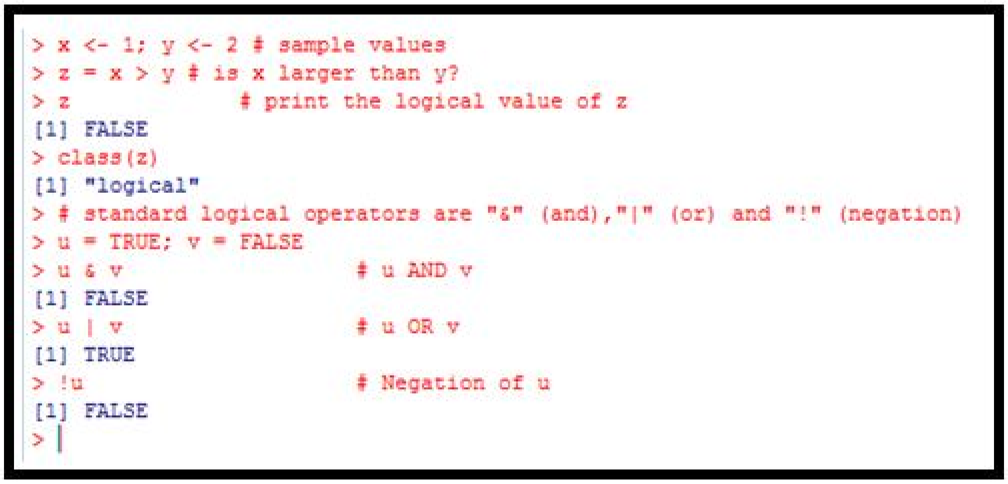
- A complex value in R is defined via the pure imaginary value i.



```
> z=1+2i      # create a complex number
> z
[1] 1+2i
> class(z)     # print the class name of z
[1] "complex"
> sqrt(-1+0i)          # square root of -1 + 0i
[1] 0+1i
> sqrt(as.complex(-1)) # coerce -1 into a complex value
[1] 0+1i
> sqrt(-1)       # square root of -1 ; returns error since -1 is not a complex value
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
> sqrt(-1+0i)          # square root of -1 + 0i
[1] 0+1i
> sqrt(as.complex(-1)) # coerce -1 into a complex value
[1] 0+1i
> |
```

d. Logical

- A logical value is often created via comparison between variables.



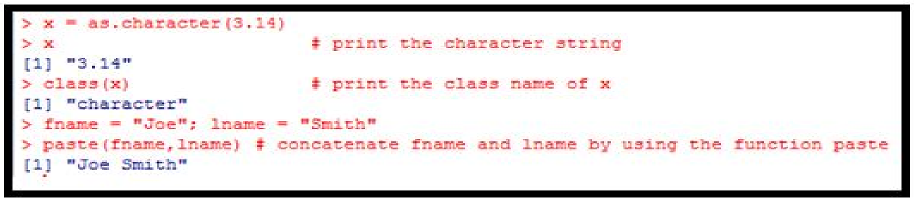
```
> x <- 1; y <- 2 # sample values
> z = x > y # is x larger than y?
> z           # print the logical value of z
[1] FALSE
> class(z)
[1] "logical"
> # standard logical operators are "&" (and),"|" (or) and "!" (negation)
> u = TRUE; v = FALSE
> u & v                # u AND v
[1] FALSE
> u | v                # u OR v
[1] TRUE
> !u                   # Negation of u
[1] FALSE
> |
```

Standard Logical Operators

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to
|	entry wise or
||	or
&	entry-wise and
&&	and
xor(a,b)	exclusive or

e. Character

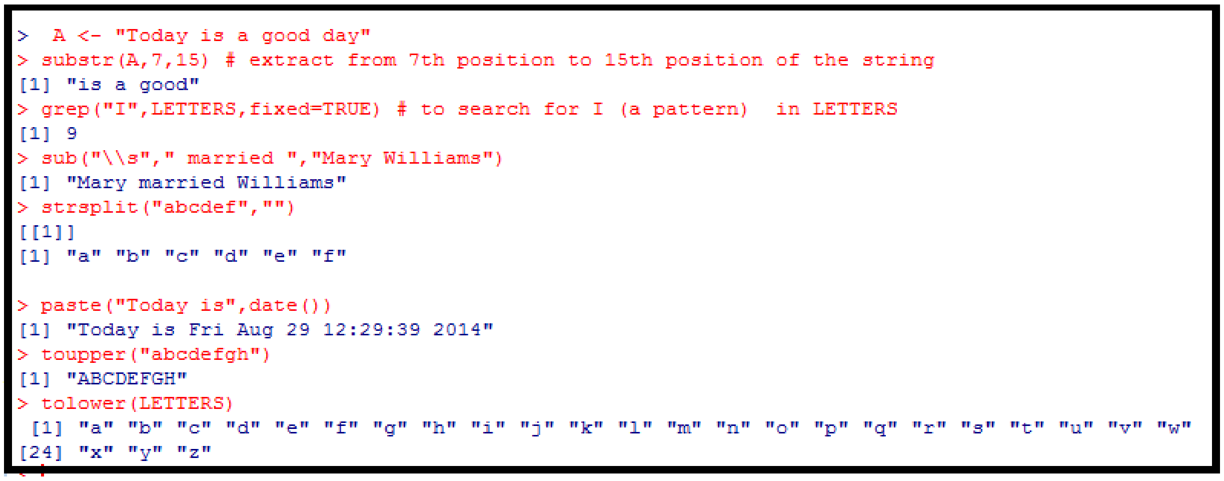
- A character object is used to represent string values in R. We convert objects into character values with the as.character() function:



```
> x = as.character(3.14)
> x                          # print the character string
[1] "3.14"
> class(x)                   # print the class name of x
[1] "character"
> fname = "Joe"; lname = "Smith"
> paste(fname,lname) # concatenate fname and lname by using the function paste
[1] "Joe Smith"
```

Character functions

#	Function	Description
1	substr(x, start=n1, stop=n2)	Extract or replace substrings in a character vector.
2	grep(pattern, x, ignore.case=FALSE, fixed=FALSE)	Search for *pattern* in *x*. If fixed =FALSE then *pattern* is a regular expression. If fixed=TRUE then *pattern* is a text string. Returns matching indices.
3	sub(pattern,replacement, x, ignore.case=FALSE, fixed=FALSE)	Find *pattern* in *x* and replace with *replacement* text. If fixed=FALSE then *pattern* is a regular expression. If fixed = T then *pattern* is a text string.
4	strsplit(x, split)	Split the elements of character vector x at split.
5	paste(..,sep="")	Concatenate strings after using sep string to separate them.
6	toupper(x)	Upper Case
7	tolower(x)	Lower Case



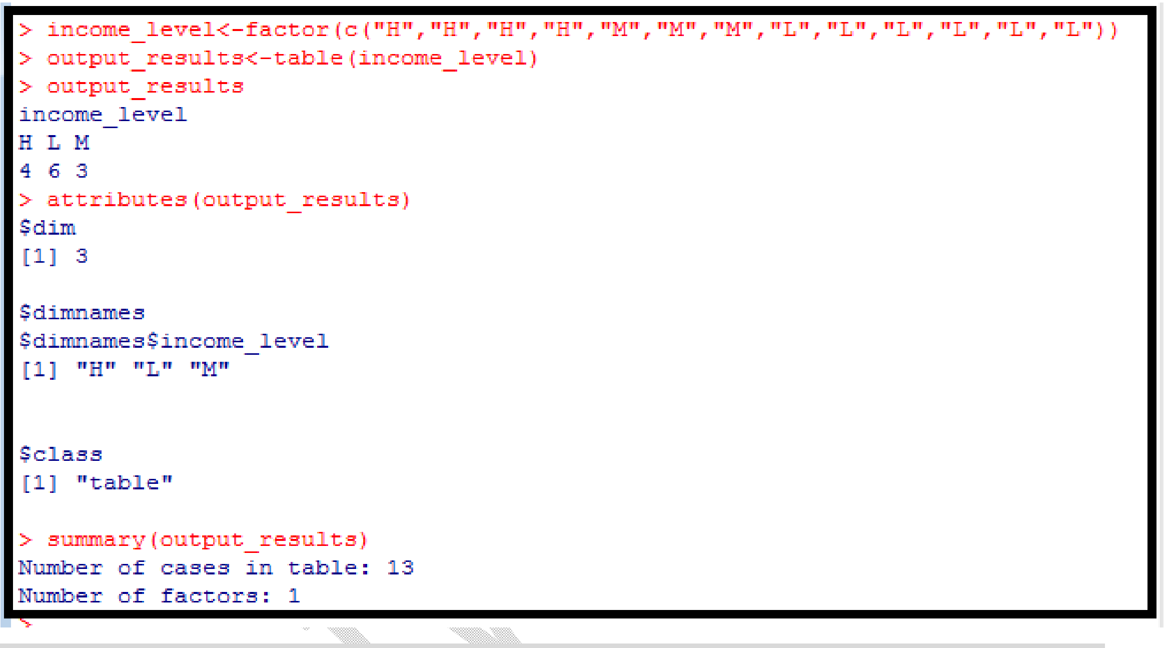
```
>  A <- "Today is a good day"
> substr(A,7,15) # extract from 7th position to 15th position of the string
[1] "is a good"
> grep("I",LETTERS,fixed=TRUE) # to search for I (a pattern)  in LETTERS
[1] 9
> sub("\\s"," married ","Mary Williams")
[1] "Mary married Williams"
> strsplit("abcdef","")
[[1]]
[1] "a" "b" "c" "d" "e" "f"

> paste("Today is",date())
[1] "Today is Fri Aug 29 12:29:39 2014"
> toupper("abcdefgh")
[1] "ABCDEFGH"
> tolower(LETTERS)
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w"
[24] "x" "y" "z"
```

1.2. Tables

- Table is a common way to store information in a table. Let us look at how to create and define tables.

One Way Table

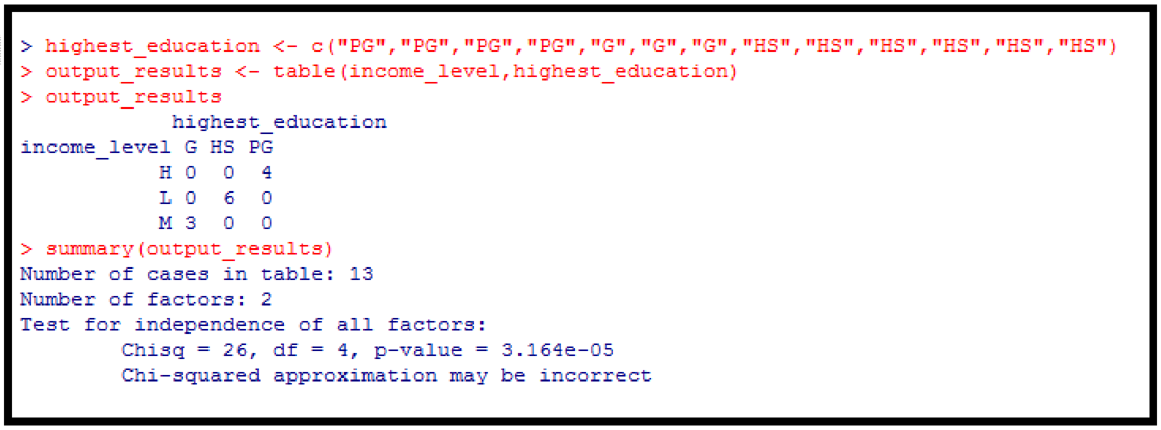
- One way tables are created using the table command with a vector of factors as arguments.
- It calculates the frequency that each factor counts. Let us look at the example of how to create a one way table.



```
> income_level<-factor(c("H","H","H","H","M","M","M","L","L","L","L","L","L"))
> output_results<-table(income_level)
> output_results
income_level
H L M
4 6 3
> attributes(output_results)
$dim
[1] 3

$dimnames
$dimnames$income_level
[1] "H" "L" "M"


$class
[1] "table"

> summary(output_results)
Number of cases in table: 13
Number of factors: 1
```

Two way tables

- If you add another vector, highest_education to the argument of the table command, you get a two way table.
- For example, we have values for Post-graduate as PG; Graduate as G; High-School as HS. Let us look at the example of how to create a two way table.



```
> highest_education <- c("PG","PG","PG","PG","G","G","G","HS","HS","HS","HS","HS","HS")
> output_results <- table(income_level,highest_education)
> output_results
            highest_education
income_level G HS PG
           H 0  0  4
           L 0  6  0
           M 3  0  0
> summary(output_results)
Number of cases in table: 13
Number of factors: 2
Test for independence of all factors:
        Chisq = 26, df = 4, p-value = 3.164e-05
        Chi-squared approximation may be incorrect
```

Fundamental Data objects

Data object	Description
Vector	a sequence of numbers or characters, or higher-dimensional arrays like matrices
list	a collection of objects that may themselves be complicated
factor	a sequence assigning a category to each index
data.frame	a table-like structure
Environment (hash-table)	A collection of key-value pairs

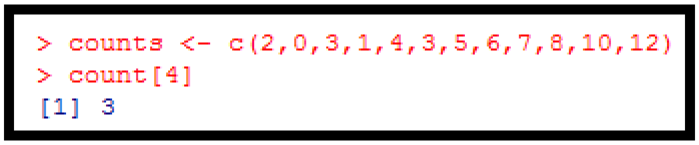
3. Indexing and Subsetting

Indexing

- R contains several constructs which allows access to individual elements or subsets through indexing operations. In case of the basic vector types one can access the ith element using x[i], but there is also indexing of lists, matrices, and multi-dimensional arrays. There are several forms of indexing in addition to indexing with a single integer.
- Indexing can be used both to extract part of an object and to replace parts of an object (or to add parts).
- R has three basic indexing operators, with syntax displayed by the following examples:
  - x[i]
  - x[i,j]
  - x[[i]]
  - x[[i,j]]
  - x$a
  - x$"a"

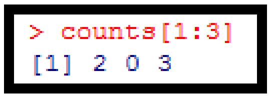
Subsetting vectors

- A subset from a vector may be obtained by appending an index within square brackets to the end of a symbol name.
- Assume we have a vector, counts containing the numbers: 2, 0, 3, 1, 4, 3, 5, 6, 7, 8, 10 and 12.
- To extract fourth of the counts by specifying the index 4, we use the following code in R:



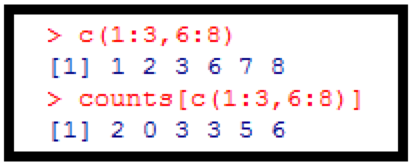
```
> counts <- c(2,0,3,1,4,3,5,6,7,8,10,12)
> count[4]
[1] 3
```

- The index can be a vector of any length.



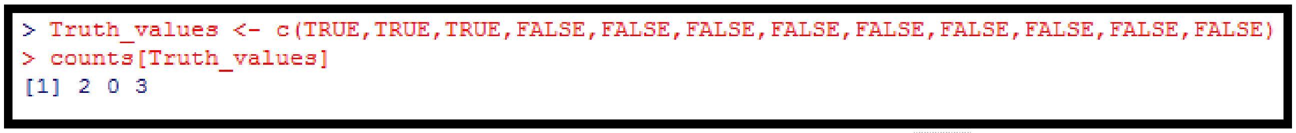
```
> counts[1:3]
[1] 2 0 3
```

- The index does not have to be a contagious sequence, and it can include repetitions.



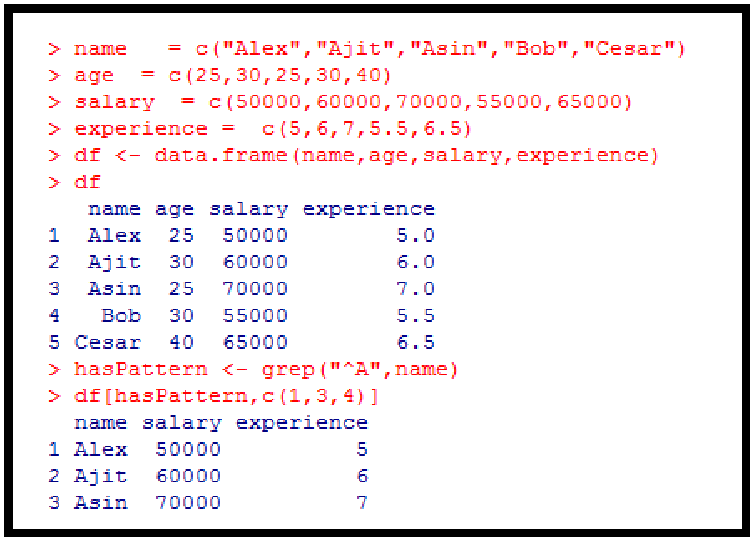
```
> c(1:3,6:8)
[1] 1 2 3 6 7 8
> counts[c(1:3,6:8)]
[1] 2 0 3 3 5 6
```

- For indices, we can use logical values.



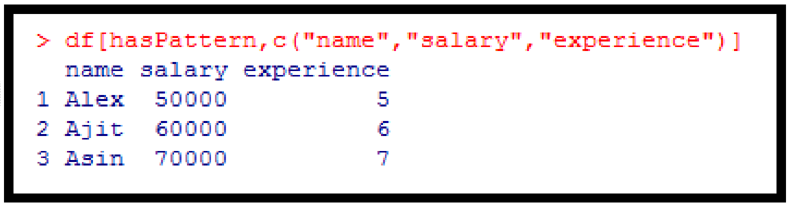
```
> Truth_values <- c(TRUE,TRUE,TRUE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE)
> counts[Truth_values]
[1] 2 0 3
```

- A data frame can also be indexed using square brackets, though slightly differently because we have to specify both which rows and which columns we want.



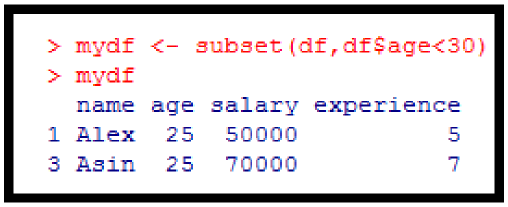
```
> name   = c("Alex","Ajit","Asin","Bob","Cesar")
> age  = c(25,30,25,30,40)
> salary  = c(50000,60000,70000,55000,65000)
> experience =  c(5,6,7,5.5,6.5)
> df <- data.frame(name,age,salary,experience)
> df
   name age salary experience
1  Alex  25  50000        5.0
2  Ajit  30  60000        6.0
3  Asin  25  70000        7.0
4   Bob  30  55000        5.5
5 Cesar  40  65000        6.5
> hasPattern <- grep("^A",name)
> df[hasPattern,c(1,3,4)]
  name salary experience
1 Alex  50000          5
2 Ajit  60000          6
3 Asin  70000          7
```

- To select this subset is to use named components for appropriate columns.



```
> df[hasPattern,c("name","salary","experience")]
  name salary experience
1 Alex  50000          5
2 Ajit  60000          6
3 Asin  70000          7
```

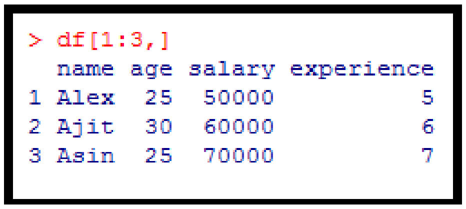
- The function subset() provides another way to subset a data frame. This function has a subset argument for specifying the rows and a select argument for specifying the columns.



```
> mydf <- subset(df,df$age<30)
> mydf
  name age salary experience
1 Alex  25  50000          5
3 Asin  25  70000          7
```

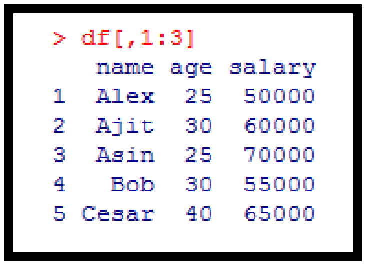
- When subsetting using square brackets it is possible to leave the row or column index completely empty. The result is that all rows or all columns, respectively are returned.

- For example, to get the first three rows and all the columns in the data frame df, we use the following code:



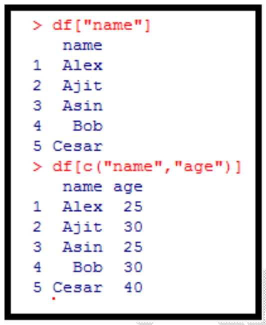
```
> df[1:3,]
  name age salary experience
1 Alex  25  50000          5
2 Ajit  30  60000          6
3 Asin  25  70000          7
```

- For example, to get the first three columns and all the rows in the data frame df, we use the following code:



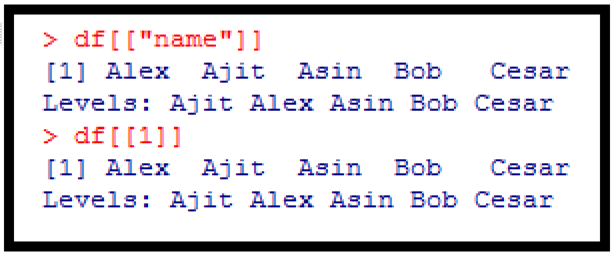
```
> df[,1:3]
   name age salary
1  Alex  25  50000
2  Ajit  30  60000
3  Asin  25  70000
4   Bob  30  55000
5 Cesar  40  65000
```

- If a single index is specified when subsetting a data frame with square brackets, the effect is to extract the appropriate columns of the data frame and all rows are returned.



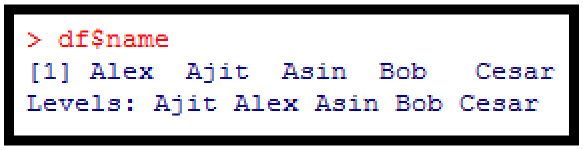
```
> df["name"]
   name
1  Alex
2  Ajit
3  Asin
4   Bob
5 Cesar
> df[c("name","age")]
   name age
1  Alex  25
2  Ajit  30
3  Asin  25
4   Bob  30
5 Cesar  40
```

- To get just the vector representing the values in the variable, we use double square brackets. The values for the variables can be column name or a number.



```
> df[["name"]]
[1] Alex  Ajit  Asin  Bob   Cesar
Levels: Ajit Alex Asin Bob Cesar
> df[[1]]
[1] Alex  Ajit  Asin  Bob   Cesar
Levels: Ajit Alex Asin Bob Cesar
```

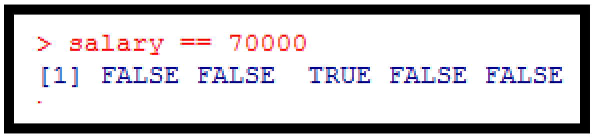
- However, with double square bracket subsetting, the index must be a single value.
- We can also use $ for getting a single variable from a data set.



```
> df$name
[1] Alex  Ajit  Asin  Bob   Cesar
Levels: Ajit Alex Asin Bob Cesar
```

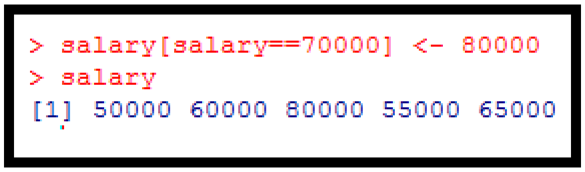
Assigning to a subset

- As with extracting subsets, the index can be a numeric vector, a character, or a logical vector. In this case, we will first develop an expression that generates a logical vector telling us where the condition is met.



```
> salary == 70000
[1] FALSE FALSE  TRUE FALSE FALSE
```

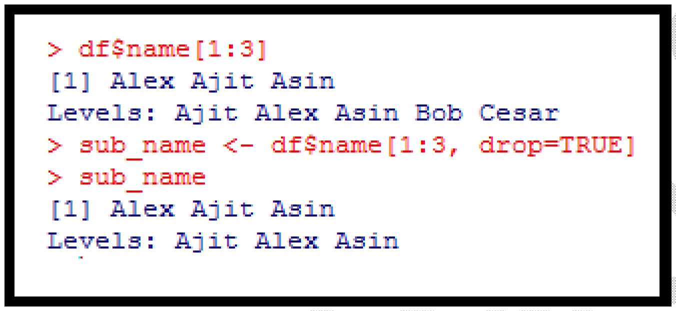
- We shall replace the amount 70000 with 80000.



```
> salary[salary==70000] <- 80000
> salary
[1] 50000 60000 80000 55000 65000
```

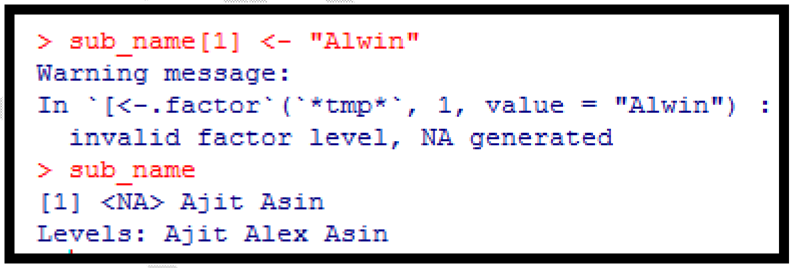
Subsetting factors

- When we subset a factor, the levels of a factor are not altered,



```
> df$name[1:3]
[1] Alex Ajit Asin
Levels: Ajit Alex Asin Bob Cesar
> sub_name <- df$name[1:3, drop=TRUE]
> sub_name
[1] Alex Ajit Asin
Levels: Ajit Alex Asin
```

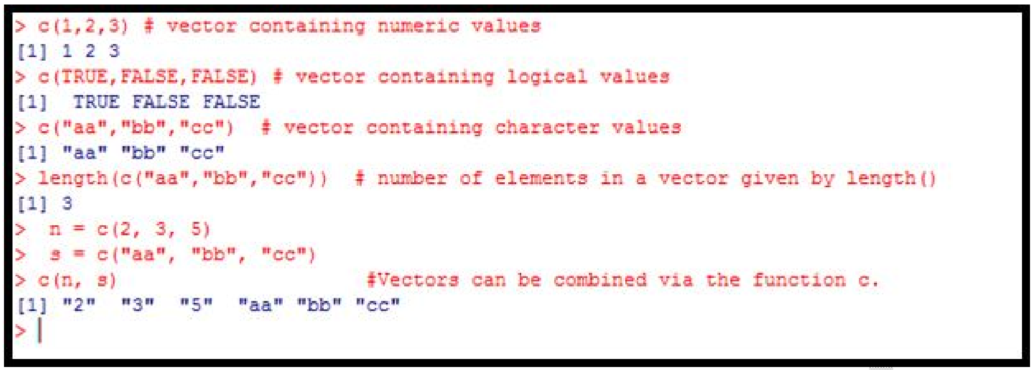
- We have seen how to force the unused levels to be dropped by specifying drop= TRUE within square brackets.
- Assuming values to a subset a factor is also a special case because only the current levels of the factor are allowed. A missing value is generated if the new value is not one of the current factor levels (and a warning is displayed).



```
> sub_name[1] <- "Alwin"
Warning message:
In `[<-.factor`(`*tmp*`, 1, value = "Alwin") :
  invalid factor level, NA generated
> sub_name
[1] <NA> Ajit Asin
Levels: Ajit Alex Asin
```

4. Vector

- A vector is a sequence of data elements of the same basic type.
- Members in a vector are also called components.

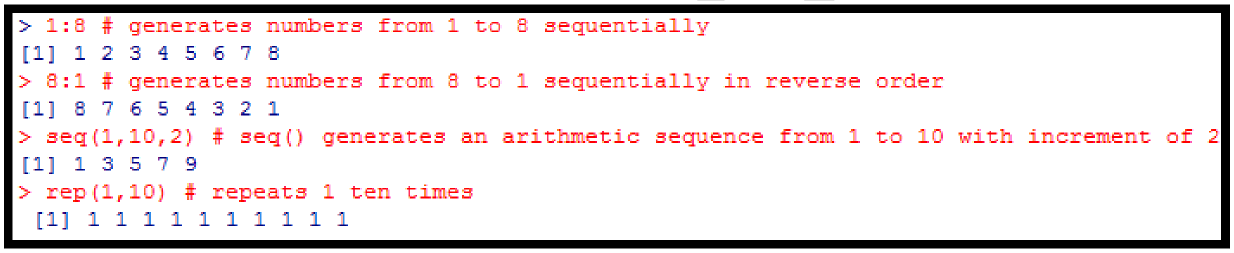


```
> c(1,2,3) # vector containing numeric values
[1] 1 2 3
> c(TRUE,FALSE,FALSE) # vector containing logical values
[1]  TRUE FALSE FALSE
> c("aa","bb","cc")  # vector containing character values
[1] "aa" "bb" "cc"
> length(c("aa","bb","cc"))  # number of elements in a vector given by length()
[1] 3
>  n = c(2, 3, 5)
>  s = c("aa", "bb", "cc")
> c(n, s)                    #Vectors can be combined via the function c.
[1] "2"  "3"  "5"  "aa" "bb" "cc"
> |
```

Value Coercion

- In the code snippet above, notice how the numeric values are being coerced into character strings when the two vectors are combined. This is necessary so as to maintain the same primitive data type for members in the same vector.

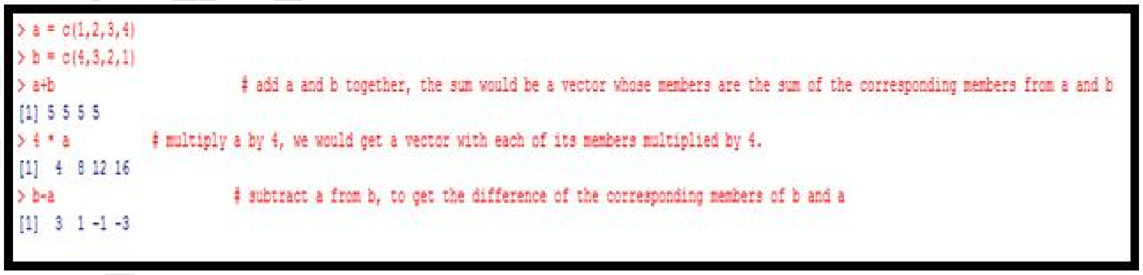
Generating vectors with ":", seq() and rep()



```
> 1:8 # generates numbers from 1 to 8 sequentially
[1] 1 2 3 4 5 6 7 8
> 8:1 # generates numbers from 8 to 1 sequentially in reverse order
[1] 8 7 6 5 4 3 2 1
> seq(1,10,2) # seq() generates an arithmetic sequence from 1 to 10 with increment of 2
[1] 1 3 5 7 9
> rep(1,10) # repeats 1 ten times
 [1] 1 1 1 1 1 1 1 1 1 1
```

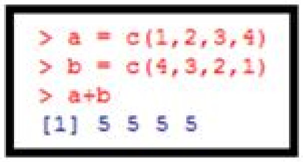
Vector Arithmetic

- Arithmetic operations of vectors are performed member-by-member, i.e., member-wise.

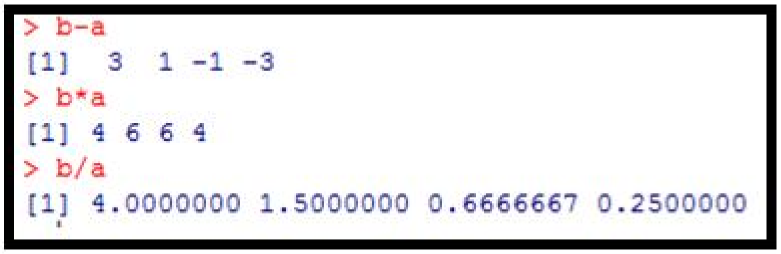


```
> a = c(1,2,3,4)
> b = c(4,3,2,1)
> a+b                # add a and b together, the sum would be a vector whose members are the sum of the corresponding members from a and b
[1] 5 5 5 5
> 4 * a          # multiply a by 4, we would get a vector with each of its members multiplied by 4.
[1]  4  8 12 16
> b-a                # subtract a from b, to get the difference of the corresponding members of b and a
[1]  3  1 -1 -3
```

- If we add two vectors a and b together, the sum would be a vector whose members are the sum of the corresponding members from a and b.
- Similarly for subtraction, multiplication and division, we get new vectors via member-wise operations.



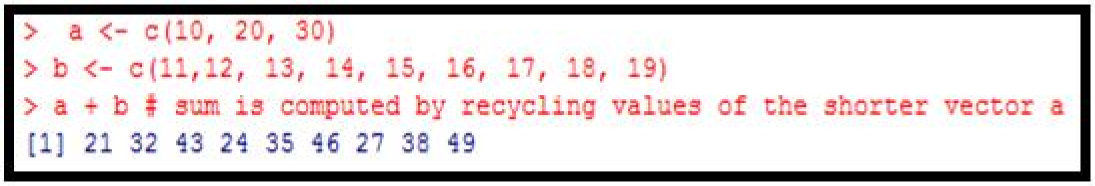
```
> a = c(1,2,3,4)
> b = c(4,3,2,1)
> a+b
[1] 5 5 5 5
```



```
> b-a
[1]  3  1 -1 -3
> b*a
[1] 4 6 6 4
> b/a
[1] 4.0000000 1.5000000 0.6666667 0.2500000
```

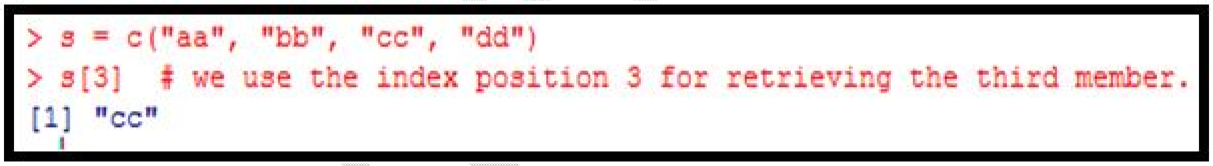
Recycling Rule

- If two vectors are of unequal length, the shorter one will be recycled in order to match the longer vector.



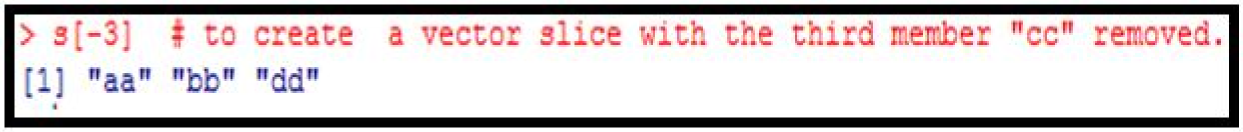
```
>  a <- c(10, 20, 30)
> b <- c(11,12, 13, 14, 15, 16, 17, 18, 19)
> a + b # sum is computed by recycling values of the shorter vector a
[1] 21 32 43 24 35 46 27 38 49
```

- We retrieve values in a vector by declaring an index inside a single square bracket "[]" operator.



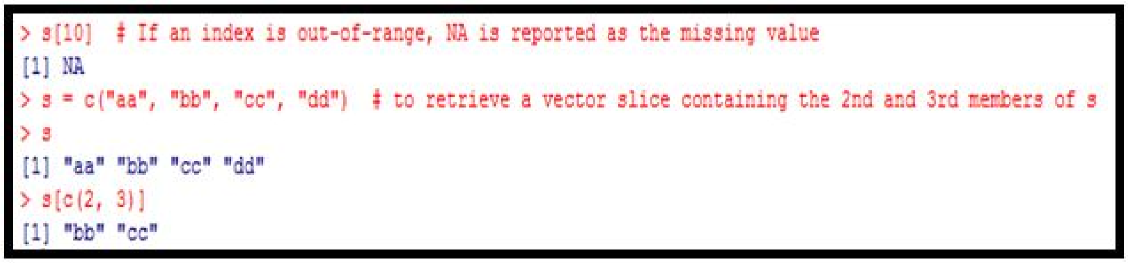
```
> s = c("aa", "bb", "cc", "dd")
> s[3]  # we use the index position 3 for retrieving the third member.
[1] "cc"
```

- Unlike other programming languages, the square bracket operator returns more than just individual members. In fact, the result of the square bracket operator is another vector, and s[3] is a vector slice containing a single member "cc".
- Negative Index would strip the member whose position has the same absolute value as the negative index.



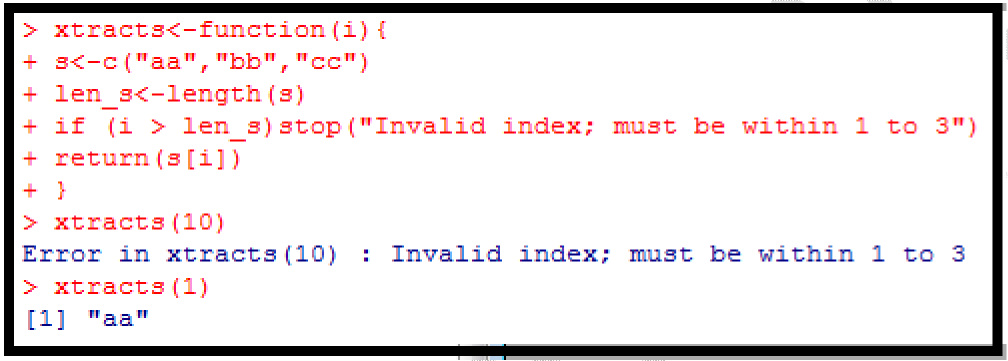
```
> s[-3]  # to create  a vector slice with the third member "cc" removed.
[1] "aa" "bb" "dd"
```

Out-of-Range Index



```
> s[10]  # If an index is out-of-range, NA is reported as the missing value
[1] NA
> s = c("aa", "bb", "cc", "dd")  # to retrieve a vector slice containing the 2nd and 3rd members of s
> s
[1] "aa" "bb" "cc" "dd"
> s[c(2, 3)]
[1] "bb" "cc"
```

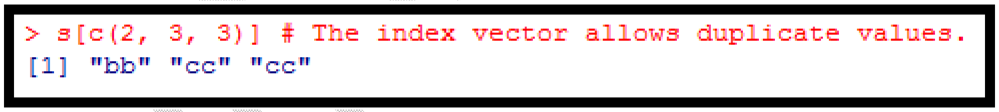
- We can validate the input to the function and throw error message when the input index is out-of-range as shown below:



```
> xtracts<-function(i){
+ s<-c("aa","bb","cc")
+ len_s<-length(s)
+ if (i > len_s)stop("Invalid index; must be within 1 to 3")
+ return(s[i])
+ }
> xtracts(10)
Error in xtracts(10) : Invalid index; must be within 1 to 3
> xtracts(1)
[1] "aa"
```

Numeric Index Vector

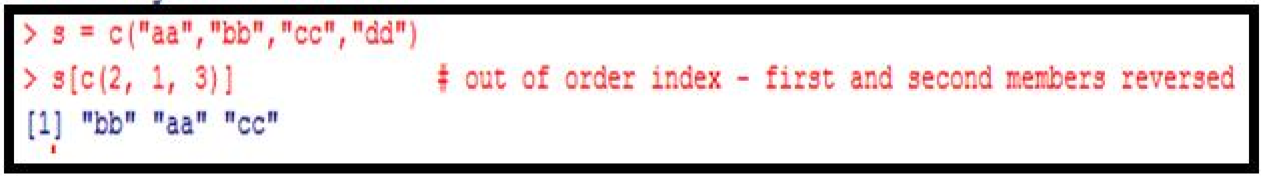
- A new vector can be sliced from a given vector with a numeric index vector, which consists of member positions of the original vector to be retrieved.



```
> s[c(2, 3, 3)] # The index vector allows duplicate values.
[1] "bb" "cc" "cc"
```

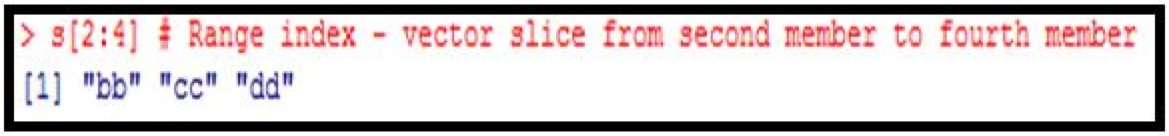
Out-of-Order Indexes

- The index vector can even be out-of-order.



```
> s = c("aa","bb","cc","dd")
> s[c(2, 1, 3)]              # out of order index - first and second members reversed
[1] "bb" "aa" "cc"
```

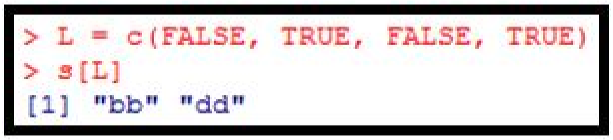
- Vector can be sliced as shown below:



```
> s[2:4] # Range index - vector slice from second member to fourth member
[1] "bb" "cc" "dd"
```

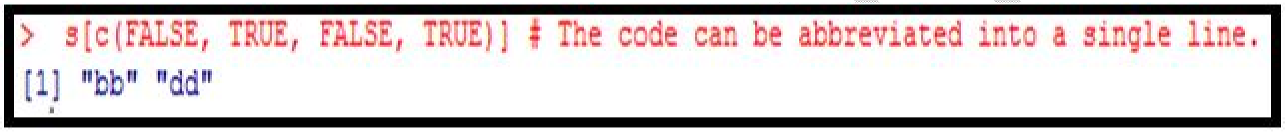
Logical Index Vector

- A new vector can be sliced from a given vector with a logical index vector, which has the same length as the original vector. Its members are TRUE if the corresponding members in the original vector are to be included in the slice, and FALSE if otherwise.



```
> L = c(FALSE, TRUE, FALSE, TRUE)
> s[L]
[1] "bb" "dd"
```

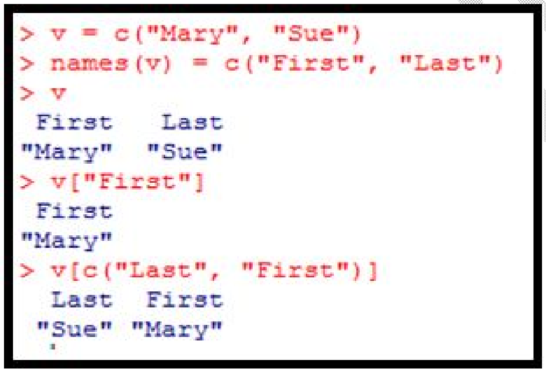
- To retrieve the second and fourth members of s, we define a logical vector L of the same length, and have its second and fourth members set as TRUE.



```
> s[c(FALSE, TRUE, FALSE, TRUE)] # The code can be abbreviated into a single line.
[1] "bb" "dd"
```

Named Vector Members

- We can assign names to vector members. For example, the following variable v is a character string vector with two members. Name the first member as First, and the second as Last.



```
> v = c("Mary", "Sue")
> names(v) = c("First", "Last")
> v
 First   Last
"Mary"  "Sue"
> v["First"]
 First
"Mary"
> v[c("Last", "First")]
  Last  First
 "Sue" "Mary"
```

Example 1: **Random numbers**

In everyday life, we come across persons making an assessment of the population through samples. A housewife tests a small quantity of rice to check if it is well cooked. The importance of the theory of sampling lies in the fact that for a large population, it is neither practical nor possible to collect data for each and every number of the population.

In testing the quality of the bulbs produced by the company it is impossible to test every bulb manufactured by the company; it is quite sufficient to if a sample is taken for testing and based on this test it is possible to draw conclusion about the population.

- A major advantage of the sampling is that it is very convenient and economical.
- Samples can be classified into two types:
  - Non-probability samples

- Probability samples

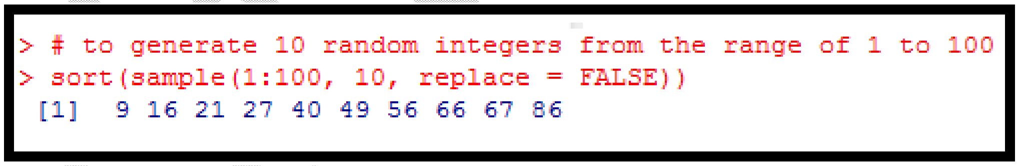
- In probability sampling, every sampling unit has a definite probability of being included in the sample.
- A sample from a population is said to be a random sample if every item of the population has equal chance of being selected.

A random sample can be divided into two types:

• **Unrestricted random sampling**	*Here every item drawn for the sample is noted and again* ***replaced*** *into the population before the next item is drawn. Here samples are drawn for sample one by one with replacement*
• **Restricted random sampling**	*Here the items for the sample are drawn one by one from the population in such a way that after each drawing the selected item* ***is not replaced*** *into the population and each time the drawing is made, everyone of the remaining members of the population has an equal chance of being selected.*

Consider B.Com final year students, total being 100 divided equally into boys and girls. Assume a researcher wants to choose 10 of them for further study. Each student is assigned a student Id ranging from 1 to 100.We need to select 10 students randomly from all the 100 students of B.Com final year class.

- The function sample generates random integers without replacement. The first argument specifies a vector containing the specified range of valid numbers. The second argument indicates the number of random integers to be returned. The function sort, sorts the output from the sample function.



```
> # to generate 10 random integers from the range of 1 to 100
> sort(sample(1:100, 10, replace = FALSE))
 [1]  9 16 21 27 40 49 56 66 67 86
```

- From the total 100 students of B.Com final year, the students with ID Numbers shown below are selected for further study:
  9, 16, 21, 27, 40, 49, 56, 66, 67 and 86

Example 2: **Basic Statistics**

Measures of averages:

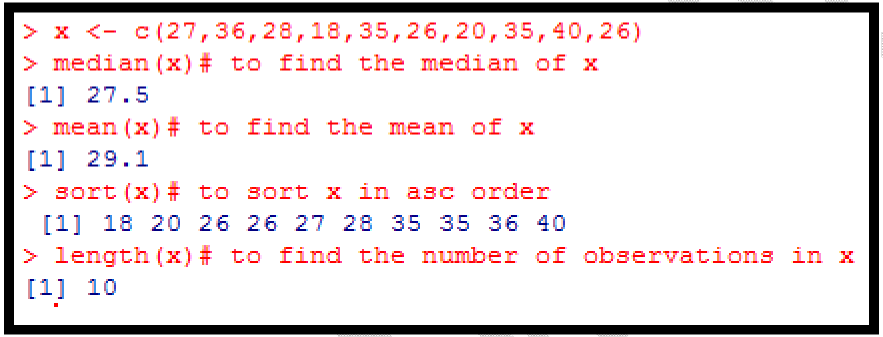
An average is considered as a typical representative of the whole data.

The various averages in common use are:

1. Mean
2. Median
3. Mode

- Arithmetic Mean is the most popular measure of central tendency. The mean is equal to the sum of all the values in the data set divided by the number of values in the data set.
- Median is the middle value for a set of data that has been arranged in order of magnitude. The median is less affected by outliers and skewed data.
- Mode is the value that occurs most often in the data set.

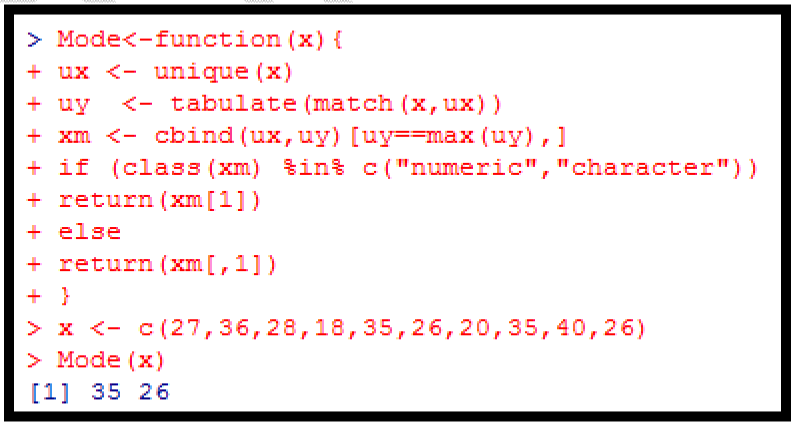
Find the mean and median of the set of observations 27,36,28,18,35,26,20,35,40,26



```
> x <- c(27,36,28,18,35,26,20,35,40,26)
> median(x)# to find the median of x
[1] 27.5
> mean(x)# to find the mean of x
[1] 29.1
> sort(x)# to sort x in asc order
 [1] 18 20 26 26 27 28 35 35 36 40
> length(x)# to find the number of observations in x
[1] 10
```

- Median is the mid value of 5th and 6th position since the total number of observations is even and it is 10.

Mode



```
> Mode<-function(x){
+ ux <- unique(x)
+ uy  <- tabulate(match(x,ux))
+ xm <- cbind(ux,uy)[uy==max(uy),]
+ if (class(xm) %in% c("numeric","character"))
+ return(xm[1])
+ else
+ return(xm[,1])
+ }
> x <- c(27,36,28,18,35,26,20,35,40,26)
> Mode(x)
[1] 35 26
```

- The function **unique** returns a vector, data frame or array like x but with duplicate elements / rows removed.
- The function **tabulate** takes the integer-valued vector bin and counts the number of times each integer occurs in it.

  Usage: tabulate(bin, nbins = max(1, bin, na.rm = TRUE))

  Arguments

bin	a numeric vector (of positive integers), or a factor. Long vectors are supported.
nbins	the number of bins to be used.

- The function **match** returns a vector of the positions of (first) matches of its first argument in its second.

  %in% is a more intuitive interface as a binary operator, which returns a logical vector indicating if there is a match or not for its left operand.

  Usage match(x, table, nomatch = NA_integer_, incomparables = NULL)

  x %in% table

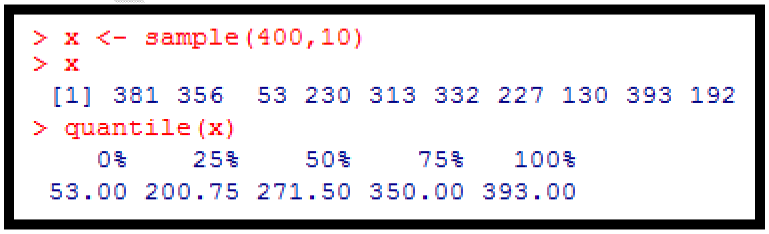
  Arguments

x	vector or NULL: the values to be matched. Long vectors are supported.
table	vector or NULL: the values to be matched against. Long vectors are not supported.
nomatch	the value to be returned in the case when no match is found. Note that it is coerced to integer.
incomparables	a vector of values that cannot be matched. Any value in x matching a value in this vector is assigned the nomatch value. For historical reasons, FALSE is equivalent to NULL.

- The function **cbind** takes a sequence of vector, matrix, or data frames arguments and combine by columns.
- We list all the x - elements with having the maximum number of occurrences
- In the current example, there are two modes namely 26 and 35.

Quantile

- The first quantile or lower quantile is the value that cuts off the first 25 % of the data when it is sorted in ascending order. The second quantile or median is the value that cuts off the first 50%. The third quantile or upper quantile is the value that cuts off the first 75%.

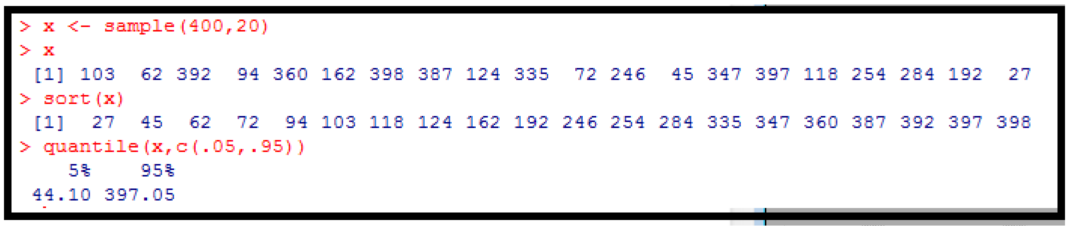


```
> x <- sample(400,10)
> x
 [1] 381 356  53 230 313 332 227 130 393 192
> quantile(x)
    0%    25%    50%    75%   100% 
 53.00 200.75 271.50 350.00 393.00 
```

Percentile

- The nth percentile of an observation variable is the value that cuts off the first n percent of the data values when it is sorted in ascending order.

  Find the 5th, 95th percentile of the data set x containing 20 random integers from 1 to 400.



```
> x <- sample(400,20)
> x
 [1] 103  62 392  94 360 162 398 387 124 335  72 246  45 347 397 118 254 284 192  27
> sort(x)
 [1]  27  45  62  72  94 103 118 124 162 192 246 254 284 335 347 360 387 392 397 398
> quantile(x,c(.05,.95))
    5%    95%
 44.10 397.05
```

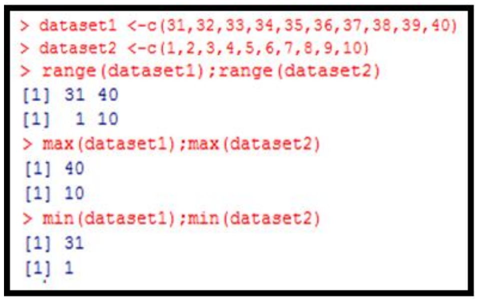
Measures of Dispersion

- By dispersion, it is meant spreading of the observations from an average.
- They measure the variability in the observed values in a data set.
- They are also called spread, scatter and measures of variation.

Range	Range is defined to be the difference between the largest and smallest of the observations.
Maxima and Minima	The functions min() and max() can be used to get the minimum and maximum values of the given data set.
Standard Deviation	• SD is defined as the positive square root of the Arithmetic Mean (AM) of the squares of all deviations of the observations from their AM. • The standard deviation measured the variability between observations in the sample or the population from the mean of that sample or that population. • SD is the most widely used measure of dispersion.
Coefficient of variation	• The relative measure of dispersion based on standard deviation is defined by (SD/Mean) * 100. • It is used to compare dispersion in two sets of data especially when the units are different.

1. Range

- The function range() gives the range of all the values - maximum and minimum values given as input.



```
> dataset1 <-c(31,32,33,34,35,36,37,38,39,40)
> dataset2 <-c(1,2,3,4,5,6,7,8,9,10)
> range(dataset1);range(dataset2)
[1] 31 40
[1]  1 10
> max(dataset1);max(dataset2)
[1] 40
[1] 10
> min(dataset1);min(dataset2)
[1] 31
[1] 1
```

- The function min() and max() gives the minimum and maximum values of the data set given as input.

2. Variance

- The function var() calculates the variance of the input values. It is Arithmetic Mean (AM) of the squares of all deviations of the observations from their AM.

3. Standard Deviation

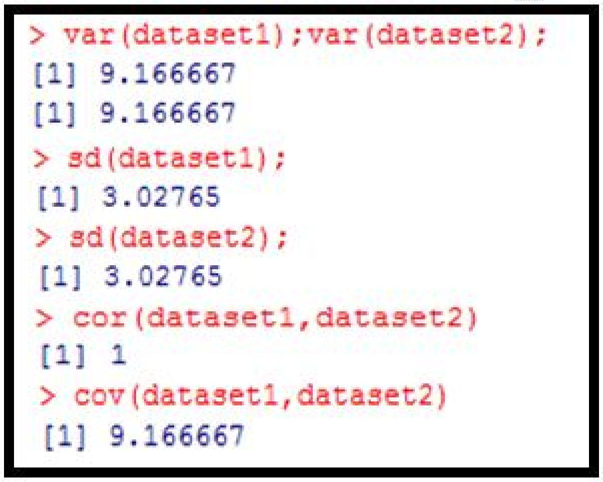
- Standard Deviation is the square root of variance. It is using the function sd().

4. Correlation coefficient

- This property measures how two variables are co-related and calculated by using the function cor(). Its value varies from -1 to 1.

5. Covariance

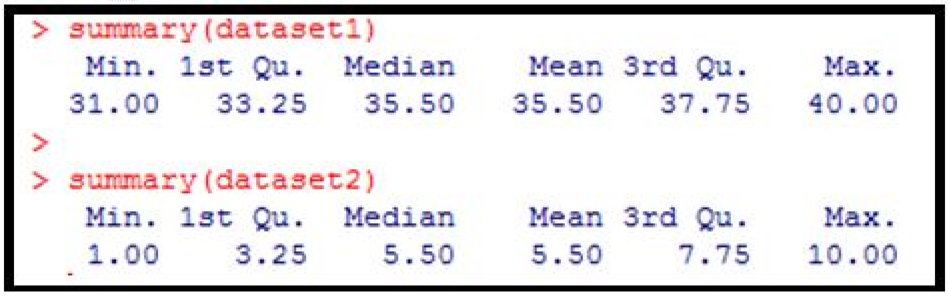
- Covariance, calculated by using the function cov() tells how two variables are varying together.



```
> var(dataset1);var(dataset2);
[1] 9.166667
[1] 9.166667
> sd(dataset1);
[1] 3.02765
> sd(dataset2);
[1] 3.02765
> cor(dataset1,dataset2)
[1] 1
> cov(dataset1,dataset2)
[1] 9.166667
```

6. Summary

- In addition to all these specific statistical functions, the function summary() will provide us with all the most basic statistical properties.



```
> summary(dataset1)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  31.00   33.25   35.50   35.50   37.75   40.00
>
> summary(dataset2)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    3.25    5.50    5.50    7.75   10.00
```

5. Matrix

- A matrix is a collection of data elements arranged in a two-dimensional rectangular layout. The following is an example of a matrix with 2 rows and 3 columns.

$$A = \begin{bmatrix} 2 & 4 & 3 \\ 1 & 5 & 7 \end{bmatrix}$$

- In Matrix, the data elements must be of the same basic type.

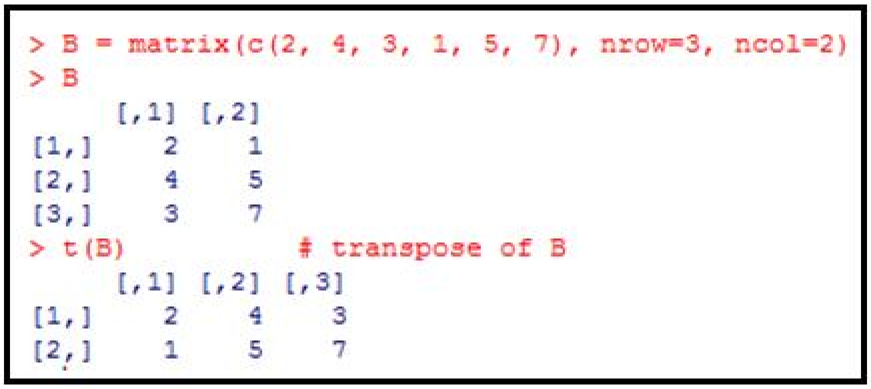
```
> A  = matrix(c(2,4,3,1,5,7), # the data elements of the matrix
+ nrow = 2,                   # number of rows
+ ncol  =  3,                 # number of columns
+ byrow = TRUE)               # fill matrix by row
> A                           # print the matrix A
     [,1] [,2] [,3]
[1,]    2    4    3
[2,]    1    5    7
>
> A[2,3]              # Access the element at 2nd row & 3rd col and print
[1] 7
> A[,3]               # Access the entire 3rd column and print
[1] 3 7
> A[2,]               # Access the entire 2nd row and print
[1] 1 5 7
> A [,c(1,3)]         # Access the entire first and third columns and print
     [,1] [,2]
[1,]    2    3
[2,]    1    7
```

- If we assign names to the rows and columns of the matrix, than we can access the elements by names.

```
> dimnames(A)  = list(
+ c("row1","row2"),              # row names
+ c("col1","col2","col3"))    # column names
> #
> A  # Print the matrix A
     col1 col2 col3
row1    2    4    3
row2    1    5    7
> #
> A["row2","col3"]  # element at 2nd row and 3rd column
[1] 7
```

Matrix Construction

- When we construct a matrix directly with data elements, the matrix content is filled along the column orientation by default.



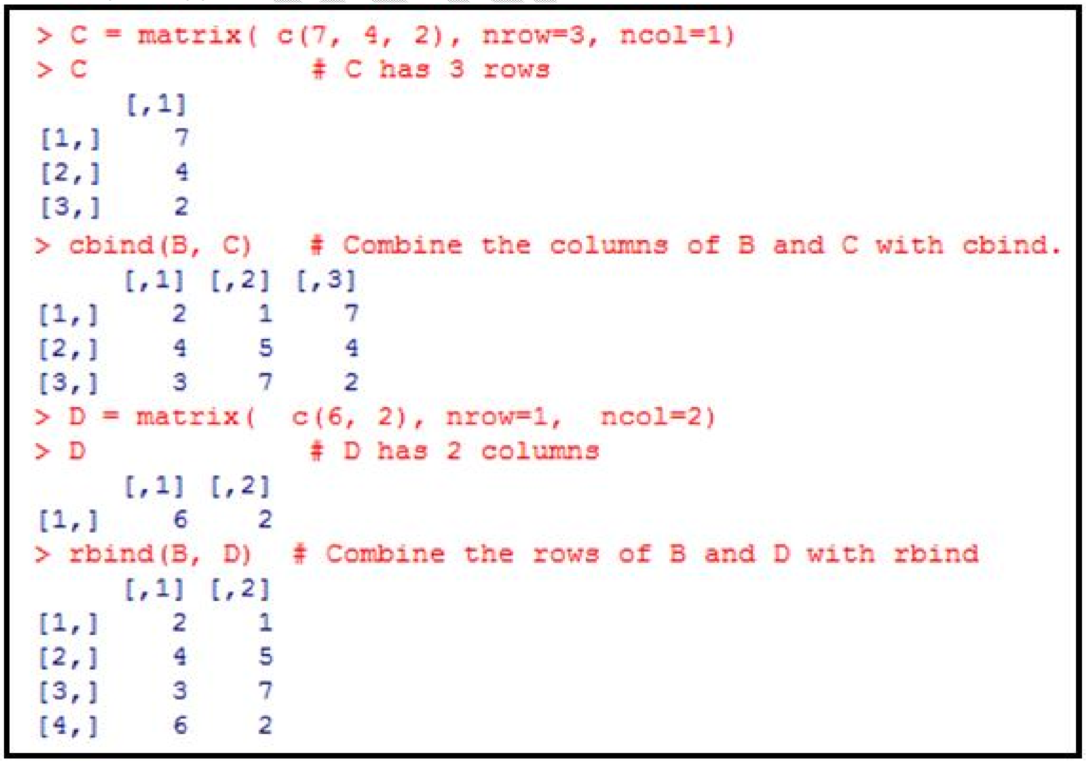
```
> B = matrix(c(2, 4, 3, 1, 5, 7), nrow=3, ncol=2)
> B
     [,1] [,2]
[1,]    2    1
[2,]    4    5
[3,]    3    7
> t(B)          # transpose of B
     [,1] [,2] [,3]
[1,]    2    4    3
[2,]    1    5    7
```

Transpose

- We construct the transpose of a matrix by interchanging its columns and rows with the function t as shown above.

Combining Matrices

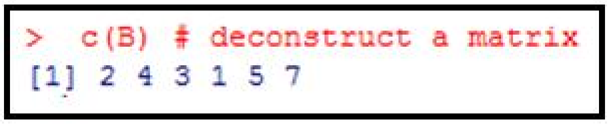
- The columns of two matrices having the same number of rows can be combined into a larger matrix.
- For example, suppose we have another matrix C with 3 rows.



```
> C = matrix( c(7, 4, 2), nrow=3, ncol=1)
> C               # C has 3 rows
     [,1]
[1,]    7
[2,]    4
[3,]    2
> cbind(B, C)     # Combine the columns of B and C with cbind.
     [,1] [,2] [,3]
[1,]    2    1    7
[2,]    4    5    4
[3,]    3    7    2
> D = matrix(  c(6, 2), nrow=1,  ncol=2)
> D               # D has 2 columns
     [,1] [,2]
[1,]    6    2
> rbind(B, D)   # Combine the rows of B and D with rbind
     [,1] [,2]
[1,]    2    1
[2,]    4    5
[3,]    3    7
[4,]    6    2
```

Deconstruction

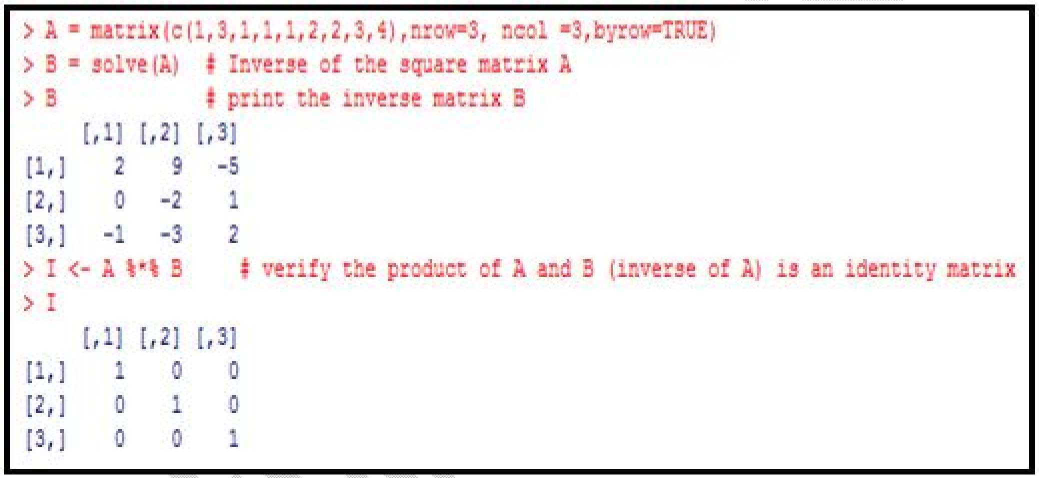
- We can deconstruct a matrix by applying the c function, which combines all column vectors into one.



```
>  c(B) # deconstruct a matrix
[1] 2 4 3 1 5 7
```

Inverse of Matrix

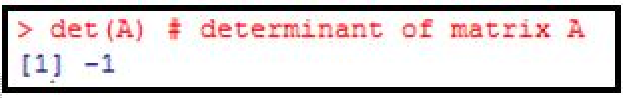
- Let A be a non-singular matrix. If there exists a square matrix B such that AB = I (Identity matrix) then B is called the inverse of matrix A and is denoted by A^{-1}.



```
> A = matrix(c(1,3,1,1,1,2,2,3,4),nrow=3, ncol =3,byrow=TRUE)
> B = solve(A)   # Inverse of the square matrix A
> B              # print the inverse matrix B
     [,1] [,2] [,3]
[1,]    2    9   -5
[2,]    0   -2    1
[3,]   -1   -3    2
> I <- A %*% B      # verify the product of A and B (inverse of A) is an identity matrix
> I
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Determinant of a matrix

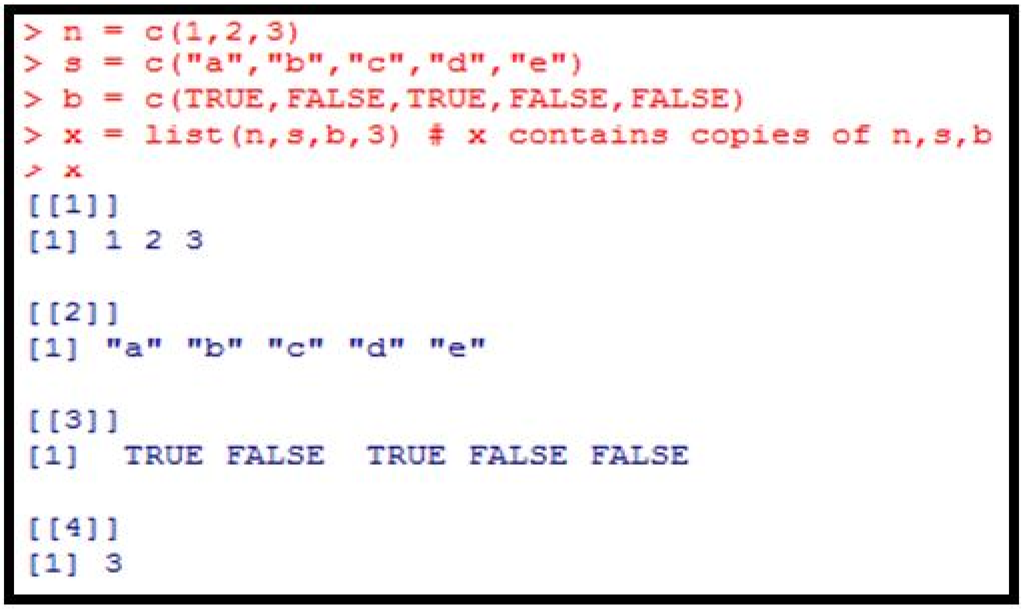
- Determinant of a square matrix is a single number calculated by combining all the elements of the matrix.
- The function det() calculates the determinant of the matrix (a1,b1,a2,b2) is a1 X b2 - a2 X b1.



```
> det(A) # determinant of matrix A
[1] -1
```

6. Lists

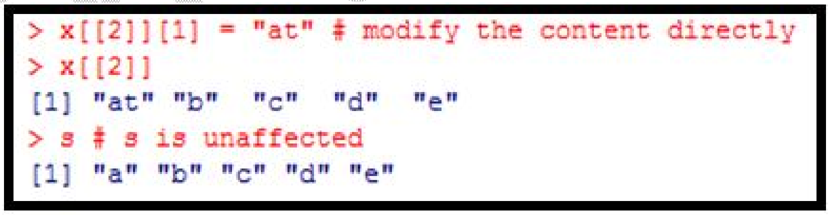
- A list is a generic vector containing other objects.
- For example, the following variable x is a list containing copies of three vectors n, s, b, and a numeric value 3..



```
> n = c(1,2,3)
> s = c("a","b","c","d","e")
> b = c(TRUE,FALSE,TRUE,FALSE,FALSE)
> x = list(n,s,b,3) # x contains copies of n,s,b
> x
[[1]]
[1] 1 2 3

[[2]]
[1] "a" "b" "c" "d" "e"

[[3]]
[1]  TRUE FALSE  TRUE FALSE FALSE

[[4]]
[1] 3
```

Member Reference

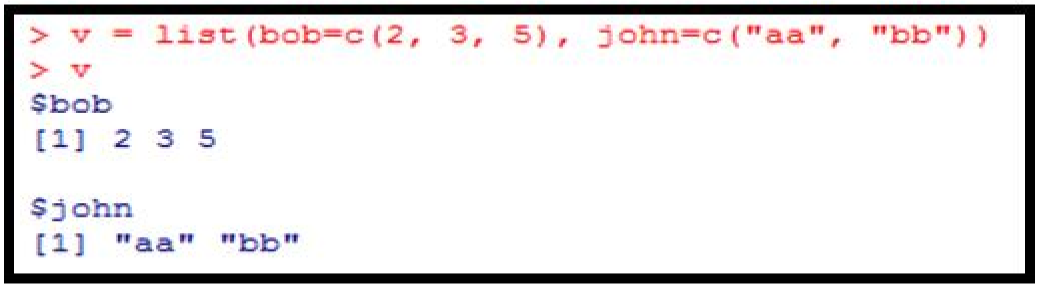
- In order to reference a list member directly, we have to use the double square bracket "[[]]" operator. The following object x[[2]] is the second member of x. In other words, x[[2]] is a copy of s, but is not a slice containing s or its copy.



```
> x[[2]][1] = "at" # modify the content directly
> x[[2]]
[1] "at" "b"  "c"  "d"  "e"
> s # s is unaffected
[1] "a" "b" "c" "d" "e"
```

Named List Members

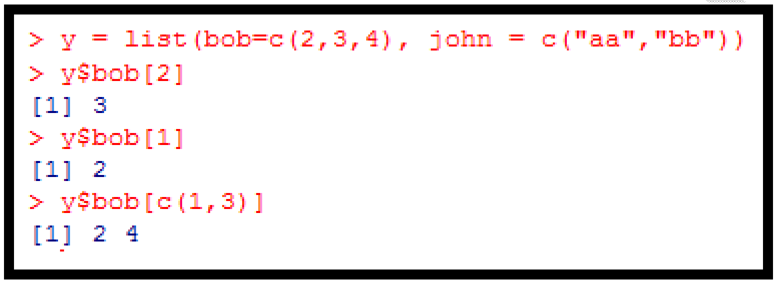
- We can assign names to list members, and reference them by names instead of numeric indexes.
- For example, in the following, v is a list of two members, named "bob" and "john".



```
> v = list(bob=c(2, 3, 5), john=c("aa", "bb"))
> v
$bob
[1] 2 3 5

$john
[1] "aa" "bb"
```

List Slicing

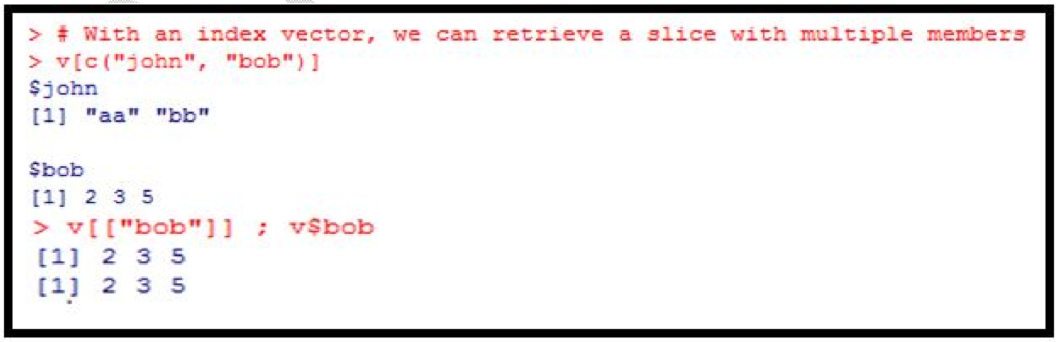
- We retrieve a list slice with the single square bracket "[]" operator. Here is a list slice containing a member of v named "bob".



```
> y = list(bob=c(2,3,4), john = c("aa","bb"))
> y$bob[2]
[1] 3
> y$bob[1]
[1] 2
> y$bob[c(1,3)]
[1] 2 4
```

Member Reference

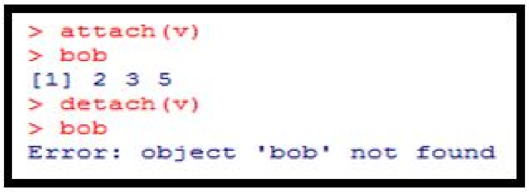
- In order to reference a list member directly, we have to use the double square bracket "[[]]" operator or by using $ operator.
- The following references a member of v by name.



```
> # With an index vector, we can retrieve a slice with multiple members
> v[c("john", "bob")]
$john
[1] "aa" "bb"

$bob
[1] 2 3 5
> v[["bob"]] ; v$bob
[1] 2 3 5
[1] 2 3 5
```

Search Path Attachment

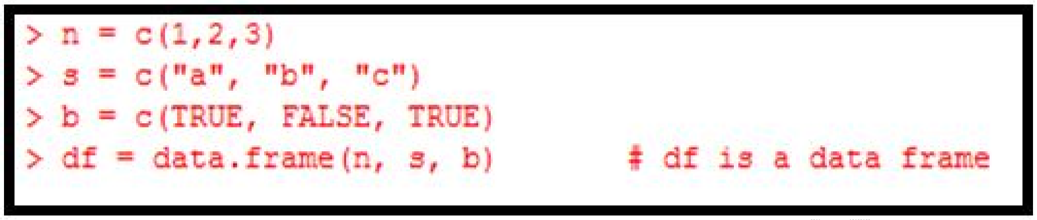
- We can attach a list to the R search path and access its members without explicitly mentioning the list. It should to be detached for cleanup.



```
> attach(v)
> bob
[1] 2 3 5
> detach(v)
> bob
Error: object 'bob' not found
```

7. Data Frame

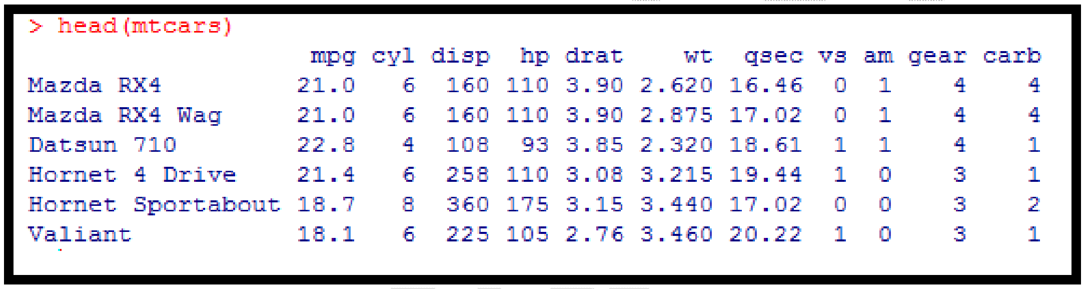
- A data frame is used for storing data tables. It is a list of vectors of equal length.
- For example, the variable df is a data frame containing three vectors n, s, b.



```
> n = c(1,2,3)
> s = c("a", "b", "c")
> b = c(TRUE, FALSE, TRUE)
> df = data.frame(n, s, b)       # df is a data frame
```

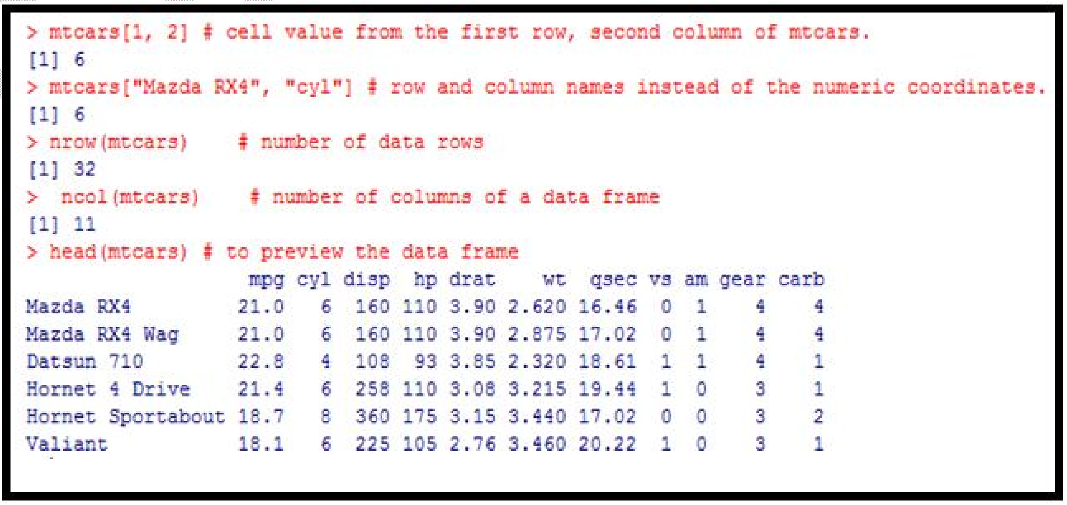
Build-in Data Frame

- We use built-in data frames in R for our tutorials. For example, here is a built-in Data frame in R, called mtcars.



```
> head(mtcars)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

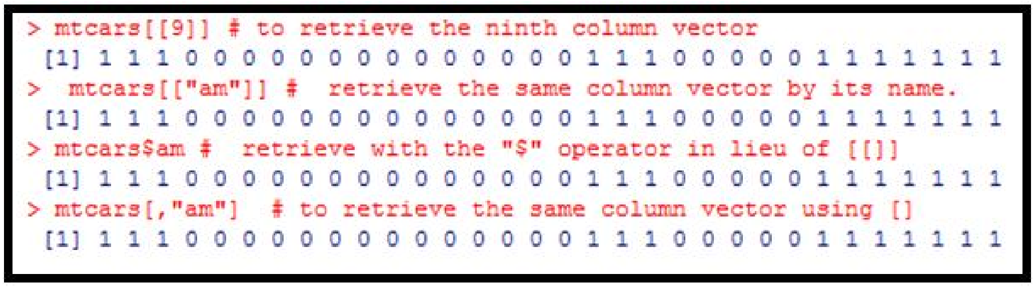
- The top line of the table, called the header, contains the column names. Each horizontal line afterward denotes a data row, which begins with the name of the row, and followed by the actual data. Each data member of a row is called a cell.
- To retrieve data in a cell, we would enter its row and column coordinates in the single square bracket "[]" operator. The two coordinates are separated by a comma.
- In other words, the coordinates begins with row position, followed by a comma, and ends with the column position. The order is important.



```
> mtcars[1, 2] # cell value from the first row, second column of mtcars.
[1] 6
> mtcars["Mazda RX4", "cyl"] # row and column names instead of the numeric coordinates.
[1] 6
> nrow(mtcars)     # number of data rows
[1] 32
>  ncol(mtcars)     # number of columns of a data frame
[1] 11
> head(mtcars) # to preview the data frame
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Data Frame Column Vector

- We reference a data frame column with the double square bracket "[[]]" operator.



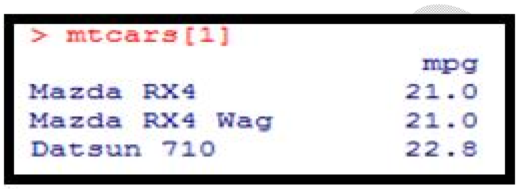
```
> mtcars[[9]] # to retrieve the ninth column vector
 [1] 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1
>  mtcars[["am"]] #  retrieve the same column vector by its name.
 [1] 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1
> mtcars$am #  retrieve with the "$" operator in lieu of [[]]
 [1] 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1
> mtcars[,"am"]  # to retrieve the same column vector using []
 [1] 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1
```

Data Frame Column Slice

- We retrieve a data frame column slice with the single square bracket "[]" operator.

Numeric Indexing

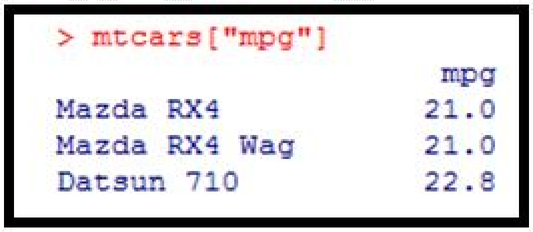
- The following is a slice containing the first column of the built-in data set mtcars.



```
> mtcars[1]
                   mpg
Mazda RX4         21.0
Mazda RX4 Wag     21.0
Datsun 710        22.8
```

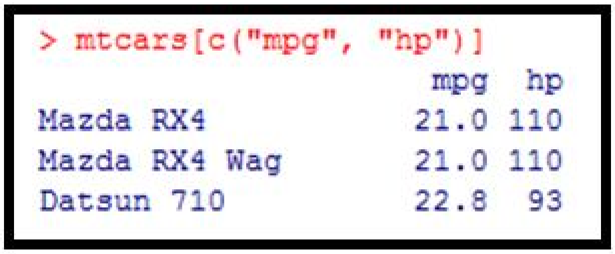
Name Indexing

- We can retrieve the same column slice by its name.



```
> mtcars["mpg"]
                   mpg
Mazda RX4         21.0
Mazda RX4 Wag     21.0
Datsun 710        22.8
```

- To retrieve a data frame slice with the two columns mpg and hp, we pack the column names in an index vector inside the single [ ] operator.



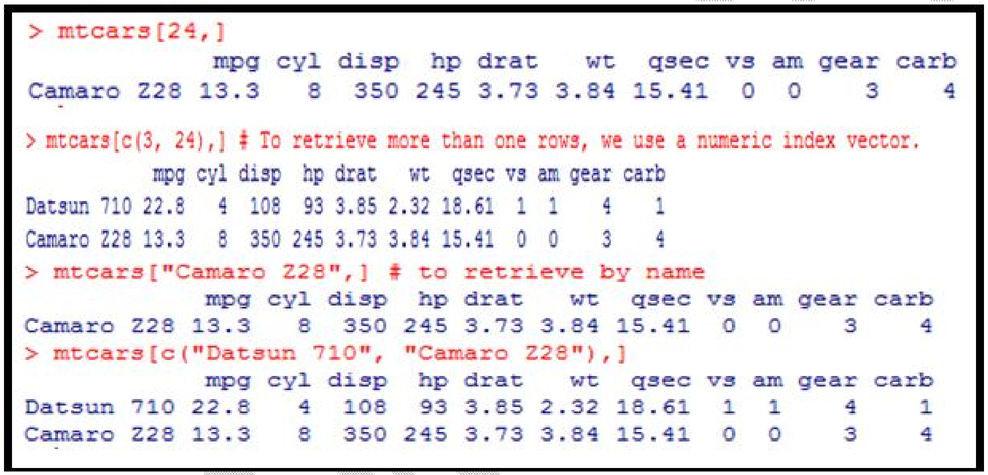
```
> mtcars[c("mpg", "hp")]
                   mpg  hp
Mazda RX4         21.0 110
Mazda RX4 Wag     21.0 110
Datsun 710        22.8  93
```

Data Frame Row Slice

- We retrieve rows from a data frame with the single square bracket operator, just like what we did with columns. However, in additional to an index vector of row positions, we append an extra comma character. This is important, as the extra comma signals a wildcard match for the second coordinate for column positions.

Numeric Indexing

- For example, the following retrieves a row record of the built-in data set mtcars.
- Please notice the extra comma in the square bracket operator.
- It states that the 1974 Camaro Z28 has a gas mileage of 13.3 miles per gallon, and an eight cylinder 245 horse power engine, ..., etc.

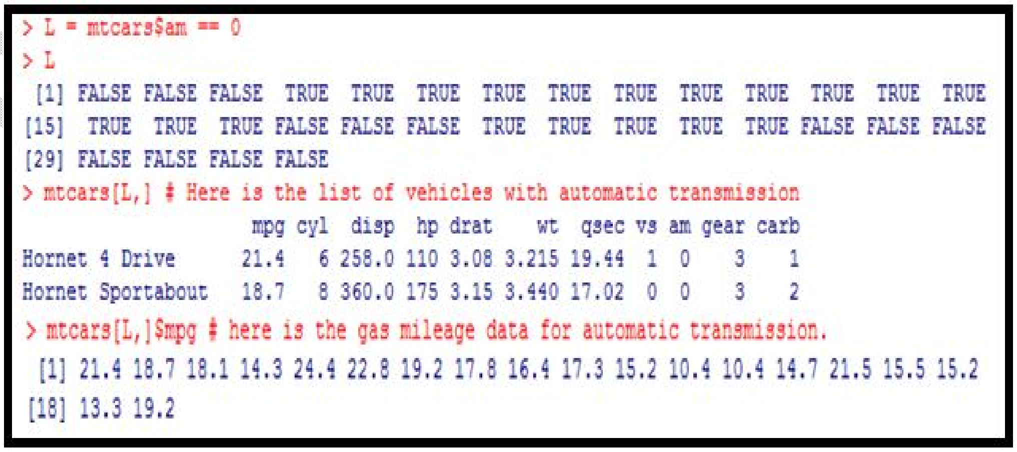


```
> mtcars[24,]
            mpg cyl disp  hp drat   wt  qsec vs am gear carb
Camaro Z28 13.3   8  350 245 3.73 3.84 15.41  0  0    3    4

> mtcars[c(3, 24),] # To retrieve more than one rows, we use a numeric index vector.
            mpg cyl disp  hp drat   wt  qsec vs am gear carb
Datsun 710 22.8   4  108  93 3.85 2.32 18.61  1  1    4    1
Camaro Z28 13.3   8  350 245 3.73 3.84 15.41  0  0    3    4
> mtcars["Camaro Z28",] # to retrieve by name
            mpg cyl disp  hp drat   wt  qsec vs am gear carb
Camaro Z28 13.3   8  350 245 3.73 3.84 15.41  0  0    3    4
> mtcars[c("Datsun 710", "Camaro Z28"),]
            mpg cyl disp  hp drat   wt  qsec vs am gear carb
Datsun 710 22.8   4  108  93 3.85 2.32 18.61  1  1    4    1
Camaro Z28 13.3   8  350 245 3.73 3.84 15.41  0  0    3    4
```

- We can pack the row names in an index vector in order to retrieve multiple rows, as shown above.

Logical Indexing

- Lastly, we can retrieve rows with a logical index vector. In the following vector L, the member value is TRUE if the car has automatic transmission, and FALSE if otherwise.



```
> L = mtcars$am == 0
> L
 [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[15]  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
[29] FALSE FALSE FALSE FALSE
> mtcars[L,] # Here is the list of vehicles with automatic transmission
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
> mtcars[L,]$mpg # here is the gas mileage data for automatic transmission.
 [1] 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4 10.4 14.7 21.5 15.5 15.2
[18] 13.3 19.2
```

8. Time Series

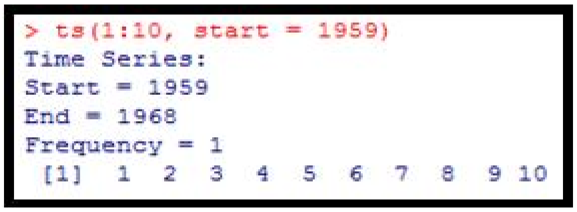
- A time series is a set of observations, measured typically at successive points in time spaced at uniform time intervals and arranged in chronological order.
- Time series are used in weather forecasting, earthquake prediction, and largely in any domain of applied science and engineering which involves temporal measurements.
- Examples of time series include:
  - the hourly series of temperature recorded by the Meteorological observatory
  - the daily series closing price of shares in the National Stock Exchange
- Time series analysis comprises methods for analyzing time series data in order to extract meaningful statistics and other characteristics of the data.
- Time series forecasting is the use of a model to predict future values based on previously observed values.
- The Holt- Winters forecasting procedure is a widely used projection method which can cope with trend and seasonal variation.
- The Holt-Winters forecasting model consists of both an exponentially smoothing component (E, w) and a trend component (T, v) with two different smoothing factors.
  - Here, $\mathbf{E_{t-1}}$ is the estimated value for period t-1
  - **w** is the smoothing constant for estimates
  - **T** is the trend for period t
  - **v** is the smoothing factor for trend

- The function ts creates an object of class "ts" from a vector (single time-series) or a matrix (multi-variate time-series), and some options which characterize the series.
- The options, with the default values, are:

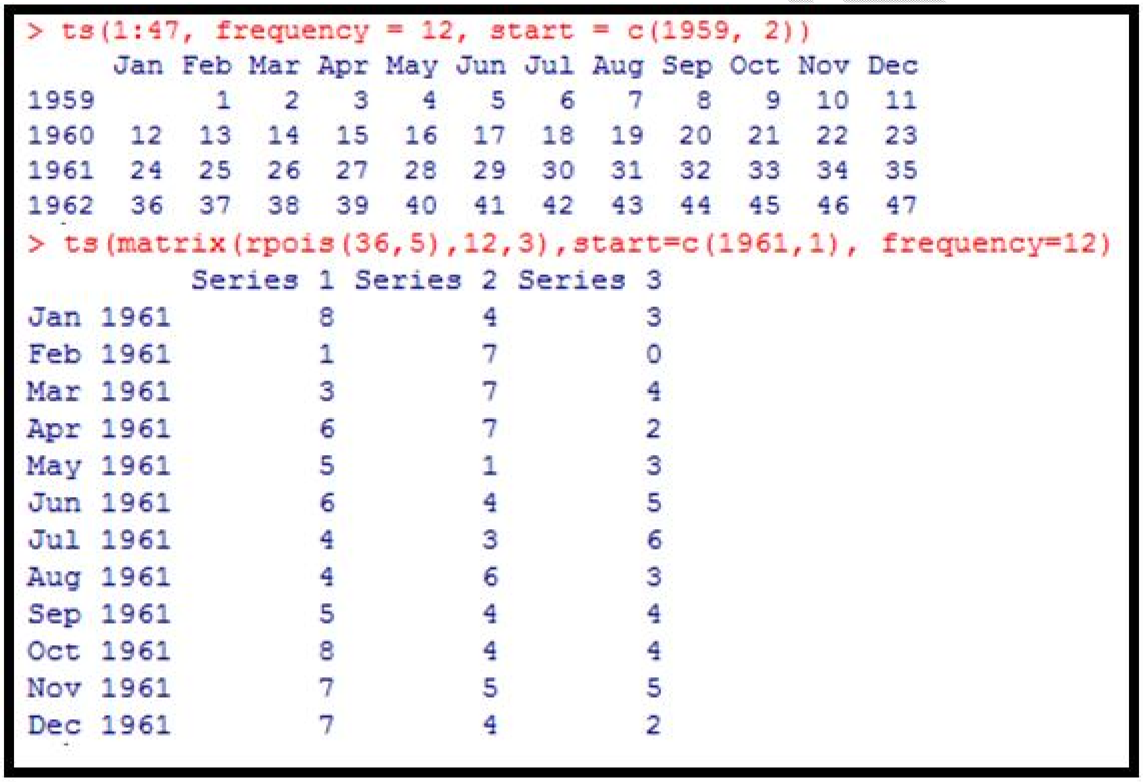
ts(data = NA, start = 1, end = numeric(0), frequency = 1, deltat = 1,ts.eps = getOption("ts.eps"), class, names)

data	a vector
start	the time of the first observation, either a number, or a vector of two integers
end	the time the last observation specified in the same way than start
frequency	the number of observations per time unit
deltat	the fraction of the sampling period between successive observations (ex. 1/12 for monthly data); only one of the frequency or deltat must be given
ts.eps	tolerance for the comparison of series. The frequencies are considered equal if their difference is less than ts.eps
class	class to give to the object; the default is "ts" for a single series, and c("mts"."ts") for a multivariate series

names	a vector of mode character with the names of the individual series in the case of a multivariate series; by default the names of the columns of data, or Series 1, Series 2. ...



```
> ts(1:10, start = 1959)
Time Series:
Start = 1959
End = 1968
Frequency = 1
 [1]  1  2  3  4  5  6  7  8  9 10
```



```
> ts(1:47, frequency = 12, start = c(1959, 2))
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1959       1   2   3   4   5   6   7   8   9  10  11
1960  12  13  14  15  16  17  18  19  20  21  22  23
1961  24  25  26  27  28  29  30  31  32  33  34  35
1962  36  37  38  39  40  41  42  43  44  45  46  47
> ts(matrix(rpois(36,5),12,3),start=c(1961,1), frequency=12)
         Series 1 Series 2 Series 3
Jan 1961        8        4        3
Feb 1961        1        7        0
Mar 1961        3        7        4
Apr 1961        6        7        2
May 1961        5        1        3
Jun 1961        6        4        5
Jul 1961        4        3        6
Aug 1961        4        6        3
Sep 1961        5        4        4
Oct 1961        8        4        4
Nov 1961        7        5        5
Dec 1961        7        4        2
```

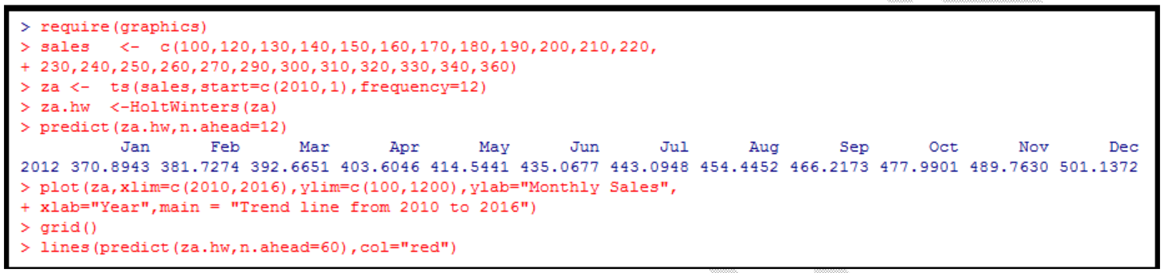
Example 3: Time series

Monthly series of income from sales in lakhs of Indian Rupees for a large retail store in Chennai for the years 2010 and 2011 are given below:

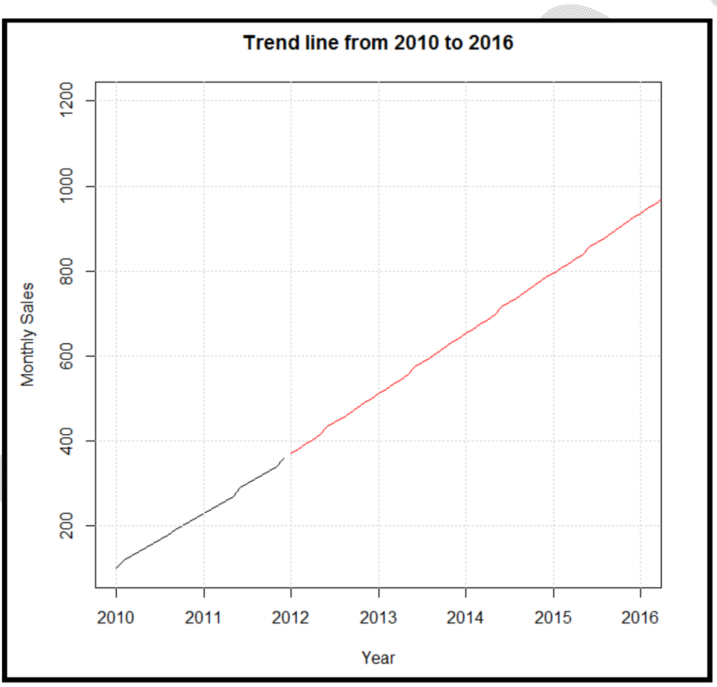
Year	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2010	100	120	130	140	150	160	170	180	190	200	210	220
2011	230	240	250	260	270	290	300	310	320	330	340	360

Predict the income from sales for the year 2012 and plot in a graph the trend line from 2012 to 2016.

Solution:



```
> require(graphics)
> sales   <-  c(100,120,130,140,150,160,170,180,190,200,210,220,
+ 230,240,250,260,270,290,300,310,320,330,340,360)
> za <-  ts(sales,start=c(2010,1),frequency=12)
> za.hw  <-HoltWinters(za)
> predict(za.hw,n.ahead=12)
          Jan      Feb      Mar      Apr      May      Jun      Jul      Aug      Sep      Oct      Nov      Dec
2012 370.8943 381.7274 392.6651 403.6046 414.5441 435.0677 443.0948 454.4452 466.2173 477.9901 489.7630 501.1372
> plot(za,xlim=c(2010,2016),ylim=c(100,1200),ylab="Monthly Sales",
+ xlab="Year",main = "Trend line from 2010 to 2016")
> grid()
> lines(predict(za.hw,n.ahead=60),col="red")
```

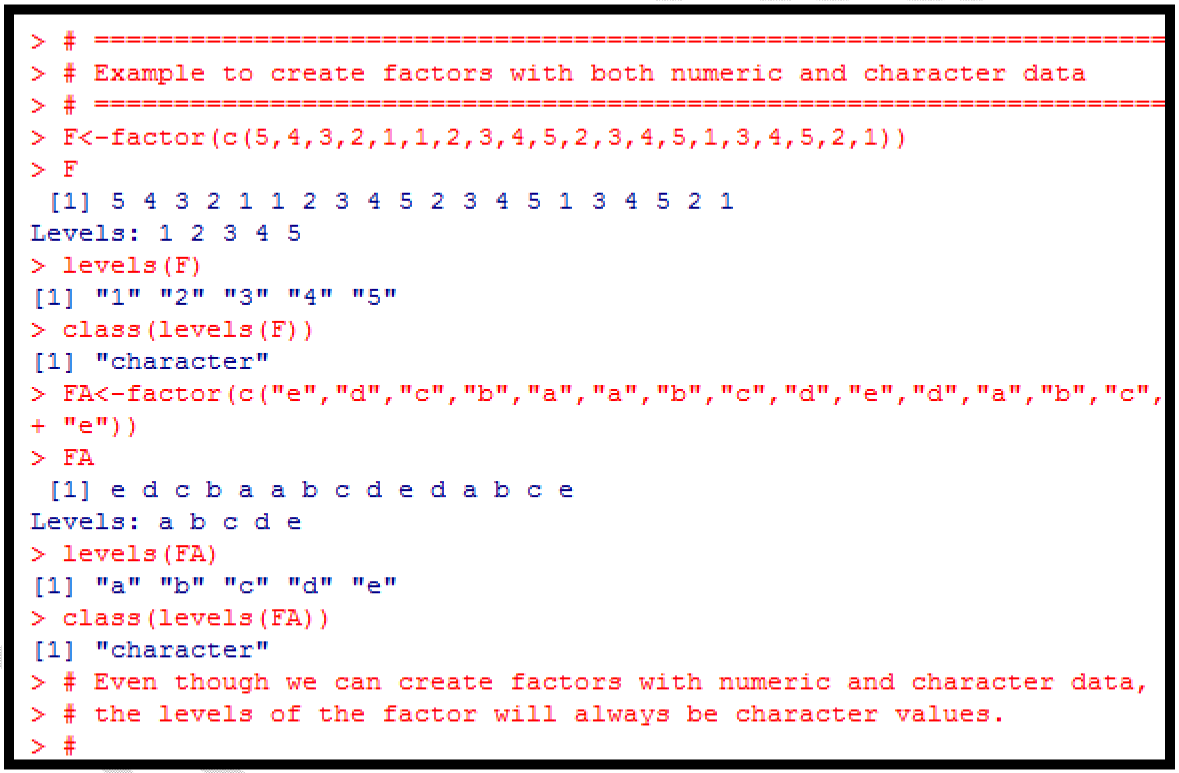


- Sales vector is converted to a time series object, za.
- HoltWinters procedure is performed on dataset, za and stored in za.hw.
- Using the predict function, sales for the next 12 months are predicted.
- The time series observed values and predicted values are plotted in a graph.
- Grid lines are drawn in the graph.

Lines are drawn through the predicted points for 60 months from 2010 using the color red.

9. Factors

- A factor is a vector object used to specify a discrete classification (grouping) of the components of other vectors of the same length. R provides both both ordered and unordered factors.
- In ordered factors, the levels of factors, which represent the category of elements are stored in alphabetical order, or in the order they were specified to factor if they were specified explicitly.
- Levels will help user to represent the data in a compact manner.
- The function factor is used to encode a vector as a factor (the terms 'category' and 'enumerated type' are also used for factors). If argument ordered is TRUE, the factor levels are assumed to be ordered.
- Factors are mostly used for experimental observations that include trials at different levels of selected variables. Factors are widely used in statistical modeling purpose.



```
> # ====================================================================
> # Example to create factors with both numeric and character data
> # ====================================================================
> F<-factor(c(5,4,3,2,1,1,2,3,4,5,2,3,4,5,1,3,4,5,2,1))
> F
 [1] 5 4 3 2 1 1 2 3 4 5 2 3 4 5 1 3 4 5 2 1
Levels: 1 2 3 4 5
> levels(F)
[1] "1" "2" "3" "4" "5"
> class(levels(F))
[1] "character"
> FA<-factor(c("e","d","c","b","a","a","b","c","d","e","d","a","b","c",
+ "e"))
> FA
 [1] e d c b a a b c d e d a b c e
Levels: a b c d e
> levels(FA)
[1] "a" "b" "c" "d" "e"
> class(levels(FA))
[1] "character"
> # Even though we can create factors with numeric and character data,
> # the levels of the factor will always be character values.
> #
```