UNIT – II

Basics of R

- Introduction
- R-Environment Setup, Programming with R, Basic Data Types
- Vectors: Creating and Naming Vectors, Vector Arithmetic Vector Subsetting
- Matrices: Creating and Naming Matrices, Matrix Subsetting
- Arrays, Class.

Factors and Dataframes

- Introduction to Factors: Factor Levels, Summarizing a Factor
- Ordered Factors, Comparing Ordered Factors
- Introduction to Data Frame, Subsetting of Data Frames, Extending Data Frames, Sorting Data Frames

Part - I
Basics of R

Introduction

- R is a language for statistical computing.
- Initially developed by *Ross Ihaka* and *Robert Gentleman* at the University of Auckland in early 90s.
- It is considered as an open source implementation of the S language.

Advantages

- open source and hence free
- it has top-notch functionalities for effective graphical representation of data
- it is easy to use
- Users can wrap their work in R scripts and this can be easily shared with colleagues

Disadvantages

- R seems to be relatively easy to learn in the beginning, but it is hard to really master it.
- As R is command-based, it becomes highly inconvenient for many of the statisticians and other non-computer science professionals to use

R-Environment setup

- Installation of R(in windows)
  - R can be installed in Windows 7/8/10/Vista and supports both the 32-bit and 64-bit versions. Go to the CRAN website and select the latest installer R 3.3.3 for Windows and download the.exe file
  - After download, clicking on the setup file opens the dialog box shown in Fig. 1.1.

Click on the ‘Next’ button starts the installation process. This redirects you to the license window shown in Fig. 1.2; it is advisable to read the terms and conditions before selecting ‘Next’.

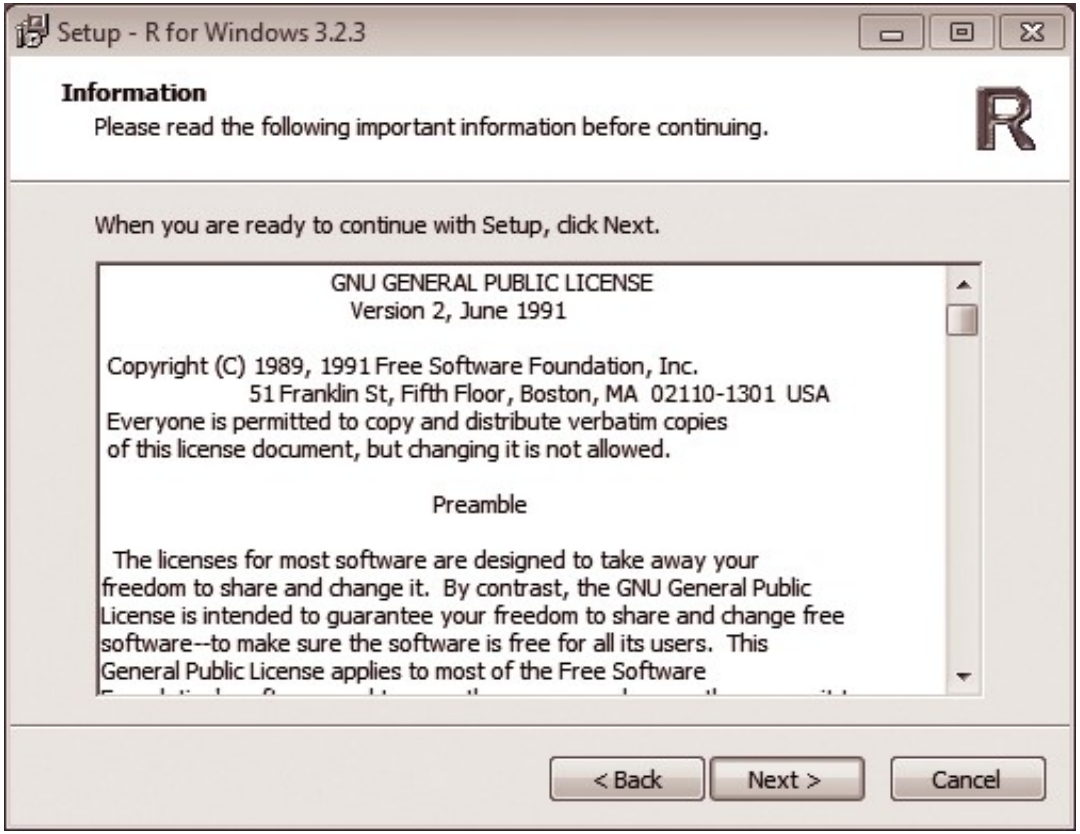
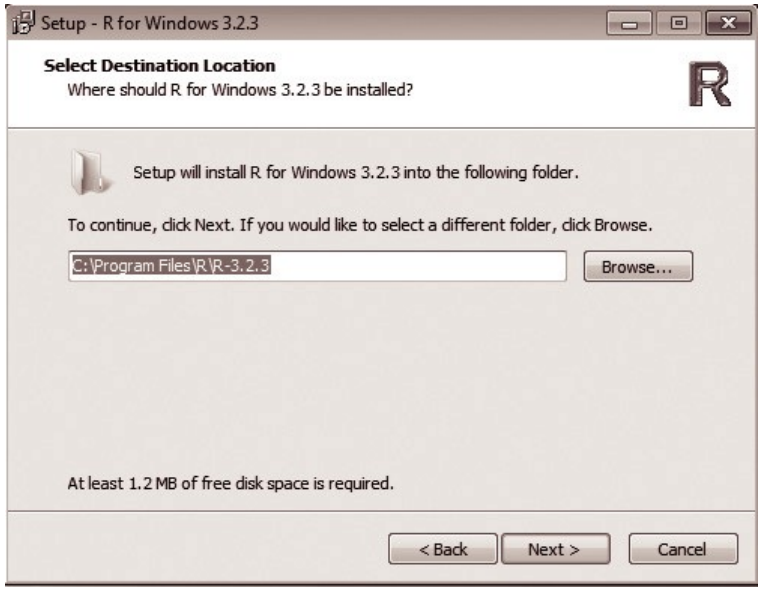


Fig. 1.2 R license agreement

4. After selecting the next button from the previous step the installation folder path is required (Fig. 1.3).
Select the desired folder for installation; it is advisable to select the C directory for smooth running of the program.



- **Fig. 1.3** Selecting the installation folder

5. Next select the components for installation based on the requirements of your operating system (OS) to avoid unwanted use of disk space (Fig. 1.4).

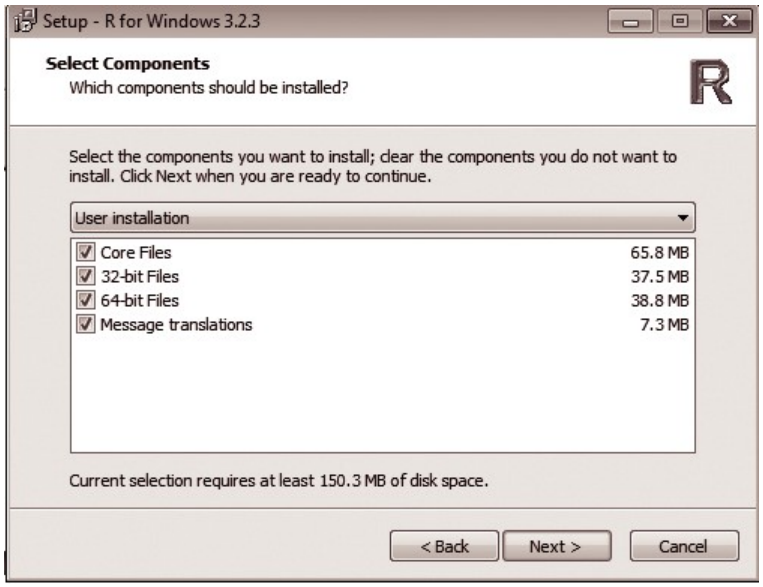


Fig. 1.4 Selecting components for installation

6. In the next dialog box (Fig. 1.5), we need to select the start menu folder. Here, it is better to go with the default option given by the installer.

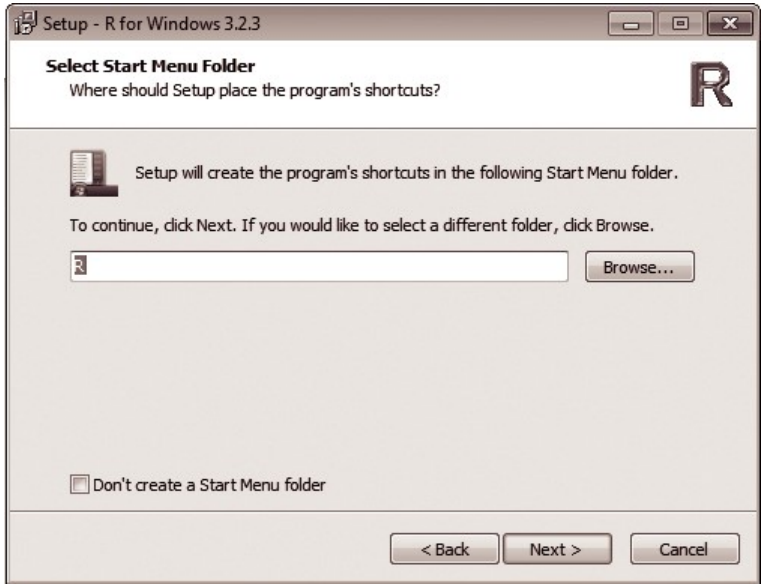


Fig. 1.5 Selecting the Start menu folder

7. After setting up the Start menu folder, check the additional options for completing the setup as shown in Fig. 1.6.

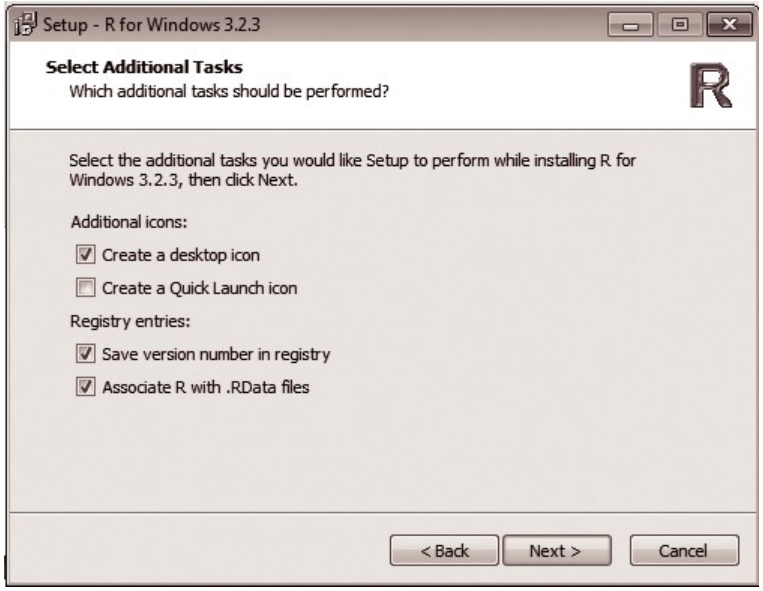


Fig. 1.6 Additional options for setup

8. After clicking next from the previous step, the installation procedure ends and the window in Fig. 1.7 is displayed. Click Finish to exit from the installation window.

Fig. 1.7 End of installation

Instalation In Ubuntu

- Go to the CRAN website or follow the link https://cran.r-project.org/bin/linux/ and it will redirect you to the index page. The index page contains four different parent directories—debain, Redhat, sesu, and ubuntu.

1. Add R to your repository by typing the following command:

   - sudo echo “dep http://cran.rstudio.com/bin/ubuntu trusty/” | sudo tee-a etc/apt/spurces.list.

2. Add R to Ubuntu keyring.

   - gpg –keyserver keyserver.ubuntu.com –recv-key E084DAB9
   - gpg -a –export E084DAB9 |sudo apt-key add –

3. Finally install R-Base.

- sudo apt-get update
- sudo apt-get install r-base r-base-dev apt-get update

- $ yum install R
- $ R

Install Rstudio

- RStudio can be installed in any of the Windows platforms such as Windows 7/8/10/Vista and can be configured within a few minutes.
- The basic requirement is R 2.11.1+ version.
- The following are the steps involved to setup RStudio:

    1. Download the latest version of RStudio just by clicking on the link provided here https://www.rstudio.com/products/rstudio/ download/

2. Download the.exe file and double click on it to initiate the installation as shown in Fig. 1.8

Fig. 1.8 RStudio Setup

3. Click on the Next button and it redirects you to select the installation folder (Fig. 1.9). Select 'C:\' as your installation directory since R and RStudio must be installed in the same directory to avoid path issues for running R programs.

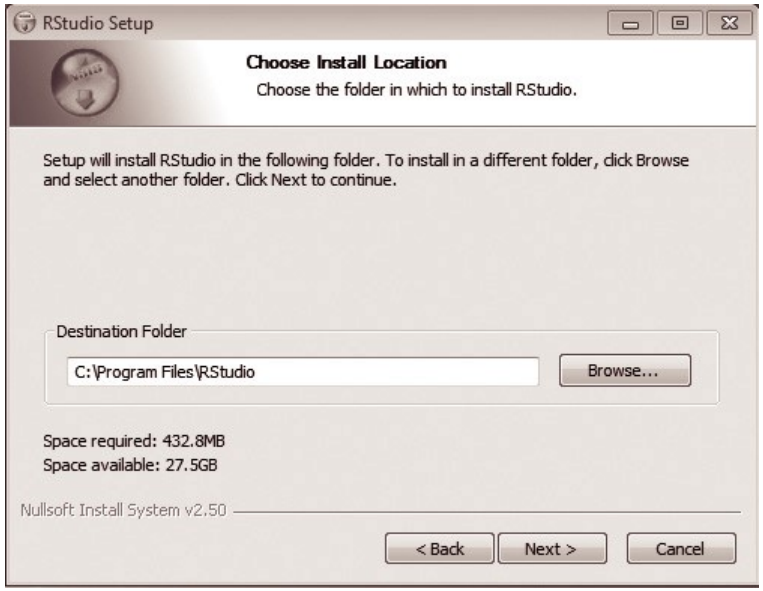


Fig. 1.9 Selecting the installation folder

4. Click Next to continue and a dialog box asking you to select the Start menu folder opens as shown in Fig. 1.10. Its is advisable to create your own folder to avoid any possible confusion and click on Install button to install RStudio.

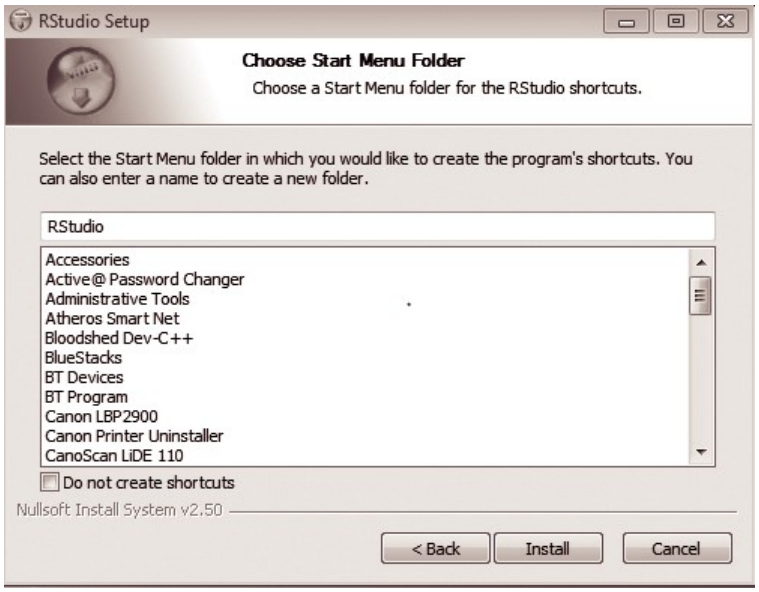


Fig. 1.10 Choosing the Start menu folder

4. After completion of installation, the following window as shown in Fig. 1.11 appears. Click on ‘Finish’ to exit.

Fig. 1.11 Installation complete

Install RStudio in Linux

- 1. First install the core debain command for installation of the debain version of RStudio.
    - >sudo apt-get install gdebi-core
- 2. Using the wget command fetch the debain version of RStudio.
    - >wget https://download1.rstudio.org/rstudio-1.0.136-amd64.deb
- 3. After fetching RStudio, the following commands install RStudio with the standard packages.
    - >sudo gdebi –n rstudio-1.0.136-amd64.deb
- 4. After installing RStudio, remove the installation file for saving disk space.
    - >rm rstudio-1.0.44-amd64.deb

Programming with R

- To begin with R, one needs to start working with R console where all the action takes place. Figure 1.12 gives a look and feel of the R console.
- R console is an execution window where users can execute R commands.
- Users need to type the required action on the command prompt and upon pressing the Enter key, R interprets the action typed by the user, executes the same, and gives the answer.

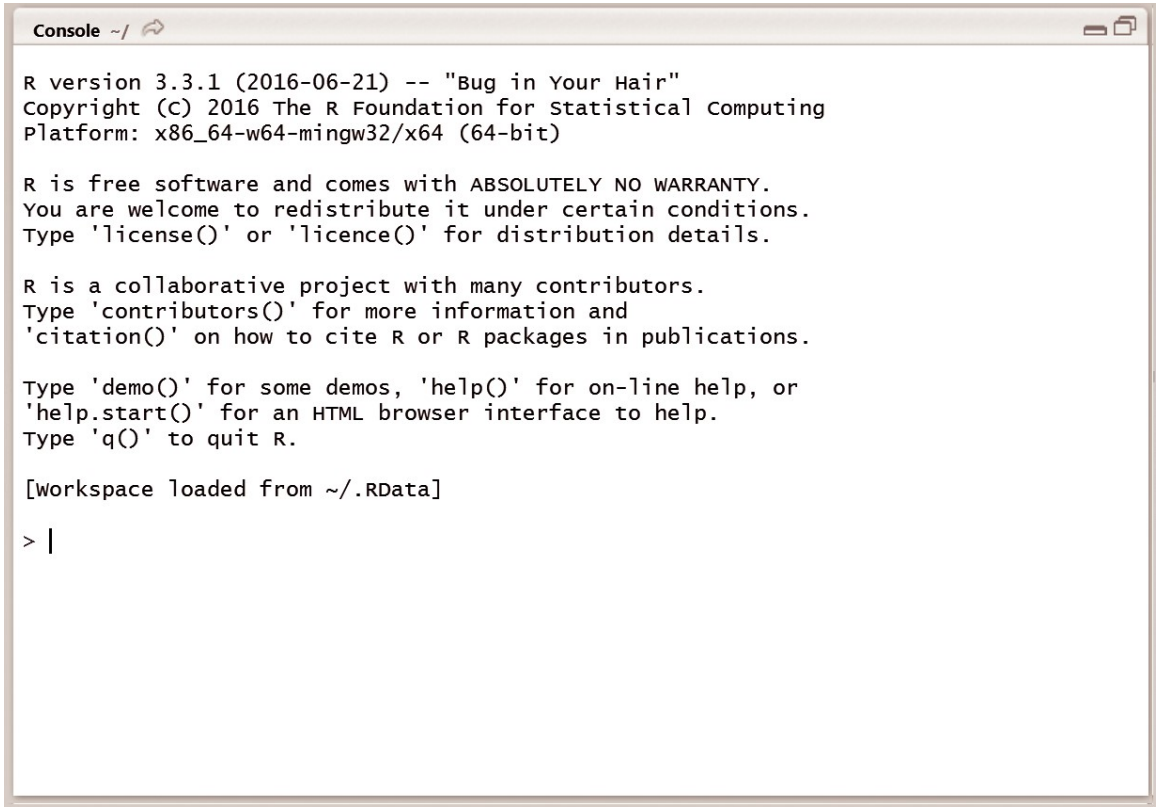


Fig. 1.12 R console

Basic Datatypes in R

There are six basic datatypes in R.

Logical	Numeric	Integer
Complex	Character	Raw

- As shown in the code below, the *class* function may be used to retrieve the datatype of the data contained in a variable.
- The *typeof* function may also be used to do the same, but it returns a different output, “double”, only in the case of the numeric datatype.

```
v1 <- TRUE
v2 <- 18311
v3 <- 18311L
v4 <- 1 + 8i
v5 <- "Roll Number"
v6 <- charToRaw("Roll")

print(class(v1))
print(class(v2))
print(class(v3))
print(class(v4))
print(class(v5))
print(class(v6))
```

```
[1] "logical"
[1] "numeric"
[1] "integer"
[1] "complex"
[1] "character"
[1] "raw"

** Process exited
```

```
print(typeof(v1))
print(typeof(v2))
print(typeof(v3))
print(typeof(v4))
print(typeof(v5))
print(typeof(v6))
```

```
[1] "logical"
[1] "double"
[1] "integer"
[1] "complex"
[1] "character"
[1] "raw"
```

Basic Datatypes in R : *logical*

logical – This is a datatype that can store either of the two values, logical **TRUE** or logical **FALSE**.

```
var1 <- TRUE
var2 <- FALSE
cat("The datatype of var1 is :",class(var1),"\n")
cat("The datatype of var2 is :",class(var2),"\n")

a <- 2
b <- 3
var3 <- a>b
cat("The datatype of var3 is :",class(var1),"\n")
```

```
The datatype of var1 is : logical
The datatype of var2 is : logical
The datatype of var3 is : logical
```

Basic Datatypes in R : *numeric*

numeric – This is a datatype that can store whole numbers and decimal (floating point) numbers.

This is the default computational datatype in R.

main.r +

```r
var1 <- 39
var2 <- 39.3939

cat("The datatype of var1 is :",class(var1),"\n")
cat("The datatype of var2 is :",class(var2),"\n")

cat("The datatype of var1 is (using typeof) :",typeof(var1),"\n")
cat("The datatype of var2 is (using typeof) :",typeof(var2),"\n")
```

```
The datatype of var1 is : numeric
The datatype of var2 is : numeric
The datatype of var1 is (using typeof) : double
The datatype of var2 is (using typeof) : double
```

Basic Datatypes in R : *integer*

integer – This datatype stores integers.

To initialize an integer variable, a suffix **L** has to be added. Not doing so will initialize the variable with the numeric datatype, as shown below.

main.r +

```r
var1 <- 39
var2 <- 39L
cat("The datatype of var1 is :",class(var1),"\n")
cat("The datatype of var2 is :",class(var2),"\n")
```

```
The datatype of var1 is : numeric
The datatype of var2 is : integer
```

Converting *numeric* to *integer*

numeric variables can be type-casted into **integer** variables using the ***as.integer()*** function. This helps initialize variables as integer variables and not as numeric variables, whenever necessary.

main.r +

```r
var1 <- 39
var2 <- as.integer(var1)

cat("var1  - Type : ",class(var1)," - Value :",var1,"\n")
cat("var2  - Type : ",class(var2)," - Value :",var2,"\n")
```

```
var1  - Type :  numeric  - Value : 39
var2  - Type :  integer  - Value : 39
```

```r
var1 <- 39.243567
var2 <- as.integer(var1)

cat("var1  - Type : ",class(var1)," - Value :",var1,"\n")
cat("var2  - Type : ",class(var2)," - Value :",var2,"\n")
```

```
var1  - Type :  numeric  - Value : 39.24357
var2  - Type :  integer  - Value : 39
```

Basic Datatypes in R : *complex*

complex – This datatype stores complex/imaginary variables.

Complex numbers can be initialized in two ways as shown below.

```
var1 <- 3 + 9i
var2 <- complex(real = 3, imaginary = 9)

cat(var1,", type :",class(var1),"\n")
cat(var2,", type :",class(var2),"\n")
```

```
3+9i , type : complex
3+9i , type : complex
```

complex variables can also be initialized from **numeric** or **integer** variables as shown below, using the ***as.complex()*** function.

```
var1 <- -1
var2 <- as.complex(var1)

cat(var1,", type :",class(var1),"\n")
cat(var2,", type :",class(var2),"\n")
```

```
-1 , type : numeric
-1+0i , type : complex
```

Operations on *complex* variables

```r
var1 <- 3 - 9i

#1. is.complex() - used to check if a variable is of the complex datatype
cat("Is var1 complex?",is.complex(var1),"\n")

#2. Re() and Im() - used to print the real and imaginary parts of a complex number, respectively
cat("Real part of var1 : ",Re(var1),"\n")
cat("Imaginary part of var1 : ",Im(var1),"\n")

#3. Mod() and Arg() - used to print the modulus and argument of a complex number ( polar form )
cat("Modulus of var1 : ",Mod(var1),"\n")
cat("Argument of var1 : ",Arg(var1),"\n")

#4. Conj() - used to print the complex conjugate of the given complex number
cat("Complex conjugate of var1 : ",Conj(var1),"\n")
#5. The square root of a negative number can be calculated only if it is complex.
#So, if you would like to compute the square root of a negative integer, typecast it to complex using as.complex()
cat("Square root of var1 : ",sqrt(var1),"\n")
cat("Square root of -1   : ",sqrt(as.complex(-1)))
```

```
Is var1 complex? TRUE
Real part of var1 :  3
Imaginary part of var1 :  -9
Modulus of var1 :  9.486833
Argument of var1 :  -1.249046
Complex conjugate of var1 :  3+9i
Square root of var1 :  2.498683-1.800949i
Square root of -1   :  0+1i
```

Basic Datatypes in R : *character*

character – This datatype stores characters and strings.

```
a <- 'i'
b <- 'ide'
print(class(a))
print(class(b))
```

```
[1] "character"
[1] "character"
```

numeric, integer or logical variables can be converted into **character** variables using the ***as.character()*** function.

main.r +

```
1  var1 <- 3 + 9i
2  var2 <- 3L
3  var3 <- 3.09
4  var4 <- TRUE
5
6  print(as.character(var1))
7  print(as.character(var2))
8  print(as.character(var3))
9  print(as.character(var4))
```

```
[1] "3+9i"
[1] "3"
[1] "3.09"
[1] "TRUE"
```

Basic Datatypes in R : *raw*

raw – This datatype holds raw bytes, i.e, it stores characters as raw bytes (hexadecimal representation of ASCII Code of the character)

ain.r +

```
var1 <- charToRaw("abcd")
cat("Value - ",var1,"\n")
cat("Type - ",typeof(var1))
```

```
Value -  61 62 63 64
Type -  raw
```

raw variables may be initialized from a **character** variable (remember, the *character* datatype comprises of characters and strings, both) using the ***charToRaw()*** function, as shown in the above example.

In the above example, the string **"abcd"** is decoded into characters, and is stored in memory as a sequence of hex representations of ASCII Codes of its constituent characters, i.e. raw bytes, and the same is printed as the output.

Some Basic Programs

1. Basic Arithmetic in R Console

To calculate the sum of two numbers, say 1 and 2, the programmer needs to type 1 + 2 in the command prompt of the console. (Fig. 1.13).

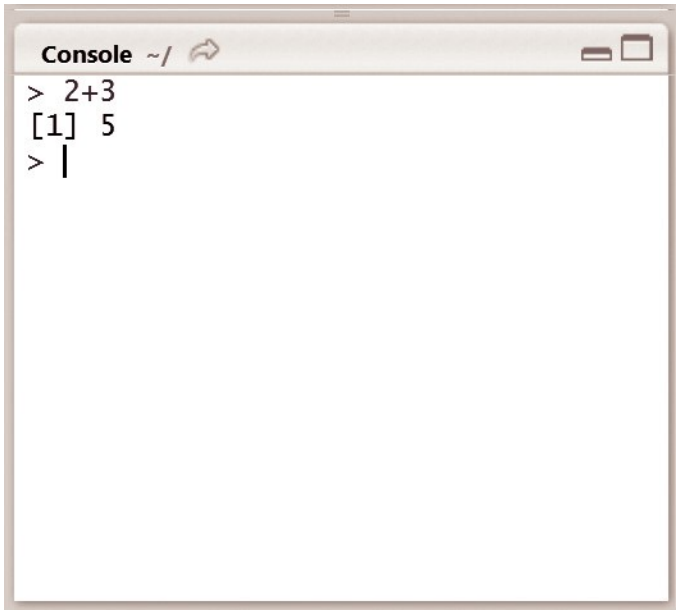
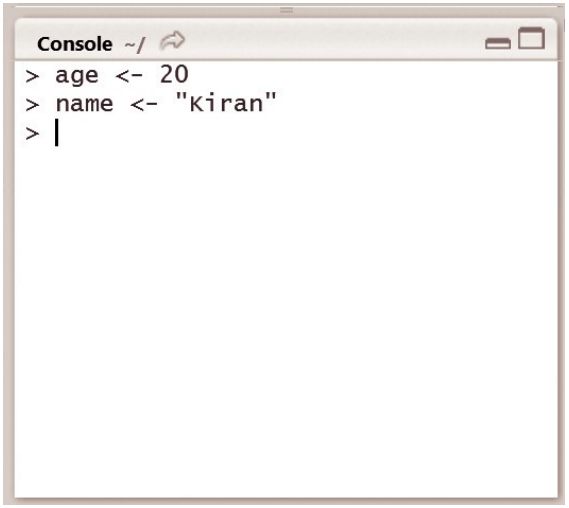


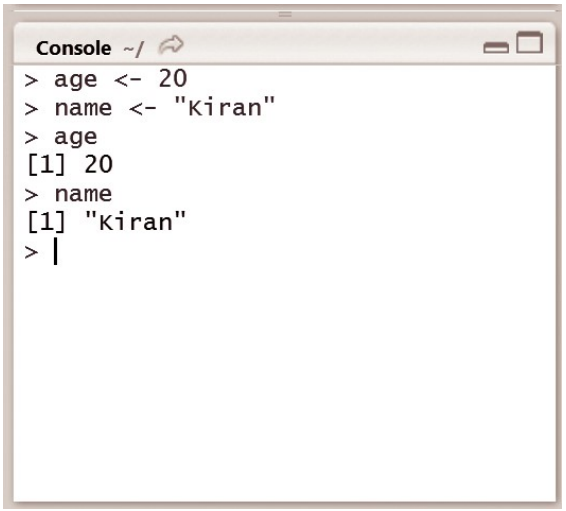
Fig. 1.13 Basic arithmetic with R

2. Declaring Variables in R Console



- Later, to retrieve the values of the variables, the user just needs to enter the name of the variable in the command prompt and R returns the stored value of the variable.

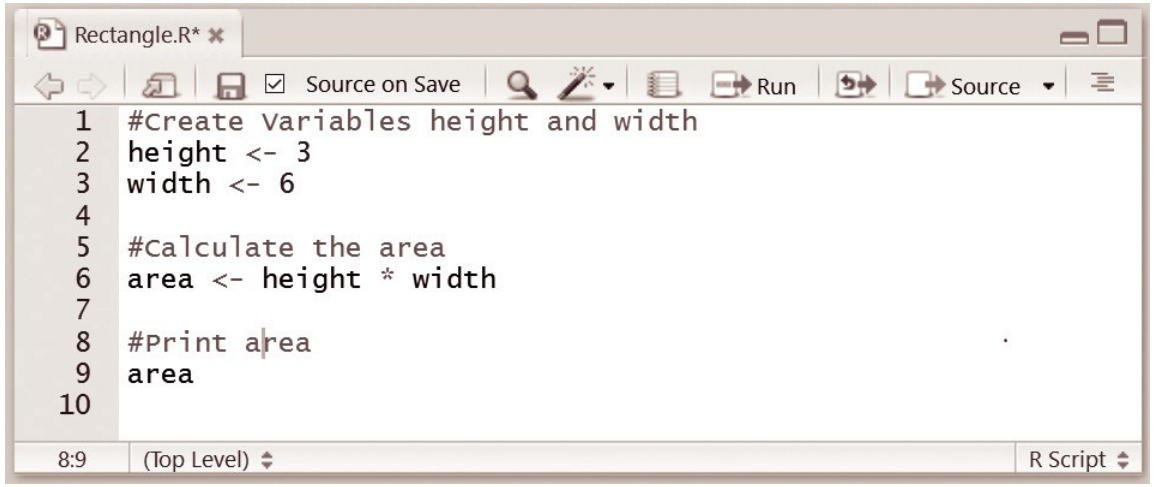
3. Retrieving the values of stored variables in R Console



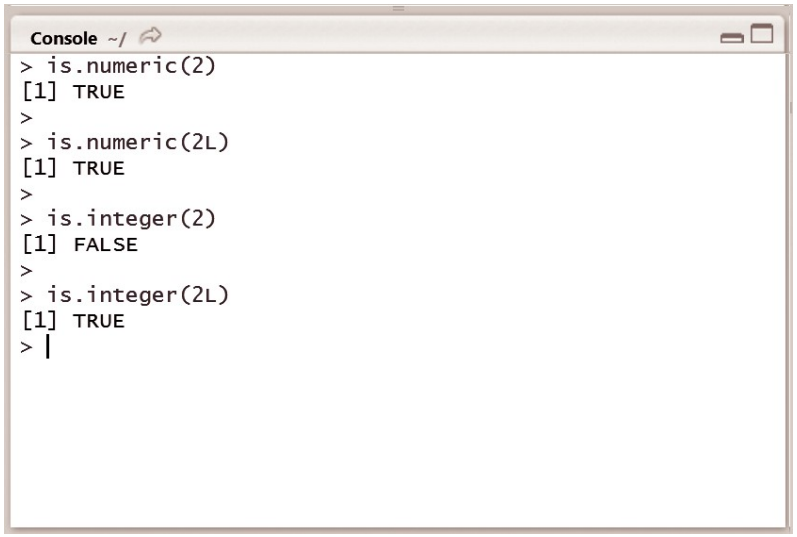
- As users assign values to variables in R console, these get accumulated in the R workspace. R workspace stores all the variables and their metadata information. Users can access the objects in the workspace using the *ls*() function. The usage of *ls*. *ls* lists all the stored variables that are created during the particular R session.

4. Comments

- The comments in R start with the # symbol.If the user wants to run this script again, the lines starting with # do not affect R's execution



5. Checking if a given input is numeric or integer using an R Function



- R provides a function for the programmers to check whether a given input is numeric or integer. This is known as the *is-dot-function*. Here, we use two functions—*is.numeric*() and *is.integer*()—as shown in Fig.
- 1.25, to check if a variable is a numeric or integer. In this particular case, it appears that both are *numeric* and the results also demonstrate the fact that all *integer* variables are *numeric*, but not all *numeric* variables are *integers*.

vectors

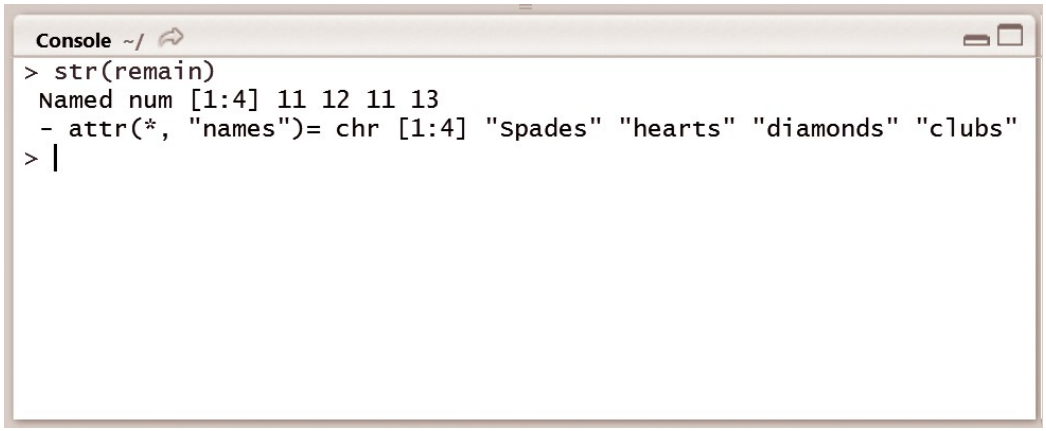
- Vectors are the most basic R data objects. A vector is a sequence of data elements of the same data type.
- There are six types of atomic vectors—*logical*, *integer*, *double*, *complex*, *character*, and *raw*.
- Programmers can create *character* vectors, *numeric* vectors, *logical* vectors, and many more

creating and naming vectors

- A function *c*() is used to create a vector in R, which further allows users to combine values into a vector
- Figure 1.33 shows the various options for creating a new vector.

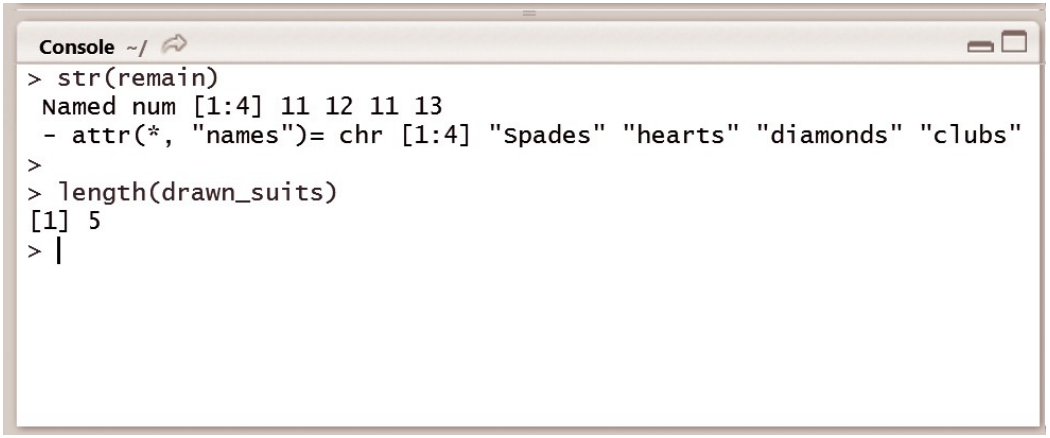
```
Console ~/
> #OPTION - 1
> #Store how much cards of each suit remain after you drew the 5 cards
> remain<-c(11,12,11,13)
>
> #Print Vector remain
> remain
[1] 11 12 11 13
>
> #Attach labels to the vector elements
> suits<-c("Spades","hearts", "diamonds","clubs")
>
> #Name the vector
> names(remain) <- suits
> #Print remain
> remain
  Spades   hearts diamonds    clubs
      11       12       11       13
>
> #OPTION - 2
> remain <- c(Spades=11, hearts=12, diamonds=11,clubs=13)
>
> #OPTION - 3
> remain <- c("Spades"=11, "hearts"=12, "diamonds"=11,"clubs"=13)
>
```

- R allows programmers to name the data elements of the vector. R uses the *names*() function to name the vector elements.



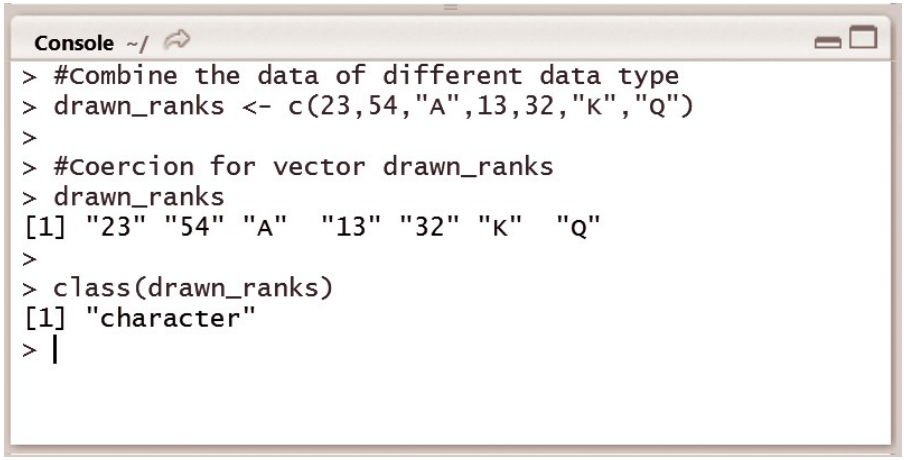
Vector Length

- A single variable in R is actually a vector of length 1.
- R provides a function utility named *length*() to determine the length of the vector.



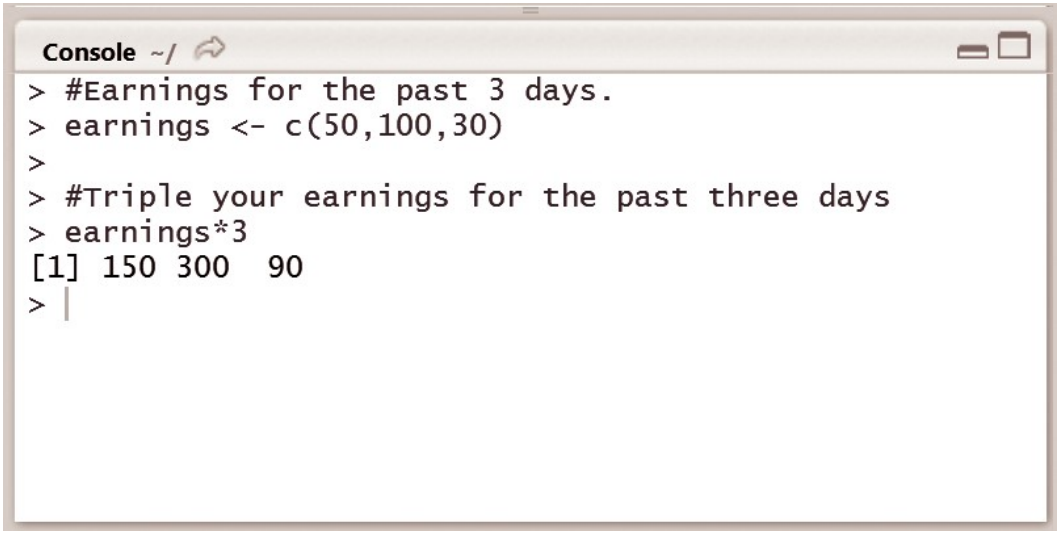
Coercion of Vector Elements

- A vector in R can only hold elements of the same type, which means that users cannot have a vector that contains both *logical* and *numeric* data types.
- If the user wants to build a mixed vector that contains both integers and characters, then automatically, R performs *coercion* to make

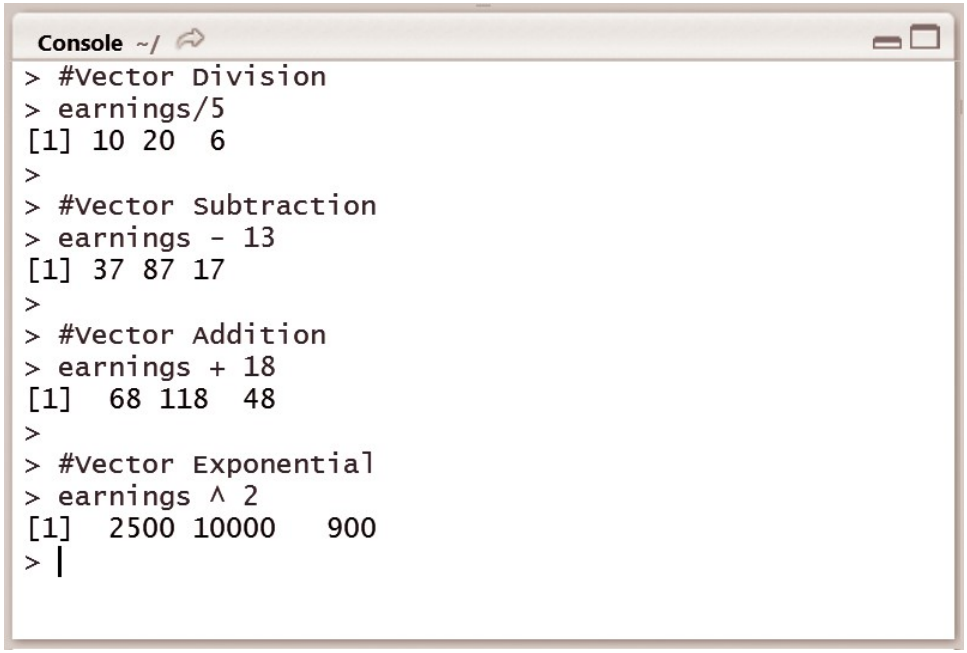


vector arithmetic

- R allows programmers to perform arithmetic calculations with vectors and the operations are applied element by element
- Fig show this simple vector arithmetic by multiplying the vector variable *earnings* by 3, where the vector *earnings* has been created with the three days of earnings in `.

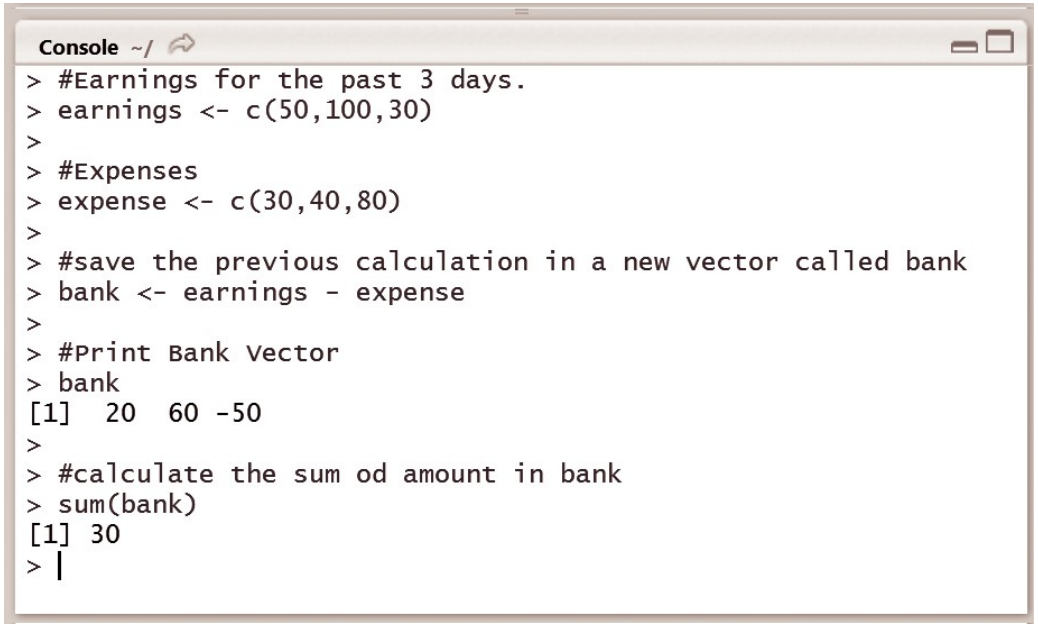


- Figure illustrates the division of the *earnings* vector by 5, subtracting 13 from the *earnings* vector elements, adding 18 to the vector elements, and finding the exponential of 2 of the elements of *earnings*



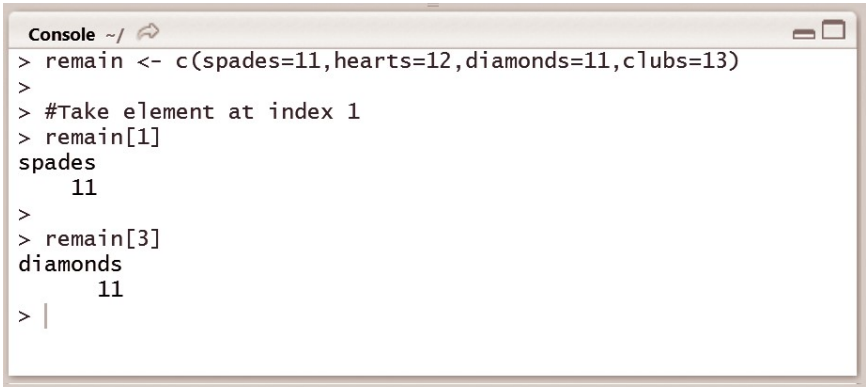
Enumerate multiplication and division operations between matrices and vectors in R console.

- the multiplication of two vectors can result in a single scalar or a matrix.
- Figure shows users how to check their bank accounts' status after these three days of earnings in the city of Bangalore.

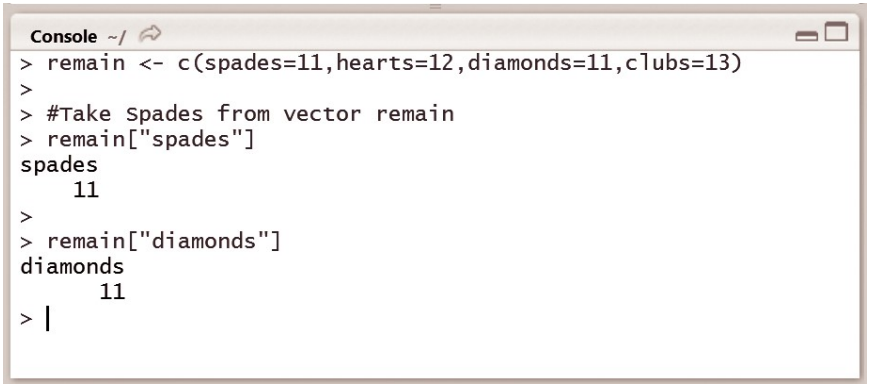


vector Subsetting

- Vector subsetting is used to break vectors into selected parts and derive a new vector known as a subset of the original vector.
- Figure illustrates the usage of vector subsetting by considering users want to select the first element from the vector, corresponding to the number of spades that are left. Users can use square brackets for this.
- Programmers write remain [1], where the number 1 inside the square brackets indicates that users want to get the first element from the *remain* vector.



- If users are dealing with named vectors, they can also use the names to perform the selection. Instead of using the index 1 to select the first element, users can use the name *spades*.



Matrices

- Matrices are the R objects in which the elements of the same atomic type are arranged in a two-dimensional rectangular layout

The basic syntax for creating a matrix in R is

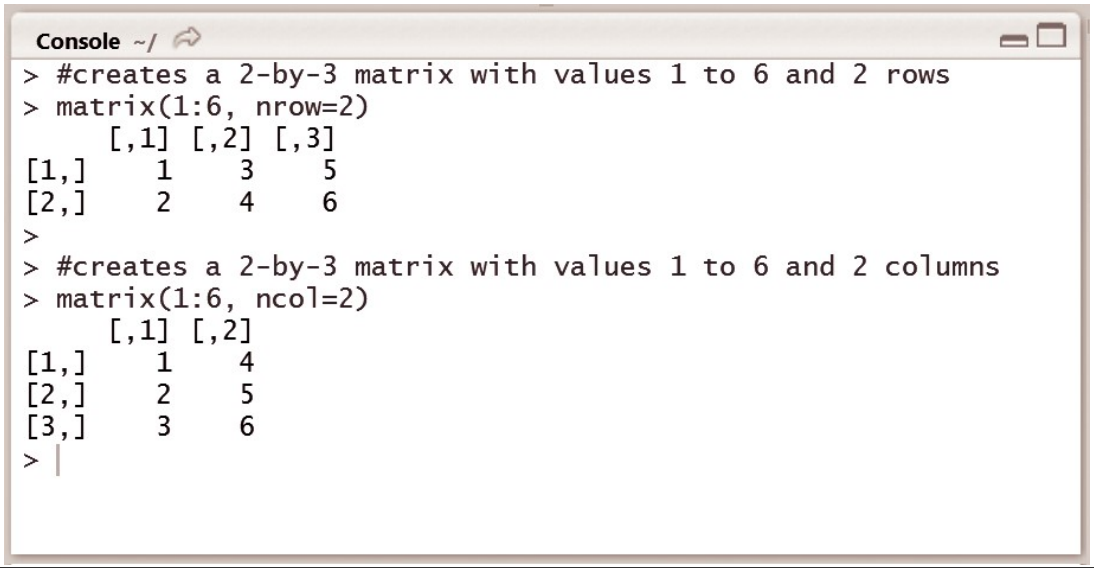
matrix(data, nrow, ncol, byrow, dimnames)

The following are the attributes used in matrix

1. data is the input vector which becomes the data elements of the matrix.
2. nrow is the number of rows to be created.
3. ncol is the number of columns to be created.
4. byrow is a logical clue. If TRUE then the input vector elements are arranged by row.
5. dimnames are the names assigned to the rows and columns.

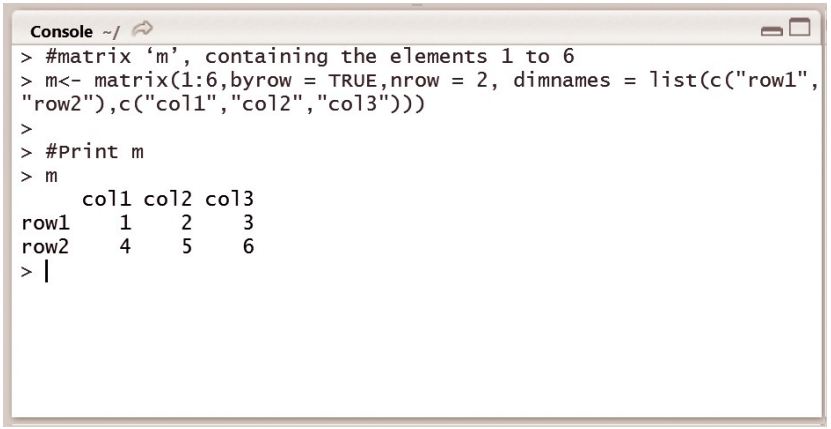
creating matrices

- As with vectors, a matrix can contain only one atomic vector type.
- To build a matrix, users use the *matrix* function. Most importantly, it needs a vector containing the values that users want to place in the matrix, and at least one matrix dimension. Users can choose to specify the number of rows or columns.
- Figure shows the creation of a 2-by-3 matrix containing the values 1 to 6, by specifying the vector and setting the *nrow* argument to 2.



naming matrices

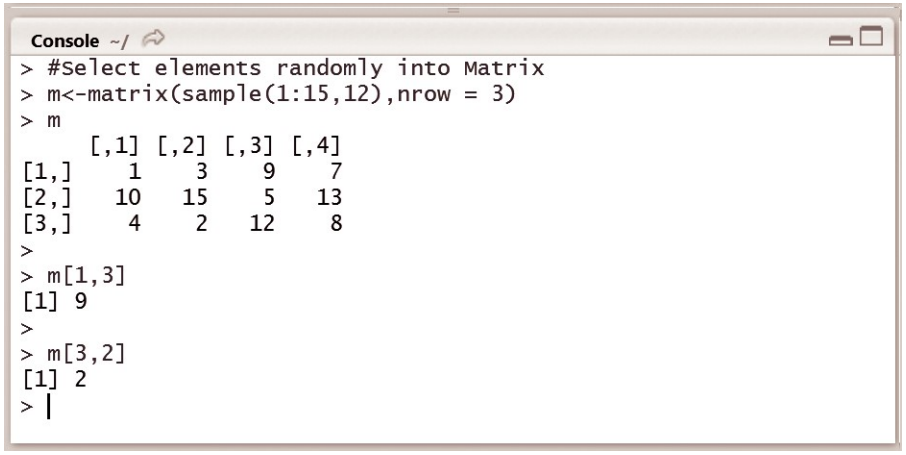
- as with vectors, there are also one-liner ways of naming matrices while we are creating it. We can use the *dimnames* argument of the matrix function for this. We need to specify a list that has a vector of row names as the first element and a vector of column names as the second element as shown in Fig



Matrix Subsetting

- to select a single element or entire parts of a matrix to continue the analysis with. Again, we can use square brackets for this
- If we want to select a single element from this matrix, we will have to specify both the row and the column of the element of interest.
- If we want to select the number 15, located at the first row and the third column we type *m[1,3]*.

- As is evident, the first index refers to the row and the second one refers to the column.
- Likewise, to select the number 1, at row 3 and column 2, we write *m[3,2]*. Notice that the results are single values, that is, vectors of length 1.



•Now, what if we want to select an entire row or column from this matrix? We can do this by missing either the row or column index within the square brackets.

• Instead of writing [3, 2] to select the element at row 3 and column 2, if we write [3,], this would select all the elements that are in row 3

Arrays

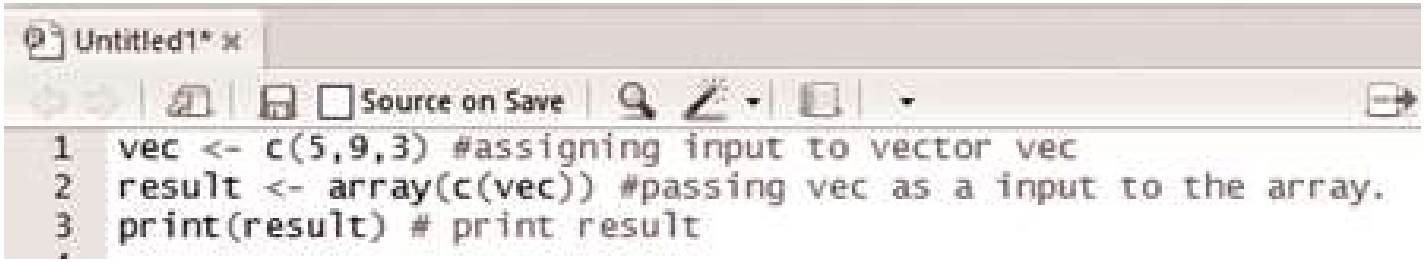
- An array is a collection of similar data of the same type, for example, numeric.
- R uses a vector to create an array only if there is a dimension vector with dim attributes.
- The *array*() function is used to create an array
- The basic syntax for creating an array in R is

array(data, dim, dimname)

where,

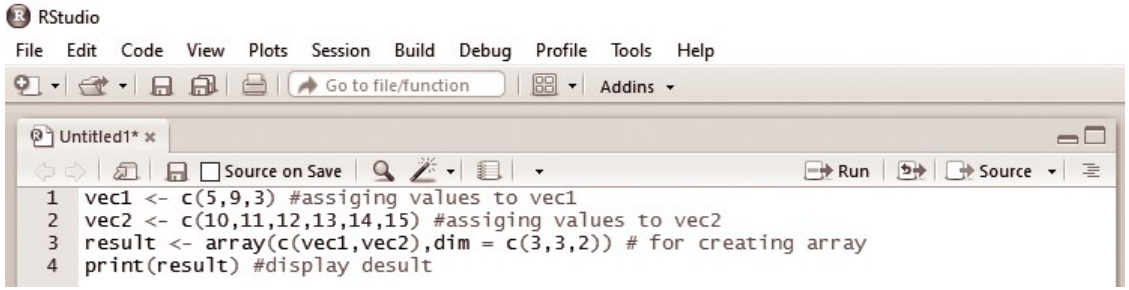
- data is the input values to the vector
- dim is used to create the dimension
- dimname is used to assign a name to dim

- Figure shows a program creating an array of 3 elements.
- First, values are assigned to the object *vec.*
- Then an array is created with *vec* passed as an input.
- Finally the array is displayed



Write a program to create an array of 3×3 matrixes with 3 rows and 3 columns.

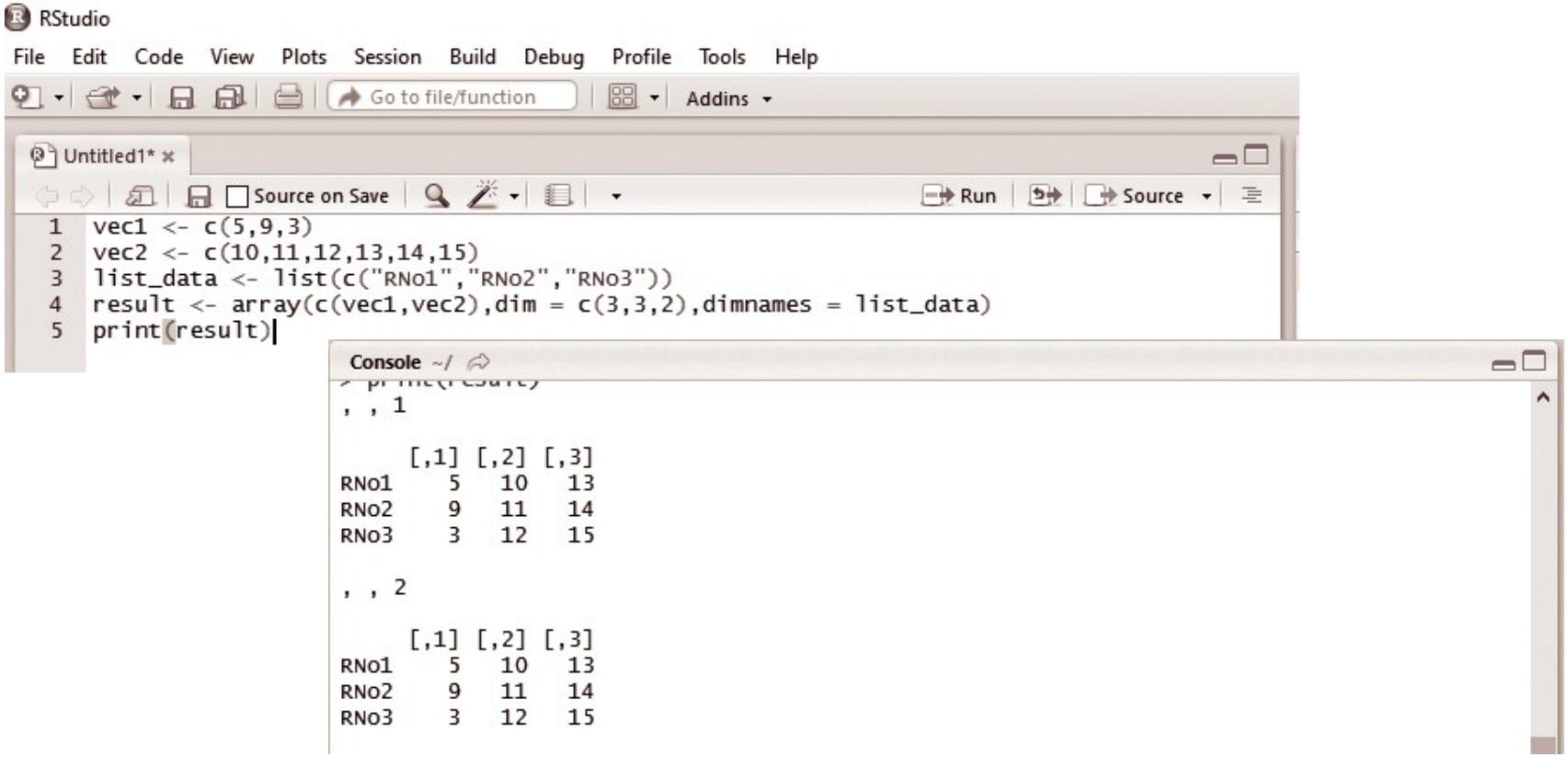
- shows the code to create an array of matrices. First values are assigned to objects *vec1* and *vec2*. Then the array is created and displayed in matrix form.




```
, , 1

     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

, , 2

     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15
```

Write a program to assign dimension name to an array of 3×3 matrixes.

- . First, values are assigned to objects *vec1* and *vec2*. Then an array is created and names RNo1, RNo2, and RNo3 are assigned to the array matrix. Finally the dimension names are displayed in a matrix form.



Class

- in R language is considered as an object and every object is associated with a class.
- To create a class in R using the *class* keyword.

S3 class

- S3 classes are used to implement an object-oriented (OO) concept known as generic function OO.
- In S3 class, we use the concept of list property, where a list is created and a list class is set and returned.

Write a program to create a class, object, and function.

RStudio

File Edit Code View Plots Session Build Debug Profile Tools Help

Go to file/function | Addins

Untitled4*

Source on Save | Run | Source

```
1  India <- function(Food=TRUE,myFavorite="fruits")
2  {
3    me <- list(
4      hasFood = Food,
5      favoriteFood = myFavorite
6    )
7
8    class(me) <- append(class(me),"India")#Assigning name to class
9    return(me)
10 }
11  obj <- India() # create funcation , object of the funcation
12  obj
13  obj$hasFood
14
15 GVeg <- India(Food = TRUE,myFavorite="Green Veg")
16 GVeg
17
```

```
> India <- function(Food=TRUE,myFavorite="fruits")
+ {
+   me <- list(
+     hasFood = Food,
+     favoriteFood = myFavorite
+   )
+
+   class(me) <- append(class(me),"India")#Assigning name to class
+   return(me)
+ }
> obj <- India() # create funcation , object of the funcation
> obj
$hasFood
[1] TRUE

$favoriteFood
[1] "fruits"

attr(,"class")
[1] "list"  "India"
> obj$hasFood
[1] TRUE
> GVeg <- India(Food = TRUE,myFavorite="Green Veg")
> GVeg
$hasFood
[1] TRUE

$favoriteFood
[1] "Green Veg"

attr(,"class")
[1] "list"  "India"
>
```

Part - II
Factors and Data Frames

Introduction to factors

- There are three kinds of variables in R programming:
  - Categorical
  - ordinal
  - interval

categorical variable

- A *categorical variable* is one that has two or more categories, but there is no intrinsic ordering to the categories.
- For example, gender is one categorical variable with male and female being the possible values.

ordinal variable

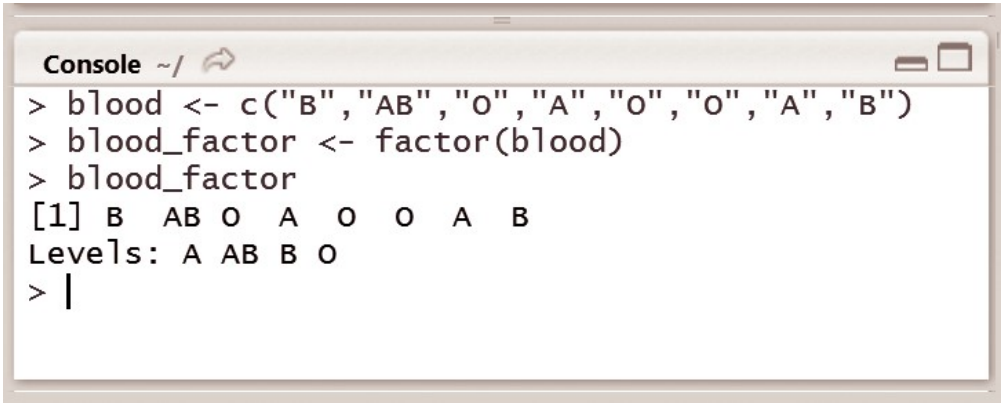
- An ordinal variable is a categorical variable, but there is an intrinsic order present for the values it can take.
- The economic status may be low, medium, or high and grades in college are S, A, B, C, D, E, F.

interval variable

- These variables are defined by specific ranges such as $10–$15, age in the range of 20–25, height ranging from 5 feet 5 inches to 5 feet 9 inches

factor

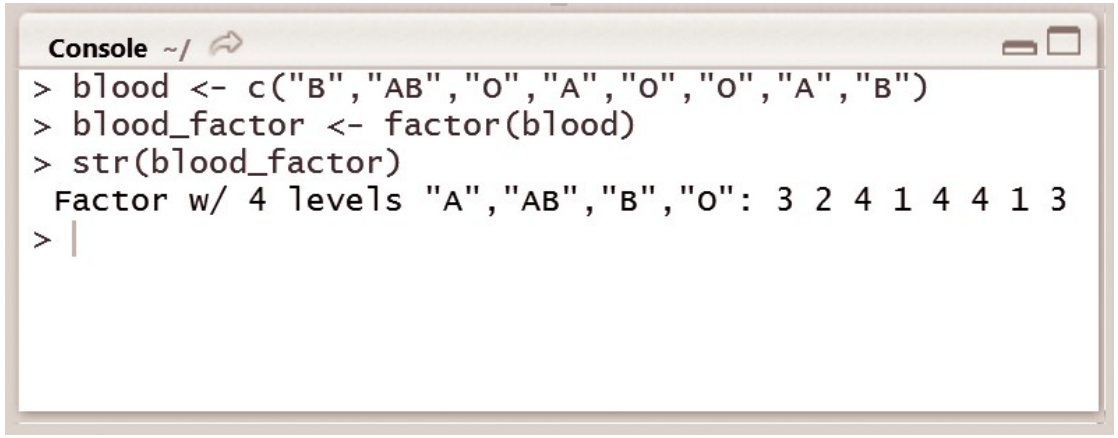
- A *factor* is a statistical data type that stores categorical variables, which is used in statistical modelling.
- Factors are stored as a vector of integer values with a set of character values.
- To convert this vector to a factor, the *factor*( ) function can be used as shown in Fig.



R performs two functions when the *factor*() function is called on a character vector:

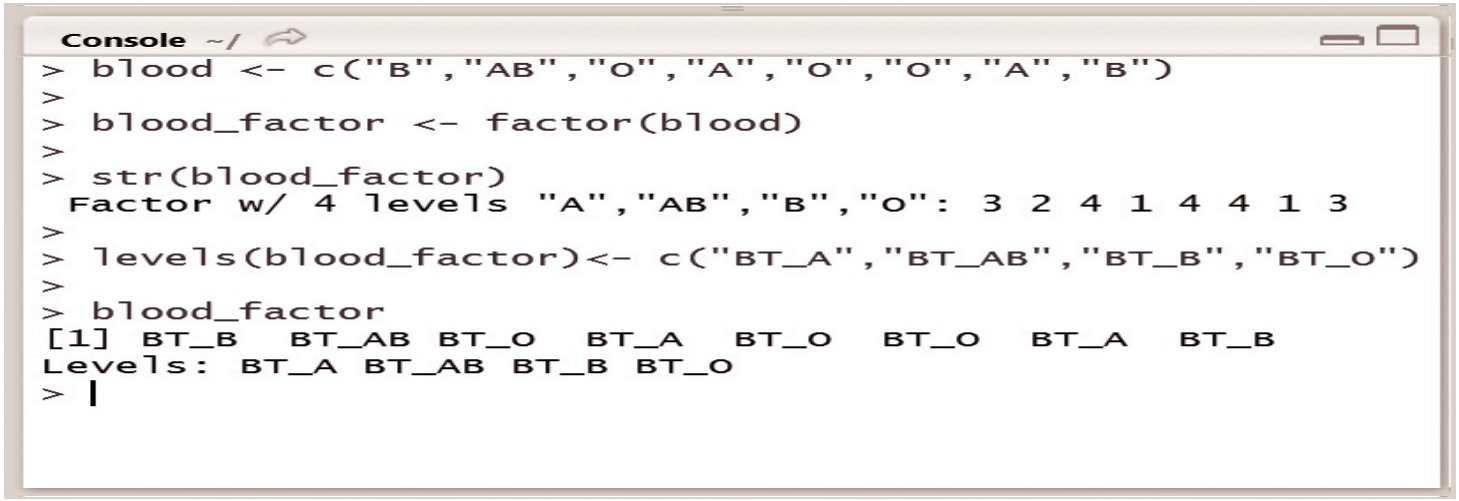
1. First, it scans through the vector to check the availability of different categories, R performs a check for categories of blood groups among "A", "AB", "B", and "O".

2. Further it sorts the levels alphabetically as can be observed from

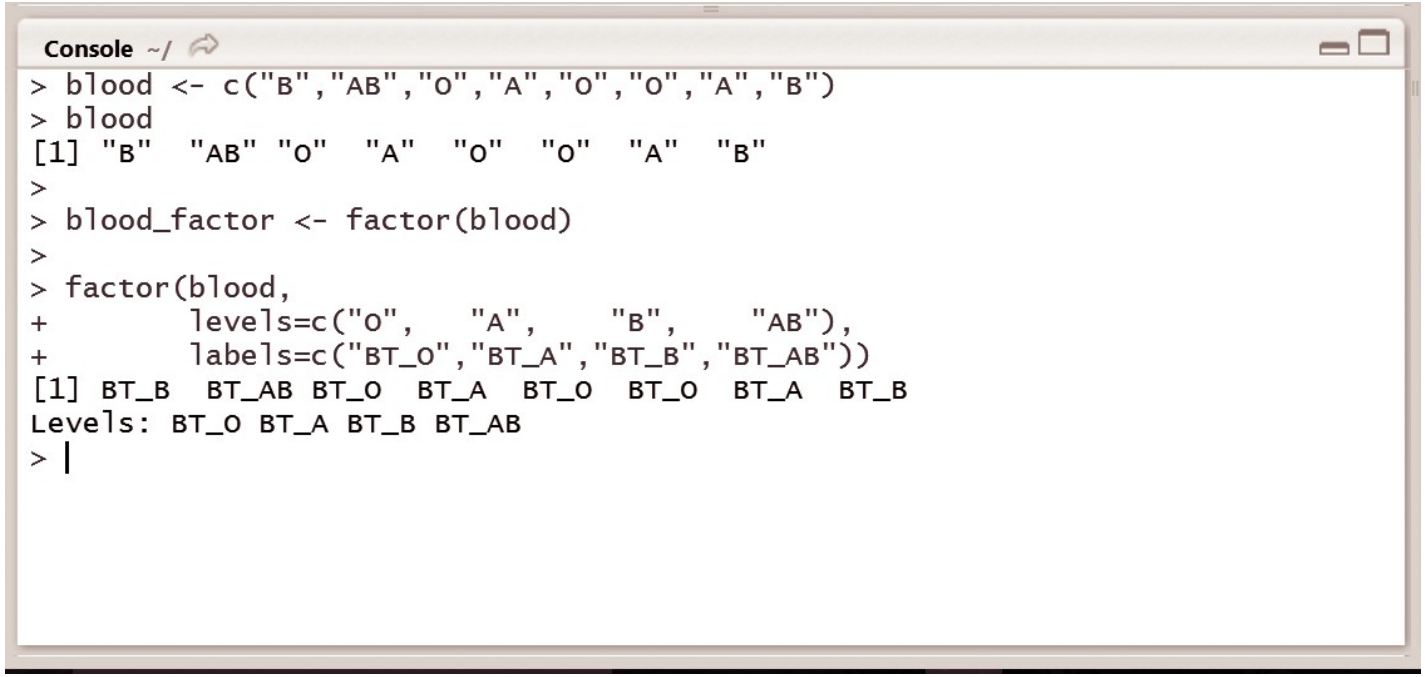


Factor Levels

- Factors are integer vectors where each integer corresponds to a category or a level.
- R allows programmers to perform this task using the function *levels*( ).
- In case a different ordering of the levels is required, the argument can be specified in the factor function.

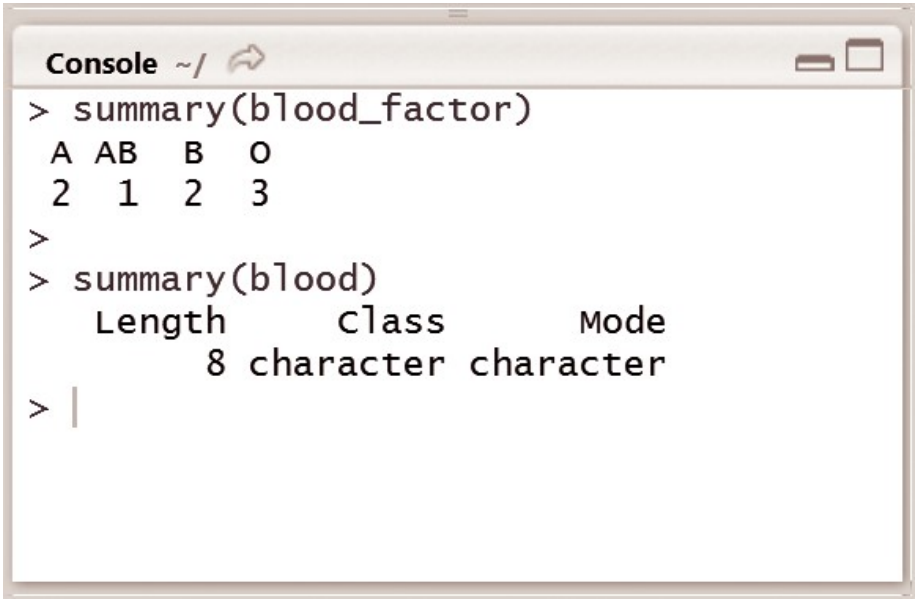


- To change the order of the levels, it is possible to manually specify the level names, instead of letting R choose them.
- To display the blood types as ‘BT_A’, ‘BT_AB’, ‘BT_B’, and ‘BT_O’, we can use the *levels*() function.
- We can specify the category names or levels by using the *labels* argument in the *factor*() function.



summarizing a Factor

- Summary is a generic function used to produce summaries of the results.
- For instance, in Fig, summary of the *blood_factor* object shows the 4 categories of blood groups A, AB, B, and O and the summary of the object *blood* shows us the number of elements in the vector (which is 8), the class, and the mode of that vector.

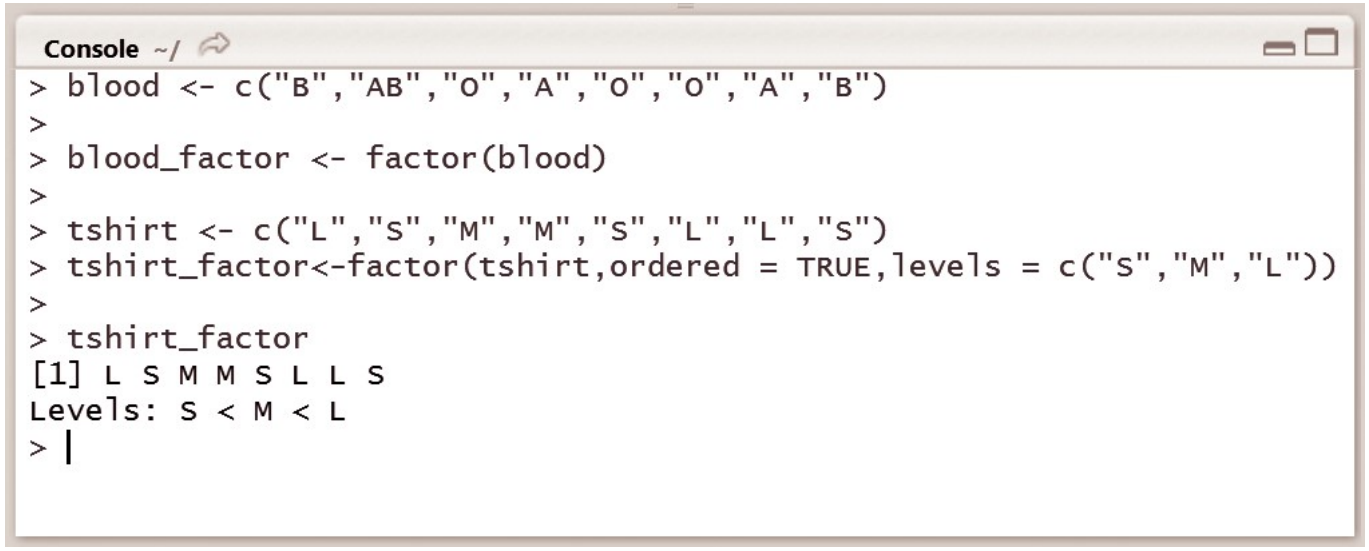


ordered Factors

- There are examples where a natural ordering exists; sometimes programmers have to deal with factors that do have a natural ordering among their categories.
- In the domain of categorical variables, there exists a difference between nominal categorical variables and ordinal categorical variables.
- A nominal categorical variable, as mentioned earlier, has no implied order.

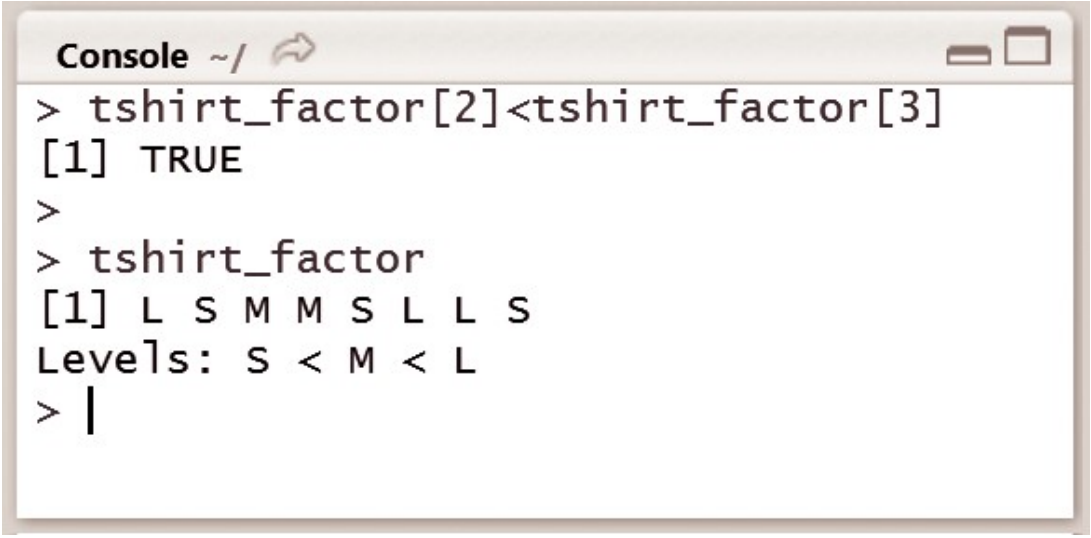
Write a command in R console to create a *tshirt_factor*, which is ordered with levels 'S', 'M', and 'L'. Is it possible to identify from the examples discussed earlier, if blood type 'O' is greater or less than blood type 'A'?

- Fig. we can see that it is difficult to identify whether blood type "O" is greater or less than blood type "A".



comparing ordered Factors

- It is possible to compare ordered factors in R.
- This is explained in Fig it is specified that L is indeed greater than M. R provides a way to impose this kind of order on a factor, thus making it an ordered factor. This is achieve by the *factor*() function, where the argument *ordered* is set to TRUE and the levels are specified in ascending order.



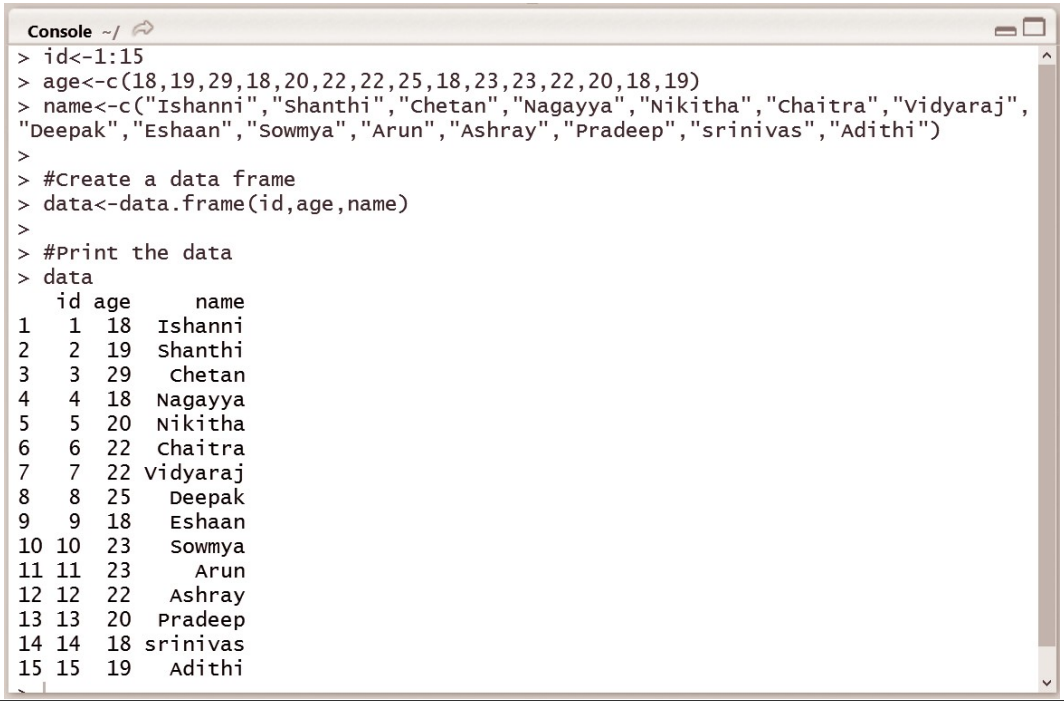
Data frame

- Data sets typically comprise observations, or instances, that have some variables associated with them.
- Data frame is a fundamental data structure to store data sets
- For example, consider a data set of five people,
- A matrix cannot be used to store such information in R, as name would be a character and age would be numeric and these will not fit in a matrix.

creating a Data Frame

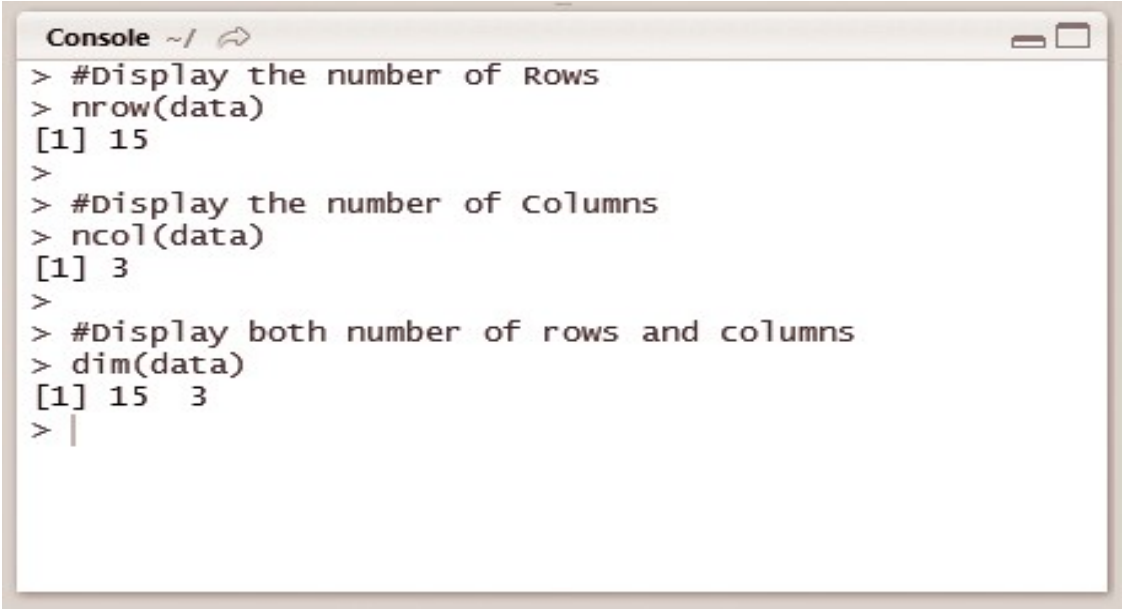
- A data frame is created using the function *data.frame*() by using a variable *data*.
- The parameters are the vectors created earlier, which are passed to the data frame as follows:

  data.frame(ID, age, name)
- The contents of the data frame can be displayed by printing the contents of the variable *data*, which looks like a spreadsheet.



Compute number of rows and columns

- The number of rows and columns in a data frame can be computed using the functions *nrow*() and *ncol*( )

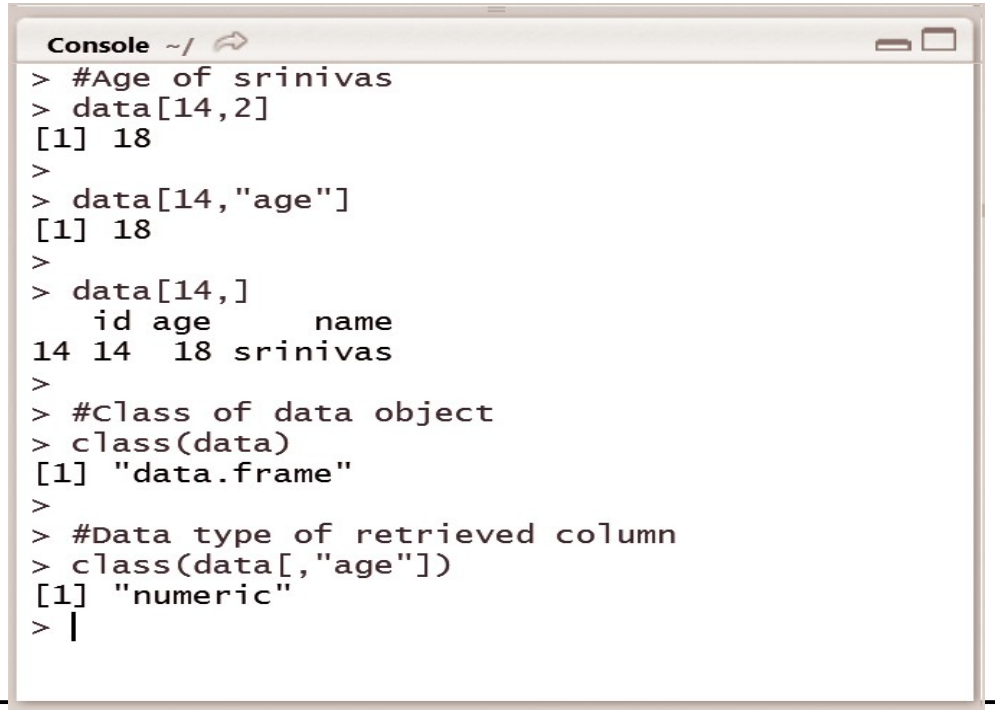


The number of rows and columns of a data frame can be visualized in an environment window

sub setting of Data Frames

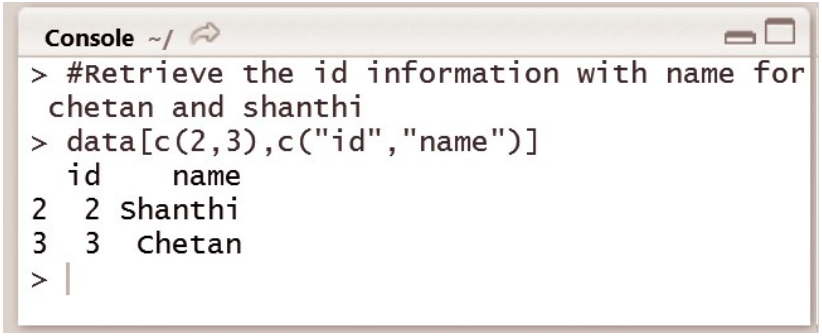
- The data frame combines the features of matrices and lists.
- To subset a data frame from both matrices and lists, the following syntax can be used;
  - Single brackets '[' are used for sub setting matrices
  - double brackets '[['
  - dollar sign '$' are used to select list elements.

Figure illustrates the process of subsetting a data frame by considering the previous data set in Fig. 2.9 containing 15 employee records



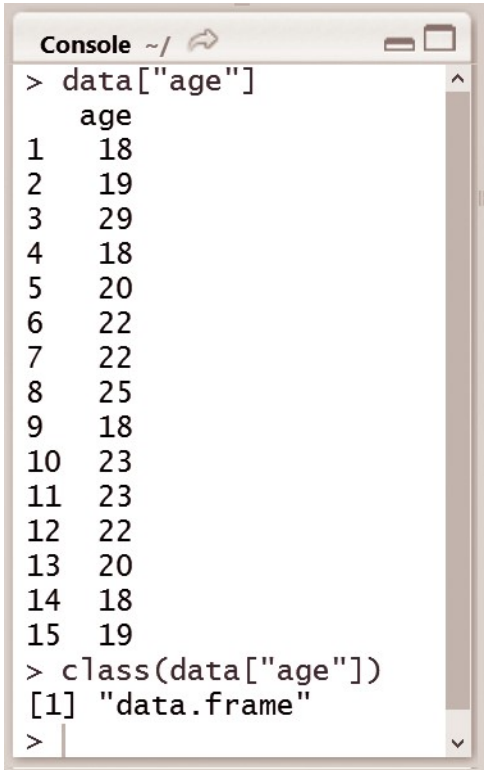
Write the command in R console to retrieve the name and id information from the 2nd and 3rd rows.

Figure illustrates retrieving data from rows 2 and 3 by selecting the name and id information of Chetan and Shanthi.



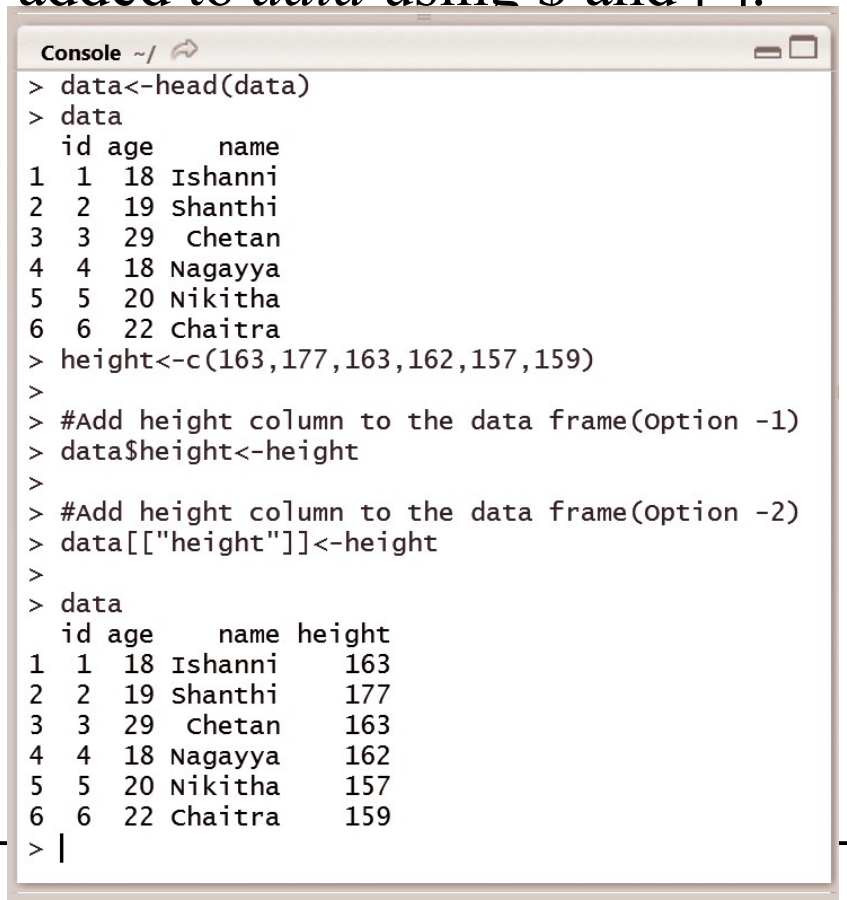
Write the command in R console to create a new data frame containing the 'age' parameter from the existing data frame. Check if the result is a data frame or not.

Single brackets can also be used to subset lists; this generates a new list containing only the specified elements. The result is still a data frame, which is a list, but containing only the "age" element as shown in Fig.

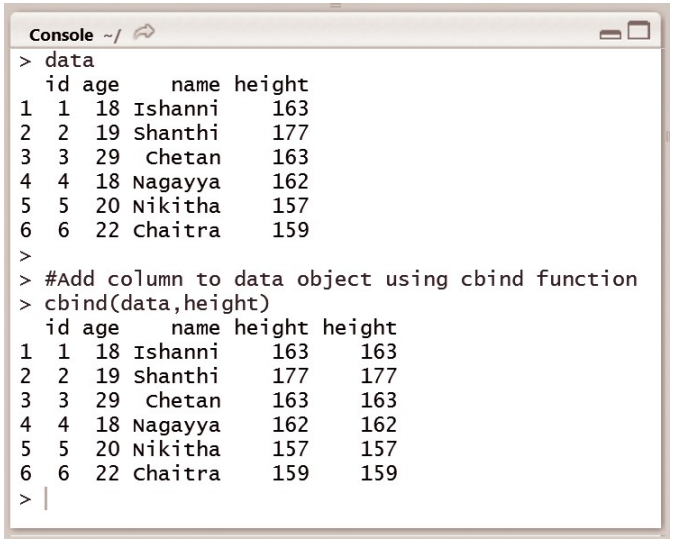


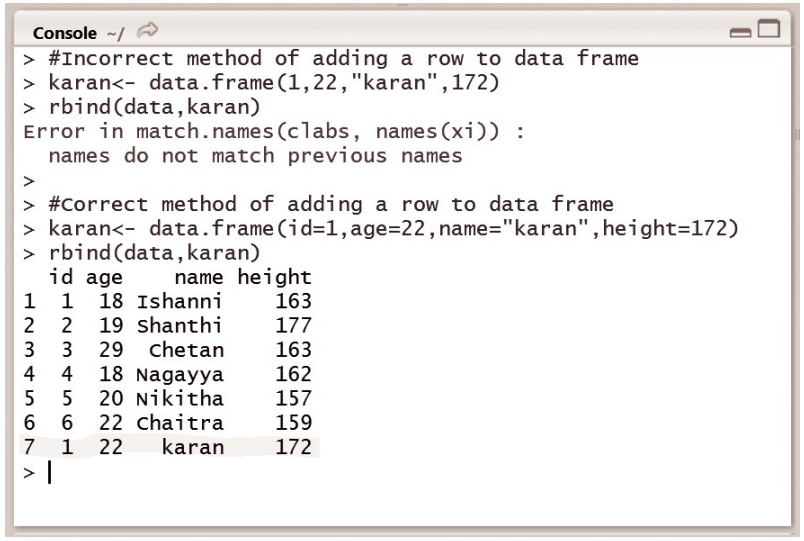
extending Data Frames

- To add a column, new variable, new row, or new observation to the existing data frame, the dollar sign or the double square brackets can be used. In Fig, a vector named '*height*' is created and is added to *data* using $ and [ ].



- The *cbind*() function can also be used to build and extend matrices. It works similar to the data frames shown in Fig.
- If *cbind*() works, than *rbind*() will work fine as well. Indeed, *rbind*() is used to add new rows to observations
- To add the information of another employee named Karan to the data frame, simply creating a vector with the *id*, *name*, *age*, and *height*, won't work, because a vector can't contain elements of different types. A new data frame needs to be created containing only a single observation, and that needs to be added to the existing data frame using *rbind*(). The usage of *cbind*() and *rbind*() are shown in Fig



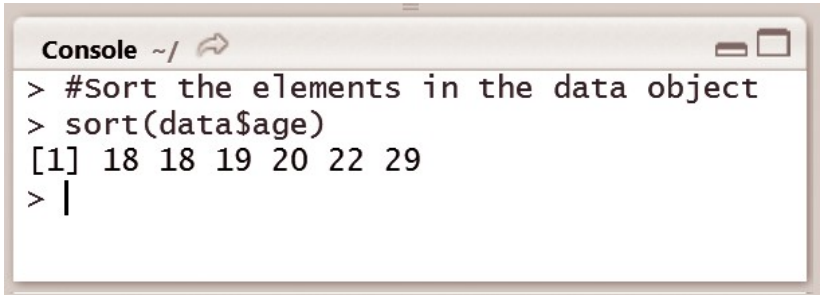


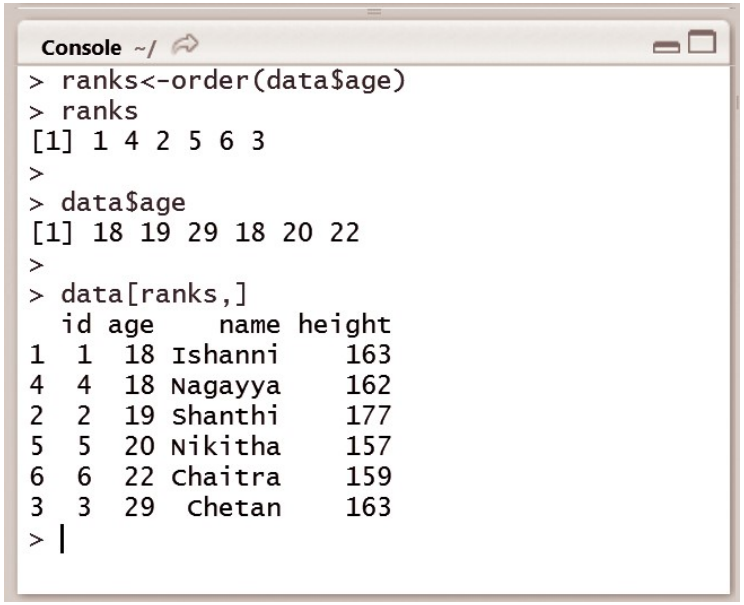
sorting Data Frames

- To sort the data frame by age, such that the youngest person is on top, and the oldest at the bottom.
- the *sort*() function can be used. This function sorts a vector into ascending or descending order.

Write the command in R console to display the values of *age* of *data* in ascending order using the *sort*() function.

- Fig. it can be observed that if the sorting is done based on the age column, then an ordered version of the age column is returned.





To change the ordering of the rows in the data frame, *order*() function can be used instead of the *sort*() function.

- To sort in descending order, the *decreasing* argument is set to TRUE