UNIT - III

Lists

- Introduction, Creating a List: Creating a Named List
- Accessing List Elements, Manipulating List Elements
- Merging Lists, Converting Lists to Vectors

Conditionals and Control Flow

- Relational Operators, Relational Operators and Vectors, Logical Operators
- Logical Operators and Vectors, Conditional Statements

Iterative Programming in R

- Introduction, While Loop, For Loop, Looping Over List

Functions in R

- Introduction, Writing a Function in R, Nested Functions
- Function Scoping, Recursion
- Loading an R Package, Mathematical Functions

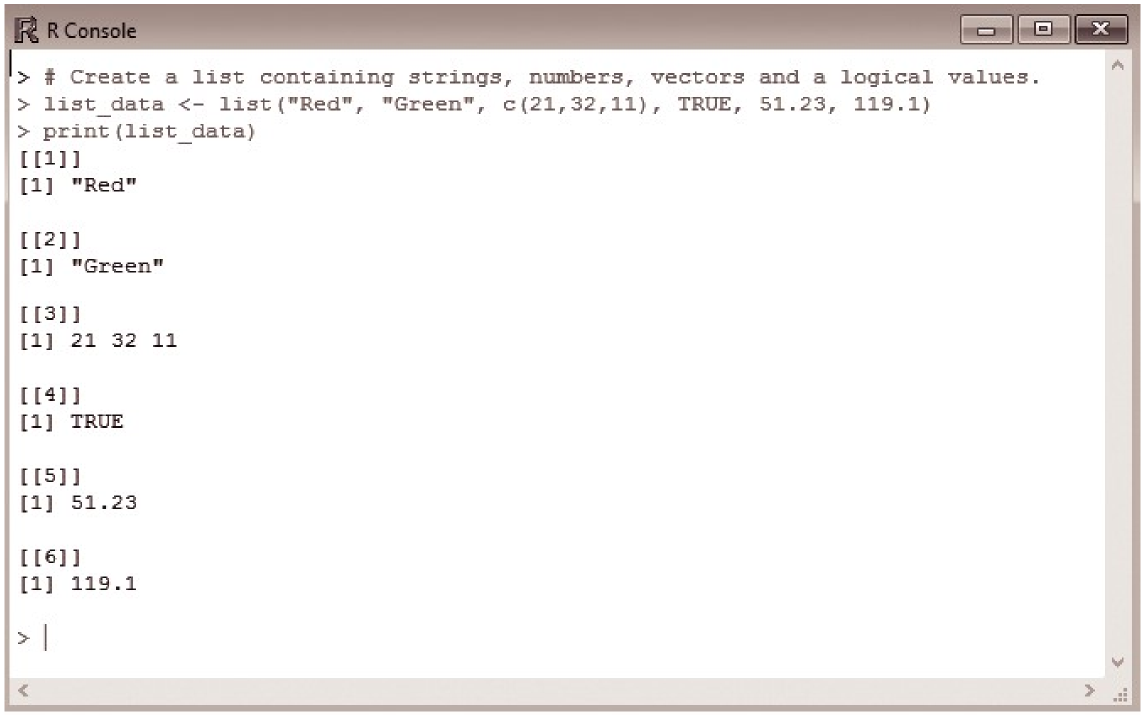
Part - I
Lists in R

List

- A list in R constitutes of different objects such as strings, numbers, and vectors.
- it can include another list within it.
- It can also include matrices and functions.

Creating a List

- A list can be created using the *list*() function.
- Figure demonstrates the creation of an anonymous list *list_data*, containing strings, numbers, vectors, and logical values as data objects, And also use *print* statement to print it.



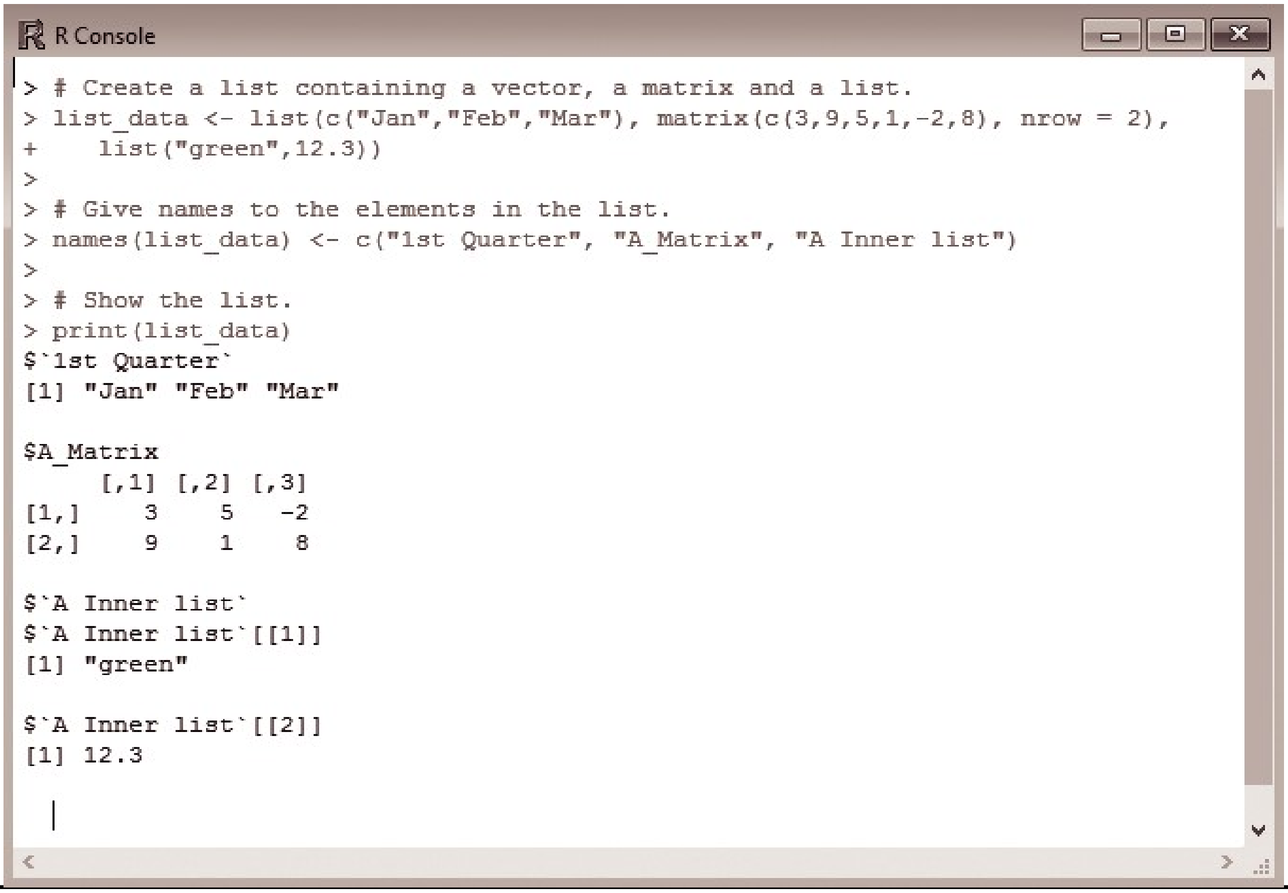
```
> # Create a list containing strings, numbers, vectors and a logical values.
> list_data <- list("Red", "Green", c(21,32,11), TRUE, 51.23, 119.1)
> print(list_data)
[[1]]
[1] "Red"

[[2]]
[1] "Green"

[[3]]
[1] 21 32 11

[[4]]
[1] TRUE

[[5]]
[1] 51.23

[[6]]
[1] 119.1

> |
```

Creating a named List

- Lists in R have a special property wherein names can be assigned to the previously defined set of elements in the list.
- Named lists can be created using the *names*() function.

Write the command in R console to create a list containing a vector, a matrix, and a list. Also give names to the elements in the list and display the list.

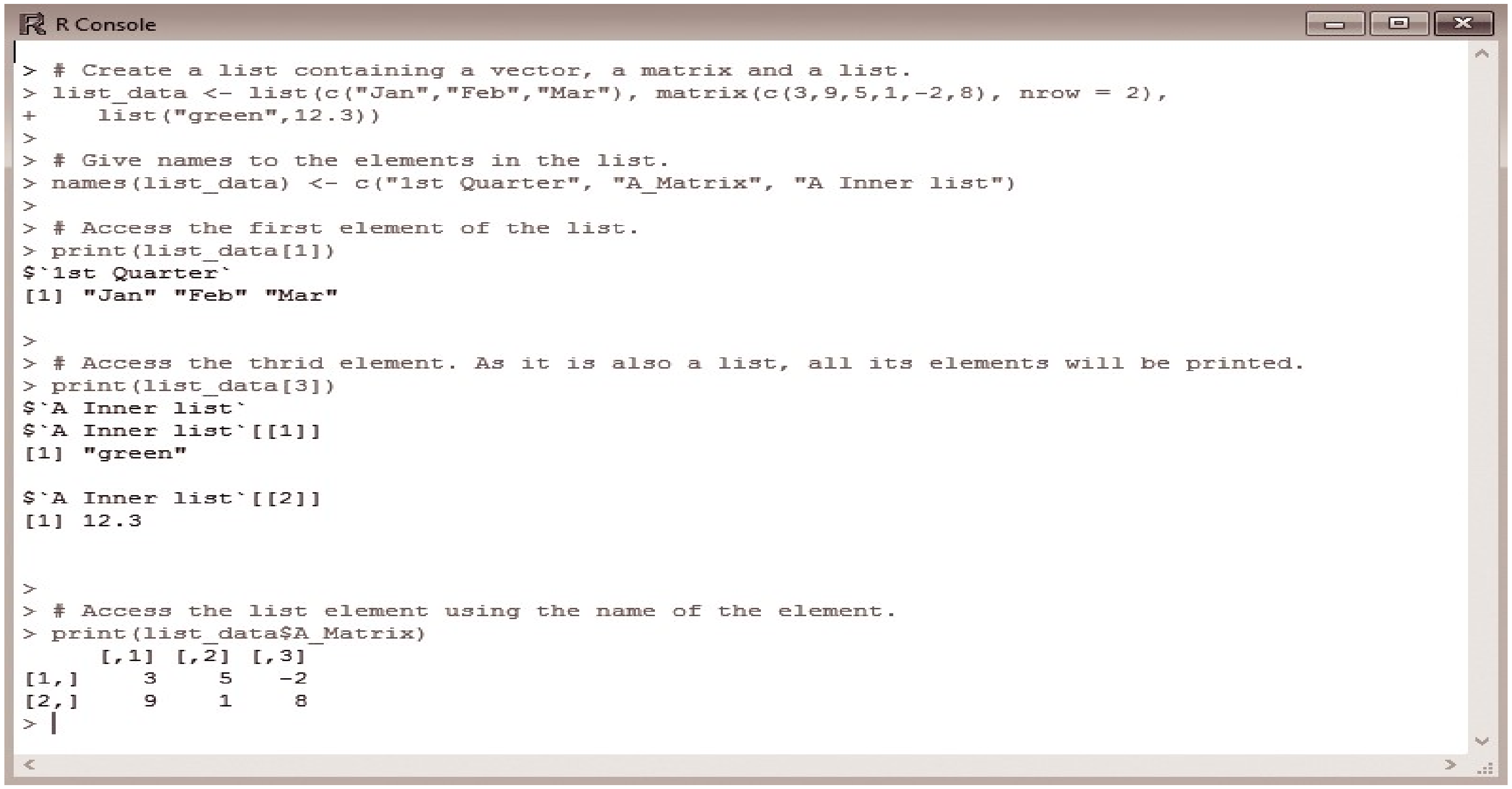
- creates a named list using the *names*() function.
- First a list called *list_data* containing a vector and a matrix and another list called *list* are created.
- The created list is viewed using the *print* statement.



```
> # Create a list containing a vector, a matrix and a list.
> list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
+    list("green",12.3))
>
> # Give names to the elements in the list.
> names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
>
> # Show the list.
> print(list_data)
$`1st Quarter`
[1] "Jan" "Feb" "Mar"

$A_Matrix
     [,1] [,2] [,3]
[1,]    3    5   -2
[2,]    9    1    8

$`A Inner list`
$`A Inner list`[[1]]
[1] "green"

$`A Inner list`[[2]]
[1] 12.3
```

Accessing List elements

- The elements can be accessed using the index value of that element. For a named list, the elements are accessed based on their names.
- In Fig., to access the first element from list_data, the index value of the list is accessed; here the list elements are indexed starting from 1 and not from 0. Therefore printing *list_data*[1] gives the names of the first and later elements in the list.



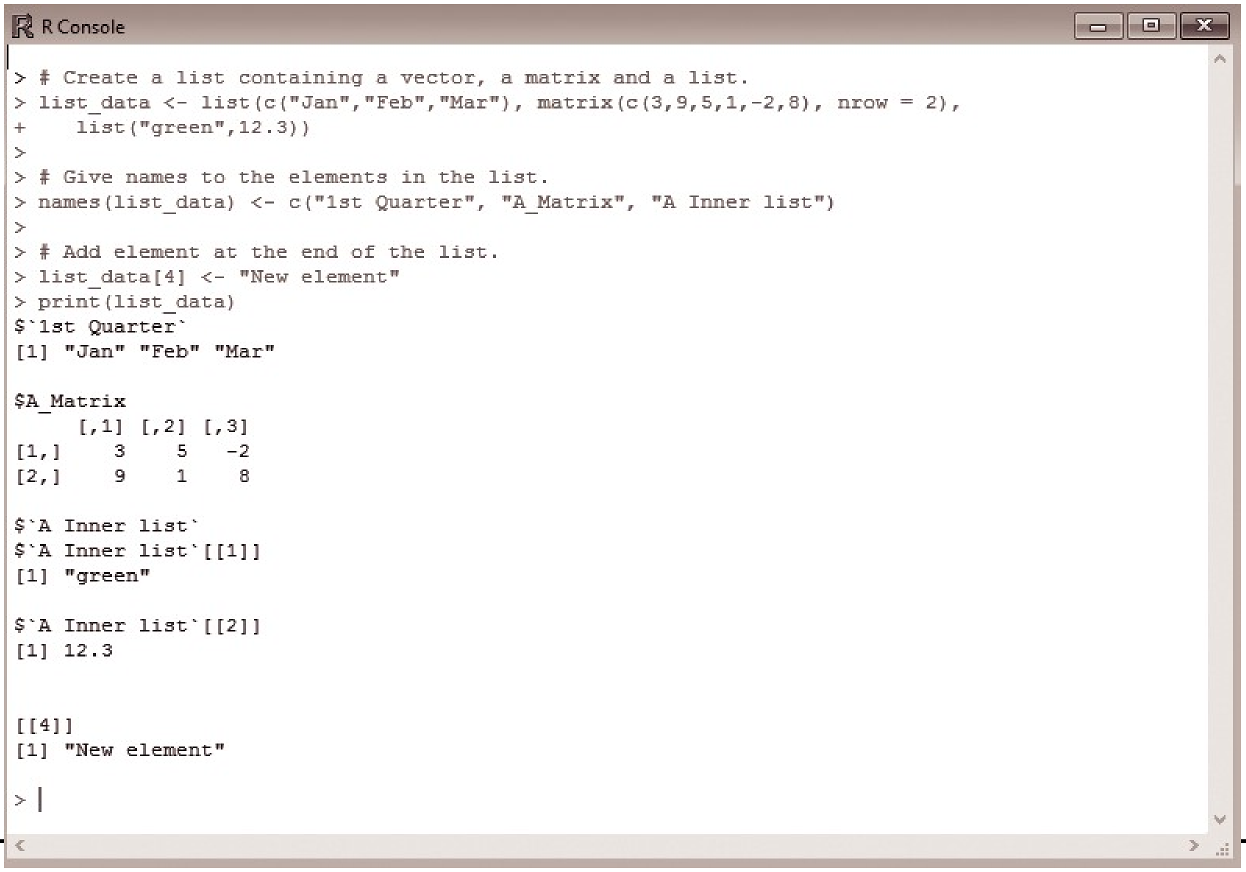
```
> # Create a list containing a vector, a matrix and a list.
> list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
+    list("green",12.3))
>
> # Give names to the elements in the list.
> names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
>
> # Access the first element of the list.
> print(list_data[1])
$`1st Quarter`
[1] "Jan" "Feb" "Mar"

>
> # Access the thrid element. As it is also a list, all its elements will be printed.
> print(list_data[3])
$`A Inner list`
$`A Inner list`[[1]]
[1] "green"

$`A Inner list`[[2]]
[1] 12.3


>
> # Access the list element using the name of the element.
> print(list_data$A_Matrix)
     [,1] [,2] [,3]
[1,]    3    5   -2
[2,]    9    1    8
> |
```

Manipulating List elements

- Addition, deletion, and updating operations can be performed on the list to manipulate the elements.
- Elements can be added and deleted only at the end of the list
- update operation can be performed on any element in the list irrespective of its position

Write the command in R console to add a new element at the end of the list and display the same.

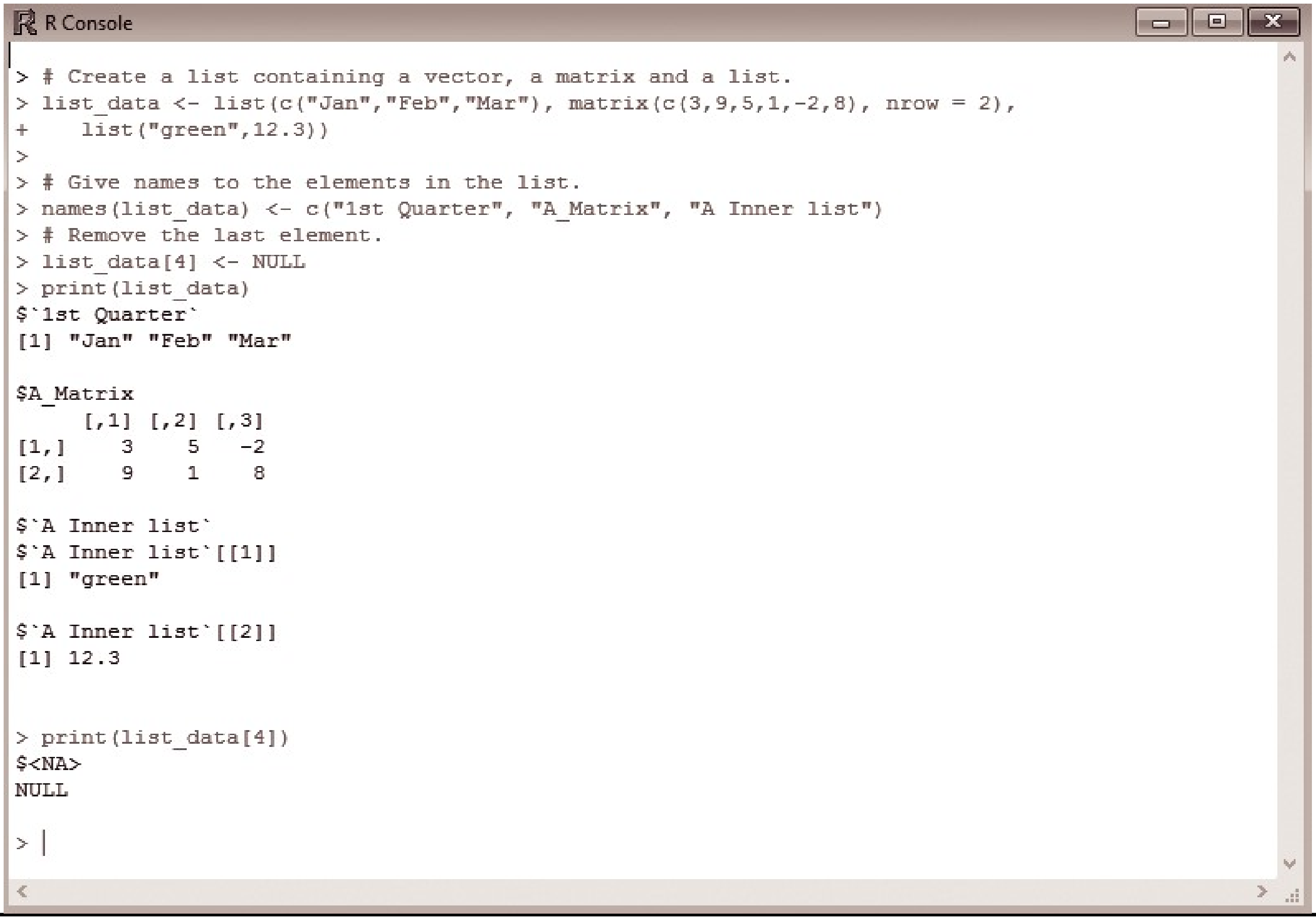
- Fig depicts the addition of an element to a list.
- Assume list consists of only three elements. To add the fourth element to this list, append the element (which is a string here) “New element” to the 4th indexed position of the list, that is, *list_data*[4].
- To view the added element in the list, print the list and the four elements are displayed.



```
> # Create a list containing a vector, a matrix and a list.
> list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
+    list("green",12.3))
>
> # Give names to the elements in the list.
> names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
>
> # Add element at the end of the list.
> list_data[4] <- "New element"
> print(list_data)
$`1st Quarter`
[1] "Jan" "Feb" "Mar"

$A_Matrix
     [,1] [,2] [,3]
[1,]    3    5   -2
[2,]    9    1    8

$`A Inner list`
$`A Inner list`[[1]]
[1] "green"

$`A Inner list`[[2]]
[1] 12.3


[[4]]
[1] "New element"

> |
```

Write the command in R console to delete the fourth element from a list and display the resultant list.

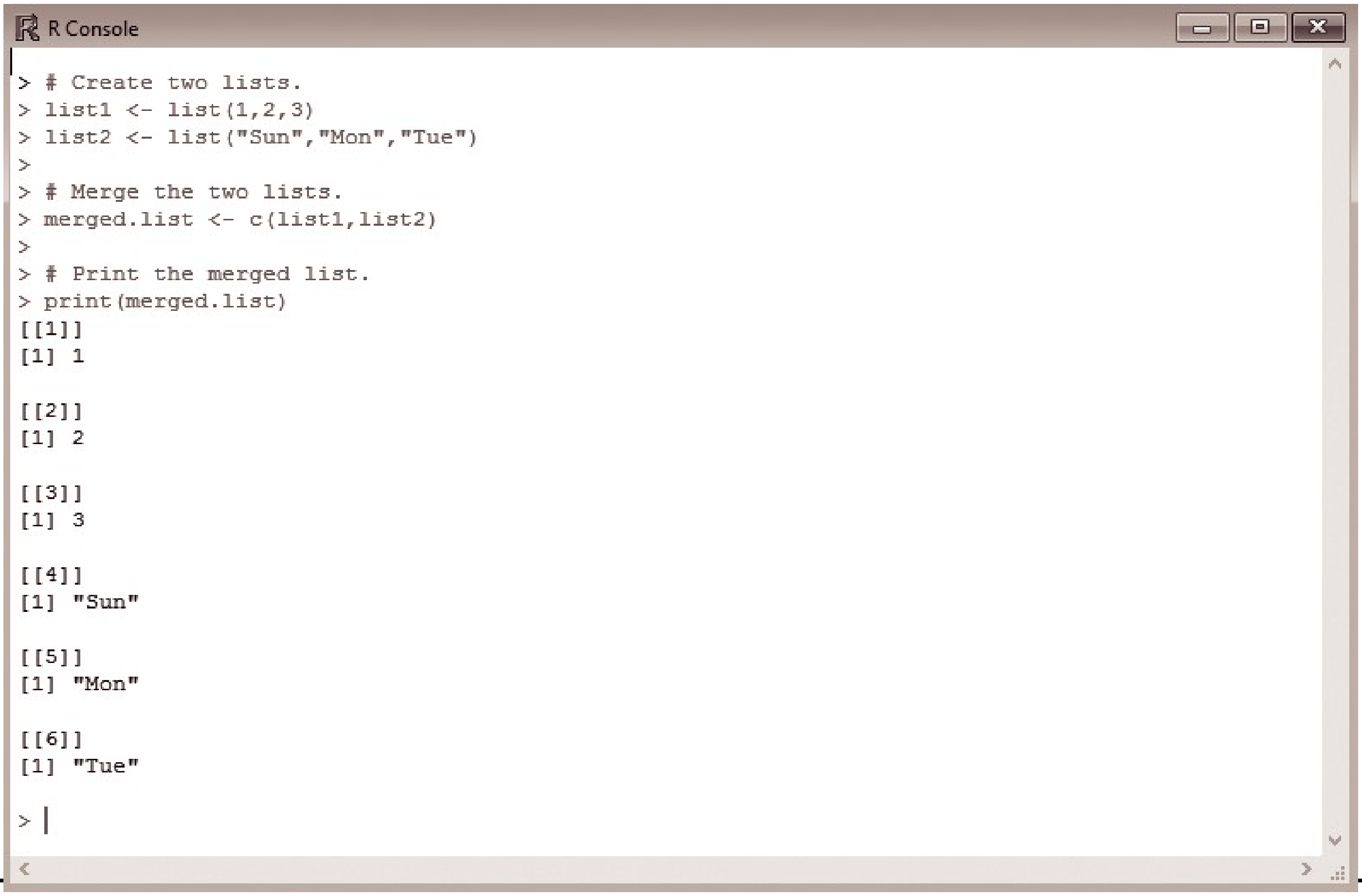
- Fig. , the fourth element, “New element” can be deleted by setting its value to NULL.
- Printing *list_data* shows only three elements in the list



```
> # Create a list containing a vector, a matrix and a list.
> list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
+    list("green",12.3))
>
> # Give names to the elements in the list.
> names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
> # Remove the last element.
> list_data[4] <- NULL
> print(list_data)
$`1st Quarter`
[1] "Jan" "Feb" "Mar"

$A_Matrix
     [,1] [,2] [,3]
[1,]    3    5   -2
[2,]    9    1    8

$`A Inner list`
$`A Inner list`[[1]]
[1] "green"

$`A Inner list`[[2]]
[1] 12.3


> print(list_data[4])
$<NA>
NULL

> |
```

Merging Lists

- Multiple lists can be merged by placing all those lists in a single *list*() function.
- Figure, shows two lists *list1* and *list2* created using the *list*() function.
- placing them in a single *list*() function gives a merged list called *merged.list*, which contains the elements of both the lists placed sequentially.



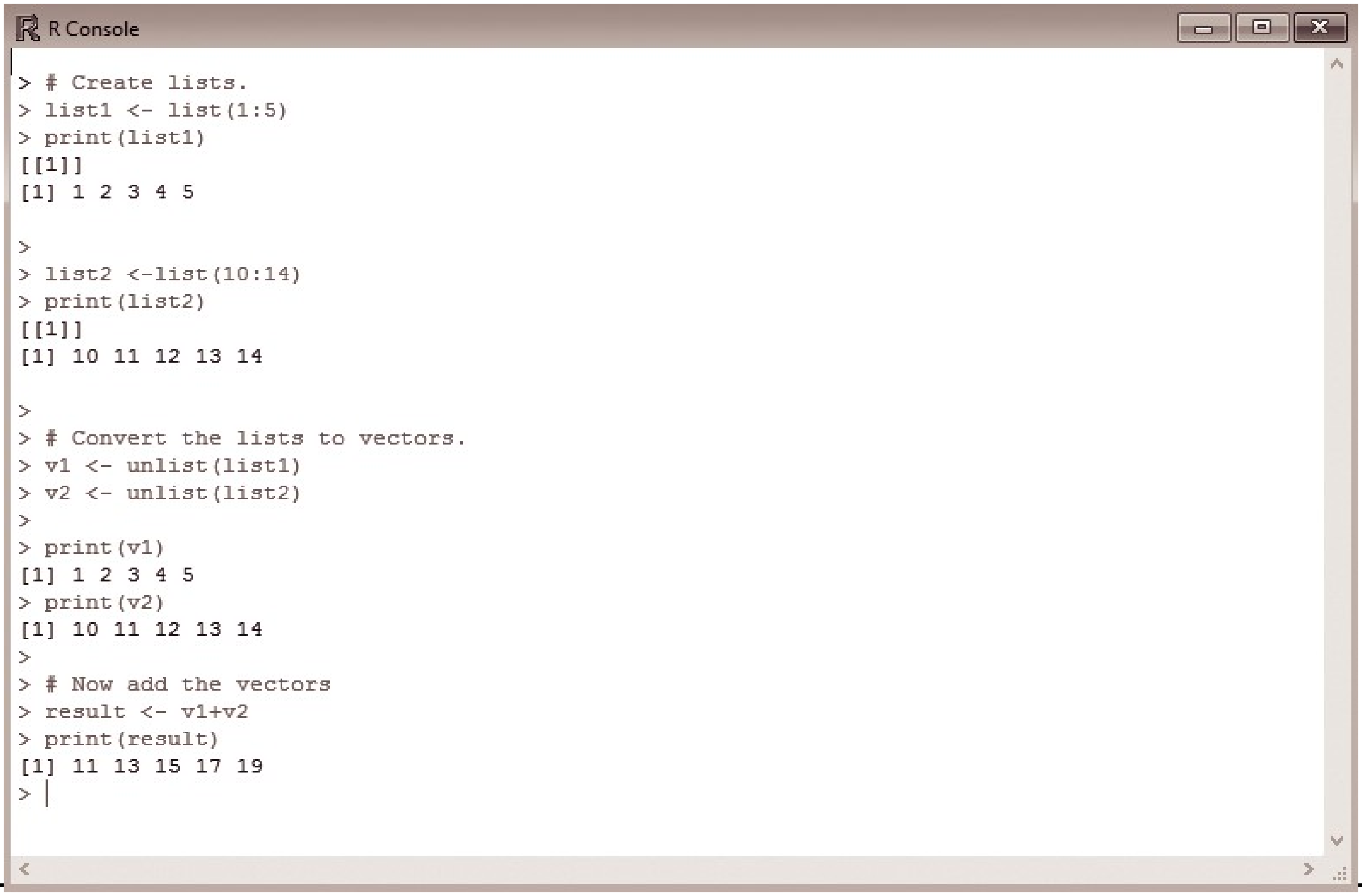
```
> # Create two lists.
> list1 <- list(1,2,3)
> list2 <- list("Sun","Mon","Tue")
>
> # Merge the two lists.
> merged.list <- c(list1,list2)
>
> # Print the merged list.
> print(merged.list)
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

[[4]]
[1] "Sun"

[[5]]
[1] "Mon"

[[6]]
[1] "Tue"

>
```

Converting Lists to vectors

- There are some special cases where lists are converted to vectors for further manipulation of the data elements.
- This is done because in addition to the basic operations of add, delete, and update, there are other arithmetic operations that can be easily applied when the operands are in vector form.
- A special function called *unlist*() is used ; this function takes the input as list and produces vectors

Write the command in R console to create two lists, each containing 5 elements. Convert the lists into vectors and perform addition on the two vectors. Display the resultant vector.

- Here two lists namely, *list1* and *list2* are initially created.
- Later with the help of the *unlist*() function, both the lists are converted to vectors named v1 and v2

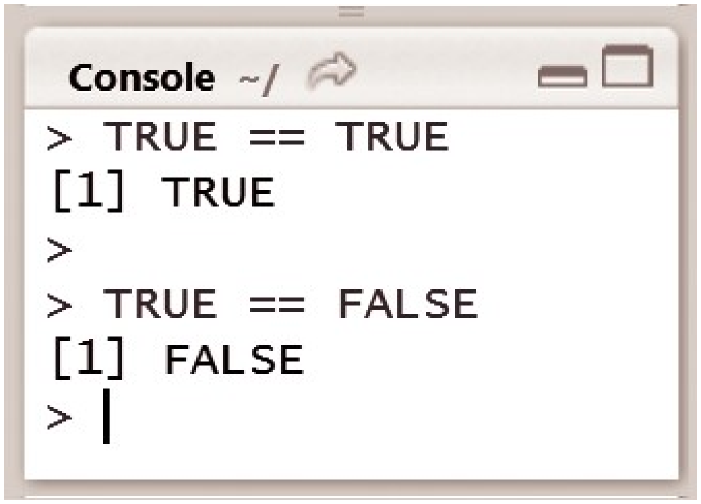


```
> # Create lists.
> list1 <- list(1:5)
> print(list1)
[[1]]
[1] 1 2 3 4 5

>
> list2 <-list(10:14)
> print(list2)
[[1]]
[1] 10 11 12 13 14

>
> # Convert the lists to vectors.
> v1 <- unlist(list1)
> v2 <- unlist(list2)
>
> print(v1)
[1] 1 2 3 4 5
> print(v2)
[1] 10 11 12 13 14
>
> # Now add the vectors
> result <- v1+v2
> print(result)
[1] 11 13 15 17 19
> |
```

Part - II
Conditionals and Control Flow in R

Relational operators

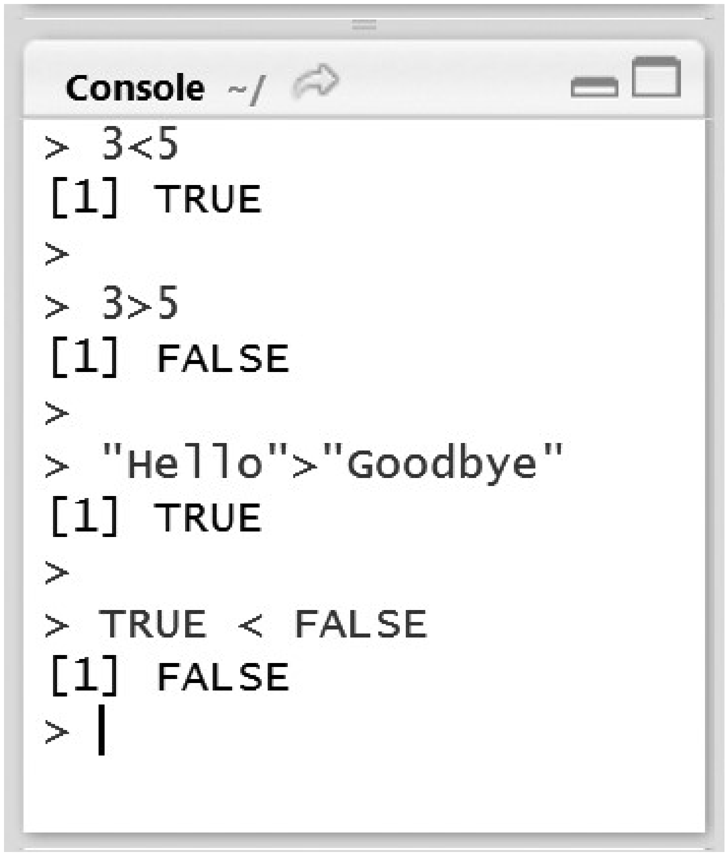
- Relational operators are comparators that help us understand how one object relates to another.
- For example, to check whether two objects are equal, the double equal operator can be used to check if the logical value TRUE equals the logical value TRUE (TRUE == TRUE) and the result is a logical value



```
> TRUE == TRUE
[1] TRUE
>
> TRUE == FALSE
[1] FALSE
> |
```

Logical variables can also be used to check the quality of other types, which compares strings and numbers

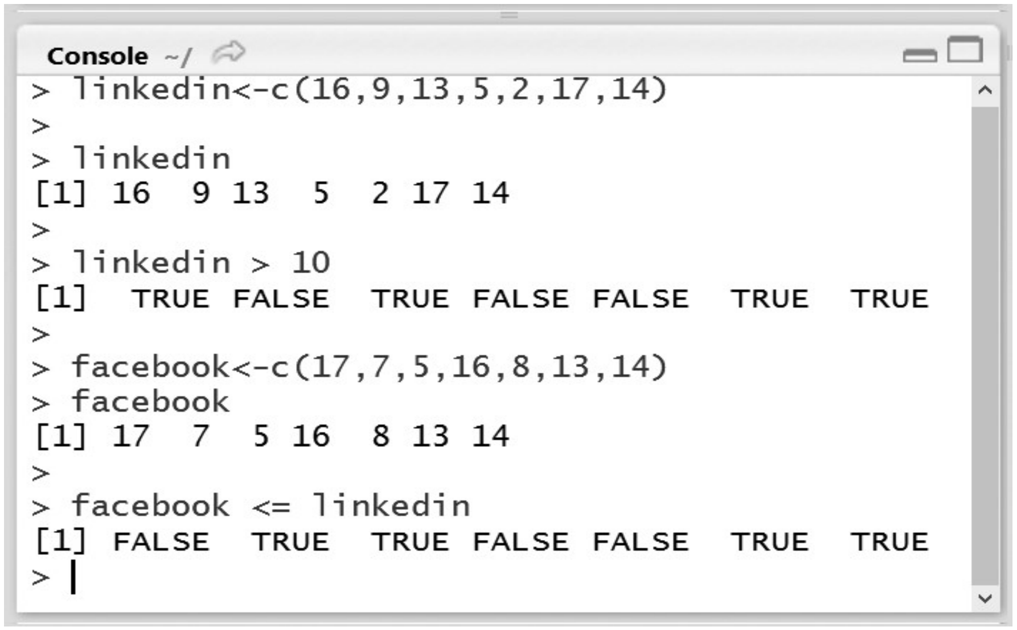
- These are the steps needed to be done to check if an object is lesser than or greater than another object.
  - In the case of numerical values 3 < 5 will evaluate to TRUE and 3 > 5 will evaluate to FALSE.
  - For character strings and logical values, R uses the alphabetic order to sort character strings
  - For logical values is TRUE < FALSE evaluates to FALSE, since TRUE corresponds to 1 and FALSE corresponds to 0



```
> 3<5
[1] TRUE
>
> 3>5
[1] FALSE
>
> "Hello">"Goodbye"
[1] TRUE
>
> TRUE < FALSE
[1] FALSE
> |
```

Relational operators and Vectors

Consider a scenario for recording the number of views of a LinkedIn profile per day using the vector *linkedin*, shown in Fig



```
> linkedin<-c(16,9,13,5,2,17,14)
>
> linkedin
[1] 16  9 13  5  2 17 14
>
> linkedin > 10
[1]  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE
>
> facebook<-c(17,7,5,16,8,13,14)
> facebook
[1] 17  7  5 16  8 13 14
>
> facebook <= linkedin
[1] FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE
> |
```

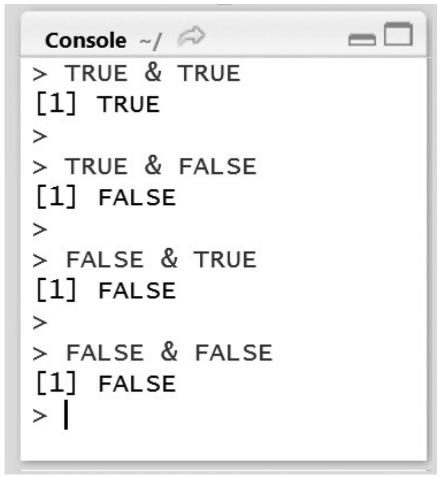
• For example, to find out on which days the number of views exceeded 10, use the greater than (>) operator,here these elements the result will be TRUE.
• Vectors can also be compared with other vectors. A record of the number of facebook user profiles for the previous week is available, which can be saved in another vector called *facebook*.

logical operators

- To change the combined results of comparisons, R uses the logical operators—AND (&), OR (|) and NOT (!).

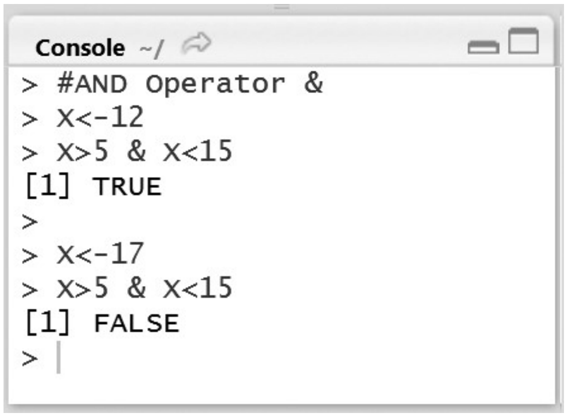
AND operator

- The AND operator takes two logical values and returns TRUE only if both these logical values are each TRUE



```
Console ~/
> TRUE & TRUE
[1] TRUE
>
> TRUE & FALSE
[1] FALSE
>
> FALSE & TRUE
[1] FALSE
>
> FALSE & FALSE
[1] FALSE
>
```

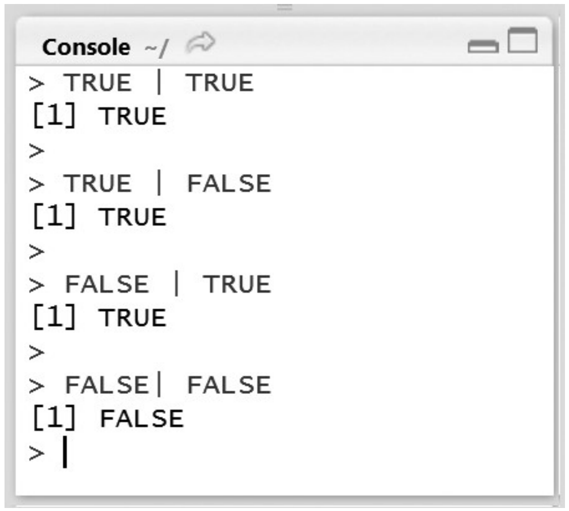
•Instead of using logical values, the results of comparisons can be used



```
Console ~/
> #AND Operator &
> X<-12
> X>5 & X<15
[1] TRUE
>
> X<-17
> X>5 & X<15
[1] FALSE
>
```

OR operator

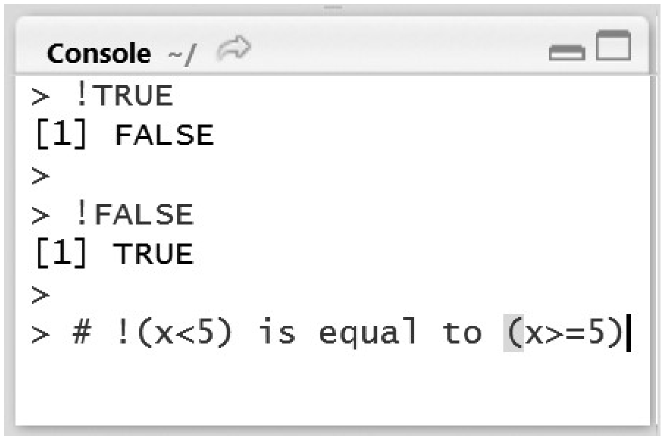
- The OR operator works similar to the AND operator with a difference that at least one of the logical values should be TRUE for the entire operation to evaluate to TRUE.



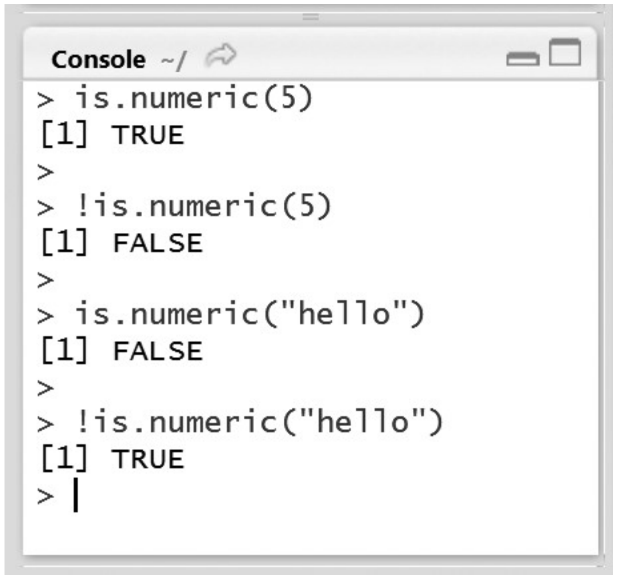
```
> TRUE | TRUE
[1] TRUE
>
> TRUE | FALSE
[1] TRUE
>
> FALSE | TRUE
[1] TRUE
>
> FALSE| FALSE
[1] FALSE
> |
```

NOT operator

- The NOT operator represented by ! simply negates the given logical value



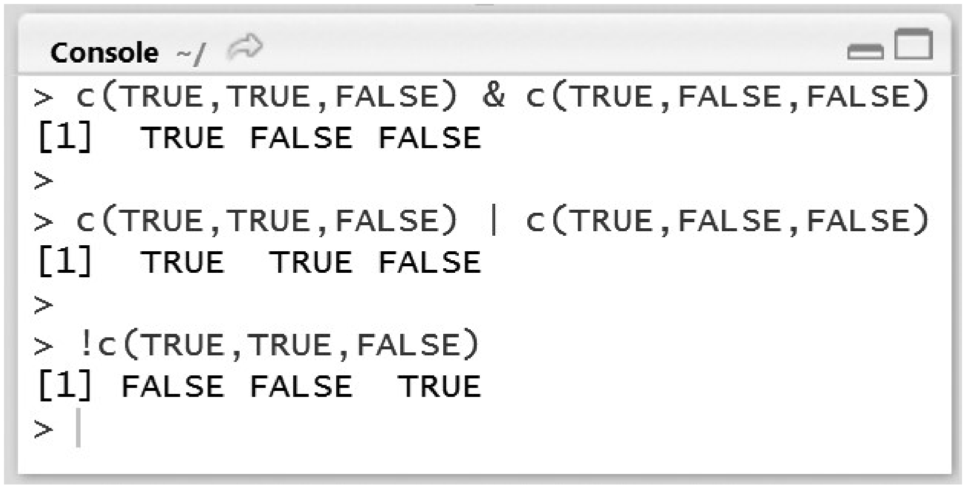
```
> !TRUE
[1] FALSE
>
> !FALSE
[1] TRUE
>
> # !(x<5) is equal to (x>=5)
```



```
> is.numeric(5)
[1] TRUE
>
> !is.numeric(5)
[1] FALSE
>
> is.numeric("hello")
[1] FALSE
>
> !is.numeric("hello")
[1] TRUE
>
```

logical operators and Vectors

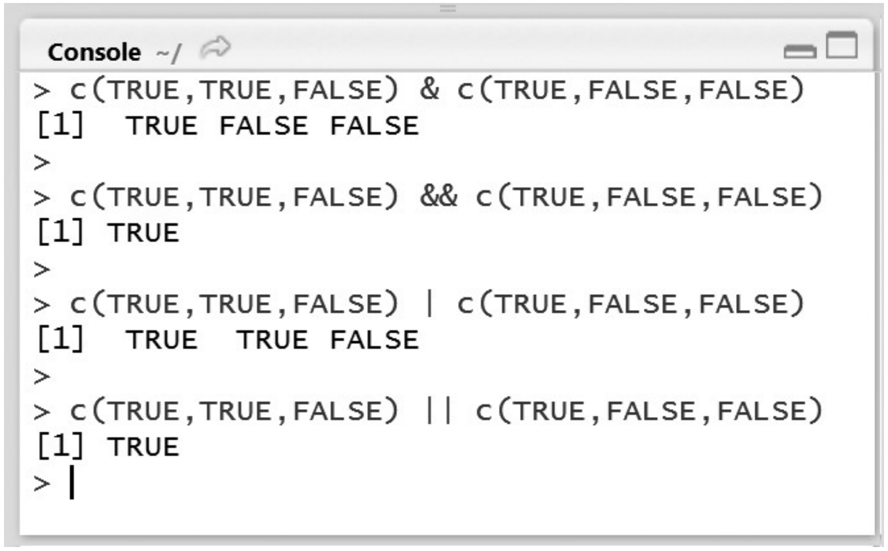
- The working of logical operators with vectors and matrices is similar to that of the relational operators; the operations are performed element-wise.
- Fig., where two vectors with the elements TRUE, TRUE, FALSE and TRUE, FALSE, FALSE are taken. AND operation on these vectors results in a vector with the elements TRUE, FALSE, and FALSE



```
> c(TRUE,TRUE,FALSE) & c(TRUE,FALSE,FALSE)
[1]  TRUE FALSE FALSE
>
> c(TRUE,TRUE,FALSE) | c(TRUE,FALSE,FALSE)
[1]  TRUE  TRUE FALSE
>
> !c(TRUE,TRUE,FALSE)
[1] FALSE FALSE  TRUE
>
```

& Vs &&, | Vs ||

- There are two versions of the AND and OR operators, namely the single operator and the double operator versions.
- Fig. , as inputs, the first expression evaluates to a vector containing TRUE, FALSE, and FALSE. However if && is used, the result will be TRUE, since the && operator only examines the first element



```
> c(TRUE,TRUE,FALSE) & c(TRUE,FALSE,FALSE)
[1]  TRUE FALSE FALSE
>
> c(TRUE,TRUE,FALSE) && c(TRUE,FALSE,FALSE)
[1] TRUE
>
> c(TRUE,TRUE,FALSE) | c(TRUE,FALSE,FALSE)
[1]  TRUE  TRUE FALSE
>
> c(TRUE,TRUE,FALSE) || c(TRUE,FALSE,FALSE)
[1] TRUE
> |
```

conditional statements

- *If-else*
- *If-else if-else*

If-else statements

- The syntax of an if statement contains a condition; if the condition evaluates to TRUE, the R code associated with the if statement is executed, the condition appears inside parentheses
- Syntax:

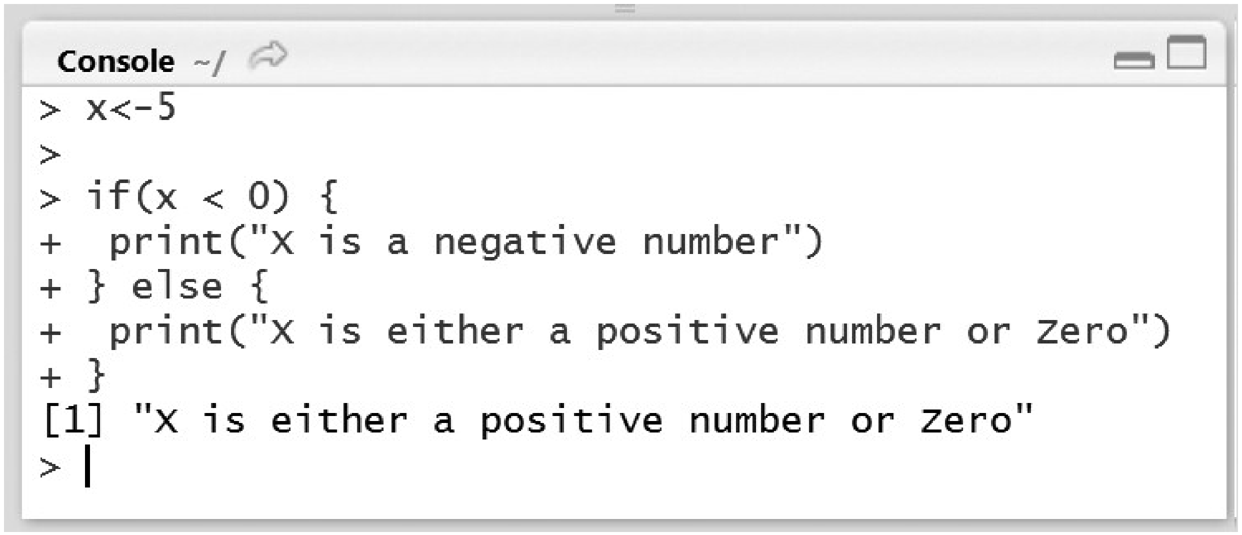
```
if(condition) {
expression
}
```

•if-else statement, the code associated with the else statement gets executed whenever the condition of the if statement is not satisfied.

Syntax:

```
if(condition) {
expression 1
 }
else {
}
expression 2
```

- Figure shows an if condition that checks if X is less than zero.
- It prints “X is either a positive number or zero” whenever the condition is not met.
- Changing X to 5, the condition is not satisfied and prints the expression following the else statement.



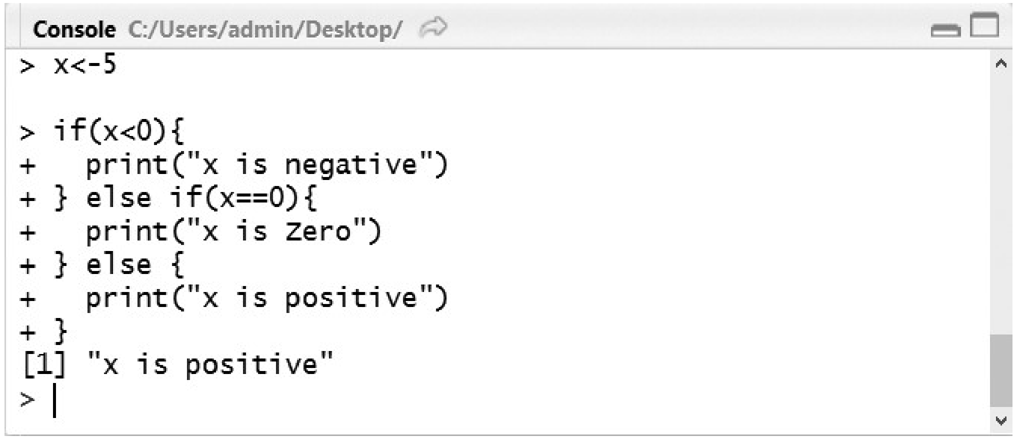
```
> x<-5
>
> if(x < 0) {
+  print("X is a negative number")
+ } else {
+  print("X is either a positive number or Zero")
+ }
[1] "X is either a positive number or Zero"
> |
```

Else – if statements

- The else if statement comes in between the if and else statements.

Syntax:

if(condition1) {
expression 1
}else if(condition2) {
expression 2
} else {expression 3
}



```
> x<-5

> if(x<0){
+   print("x is negative")
+ } else if(x==0){
+   print("x is Zero")
+ } else {
+   print("x is positive")
+ }
[1] "x is positive"
> |
```

Part - III
Iterative Programming in R

5.2 While Loop

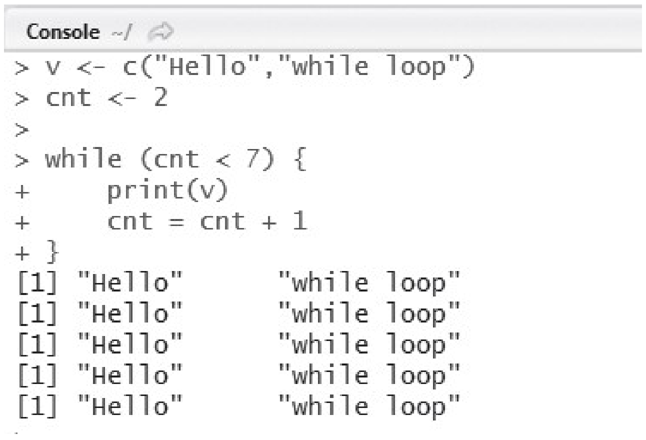
- The syntax for the while loop consists of *test_expression that acts as the minimal condition for the loop to* check how many times it has to be executed

  while(test_expression){

  statement

  }

Printing values in a vector using a while loop


Console ~/

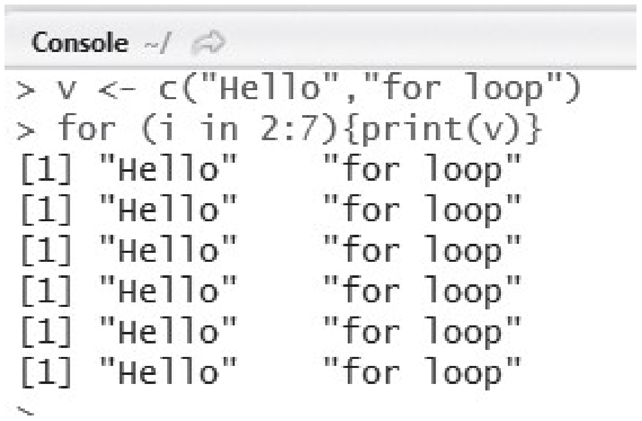
```
> v <- c("Hello","while loop")
> cnt <- 2
>
> while (cnt < 7) {
+     print(v)
+     cnt = cnt + 1
+ }
[1] "Hello"      "while loop"
[1] "Hello"      "while loop"
[1] "Hello"      "while loop"
[1] "Hello"      "while loop"
[1] "Hello"      "while loop"
```

5.3 For Loop

- The syntax of for loop consists of *variable that can be used inside the loop for iterating over the list of* elements in a vector or a list.

```
for (variable in start:end){
statement
}
```

Printing values in a vector using a for loop

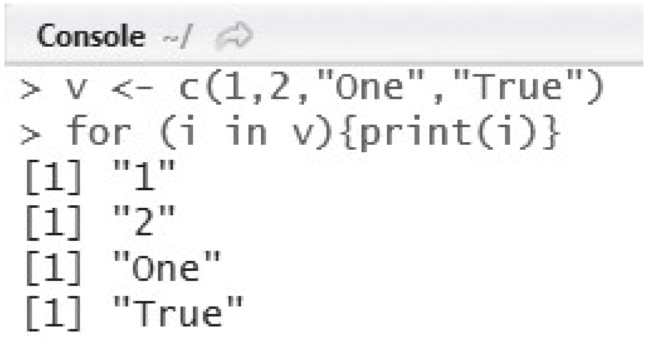


Console ~/

```
> v <- c("Hello","for loop")
> for (i in 2:7){print(v)}
[1] "Hello"    "for loop"
[1] "Hello"    "for loop"
[1] "Hello"    "for loop"
[1] "Hello"    "for loop"
[1] "Hello"    "for loop"
[1] "Hello"    "for loop"
>
```

5.4 Looping Over List

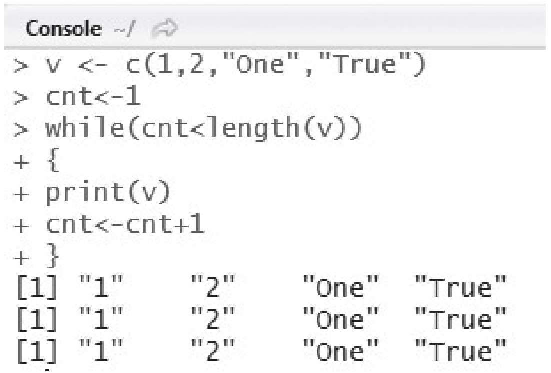
- L oops for Vectors



Console ~/

```
> v <- c(1,2,"One","True")
> for (i in v){print(i)}
[1] "1"
[1] "2"
[1] "One"
[1] "True"
```

Fig. 5.3 Vectors using for loop

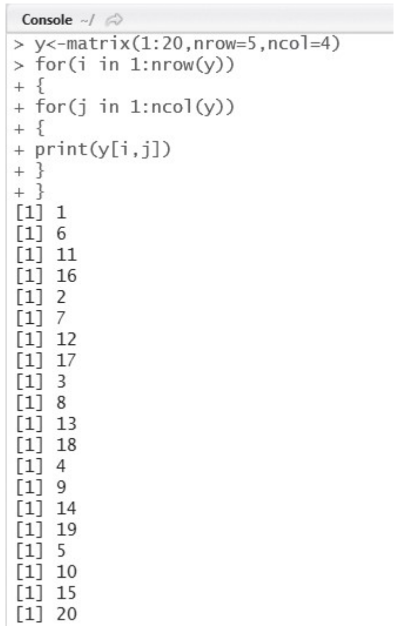


Console ~/

```
> v <- c(1,2,"One","True")
> cnt<-1
> while(cnt<length(v))
+ {
+ print(v)
+ cnt<-cnt+1
+ }
[1] "1"    "2"    "One"  "True"
[1] "1"    "2"    "One"  "True"
[1] "1"    "2"    "One"  "True"
```

Fig. 5.4 Printing vectors using while loop

- **Loops for Matrices**
  - A matrix can be created in R using the *matrix() function*



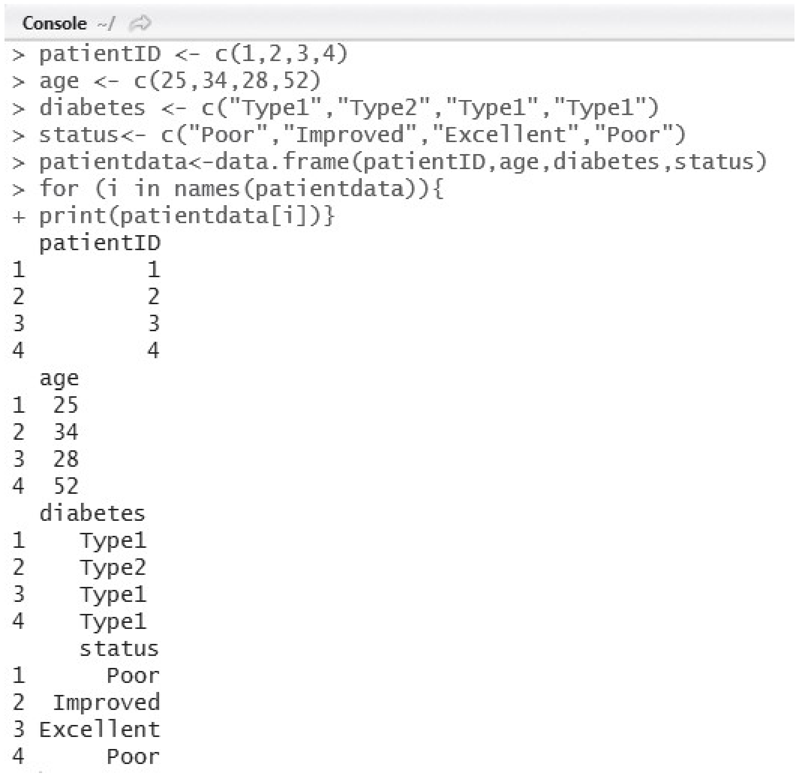
```
> y<-matrix(1:20,nrow=5,ncol=4)
> for(i in 1:nrow(y))
+ {
+ for(j in 1:ncol(y))
+ {
+ print(y[i,j])
+ }
+ }
[1] 1
[1] 6
[1] 11
[1] 16
[1] 2
[1] 7
[1] 12
[1] 17
[1] 3
[1] 8
[1] 13
[1] 18
[1] 4
[1] 9
[1] 14
[1] 19
[1] 5
[1] 10
[1] 15
[1] 20
```

Fig. 5.5 Printing the elements of a matrix using for loop

- **L oops for Data Frames**
    - A data frame in R consists of a group of vectors
    - It is created using the *data.frame() function*

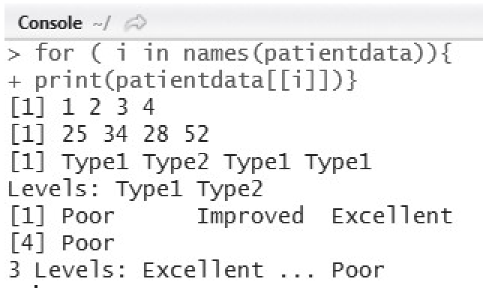
- Example

Write an R program to create four vectors namely *patientid, age, diabetes, and status. Put* these four vectors into a data frame *patientdata and print the values using a for loop*



```
Console ~/
> patientID <- c(1,2,3,4)
> age <- c(25,34,28,52)
> diabetes <- c("Type1","Type2","Type1","Type1")
> status<- c("Poor","Improved","Excellent","Poor")
> patientdata<-data.frame(patientID,age,diabetes,status)
> for (i in names(patientdata)){
+ print(patientdata[i])}
  patientID
1         1
2         2
3         3
4         4
  age
1  25
2  34
3  28
4  52
  diabetes
1    Type1
2    Type2
3    Type1
4    Type1
     status
1      Poor
2  Improved
3 Excellent
4      Poor
```

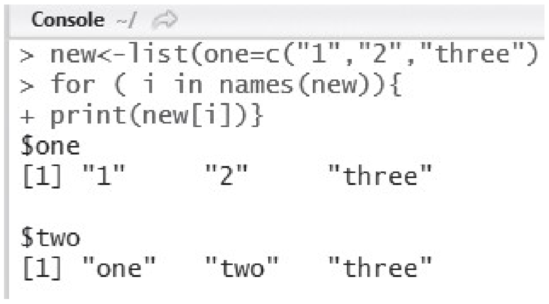
Creation of *data.frame()*



```
Console ~/
> for ( i in names(patientdata)){
+ print(patientdata[[i]])}
[1] 1 2 3 4
[1] 25 34 28 52
[1] Type1 Type2 Type1 Type1
Levels: Type1 Type2
[1] Poor      Improved  Excellent
[4] Poor
3 Levels: Excellent ... Poor
```

Listing the contents of *data.frame()*

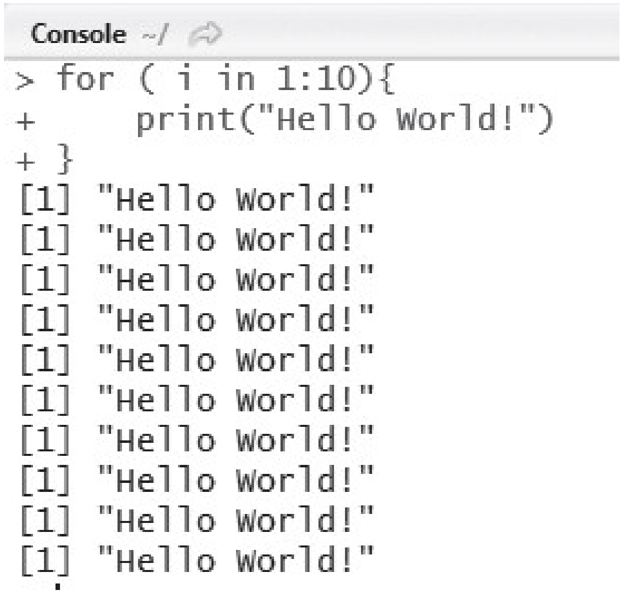
- **Loops for Lists**



```
Console ~/
> new<-list(one=c("1","2","three")
> for ( i in names(new)){
+ print(new[i])}
$one
[1] "1"     "2"     "three"

$two
[1] "one"   "two"   "three"
```

Creation of *list()*

- Write an R program to print HELLO 10 times using the for loop.



```
Console ~/
> for ( i in 1:10){
+     print("Hello World!")
+ }
[1] "Hello World!"
[1] "Hello World!"
[1] "Hello World!"
[1] "Hello World!"
[1] "Hello World!"
[1] "Hello World!"
[1] "Hello World!"
[1] "Hello World!"
[1] "Hello World!"
[1] "Hello World!"
```

Printing HELLO 10 times

Part - IV
Functions in R

6.1: Introduction

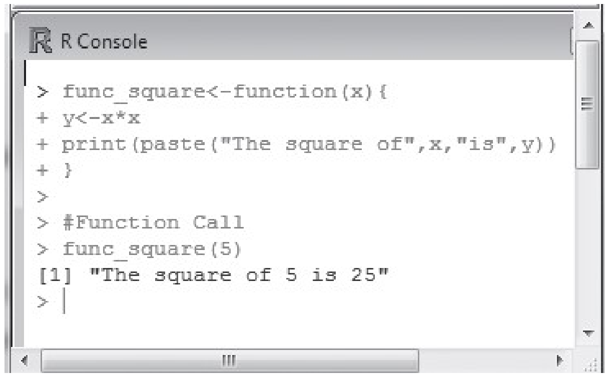
- A function is a block or chunk of code having a specific structure, which is often singular or atomic in nature and can be reused to accomplish a specific task.
- Functions serve as tools to repeatedly execute certain complex instructions coupled together, since they are self-contained
- A function has:
  - an input(in the form of arguments)
  - a task (set of statements or lines of code)
  - an output (a return value).
- A function is known by many names in various programming languages such as methods, procedures, and sub-routines.

6.2: Writing a Function in R

- The structure of a function in R is similar to that of most of other programming languages which looks something like this:

```
function_name <- function(arguments)
{
    computations on the arguments
    task-specific code statements
}
```

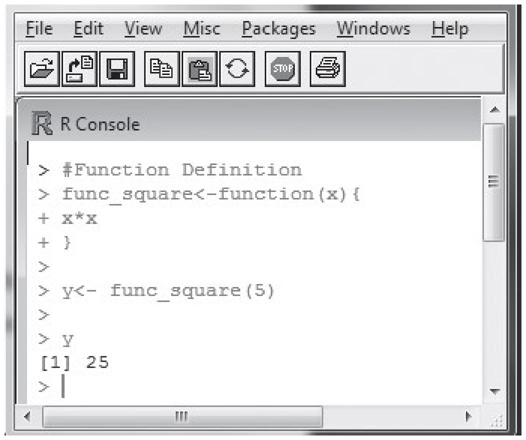
• Figure 6.1 illustrates a simple example to create a user-defined function to compute the square of an integer in R.



```
> func_square<-function(x){
+ y<-x*x
+ print(paste("The square of",x,"is",y))
+ }
>
> #Function Call
> func_square(5)
[1] "The square of 5 is 25"
>
```

Fig. 6.1 User-defined function in R

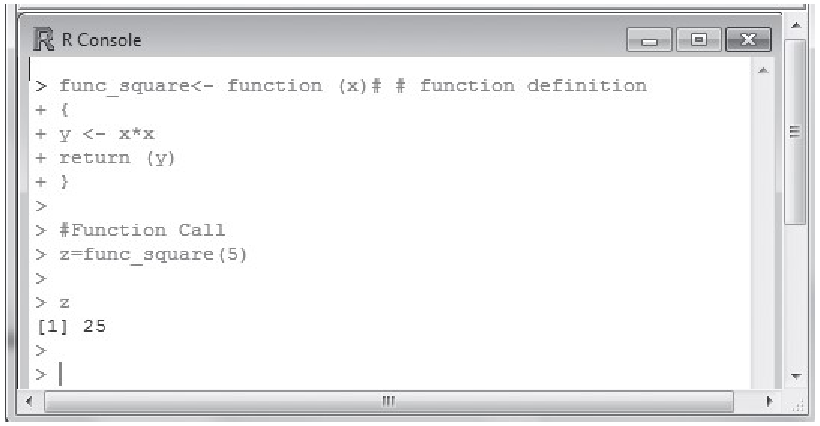
- The method shown in Fig. 6.2 does not make use of the ‘return’ keyword explicitly.



```
> #Function Definition
> func_square<-function(x){
+ x*x
+ }
>
> y<- func_square(5)
>
> y
[1] 25
> 
```

Fig. 6.2 Returning value from function in R

- In Fig. 6.3, the *return keyword is used and the returned value is stored in a variable using the = sign.*



```
> func_square<- function (x)# # function definition
+ {
+ y <- x*x
+ return (y)
+ }
>
> #Function Call
> z=func_square(5)
>
> z
[1] 25
>
> |
```

Fig. 6.3 Returning the value using return keyword

6.3: Nested Functions

- R allows defining a function inside a function, which is commonly known as *nesting of functions. Function nesting can be achieved by defining one function inside* another, or by calling one function from another.
- For example,

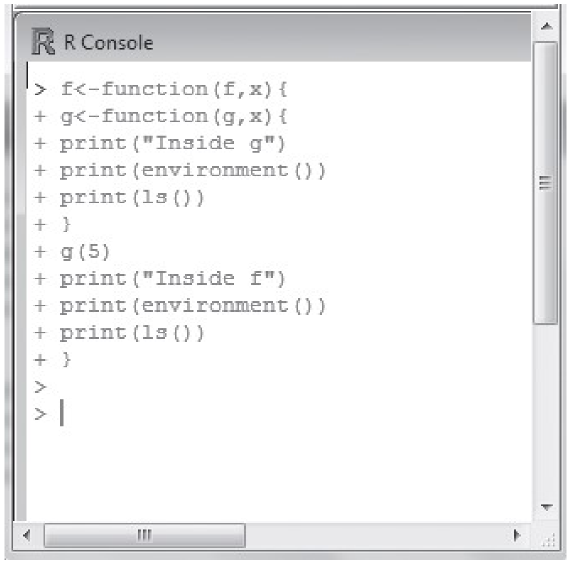
```
first_func <- function (arguments1)
{
computational code on arguments of first_func
task-specific statements
second_func <- function (arguments2)
{
computational code on arguments of second_func
task-specific statements
return statement 2 (if any)
}
rest of function body
return statement 1 (if any)
}
```

The same effect as that of the aforementioned code can be achieved in the following manner:

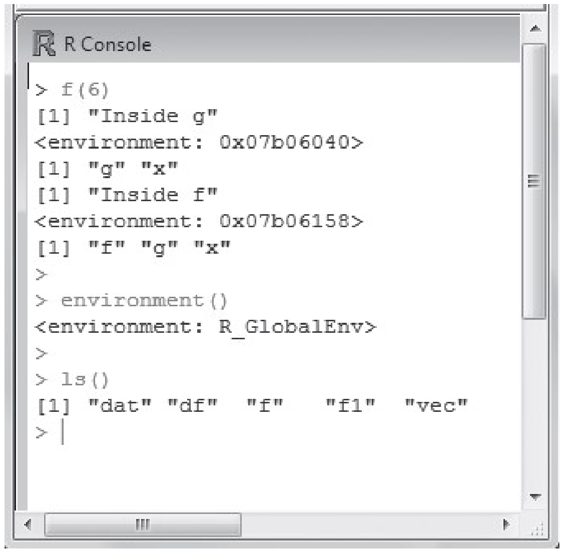
```
second_func <- function (arguments2)
{
computational code on arguments of second_func
task-specific statements
return statement 2 (if any)
}
first_func <- function (arguments1)
{
computational code on arguments of first_func
task-specific statements
my_value <- second_func (arguments2)
rest of function body
return statement 1 (if any)
}
```

6.4 Function Scoping

- The notion of environment and scope is important.
- **Function Environment**
  - The first thing that gets created when the R interpretation begins is its environment. All variables defined in the R script are the part of this environment and are listed as environment variables as shown in Fig.



```
> f<-function(f,x){
+ g<-function(g,x){
+ print("Inside g")
+ print(environment())
+ print(ls())
+ }
+ g(5)
+ print("Inside f")
+ print(environment())
+ print(ls())
+ }
>
> 
```



```
> f(6)
[1] "Inside g"
<environment: 0x07b06040>
[1] "g" "x"
[1] "Inside f"
<environment: 0x07b06158>
[1] "f" "g" "x"
>
> environment()
<environment: R_GlobalEnv>
>
> ls()
[1] "dat" "df"  "f"   "f1"  "vec"
> 
```

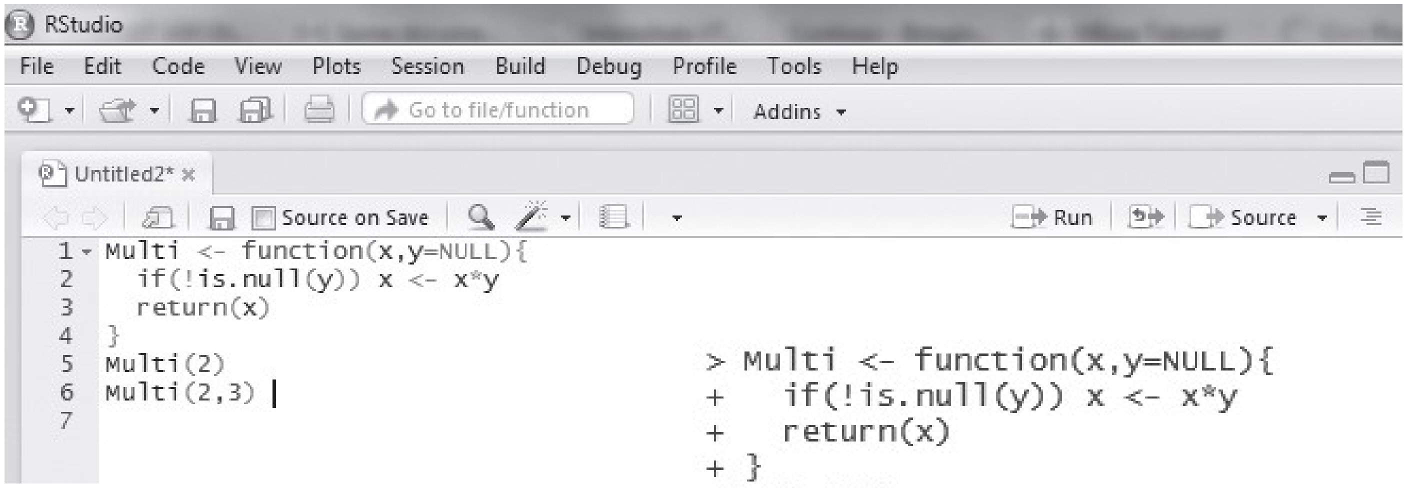
Function Scope

- Scoping is the set of rules that governs how R looks up the value of a symbol
- Global variables are those that are available throughout the program changes are reflected in all further references to that variable
- A variable defined in a function is not accessible outside the function. It is released when the function ends.

Example 2, Multiplying two numbers using a function with a default value. Assuming default values as NULL

Default Values for Arguments

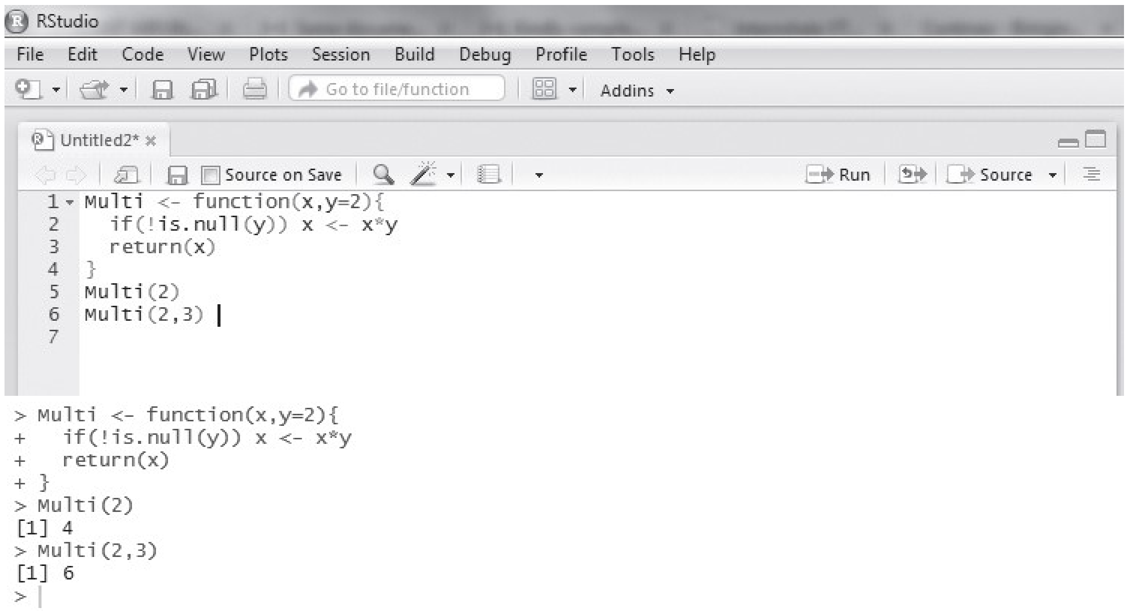
- the concept of a default value is legal
- The default argument value can be passed only once
- The value of the default argument needs to be declared while declaring the function.



```
Multi <- function(x,y=NULL){
  if(!is.null(y)) x <- x*y
  return(x)
}
Multi(2)
Multi(2,3)
```

```
> Multi <- function(x,y=NULL){
+   if(!is.null(y)) x <- x*y
+   return(x)
+ }
> Multi(2)
[1] 2
> Multi(2,3)
[1] 6
>
```

- Figure 6.8 shows a code construct where the default value *y is passed as 2 instead of NULL.*



```
Multi <- function(x,y=2){
  if(!is.null(y)) x <- x*y
  return(x)
}
Multi(2)
Multi(2,3)
```

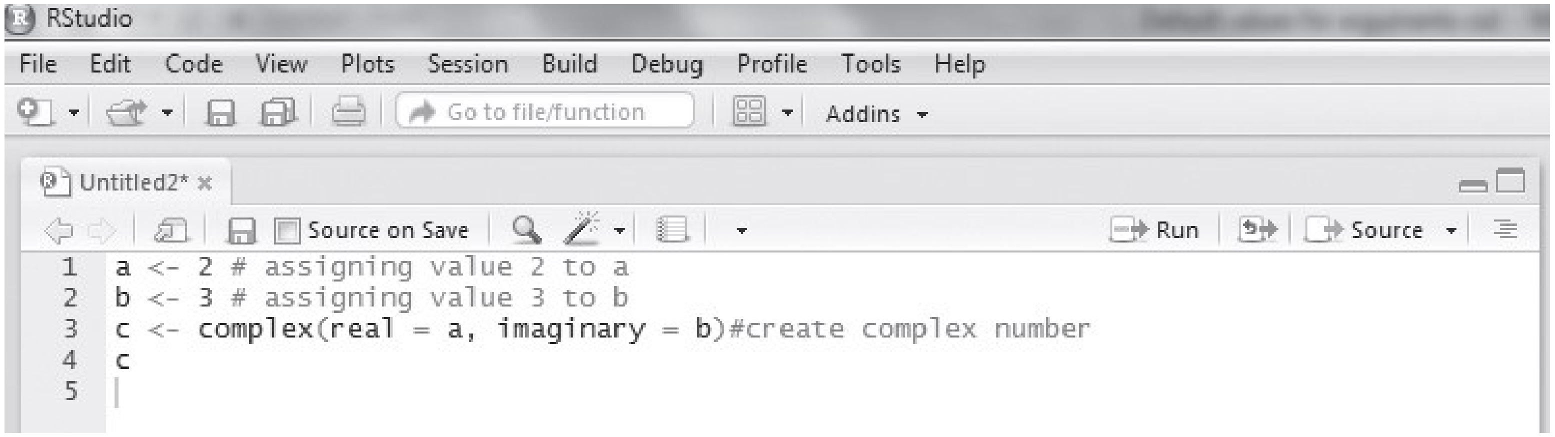
```
> Multi <- function(x,y=2){
+   if(!is.null(y)) x <- x*y
+   return(x)
+ }
> Multi(2)
[1] 4
> Multi(2,3)
[1] 6
>
```

Fig. 6.8 Function with a non-NULL default value

Returning Complex Objects

- Complex functions are incorporated in base package of R language. To generate a complex number, *a* + *ib, a* call is made to the *complex function in R. Example 3 shows the creation of a complex number.*

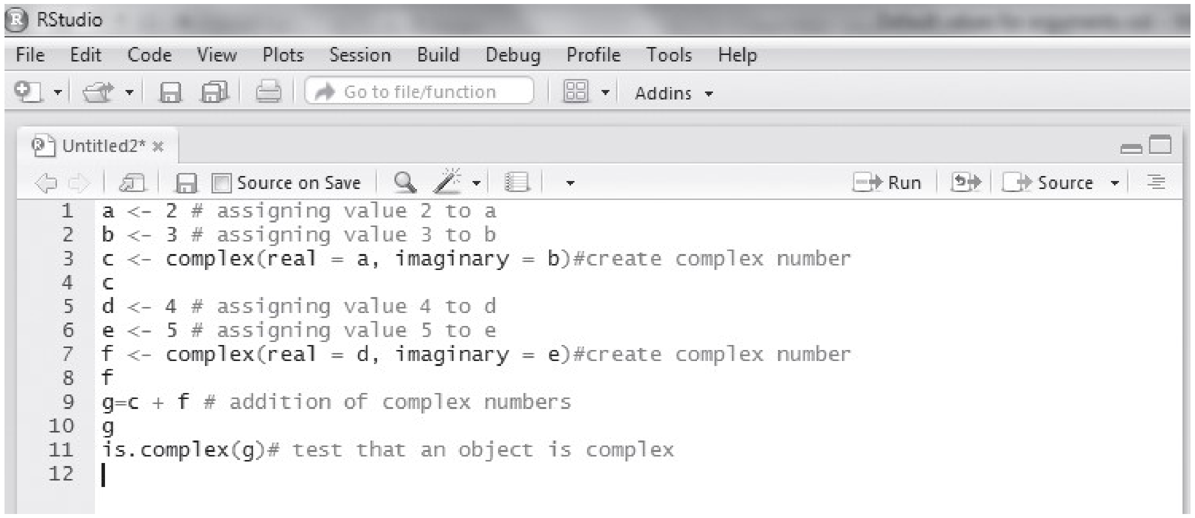
Example 3, Creating a Complex Number



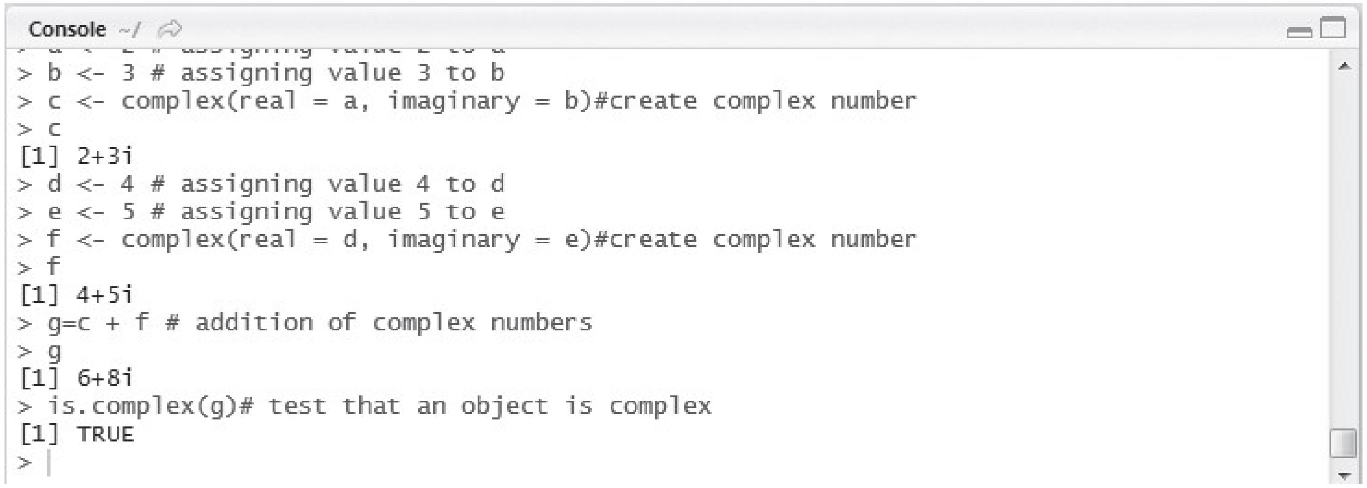
```
a <- 2 # assigning value 2 to a
b <- 3 # assigning value 3 to b
c <- complex(real = a, imaginary = b)#create complex number
c
```

```
> a <- 2 # assigning value 2 to a
> b <- 3 # assigning value 3 to b
> c <- complex(real = a, imaginary = b)#create complex number
> c
[1] 2+3i
> 
```

Example 4, Addition of two numbers using the *complex* function and testing if the sum is complex or not.

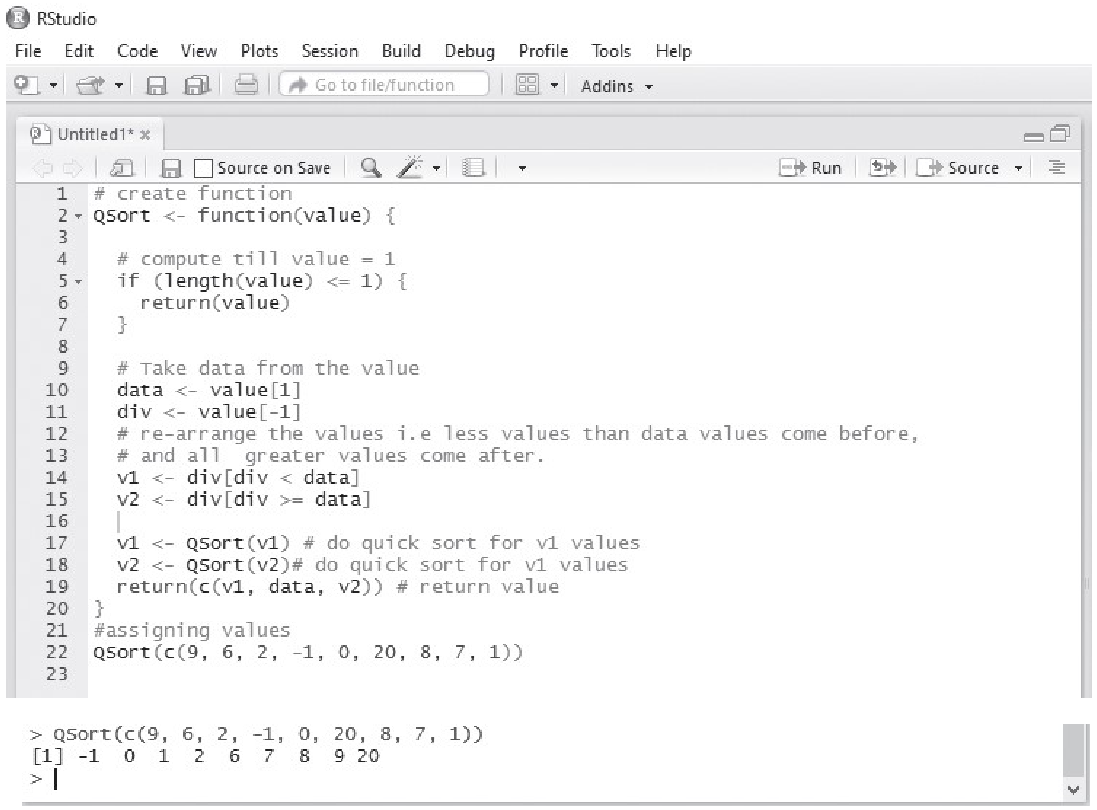


```
a <- 2 # assigning value 2 to a
b <- 3 # assigning value 3 to b
c <- complex(real = a, imaginary = b)#create complex number
c
d <- 4 # assigning value 4 to d
e <- 5 # assigning value 5 to e
f <- complex(real = d, imaginary = e)#create complex number
f
g=c + f # addition of complex numbers
g
is.complex(g)# test that an object is complex
```



```
> b <- 3 # assigning value 3 to b
> c <- complex(real = a, imaginary = b)#create complex number
> c
[1] 2+3i
> d <- 4 # assigning value 4 to d
> e <- 5 # assigning value 5 to e
> f <- complex(real = d, imaginary = e)#create complex number
> f
[1] 4+5i
> g=c + f # addition of complex numbers
> g
[1] 6+8i
> is.complex(g)# test that an object is complex
[1] TRUE
>
```

Illustration of Quicksort



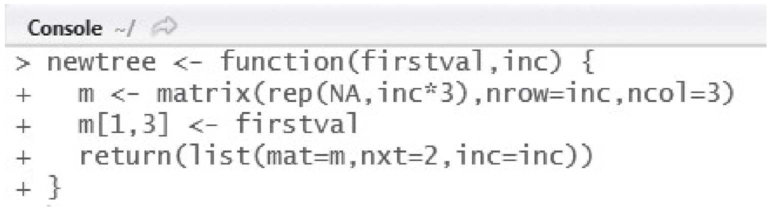
```r
# create function
QSort <- function(value) {

  # compute till value = 1
  if (length(value) <= 1) {
    return(value)
  }

  # Take data from the value
  data <- value[1]
  div <- value[-1]
  # re-arrange the values i.e less values than data values come before,
  # and all  greater values come after.
  v1 <- div[div < data]
  v2 <- div[div >= data]

  v1 <- QSort(v1) # do quick sort for v1 values
  v2 <- QSort(v2)# do quick sort for v1 values
  return(c(v1, data, v2)) # return value
}
#assigning values
QSort(c(9, 6, 2, -1, 0, 20, 8, 7, 1))
```

```
> QSort(c(9, 6, 2, -1, 0, 20, 8, 7, 1))
[1] -1  0  1  2  6  7  8  9 20
> 
```

Binary Search Trees

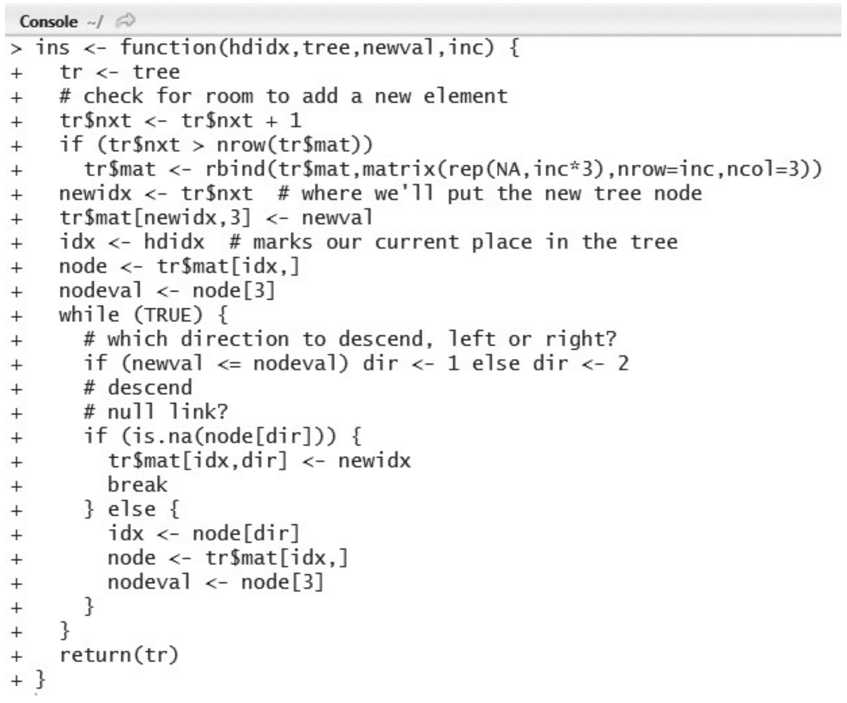
- A binary search tree is a data structure where the left child of the root is lesser than the right child of the root.
- In order to create the binary search tree in R, the following program is used.

Function 1: To create a new tree



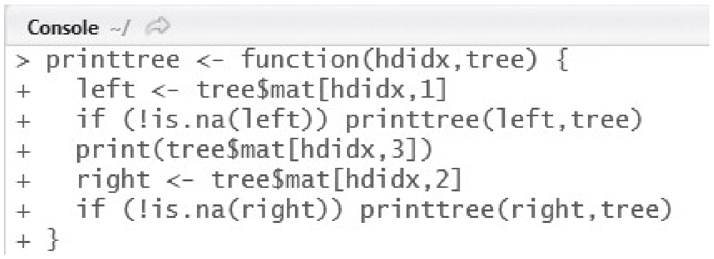
```
> newtree <- function(firstval,inc) {
+   m <- matrix(rep(NA,inc*3),nrow=inc,ncol=3)
+   m[1,3] <- firstval
+   return(list(mat=m,nxt=2,inc=inc))
+ }
```

Function 2: To insert the values into the tree such that it has the properties of the binary search tree

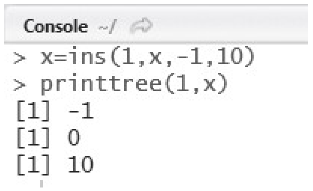


```
> ins <- function(hdidx,tree,newval,inc) {
+   tr <- tree
+   # check for room to add a new element
+   tr$nxt <- tr$nxt + 1
+   if (tr$nxt > nrow(tr$mat))
+     tr$mat <- rbind(tr$mat,matrix(rep(NA,inc*3),nrow=inc,ncol=3))
+   newidx <- tr$nxt  # where we'll put the new tree node
+   tr$mat[newidx,3] <- newval
+   idx <- hdidx  # marks our current place in the tree
+   node <- tr$mat[idx,]
+   nodeval <- node[3]
+   while (TRUE) {
+     # which direction to descend, left or right?
+     if (newval <= nodeval) dir <- 1 else dir <- 2
+     # descend
+     # null link?
+     if (is.na(node[dir])) {
+       tr$mat[idx,dir] <- newidx
+       break
+     } else {
+       idx <- node[dir]
+       node <- tr$mat[idx,]
+       nodeval <- node[3]
+     }
+   }
+   return(tr)
+ }
```

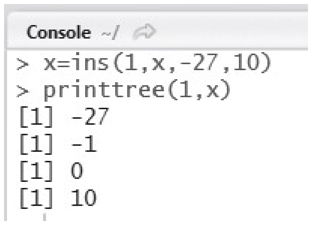
Function 3: To print the values in sorted order using in-order traversal In order to print the values of the tree in in-order



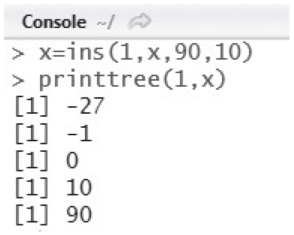
```
Console ~/
> printtree <- function(hdidx,tree) {
+   left <- tree$mat[hdidx,1]
+   if (!is.na(left)) printtree(left,tree)
+   print(tree$mat[hdidx,3])
+   right <- tree$mat[hdidx,2]
+   if (!is.na(right)) printtree(right,tree)
+ }
```



```
Console ~/
> x=ins(1,x,-1,10)
> printtree(1,x)
[1] -1
[1] 0
[1] 10
```



```
Console ~/
> x=ins(1,x,-27,10)
> printtree(1,x)
[1] -27
[1] -1
[1] 0
[1] 10
```



```
Console ~/
> x=ins(1,x,90,10)
> printtree(1,x)
[1] -27
[1] -1
[1] 0
[1] 10
[1] 90
```

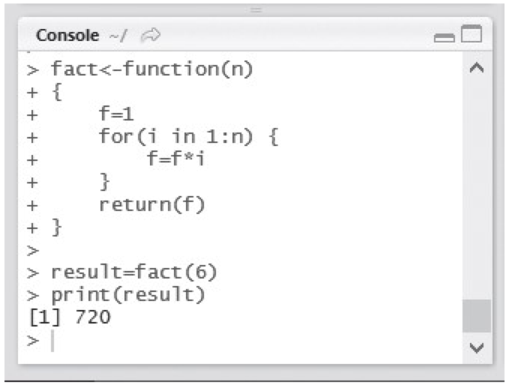
6.5 Recursion

Recursion is a technique where a function calls itself. The concept of recursion is most used when a large problem needs to be broken down into simpler sub-problems in order to simplify the complexity of computation.

The factorial of a number *n is denoted by n! and is defined as*

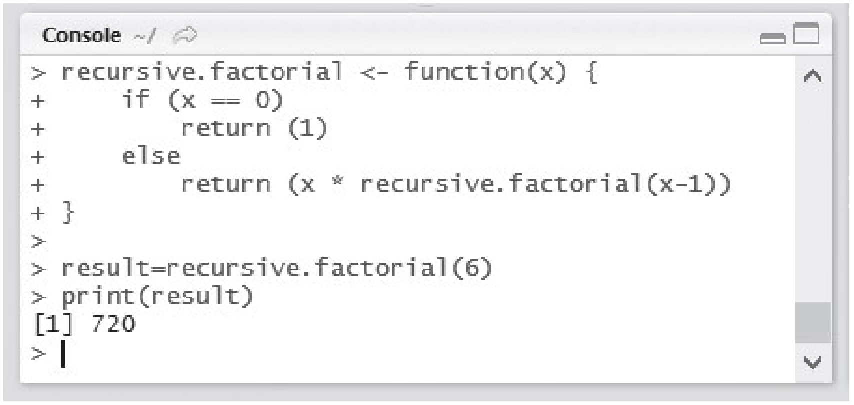
n! = n x n x (n – 1) x (n – 2) x … x 2 x 1

Following this definition, 6! = 6 x 5 x 4 x 3 x 2 x 1 = 720

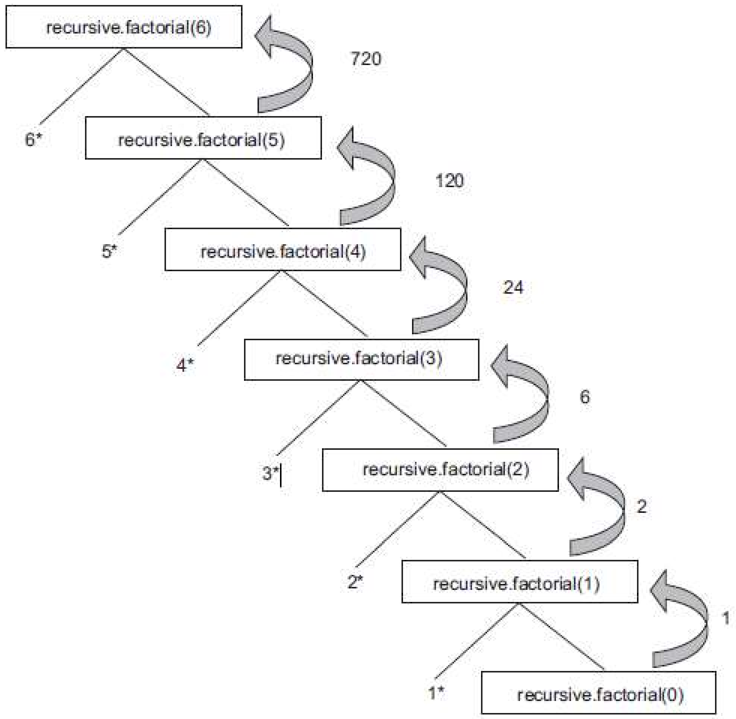


```
> fact<-function(n)
+ {
+     f=1
+     for(i in 1:n) {
+         f=f*i
+     }
+     return(f)
+ }
>
> result=fact(6)
> print(result)
[1] 720
>
```

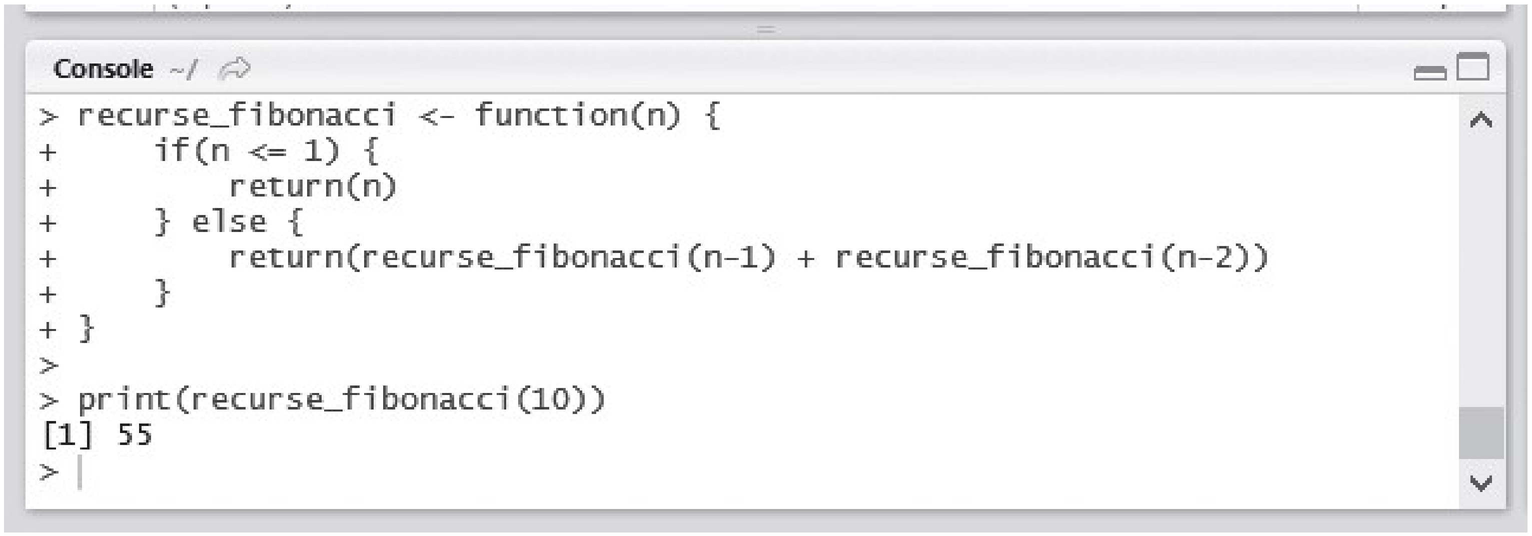
Example 5, Finding Factorial of a Number using Recursion



```
> recursive.factorial <- function(x) {
+     if (x == 0)
+         return (1)
+     else
+         return (x * recursive.factorial(x-1))
+ }
>
> result=recursive.factorial(6)
> print(result)
[1] 720
> 
```

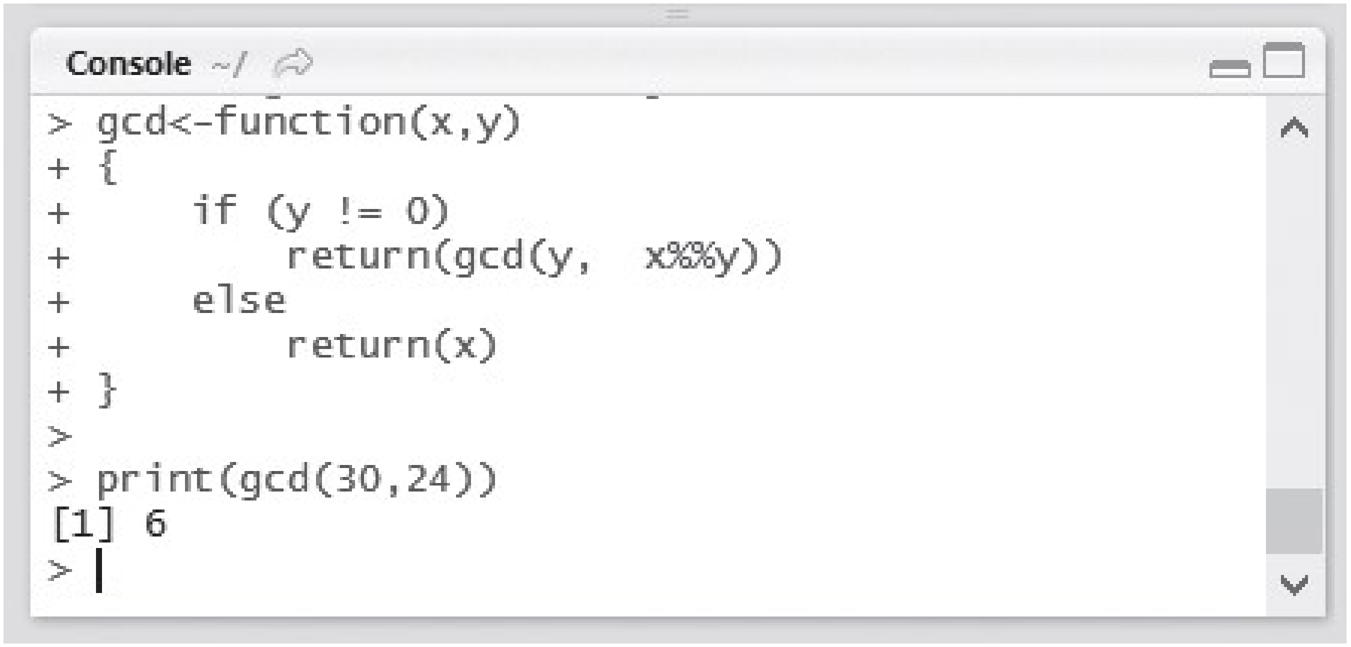


Example 7, Finding nth Fibonacci Number using Recursion



```
> recurse_fibonacci <- function(n) {
+     if(n <= 1) {
+         return(n)
+     } else {
+         return(recurse_fibonacci(n-1) + recurse_fibonacci(n-2))
+     }
+ }
>
> print(recurse_fibonacci(10))
[1] 55
> 
```

Example 8, GCD using Recursion



```
> gcd<-function(x,y)
+ {
+     if (y != 0)
+         return(gcd(y,  x%%y))
+     else
+         return(x)
+ }
>
> print(gcd(30,24))
[1] 6
> 
```

6.6 Loading an R Package

- R comes with numerous packages that are made available to install R onto the computer.
- packages need to be first loaded into the workspace

  library(package_name)
- To unload a package from the workspace, the following statement is used

  detach(package: package_name)

- **Methods of Loading**
- Loading a package is fairly simple in R. This can be done using the following statement:

  install.packages(“package_name”)
- Example:

  install.packages(“ggplot”)

  following two commands are used to include the packages:

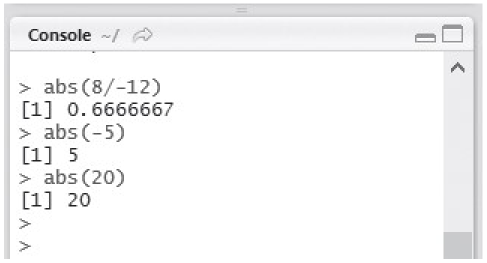
  install.packages(“ggplot2”, lib=”/data/Rpackages/”)

  library(ggplot2, lib.loc=”/data/Rpackages/”)

6.7 Mathematical Functions in R

- R provides a large number of mathematical and statistical functions to perform easy calculations.
- abs(numeric) This function takes in a numeric value and returns the absolute value of the input. This operation is equivalent to finding the mod of a number.

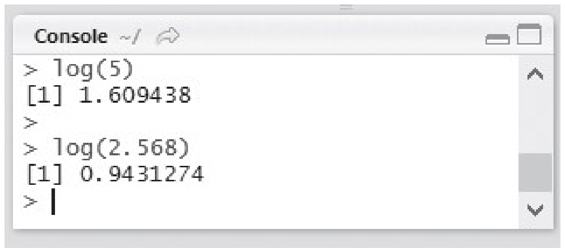
$$\text{abs}(x) = \begin{cases} -x, & x < 0 \\ x, & x \geq 0 \end{cases}$$



```
> abs(8/-12)
[1] 0.6666667
> abs(-5)
[1] 5
> abs(20)
[1] 20
>
>
```

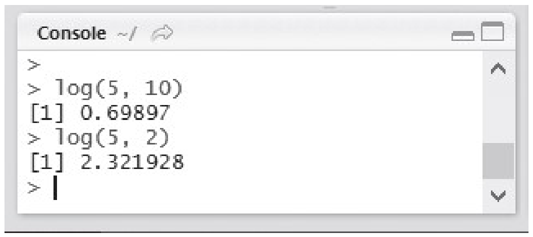
Fig. 6.25 Absolute value

- log(numeric) This function returns the natural logarithm (to base e) of a numeric value as shown in Fig. 6.26
- The log function also allows us to provide an optional argument, which explicitly change the base from *e* to any other base, as shown in Fig. 6.27



```
> log(5)
[1] 1.609438
>
> log(2.568)
[1] 0.9431274
>
```

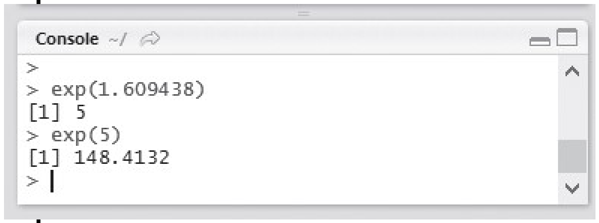
Fig. 6.26 Logarithm



```
>
> log(5, 10)
[1] 0.69897
> log(5, 2)
[1] 2.321928
>
```

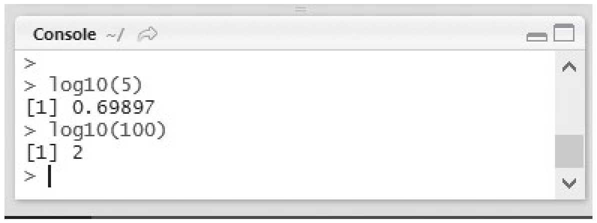
Fig. 6.27 Logarithm to variable base

- *exp(numeric)* This function is the inverse of the log function, or the antilog. The output is calculated at e raised to the power of the numeric value (refer to Fig. 6.28).
- *log10(numeric)* This function returns the logarithm of the numeric value but with base 10.
- *sqrt(numeric)* This function calculates the positive square root of the given numeric value.



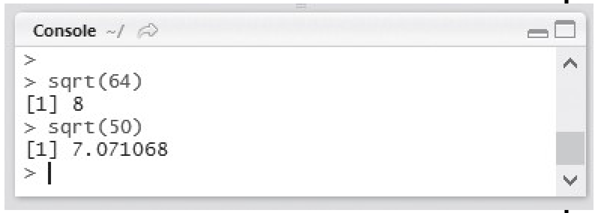
```
>
> exp(1.609438)
[1] 5
> exp(5)
[1] 148.4132
>
```

Fig. 6.28 Antilog



```
>
> log10(5)
[1] 0.69897
> log10(100)
[1] 2
>
```

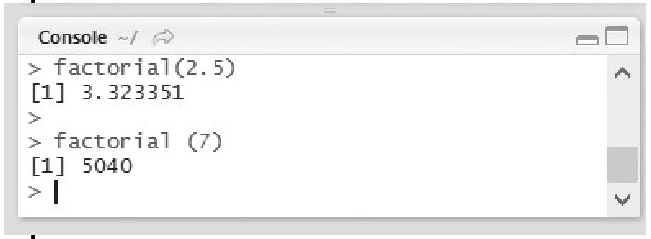
Fig. 6.29 Logarithm to base 10



```
>
> sqrt(64)
[1] 8
> sqrt(50)
[1] 7.071068
>
```

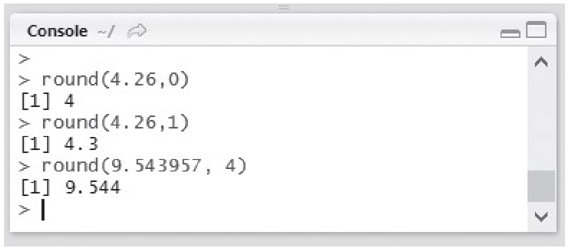
Fig. 6.30 Square root

- *factorial(numeric)* This function returns the factorial of a number.
- *round (numeric, digit)* This function takes in two arguments—a numeric and the precision (digits).
- *signif(numeric, digit)* This function takes in two arguments again—a numeric value and the precision argument.



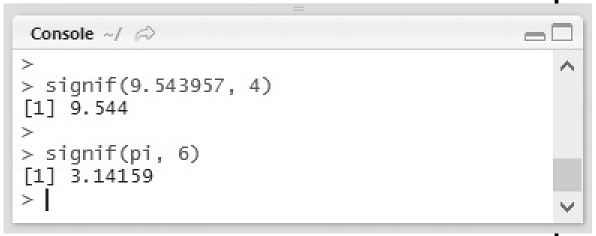
```
> factorial(2.5)
[1] 3.323351
>
> factorial (7)
[1] 5040
>
```

Fig. 6.31 Factorial



```
>
> round(4.26,0)
[1] 4
> round(4.26,1)
[1] 4.3
> round(9.543957, 4)
[1] 9.544
>
```

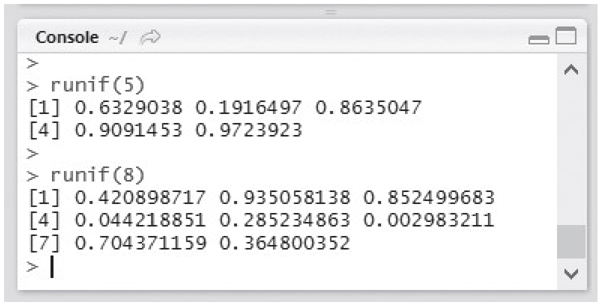
Fig. 6.32 Rounding off



```
>
> signif(9.543957, 4)
[1] 9.544
>
> signif(pi, 6)
[1] 3.14159
>
```

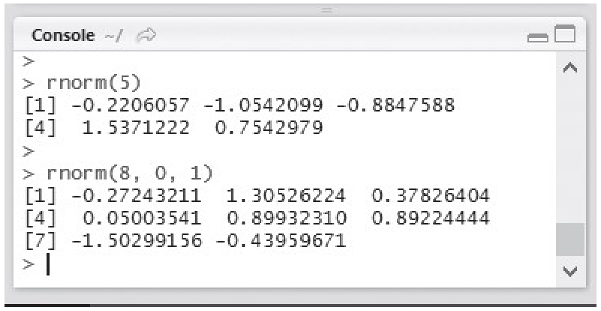
Fig. 6.33 Significant digits

- *runif(numeric)* This produces numeric number of uniform random numbers between 0 and 1.
- *rnorm(numeric)* This produces numeric number of random numbers from a normal distribution.



```
>
> runif(5)
[1] 0.6329038 0.1916497 0.8635047
[4] 0.9091453 0.9723923
>
> runif(8)
[1] 0.420898717 0.935058138 0.852499683
[4] 0.044218851 0.285234863 0.002983211
[7] 0.704371159 0.364800352
>
```

Fig. 6.34 Random number generator



```
>
> rnorm(5)
[1] -0.2206057 -1.0542099 -0.8847588
[4]  1.5371222  0.7542979
>
> rnorm(8, 0, 1)
[1] -0.27243211  1.30526224  0.37826404
[4]  0.05003541  0.89932310  0.89224444
[7] -1.50299156 -0.43959671
>
```

Fig. 6.35 Random number generator from normal distribution

6.8 Cumulative Sums and Products

- The cumulative functions in R provide a way of progressively calculating the sum or product of a sequence

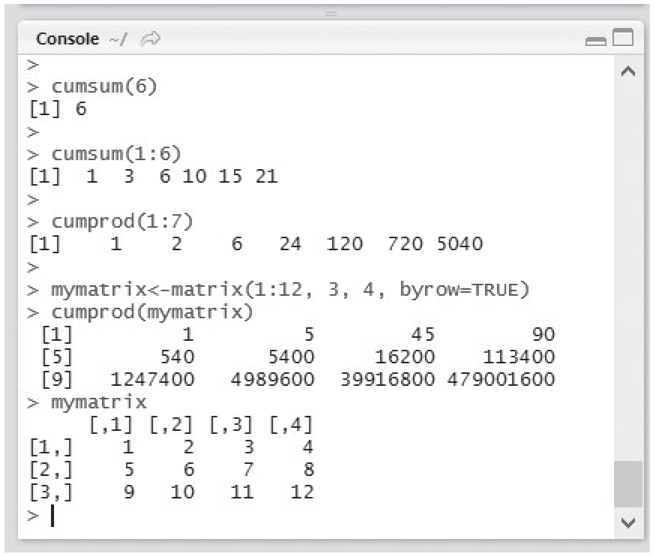
cumsum(x)

cumprod(x)

- Two interesting observations can be noted in the case of cumulative functions:

1. If the supplied object is a matrix, cumprod() using the column major approach to convert x into a vector.

2. If a data frame is sent as an argument, the function is applied only to each row.

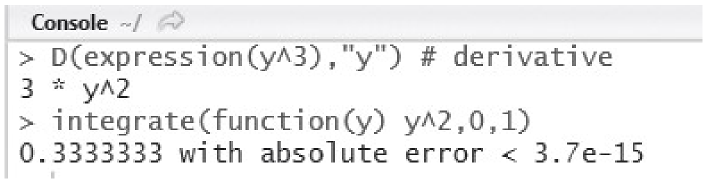


```
>
> cumsum(6)
[1] 6
>
> cumsum(1:6)
[1]  1  3  6 10 15 21
>
> cumprod(1:7)
[1]    1    2    6   24  120  720 5040
>
> mymatrix<-matrix(1:12, 3, 4, byrow=TRUE)
> cumprod(mymatrix)
 [1]         1         5        45        90
 [5]       540      5400     16200    113400
 [9]   1247400   4989600  39916800 479001600
> mymatrix
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
>
```

Fig. 6.37 Cumulative functions

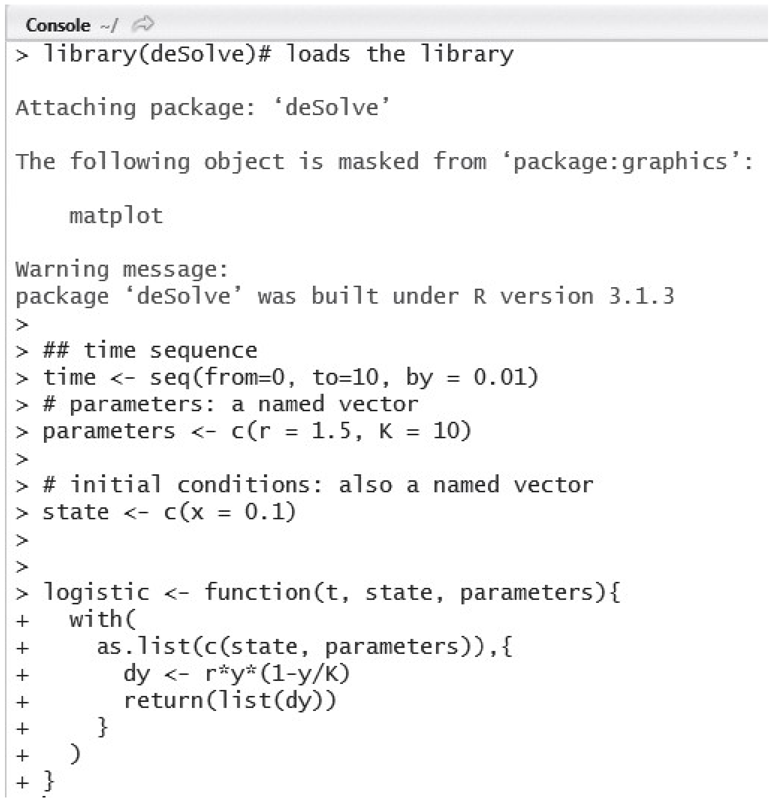
6.9 Calculus in R

- R has calculus capabilities for calculating derivative expressions and numerical integration
- Figure 6.38 demonstrates the use of the *D() function in R to calculate* the derivative of a simple function in R.



```
> D(expression(y^3),"y") # derivative
3 * y^2
> integrate(function(y) y^2,0,1)
0.3333333 with absolute error < 3.7e-15
```

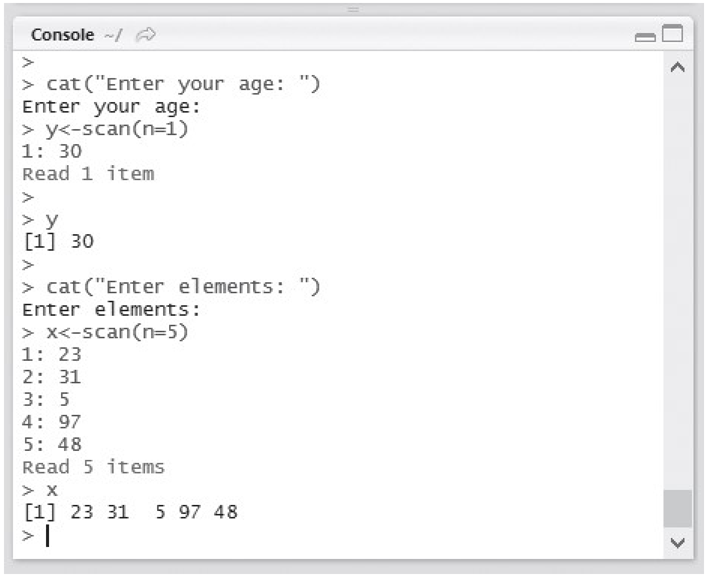
Fig. 6.38 Derivative calculation


Console ~/

```
> library(deSolve)# loads the library

Attaching package: ‘deSolve’

The following object is masked from ‘package:graphics’:

    matplot

Warning message:
package ‘deSolve’ was built under R version 3.1.3
>
> ## time sequence
> time <- seq(from=0, to=10, by = 0.01)
> # parameters: a named vector
> parameters <- c(r = 1.5, K = 10)
>
> # initial conditions: also a named vector
> state <- c(x = 0.1)
>
>
> logistic <- function(t, state, parameters){
+   with(
+     as.list(c(state, parameters)),{
+       dy <- r*y*(1-y/K)
+       return(list(dy))
+     }
+   )
+ }
```

Fig. 6.39 Use of deSolve package

Using scan() Function



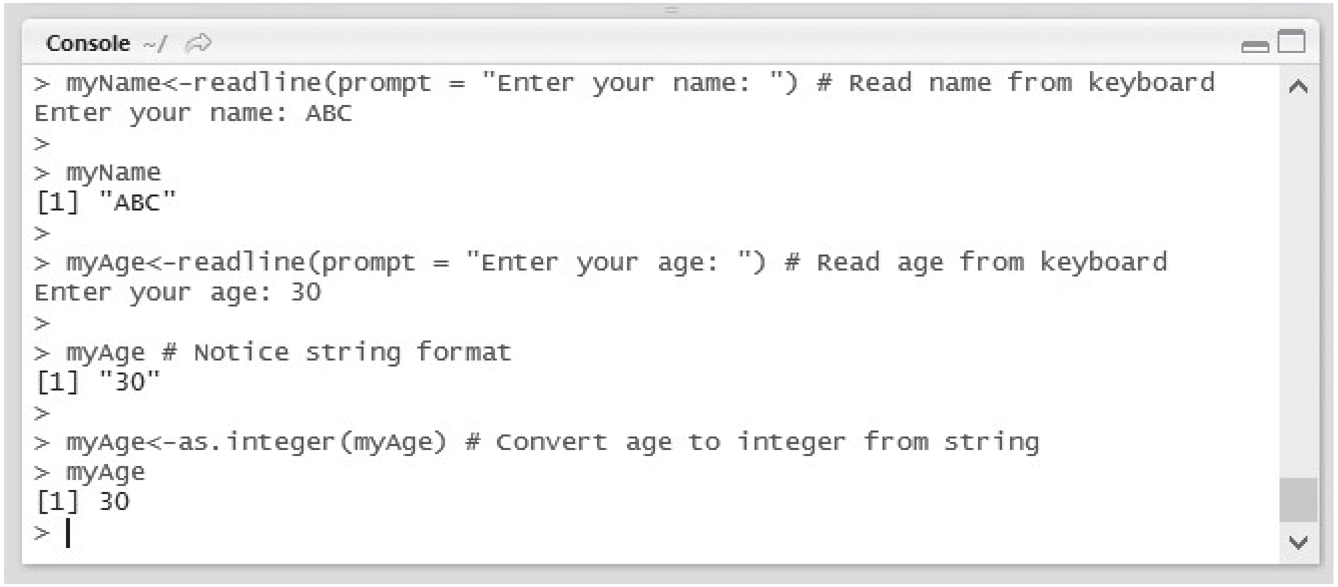
```
>
> cat("Enter your age: ")
Enter your age:
> y<-scan(n=1)
1: 30
Read 1 item
>
> y
[1] 30
>
> cat("Enter elements: ")
Enter elements:
> x<-scan(n=5)
1: 23
2: 31
3: 5
4: 97
5: 48
Read 5 items
> x
[1] 23 31  5 97 48
> 
```

Fig. 6.42 Using *scan() function*

6.10 Input and Output Operations

Using readline() Function

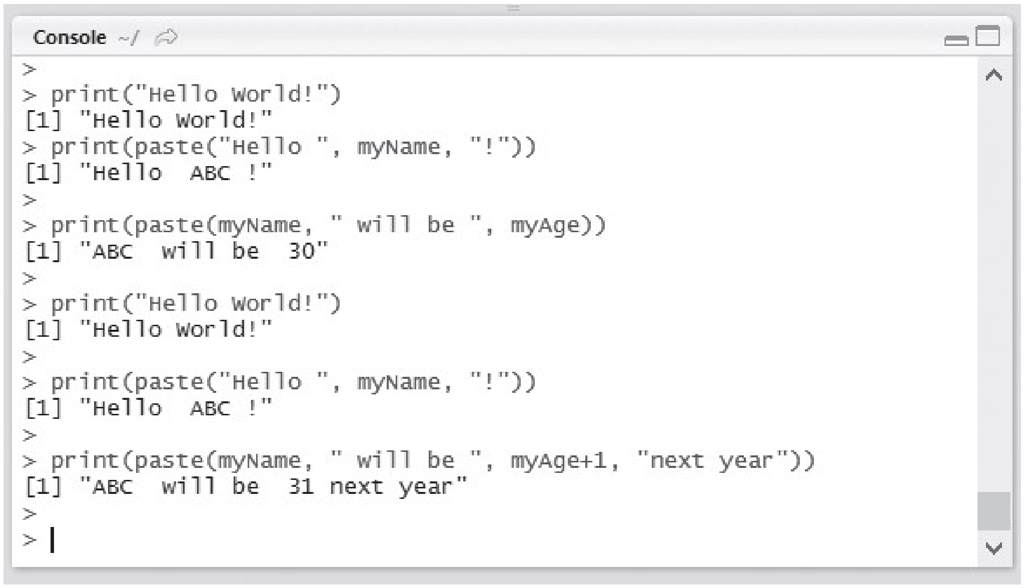
readline (prompt = “ ”)



```
> myName<-readline(prompt = "Enter your name: ") # Read name from keyboard
Enter your name: ABC
>
> myName
[1] "ABC"
>
> myAge<-readline(prompt = "Enter your age: ") # Read age from keyboard
Enter your age: 30
>
> myAge # Notice string format
[1] "30"
>
> myAge<-as.integer(myAge) # Convert age to integer from string
> myAge
[1] 30
> 
```

Displaying Output and Results

- Function that can be used to produce output is the *print() function, which is similar to the I/O* function in most other programming languages.



Console ~/

```
>
> print("Hello World!")
[1] "Hello World!"
> print(paste("Hello ", myName, "!"))
[1] "Hello  ABC !"
>
> print(paste(myName, " will be ", myAge))
[1] "ABC  will be  30"
>
> print("Hello World!")
[1] "Hello World!"
>
> print(paste("Hello ", myName, "!"))
[1] "Hello  ABC !"
>
> print(paste(myName, " will be ", myAge+1, "next year"))
[1] "ABC  will be  31 next year"
>
> |
```