UNIT V R LANGUAGE

Overview, Programming structures: Control statements - Operators - Functions - Environment and scope issues - Recursion -Replacement functions, R data structures: Vectors - Matrices and arrays - Lists -Data frames -Classes, Input/output, String manipulations

---

OVERVIEW OF R PROGRAMMING

R is a programming language and software environment for statistical analysis, graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team.

The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions. R allows integration with the procedures written in the C, C++, .Net, Python or FORTRAN languages for efficiency.

R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac. R is free software distributed under a GNU-style copy left, and an official part of the GNU project called GNU S.

Evolution of R

R was initially written by **Ross Ihaka** and **Robert Gentleman** at the Department of Statistics of the University of Auckland in Auckland, New Zealand. R made its first appearance in 1993.

- A large group of individuals has contributed to R by sending code and bug reports.
- Since mid-1997 there has been a core group (the "R Core Team") who can modify the R source code archive.

Features of R

As stated earlier, R is a programming language and software environment for statistical analysis, graphics representation and reporting. The following are the important features of R

- R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.
- R has an effective data handling and storage facility,
- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
- R provides a large, coherent and integrated collection of tools for data analysis.
- R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.

Basic operations of R

```
1+1          [1] 2
2*3          [1] 6
1==1          [1] TRUE
3<4           [1] TRUE
2+2==5       [1] FALSE
```

Variables

R is case Sensitive

```
x<-42
x/2
> x<-42                               x> x/2
x<-"ABC"                              [1] 21
> x                                   > x
[1] "ABC"                             [1] 4
```

Control statements are expressions used to control the execution and flow of the program based on the conditions provided in the statements. These structures are used to make a decision after assessing the variable. In this article, we'll discuss all the control statements with the examples.

In R programming, there are 8 types of control statements as follows:

- if condition
- if-else condition
- for loop
- nested loops
- while loop
- repeat and break statement
- return statement
- next statement

if condition

This control structure checks if the expression provided in parenthesis is true or not. If true, the execution of the statements in braces {} continues.

Syntax:

```
if(expression){
   statements
   ....
```

```
    ....
}
```

Example:

```
x <- 100

if(x > 10){
print(paste(x, "is greater than 10"))
}
```

Output:

```
[1] "100 is greater than 10"
```

if-else condition

It is similar to if condition but when the test expression in if condition fails, then statements in else condition are executed.

Syntax:

```
if(expression){
    statements
    ....
    ....
}
else{
    statements
    ....
    ....
}
```

Example:

```
x <- 5
# Check value is less than or greater than 10
if(x > 10){
  print(paste(x, "is greater than 10"))
}else{
  print(paste(x, "is less than 10"))
}
```

Output:

```
[1] "5 is less than 10"
```

for loop

It is a type of loop or sequence of statements executed repeatedly until exit condition is reached.

Syntax:

```
for(value in vector){
    statements
    ....
    ....
}
```

Example:

```
x <- letters[4:10]
for(i in x){
  print(i)
}
```

Output:

```
[1] "d"
[1] "e"
[1] "f"
[1] "g"
[1] "h"
[1] "i"
[1] "j"
```

Nested loops

Nested loops are similar to simple loops. Nested means loops inside loop. Moreover, nested loops are used to manipulate the matrix.

Example:

```
# Defining matrix
m <- matrix(2:15, 2)
for (r in seq(nrow(m))) {
  for (c in seq(ncol(m))) {
    print(m[r, c])
  }
}
```

Output:

```
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
[1] 12
```

```
[1] 14
[1] 3
[1] 5
[1] 7
[1] 9
[1] 11
[1] 13
[1] 15
```

while loop

while loop is another kind of loop iterated until a condition is satisfied. The testing expression is checked first before executing the body of loop.

Syntax:

```
while(expression){
   statement
   ....
   ....
}
```

Example:

```
x = 1
# Print 1 to 5
while(x <= 5){
  print(x)
  x = x + 1
}
```

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

repeat loop and break statement

repeat is a loop which can be iterated many number of times but there is no exit condition to come out from the loop. So, break statement is used to exit from the loop. break statement can be used in any type of loop to exit from the loop.

Syntax:

```
repeat {
  statements
```

```
   ....
   ....
   if(expression) {
     break
   }
}
```

Example:

```
x = 1

# Print 1 to 5
repeat{
  print(x)
  x = x + 1
  if(x > 5){
    break
   }
}
```

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

return statement

return statement is used to return the result of an executed function and returns control to the calling function.

Syntax:

```
return(expression)
```

Example:

```
func <- function(x){
  if(x > 0){
    return("Positive")
  }else if(x < 0){
    return("Negative")
  }else{
    return("Zero")
  }
}
func(1)
func(0)
func(-1)
```

Output:

```
[1] "Positive"
[1] "Zero"
[1] "Negative"
```

next statement

next statement is used to skip the current iteration without executing the further statements and continues the next iteration cycle without terminating the loop.

Example:

```
# Defining vector
x <- 1:10
# Print even numbers
for(i in x){
  if(i%%2 != 0){
    next #Jumps to next loop
  }
  print(i)
}
```

Output:

```
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
```

OPERATORS

R supports majorly four kinds of binary operators between a set of operands. In this article, we will see various types of operators in R Programming language and their usage.

Types of the operator in R language

- Arithmetic Operators
- Logical Operators
- Relational Operators

- Assignment Operators
- Miscellaneous Operators

Arithmetic Operators

Arithmetic Operators modulo using the specified operator between operands, which may be either scalar values, complex numbers, or vectors. The R operators are performed element-wise at the corresponding positions of the vectors.

1. **Addition operator (+)**

   The values at the corresponding positions of both operands are added. Consider the following R operator snippet to add two vectors:

   ```
   a <- c (1, 0.1)
   b <- c (2.33, 4)
   print (a+b)
   ```

   Output : 3.33 4.10

2. **Subtraction Operator (-)**

   The second operand values are subtracted from the first. Consider the following R operator snippet to subtract two variables:

   ```
   a <- 6
    b <- 8.4
   print (a-b)
   ```

   Output : -2.4

3. **Multiplication Operator (*)**

   The multiplication of corresponding elements of vectors and Integers are multiplied with the use of the '*' operator.

   ```
   B= c(4,4)
   C= c(5,5)
   print (B*C)
   ```

   Output : 20 20

4. **Division Operator (/)**

   The first operand is divided by the second operand with the use of the '/' operator.

   ```
   a <- 10
   b <- 5
   print (a/b)
   ```

   Output : 2

5. **Power Operator (^)**

   The first operand is raised to the power of the second operand.

   ```
   a <- 4
   b <- 5
   ```

```
print(a^b)
```
Output : 1024

6. **Modulo Operator (%%)**

    The remainder of the first operand divided by the second operand is returned.
    ```
    list1<- c(2, 22)
    list2<-c(2,4)
    print(list1 %% list2)
    ```
    Output : 0 2

# R program to illustrate the use of Arithmetic operators
```
vec1 <- c(0, 2)
vec2 <- c(2, 3)
cat ("Addition of vectors :", vec1 + vec2, "\n")
cat ("Subtraction of vectors :", vec1 - vec2, "\n")
cat ("Multiplication of vectors :", vec1 * vec2, "\n")
cat ("Division of vectors :", vec1 / vec2, "\n")
cat ("Modulo of vectors :", vec1 %% vec2, "\n")
cat ("Power operator :", vec1 ^ vec2)
```

Output
```
Addition of vectors : 2 5
Subtraction of vectors : -2 -1
Multiplication of vectors : 0 6
Division of vectors : 0 0.6666667
Modulo of vectors : 0 2
Power operator : 0 8
```

Logical Operators

Logical Operators in R simulate element-wise decision operations, based on the specified operator between the operands, which are then evaluated to either a True or False boolean value. Any non-zero integer value is considered as a TRUE value, be it a complex or real number.

1. **Element-wise Logical AND operator (&)**

    Returns True if both the operands are True.
    ```
    list1 <- c(TRUE, 0.1)
    list2 <- c(0,4+3i)
    print(list1 & list2)
    ```
    Output : FALSE   TRUE

Any non zero integer value is considered as a TRUE value, be it complex or real number.

2. **Element-wise Logical OR operator (|)**

   Returns True if either of the operands is True.

   ```
   list1 <- c(TRUE, 0.1)
   list2 <- c(0,4+3i)
   print(list1|list2)
   ```

   Output : TRUE TRUE

   NOT operator (!)

   A unary operator that negates the status of the elements of the operand.

   ```
   list1 <- c(0,FALSE)
   print(!list1)
   ```

   Output : TRUE TRUE

3. **Logical AND operator (&&)**

   Returns True if both the first elements of the operands are True.

   ```
   list1 <- c(TRUE, 0.1)
   list2 <- c(0,4+3i)
   print(list1[1] && list2[1])
   ```

   Output : FALSE

   Compares just the first elements of both the lists.

4. **Logical OR operator (||)**

   Returns True if either of the first elements of the operands is True.

   ```
   list1 <- c(TRUE, 0.1)
   list2 <- c(0,4+3i)
   print(list1[1]||list2[1])
   ```

   Output : TRUE

```
# R program to illustrate the use of Logical operators
vec1 <- c(0,2)
vec2 <- c(TRUE,FALSE)
# Performing operations on Operands
cat ("Element wise AND :", vec1 & vec2, "\n")
cat ("Element wise OR :", vec1 | vec2, "\n")
cat ("Logical AND :", vec1[1] && vec2[1], "\n")
cat ("Logical OR :", vec1[1] || vec2[1], "\n")
cat ("Negation :", !vec1)
```

Output

Element wise AND : FALSE FALSE
Element wise OR : TRUE TRUE
Logical AND : FALSE
Logical OR : TRUE
Negation : TRUE FALSE

Relational Operators

The Relational Operators in R carry out comparison operations between the corresponding elements of the operands. Returns a boolean TRUE value if the first operand satisfies the relation compared to the second. A TRUE value is always considered to be greater than the FALSE.

1. **Less than (<)**

   Returns TRUE if the corresponding element of the first operand is less than that of the second operand. Else returns FALSE.
   ```
   list1 <- c(TRUE, 0.1,"apple")
   list2 <- c(0,0.1,"bat")
   print(list1<list2)
   ```
   Output : FALSE FALSE TRUE
2. **Less than equal to (<=)**

   Returns TRUE if the corresponding element of the first operand is less than or equal to that of the second operand. Else returns FALSE.
   ```
   list1 <- c(TRUE, 0.1, "apple")
   list2 <- c(TRUE, 0.1, "bat")
   # Convert lists to character strings
   list1_char <- as.character(list1)
   list2_char <- as.character(list2)
   # Compare character strings
   print(list1_char <= list2_char)
   ```
   Output : TRUE TRUE TRUE
3. **Greater than (>)**

   Returns TRUE if the corresponding element of the first operand is greater than that of the second operand. Else returns FALSE.
   ```
   list1 <- c(TRUE, 0.1, "apple")
   list2 <- c(TRUE, 0.1, "bat")
   print(list1_char > list2_char)
   ```
   Output : FALSE FALSE FALSE
4. **Greater than equal to (>=)**

Returns TRUE if the corresponding element of the first operand is greater or equal to that of the second operand. Else returns FALSE.

```
list1 <- c(TRUE, 0.1, "apple")
list2 <- c(TRUE, 0.1, "bat")
print(list1_char >= list2_char)
```

Output : TRUE TRUE FALSE

5. **Not equal to (!=)**

Returns TRUE if the corresponding element of the first operand is not equal to the second operand. Else returns FALSE.

```
list1 <- c(TRUE, 0.1,'apple')
list2 <- c(0,0.1,"bat")
print(list1!=list2)
```

Output : TRUE FALSE TRUE

The following R code illustrates the usage of all Relational Operators in R:

```
# R program to illustrate the use of Relational operators
vec1 <- c(0, 2)
vec2 <- c(2, 3)
cat ("Vector1 less than Vector2 :", vec1 < vec2, "\n")
cat ("Vector1 less than equal to Vector2 :", vec1 <= vec2, "\n")
cat ("Vector1 greater than Vector2 :", vec1 > vec2, "\n")
cat ("Vector1 greater than equal to Vector2 :", vec1 >= vec2, "\n")
cat ("Vector1 not equal to Vector2 :", vec1 != vec2, "\n")
```

Output

```
Vector1 less than Vector2 : TRUE TRUE
Vector1 less than equal to Vector2 : TRUE TRUE
Vector1 greater than Vector2 : FALSE FALSE
Vector1 greater than equal to Vector2 : FALSE FALSE
Vector1 not equal to Vector2 : TRUE TRUE
```

Assignment Operators

Assignment Operators in R are used to assign values to various data objects in R. The objects may be integers, vectors, or functions. These values are then stored by the assigned variable names. There are two kinds of assignment operators: Left and Right

1. **Left Assignment (<- or <<- or =)**

Assigns a value to a vector.

```
vec1 = c("ab", TRUE)
print (vec1)
```

Output : "ab"   "TRUE"

2. **Right Assignment (-> or ->>)**

Assigns value to a vector.

```
c("ab", TRUE) ->> vec1
print (vec1)
```

Output : "ab"   "TRUE"

```
# R program to illustrate the use of Assignment operators
vec1 <- c(2:5)
c(2:5) ->> vec2
vec3 <<- c(2:5)
vec4 = c(2:5)
c(2:5) -> vec5
# Performing operations on Operands
cat ("vector 1 :", vec1, "\n")
cat("vector 2 :", vec2, "\n")
cat ("vector 3 :", vec3, "\n")
cat("vector 4 :", vec4, "\n")
cat("vector 5 :", vec5)
```

Output

```
vector 1 : 2 3 4 5
vector 2 : 2 3 4 5
vector 3 : 2 3 4 5
vector 4 : 2 3 4 5
vector 5 : 2 3 4 5
```

Miscellaneous Operators

Miscellaneous Operator are the mixed operators in R that simulate the printing of sequences and assignment of vectors, either left or right-handed.

1. **%in% Operator**

Checks if an element belongs to a list and returns a boolean value TRUE if the value is present else FALSE.

```
val <- 0.1
list1 <- c(TRUE, 0.1,"apple")
print (val %in% list1)
```

Output : TRUE

Checks for the value 0.1 in the specified list. It exists, therefore, prints TRUE.

2. **%*% Operator**

This operator is used to multiply a matrix with its transpose. Transpose of the matrix is obtained by interchanging the rows to columns and columns to rows. The number of columns of the first matrix must be equal to the number of rows of the second matrix. Multiplication of the matrix A with its transpose, B, produces a square matrix.

A_{r*c} x B_{c*r}–> P_{r*r}

```
mat = matrix(c(1,2,3,4,5,6),nrow=2,ncol=3)
    print (mat)
    print( t(mat))
    pro = mat %*% t(mat)
    print(pro)
```

Output :

```
      [,1] [,2] [,3]      #original matrix of order 2x3
   [1,]   1    3    5
   [2,]   2    4    6
      [,1] [,2]           #transposed matrix of order 3x2
   [1,]    1    2
   [2,]    3    4
   [3,]    5    6
      [,1] [,2]           #product matrix of order 2x2
   [1,]   35   44
   [2,]   44   56
```

The following R code illustrates the usage of all Miscellaneous Operators in R:

```
# R program to illustrate the use of Miscellaneous operators
mat <- matrix (1:4, nrow = 1, ncol = 4)
print("Matrix elements using : ")
print(mat)
product = mat %*% t(mat)
print("Product of matrices")
print(product,)
cat ("does 1 exist in prod matrix :", "1" %in% product)
```

Output

```
[1] "Matrix elements using : "
    [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[1] "Product of matrices"
    [,1]
[1,]   30
```

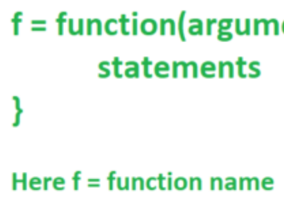
does 1 exist in prod matrix : FALSE

FUNCTIONS IN R PROGRAMMING

A function accepts input arguments and produces the output by executing valid R commands that are inside the function. Functions are useful when you want to perform a certain task multiple times. In R Programming Language when you are creating a function the function name and the file in which you are creating the function need not be the same and you can have one or more functions in R.

Creating a Function in R Programming

Functions are created in R by using the command function(). The general structure of the function file is as follows:



```
f = function(arguments){
        statements
}

Here f = function name
```

Parameters or Arguments in R Functions:

Parameters and arguments are the same term in functions. Parameters or arguments are the values passed into a function. A function can have any number of arguments, they are separated by comma in parentheses.

Example:

```
add_num <- function(a,b)
{
  sum_result <- a+b
  return(sum_result)
}
# calling add_num function
sum = add_num(35,34)
#printing result
print(sum)
```

Output

```
[1] 69
```

No. of Parameters:

Function should be called with right no. of parameters, neither less nor more or else it will give error.

Default Value of Parameter:

Some functions have default values, and you can also give default value in your user-defined functions. These values are used by functions if user doesn't pass any parameter value while calling a function.

Return Value:

You can use the return() function if you want your function to return the result.

Calling a Function in R

After creating a Function, you have to call the function to use it. Calling a function in R is very easy, you can call a function by writing it's name and passing possible parameters value.

Passing Arguments to Functions in R Programming Language

There are several ways you can pass the arguments to the function:

- **Case 1:** Generally in R, the arguments are passed to the function in the same order as in the function definition.
- **Case 2:** If you do not want to follow any order what you can do is you can pass the arguments using the names of the arguments in any order.
- **Case 3:** If the arguments are not passed the default values are used to execute the function.

```
# A simple R program to demonstrate passing arguments to a function
 Rectangle = function(length=5, width=4)
{
  area = length * width
  return(area)
}
print(Rectangle(2, 3))
print(Rectangle(width = 8, length = 4))
print(Rectangle())
```

Output

```
[1] 6
[1] 32
[1] 20
```

Types of Function in R Language

- **Built-in Function:** Built-in functions in R are pre-defined functions that are available in R programming languages to perform common tasks or operations.
- **User-defined Function:** R language allows us to write our own function.

Built-in Function in R Programming Language

Built-in Function are the functions that are already existing in R language and you just need to call them to use.

Here we will use built-in functions like sum(), max() and min().

```
# Find sum of numbers 4 to 6.
print(sum(4:6))
# Find max of numbers 4 and 6.
print(max(4:6))
# Find min of numbers 4 and 6.
print(min(4:6))
```

Output

```
[1] 15
[1] 6
[1] 4
```

Other Built-in Functions in R:

Functions	**Syntax**
MATHEMATICAL FUNCTIONS	
abs()	calculates a number’s absolute value.
sqrt()	calculates a number’s square root.
round()	rounds a number to the nearest integer.
exp()	calculates a number’s exponential value
log()	which calculates a number’s natural logarithm.
cos(), sin(), and tan()	calculates a number’s cosine, sine, and tang
STATISTICAL FUNCTIONS	
mean()	A vector’s arithmetic mean is determined by the mean() function.
median()	A vector’s median value is determined by the median() function.
cor()	calculates the correlation between two vectors.
var()	calculates the variance of a vector and calculates the standard deviation of a vector.

DATA MANIPULATION FUNCTIONS	
unique()	returns the unique values in a vector.
subset()	subsets a data frame based on conditions.
aggregate()	groups data according to a grouping variable.
order()	uses ascending or descending order to sort a vector.
FILE INPUT / OUTPUT FUNCTIONS	
read.csv()	reads information from a CSV file.
write.csv()	publishes information to write a CSV file.
read. table()	reads information from a tabular.
write.table()	creates a tabular file with data.

User-defined Functions in R Programming Language

User-defined functions are the functions that are created by the user. User defines the working, parameters, default parameter, etc. of that user-defined function. They can be only used in that specific code.

Example

```
evenOdd = function(x)
{
 if(x %% 2 == 0)
   return("even")
 else
   return("odd")
}
print(evenOdd(4))
print(evenOdd(3))
```

Output

```
[1] "even"
[1] "odd"
```

Multiple Input Multiple Output

The functions in R Language take multiple input objects but returned only one object as output, this is, however, not a limitation because you can create lists of all the outputs which you want to create and once the list is created you can access them into the elements of the list and get the answers which you want.

Let us consider this example to create a function “Rectangle” which takes “length” and “width” of the rectangle and returns area and perimeter of that rectangle. Since R Language can return only one object. Hence, create one object which is a list that contains “area” and “perimeter” and return the list.

```
Rectangle = function(length, width)
{
  area = length * width
  perimeter = 2 * (length + width)
  # create an object called result which is
  # a list of area and perimeter
  result = list("Area" = area, "Perimeter" = perimeter)
  return(result)
}
resultList = Rectangle(2, 3)
print(resultList["Area"])
print(resultList["Perimeter"])
```

Output

```
$Area
[1] 6
$Perimeter
[1] 10
```

Inline Functions in R

Sometimes creating an R script file, loading it, executing it is a lot of work when you want to just create a very small function. So, what we can do in this kind of situation is an inline function.

To create an inline function you have to use the function command with the argument x and then the expression of the function.

Example

```
f = function(x) x^2*4+x/3
print(f(4))
print(f(-2))
print(0)
```

Output

```
[1] 65.33333
[1] 15.33333
```

```
[1] 0
```

ENVIRONMENT AND SCOPE ISSUES

The environment is a virtual space that is triggered when an interpreter of a programming language is launched. Simply, the environment is a collection of all the objects, variables, and functions. Or, Environment can be assumed as a top-level object that contains the set of names/variables associated with some values. In this article, let us discuss creating a new environment in R programming, listing all environments, removing a variable from the environment, searching for a variable or function among environments and function environments with the help of examples.

Environment Differ from the List?

- Every object in an environment has a name.
- The environment has a parent environment.
- Environments follow reference semantics.

Create a New Environment

An environment in R programming can be created using new.env() function. Further, the variables can be accessed using the $ or [[ ]] operator. But, each variable is stored in different memory locations. There are four special environments: globalenv(), baseenv(), emptyenv() and environment()

Syntax: new.env(hash = TRUE)

Parameters:

hash: indicates logical value. If TRUE, environments uses a hash table

To know about more optional parameters, use below command in console: help("new.env")

```
# R program to illustrate Environments in R
# Create new environment
newEnv <- new.env()
# Assigning variables
newEnv$x <- 1
newEnv$y <- "GFG"
newEnv$z <- 1:10
# Print
print(newEnv$z)
```

Output:

```
[1]  1  2  3  4  5  6  7  8  9 10
```

List all Environments

Every environment has a parent environment but there is an empty environment that does not have any parent environment. All the environments can be listed using ls() function and search() function. ls() function also list out all the bindings of the variables in a particular environment.

Syntax:

ls()

search()

Parameters:

These functions need no argument

```
# R program to illustrate Environments in R

# Prints all the bindings and environments
# attached to Global Environment
ls()
# Prints bindings of newEnv
ls(newEnv)
# Lists all the environments of the parent environment
search()
```

Output:

```
[1] "al"     "e"      "e1"     "f"      "newEnv" "pts"    "x"      "y"
[9] "z"
[1] "x" "y" "z"
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

Removing a Variable From an Environment

A variable in an environment is deleted using rm() function. It is different from deleting entries from lists as entries in lists are set as NULL to be deleted. But, using rm() function, bindings are removed from the environment.

Syntax: rm(…)

Parameters:

…: indicates list of objects

Example:

```
# R program to illustrate Environments in R
# Remove newEnv
rm(newEnv)
# List
```

```
ls()
```
Output:
```
[1] "al"  "e"   "e1"  "f"   "pts" "x"   "y"   "z"
```
Search a Variable or Function Among Environments

A variable or a function can be searched in R programming by using where() function among all the environments and packages present. where() function is present in pryr package. This function takes only two arguments, the name of the object to search for and the environment from where to start the search.

Syntax: where(name)

Parameters:

name: indicates object to look for

Example:
```
# R program to illustrate Environments in R
# Install pryr package
install.packages("pryr")
# Load the package
library(pryr)
# Search
where("x")
where("mode")
```
Output:
```
<environment: R_GlobalEnv>
<environment: base>
```
Scope of Variable in R

Variables:

In R, variables are the containers for storing data values. They are reference, or pointers, to an object in memory which means that whenever a variable is assigned to an instance, it gets mapped to that instance. A variable in R can store a vector, a group of vectors or a combination of many R objects.

Example:
```
# R program to demonstrate variable assignment
# Assignment using equal operator
var1 = c(0, 1, 2, 3)
print(var1)
# Assignment using leftward operator
var2 <- c("Python", "R")
print(var2)
```

```
# A Vector Assignment
a = c(1, 2, 3, 4)
print(a)
b = c("Debi", "Sandeep", "Subham", "Shiba")
print(b)
# A group of vectors Assignment using list
c = list(a, b)
print(c)
```

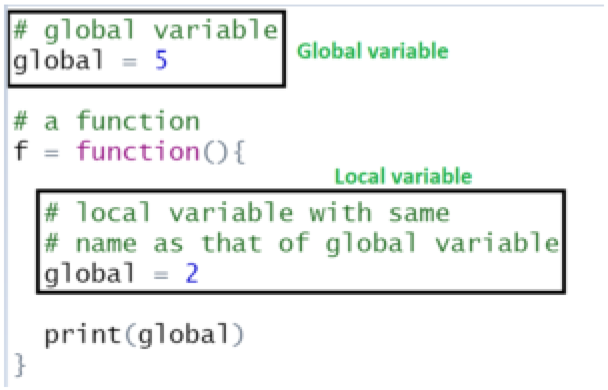
Output:

```
[1] 0 1 2 3
[1] "Python" "R"
[1] 1 2 3 4
[1] "Debi"    "Sandeep" "Subham"  "Shiba"
[[1]]
[1] 1 2 3 4
[[2]]
[1] "Debi"    "Sandeep" "Subham"  "Shiba"
```

Scope of a variable

The location where we can find a variable and also access it if required is called the scope of a variable. There are mainly two types of variable scopes:

- **Global Variables:** Global variables are those variables that exist throughout the execution of a program. It can be changed and accessed from any part of the program.
- **Local Variables:** Local variables are those variables that exist only within a certain part of a program like a function and are released when the function call ends.



```
# global variable
global = 5

# a function
f = function(){

  # local variable with same
  # name as that of global variable
  global = 2

  print(global)
}
```

Global Variable

As the name suggests, Global Variables can be accessed from any part of the program. They are available throughout the lifetime of a program. They are declared anywhere in the program outside all of the functions or blocks. Declaring global variables: Global variables are usually declared outside of all of the functions and blocks. They can be accessed from any portion of the program.

```
# R program to illustrate usage of global variables
# global variable
global = 5
# global variable accessed from within a function
display = function()
{
  print(global)
}
display()

# changing value of global variable
global = 10
display()
```

Output:

```
[1] 5
[1] 10
```

Local Variable

Variables defined within a function or block are said to be local to those functions. Local variables do not exist outside the block in which they are declared, i.e. they can not be accessed or used outside that block. Local variables are declared inside a block.

Example:

```
# R program to illustrate usage of local variables
func = function()
{
   # this variable is local to the function func() and cannot be accessed outside this function
  age = 18
}
print(age)
```

Output:

```
Error in print(age) : object 'age' not found
```

Accessing Global Variables

Global Variables can be accessed from anywhere in the code unlike local variables that have a scope restricted to the block of code in which they are created.

Example:

```
f = function() {
# a is a local variable here
a <-1
}
f()
# Can't access outside the function
print(a) # This'll give error
```

Output:

```
Error in print(a) : object 'a' not found
```