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

---

ARRAYS

Arrays are essential data storage structures defined by a fixed number of dimensions. Arrays are used for the allocation of space at contiguous memory locations.

In R Programming Language Uni-dimensional arrays are called vectors with the length being their only dimension. Two-dimensional arrays are called matrices, consisting of fixed numbers of rows and columns. R Arrays consist of all elements of the same data type. Vectors are supplied as input to the function and then create an array based on the number of dimensions.

Creating an Array

An R array can be created with the use of array() the function. A list of elements is passed to the array() functions along with the dimensions as required.

Syntax:

array(data, dim = (nrow, ncol, nmat), dimnames=names)

where

nrow: Number of rows

ncol : Number of columns

nmat: Number of matrices of dimensions nrow * ncol

dimnames : Default value = NULL.

Otherwise, a list has to be specified which has a name for each component of the dimension. Each component is either a null or a vector of length equal to the dim value of that corresponding dimension.

Uni-Dimensional Array

A vector is a uni-dimensional array, which is specified by a single dimension, length. A Vector can be created using ‘c()‘ function. A list of values is passed to the c() function to create a vector.

```
vec1 <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
print (vec1)
cat ("Length of vector : ", length(vec1))
```

Output:

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

Length of vector : 9

Multi-Dimensional Array

A two-dimensional matrix is an array specified by a fixed number of rows and columns, each containing the same data type. A matrix is created by using array() function to which the values and the dimensions are passed.

```
arr = array(2:13, dim = c(2, 3, 2))
print(arr)
```

Output:

```
, , 1
   [,1] [,2] [,3]
[1,]    2    4    6
[2,]    3    5    7
, , 2
   [,1] [,2] [,3]
[1,]    8   10   12
[2,]    9   11   13
```

Vectors of different lengths can also be fed as input into the array() function. However, the total number of elements in all the vectors combined should be equal to the number of elements in the matrices. The elements are arranged in the order in which they are specified in the function.

```
vec1 <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
vec2 <- c(10, 11, 12)
arr = array(c(vec1, vec2), dim = c(2, 3, 2))
print (arr)
```

Output:

```
, , 1
   [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
, , 2
   [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

Dimension of the Array

We will use dim function to find out the dimension of the R array.

```
arr = array(2:13, dim = c(2, 3, 2))
dim(arr)
```

Output:

[1] 2 3 2

This specifies the dimensions of the R array. In this case, we are creating a 3D array with dimensions 2x3x2. The first dimension has size 2, the second dimension has size 3, and the third dimension has size 2.

Naming of Arrays

The row names, column names and matrices names are specified as a vector of the number of rows, number of columns and number of matrices respectively. By default, the rows, columns and matrices are named by their index values.

```
row_names <- c("row1", "row2")
col_names <- c("col1", "col2", "col3")
mat_names <- c("Mat1", "Mat2")
arr = array(2:14, dim = c(2, 3, 2), dimnames = list(row_names, col_names, mat_names))
print (arr)
```

Output:

```
, , Mat1
    col1 col2 col3
row1    2    4    6
row2    3    5    7
, , Mat2
    col1 col2 col3
row1    8   10   12
row2    9   11   13
```

Accessing arrays

The R arrays can be accessed by using indices for different dimensions separated by commas. Different components can be specified by any combination of elements' names or positions.

Accessing Uni-Dimensional Array

The elements can be accessed by using indexes of the corresponding elements.

```
vec <- c(1:10)
cat ("Vector is : ", vec)
cat ("Third element of vector is : ", vec[3])
```

Output:

```
Vector is :  1 2 3 4 5 6 7 8 9 10
Third element of vector is :  3
```

Accessing entire matrices

```
vec1 <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
vec2 <- c(10, 11, 12)
row_names <- c("row1", "row2")
col_names <- c("col1", "col2", "col3")
mat_names <- c("Mat1", "Mat2")
arr = array(c(vec1, vec2), dim = c(2, 3, 2), dimnames = list(row_names, col_names, mat_names))
arr
print ("Matrix 1")
print (arr[,,1])
print ("Matrix 2")
print(arr[,,"Mat2"])
```

Output:

```
, , Mat1
    col1 col2 col3
row1    1    3    5
row2    2    4    6
, , Mat2
    col1 col2 col3
row1    7    9   11
row2    8   10   12
accessing matrix 1 by index value
[1] "Matrix 1"
    col1 col2 col3
row1    1    3    5
row2    2    4    6
accessing matrix 2 by its name
[1] "Matrix 2"
    col1 col2 col3
row1    7    9   11
row2    8   10   12
```

Accessing specific rows and columns of matrices

Rows and columns can also be accessed by both names as well as indices.

```
vec1 <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
vec2 <- c(10, 11, 12)
row_names <- c("row1", "row2")
col_names <- c("col1", "col2", "col3")
mat_names <- c("Mat1", "Mat2")
```

```
arr = array(c(vec1, vec2), dim = c(2, 3, 2), dimnames = list(row_names, col_names, mat_names))
arr
print ("1st column of matrix 1")
print (arr[, 1, 1])
print ("2nd row of matrix 2")
print(arr["row2",,"Mat2"])
```

Output:

```
, , Mat1
    col1 col2 col3
row1    1    3    5
row2    2    4    6
, , Mat2
    col1 col2 col3
row1    7    9   11
row2    8   10   12
accessing matrix 1 by index value
[1] "1st column of matrix 1"
row1 row2
  1    2
accessing matrix 2 by its name
[1] "2nd row of matrix 2"
col1 col2 col3
  8   10   12
```

Adding elements to array

Elements can be appended at the different positions in the array. The sequence of elements is retained in order of their addition to the array. The time complexity required to add new elements is O(n) where n is the length of the array. The length of the array increases by the number of element additions. There are various in-built functions available in R to add new values:

- **c(vector, values):** c() function allows us to append values to the end of the array. Multiple values can also be added together.
- **append(vector, values):** This method allows the values to be appended at any position in the vector. By default, this function adds the element at end.
- **append(vector, values, after=length(vector))** adds new values after specified length of the array specified in the last argument of the function.

- **Using the length function of the array:** Elements can be added at length+x indices where x>0.

Example:

```
x <- c(1, 2, 3, 4, 5)
x <- c(x, 6)
print ("Array after 1st modification ")
print (x)
x <- append(x, 7)
print ("Array after 2nd modification ")
print (x)
len <- length(x)
x[len + 1] <- 8
print ("Array after 3rd modification ")
print (x)
x[len + 3]<-9
print ("Array after 4th modification ")
print (x)
print ("Array after 5th modification")
x <- append(x, c(10, 11, 12), after = length(x)+3)
print (x)
print ("Array after 6th modification")
x <- append(x, c(-1, -1), after = 3)
print (x)
```

Output:

```
[1] "Array after 1st modification "
[1] 1 2 3 4 5 6
[1] "Array after 2nd modification "
[1] 1 2 3 4 5 6 7
[1] "Array after 3rd modification "
[1] 1 2 3 4 5 6 7 8
[1] "Array after 4th modification "
[1]  1  2  3  4  5  6  7  8 NA  9
[1] "Array after 5th modification"
[1]  1  2  3  4  5  6  7  8 NA  9 10 11 12
[1] "Array after 6th modification"
[1]  1  2  3 -1 -1  4  5  6  7  8 NA  9 10 11 12
```

Removing Elements from Array

Elements can be removed from arrays in R, either one at a time or multiple together. These elements are specified as indexes to the array, wherein the array values satisfying the conditions are retained and rest removed. The comparison for removal is based on array values. Multiple conditions can also be combined together to remove a range of elements. Another way to remove elements is by using %in% operator wherein the set of element values belonging to the TRUE values of the operator are displayed as result and the rest are removed.

Example:

```
m <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
print("Original Array")
print(m)
m <- m[m != 3]
print("After 1st modification")
print(m)
m <- m[m > 2 & m <= 8]
print("After 2nd modification")
print(m)
remove <- c(4, 6, 8)
print(m %in% remove)
print("After 3rd modification")
print(m[!m %in% remove])
```

Output:

```
[1] "Original Array"
[1] 1 2 3 4 5 6 7 8 9
[1] "After 1st modification"
[1] 1 2 4 5 6 7 8 9
[1] "After 2nd modification"
[1] 4 5 6 7 8
[1]  TRUE FALSE  TRUE FALSE  TRUE
[1] "After 3rd modification"
[1] 5 7
```

Updating Existing Elements of Array

The elements of the array can be updated with new values by assignment of the desired index of the array with the modified value. The changes are retained in the original array. If the index value to be updated is within the length of the array, then the value is changed, otherwise, the new element is added at the specified index. Multiple

elements can also be updated at once, either with the same element value or multiple values in case the new values are specified as a vector.
Example:

```
m <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
print ("Original Array")
print (m)
m[1] <- 0
print ("After 1st modification")
print (m)
m[7:9] <- -1
print ("After 2nd modification")
print (m)
m[c(2, 5)] <- c(-1, -2)
print ("After 3rd modification")
print (m)
m[10] <- 10
print ("After 4th modification")
print (m)
```

Output:

```
[1] "Original Array"
[1] 1 2 3 4 5 6 7 8 9
[1] "After 1st modification"
[1] 0 2 3 4 5 6 7 8 9
[1] "After 2nd modification"
[1]  0  2  3  4  5  6 -1 -1 -1
[1] "After 3rd modification"
[1]  0 -1  3  4 -2  6 -1 -1 -1
[1] "After 4th modification"
[1]  0 -1  3  4 -2  6 -1 -1 -1 10
```

CLASSES IN R PROGRAMMING

Classes and Objects are basic concepts of Object-Oriented Programming that revolve around real-life entities. Everything in R is an object. An object is simply a data structure that has some methods and attributes. A class is just a blueprint or a sketch of these objects. It represents the set of properties or methods that are common to all objects of one type.

Unlike most other programming languages, R has a three-class system. These are S3, S4, and Reference Classes.

S3 Class

S3 is the simplest yet the most popular OOP system and it lacks formal definition and structure. An object of this type can be created by just adding an attribute to it. Following is an example to make things more clear:

Example:

```
movieList <- list(name = "Iron man", leadActor = "Robert Downey Jr")
class(movieList) <- "movie"
movieList
```

Output:

```
$name
[1] "Iron man"
$leadActor
[1] "Robert Downey Jr"
```

In S3 systems, methods don't belong to the class. They belong to generic functions. It means that we can't create our own methods here, as we do in other programming languages like C++ or Java. But we can define what a generic method (for example print) does when applied to our objects.

```
print(movieList)
```

Output:

```
$name
[1] "Iron man"
$leadActor
[1] "Robert Downey Jr"
```

INPUT IN R

Developers often have a need to interact with users, either to get data or to provide some sort of result. Most programs today use a dialog box as a way of asking the user to provide some type of input. Like other programming languages in R it's also possible to take input from the user. For doing so, there are two methods in R.

- Using readline() method
- Using scan() method

Using readline() method

In R language readline() method takes input in string format. If one inputs an integer then it is inputted as a string, lets say, one wants to input 255, then it will input as "255", like a string. So one needs to convert that inputted value to the format that he

needs. In this case, string “255” is converted to integer 255. To convert the inputted value to the desired data type, there are some functions in R,

as.integer(n); —> convert to integer

as.numeric(n); —> convert to numeric type (float, double etc)

as.complex(n); —> convert to complex number (i.e 3+2i)

as.Date(n) —> convert to date …, etc

```
var = readline();
# convert the inputted value to integer
var = as.integer(var);
# print the value
print(var)
```

Output:

```
255
[1] 255
```

Taking multiple inputs in R

Taking multiple inputs in R language is same as taking single input, just need to define multiple readline() for inputs. One can use braces for define multiple readline() inside it.

Syntax:

```
var1 = readline(“Enter 1st number : “);
var2 = readline(“Enter 2nd number : “);
var3 = readline(“Enter 3rd number : “);
var4 = readline(“Enter 4th number : “);
```

or,

```
{
var1 = readline(“Enter 1st number : “);
var2 = readline(“Enter 2nd number : “);
var3 = readline(“Enter 3rd number : “);
var4 = readline(“Enter 4th number : “);
}
```

Example:

```
# using braces
{
   var1 = readline("Enter 1st number : ");
   var2 = readline("Enter 2nd number : ");
   var3 = readline("Enter 3rd number : ");
   var4 = readline("Enter 4th number : ");
```

```
}
# converting each value
var1 = as.integer(var1);
var2 = as.integer(var2);
var3 = as.integer(var3);
var4 = as.integer(var4);
# print the sum of the 4 number
print(var1 + var2 + var3 + var4)
```

Output:

```
Enter 1st number : 12
Enter 2nd number : 13
Enter 3rd number : 14
Enter 4th number : 15
[1] 54
```

Using scan() method

Another way to take user input in R language is using a method, called scan() method. This method takes input from the console. This method is a very handy method while inputs are needed to taken quickly for any mathematical calculation or for any dataset. This method reads data in the form of a vector or list. This method also uses to reads input from a file also.

Syntax:

```
x = scan()
```

scan() method is taking input continuously, to terminate the input process, need to press Enter key 2 times on the console.

Example:

This is simple method to take input using scan() method, where some integer number is taking as input and print those values in the next line on the console.

```
x = scan()
print(x)
```

Output:

```
1: 1 2 3 4 5 6
7: 7 8 9 4 5 6
13:
Read 12 items
[1] 1 2 3 4 5 6 7 8 9 4 5 6
```

In R there are various methods to print the output. Most common method to print output in R program, there is a function called print() is used. Also if the program of R is

written over the console line by line then the output is printed normally, no need to use any function for print that output. To do this just select the output variable and press run button.

OUTPUT IN R

Print output using print() function

Using print() function to print output is the most common method in R. Implementation of this method is very simple.

Syntax: print("any string") or, print(variable)

```
print("GFG")
x <- "GeeksforGeeks"
print(x)
```

Output:

```
[1] "GFG"
[1] "GeeksforGeeks"
```

Print output using paste() function inside print() function

R provides a method paste() to print output with string and variable together. This method defined inside the print() function. paste() converts its arguments to character strings.

```
x <- "GeeksforGeeks"
print(paste(x, "is best (paste inside print())"))
print(paste0(x, "is best (paste0 inside print())"))
```

Output:

```
[1] "GeeksforGeeks is best (paste inside print())"
[1] "GeeksforGeeksis best (paste0 inside print())"
```

Print output using sprintf() function

sprintf() is basically a C library function. This function is use to print string as C language. This is working as a wrapper function to print values and strings together like C language. This function returns a character vector containing a formatted combination of string and variable to be printed.

```
x = "GeeksforGeeks" # string
x1 = 255            # integer
x2 = 23.14          # float
sprintf("%s is best", x)
sprintf("%d is integer", x1)
sprintf("%f is float", x2)
```

Output:

```
> sprintf("%s is best", x)
```

```
[1] "GeeksforGeeks is best"
> sprintf("%d is integer", x1)
[1] "255 is integer"
> sprintf("%f is float", x2)
[1] "23.140000 is float"
```

Print output using cat() function

Another way to print output in R is using of cat() function. It's same as print() function. cat() converts its arguments to character strings. This is useful for printing output in user defined functions.

Syntax: cat("any string") or, cat("any string", variable)

Example:

```
x = "GeeksforGeeks"
cat(x, "is best\n")
cat("This is R language")
```

Output:

```
GeeksforGeeks is best
This is R language
```

Print output using message() function

Another way to print something in R by using message() function. This is not used for print output but its use for showing simple diagnostic messages which are no warnings or errors in the program. But it can be used for normal uses for printing output.

Syntax: message("any string") or, message("any string", variable)

Example:

```
x = "GeeksforGeeks"
message(x, "is best")
message("This is R language")
```

Output:

```
GeeksforGeeks is best
This is R language
```

STRING MANIPULATION IN R

String manipulation basically refers to the process of handling and analyzing strings. It involves various operations concerned with modification and parsing of strings to use and change its data. R offers a series of in-built functions to manipulate the contents of a string. In this article, we will study different functions concerned with the manipulation of strings in R.

Concatenation of Strings

String Concatenation is the technique of combining two strings. String Concatenation can be done using many ways:

1. paste() function Any number of strings can be concatenated together using the paste() function to form a larger string. This function takes a separator as an argument which is used between the individual string elements and another argument 'collapse' which reflects if we wish to print the strings together as a single larger string. By default, the value of collapse is NULL.

Syntax:

paste(..., sep=" ", collapse = NULL)

Example:

```
str <- paste("Learn", "Code")
print (str)
```

Output:

"Learn Code"

In case no separator is specified the default separator " " is inserted between individual strings.

Example:

```
str <- paste(c(1:3), "4", sep = ":")
print (str)
```

Output:

"1:4" "2:4" "3:4"

2. cat() function Different types of strings can be concatenated together using the cat()) function in R, where sep specifies the separator to give between the strings and file name, in case we wish to write the contents onto a file.

Syntax:

cat(..., sep=" ", file)

Example:

```
str <- cat("learn", "code", "tech", sep = ":")
print (str)
```

Output:

learn:code:techNULL

The output string is printed without any quotes and the default separator is ':'.NULL value is appended at the end.

Example:

```
cat(c(1:5), file ='sample.txt')
```

Output:

1 2 3 4 5

Calculating Length of strings

1. length() function The length() function determines the number of strings specified in the function.

Example:

```
print (length(c("Learn to", "Code")))
```

Output:

2

There are two strings specified in the function.

nchar() function nchar() counts the number of characters in each of the strings specified as arguments to the function individually.

Example:

```
print (nchar(c("Learn", "Code")))
```

Output:

5 4

Case Conversion of strings

1. Conversion to uppercase All the characters of the strings specified are converted to upper case.

Example:

```
print (toupper(c("Learn Code", "hI")))
```

2. Conversion to lowercase All the characters of the strings specified are converted to lowercase.

Example:

```
print (tolower(c("Learn Code", "hI")))
```

Output :

"learn code" "hi"

3. casefold() function All the characters of the strings specified are converted to lowercase or uppercase according to the arguments in casefold(…, upper=TRUE).

Examples:

```
print (casefold(c("Learn Code", "hI"), upper = TRUE)))
```

Output :

"LEARN CODE" "HI"
