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

---

RECURSION

Recursion is when the function calls itself. This forms a loop, where every time the function is called, it calls itself again and again and this technique is known as recursion. Since the loops increase the memory we use the recursion. The recursive function uses the concept of recursion to perform iterative tasks they call themselves, again and again, which acts as a loop. These kinds of functions need a stopping condition so that they can stop looping continuously. Recursive functions call themselves. They break down the problem into smaller components. The function() calls itself within the original function() on each of the smaller components. After this, the results will be put together to solve the original problem.

Example 1: Factorial using Recursion in R

```
rec_fac <- function(x)
{
   if(x==0 || x==1)
   {
      return(1)
   }
   else
   {
      return(x*rec_fac(x-1))
   }
}
rec_fac(5)
```

Output:

```
[1] 120
```

Example 2: Sum of Series Using Recursion

Recursion in R is most useful for finding the sum of self-repeating series. In this example, we will find the sum of squares of a given series of numbers. Sum = $1^2+2^2+…+N^2$

```
sum_series <- function(vec)
```

```
{
   if(length(vec)<=1)
   {
      return(vec^2)
   }
   else
   {
      return(vec[1]^2+sum_series(vec[-1]))
   }
}
series <- c(1:10)
sum_series(series)
```

Output:

```
[1] 385
```

Example 3: Sum of n numbers Using Recursion

```
sum_n <- function(n)
{
  if (n == 1) {
    return(1)
  } else {
    return(n + sum_n(n-1))
  }
}
# Test the sum_n function
sum_n(5)
```

Output:

```
[1] 15
```

In this example, the sum_n function recursively increases n until it reaches 1, which is the base case of the recursion, by adding the current value of n to the sum of the first n-1 values.

Example 4: Finding the exponentiation Using Recursion

```
exp_n <- function(base, n)
{
  if (n == 0) {
    return(1)
  } else {
    return(base * exp_n(base, n-1))
  }
}
# Test the exp_n function
exp_n(4, 5)
```

Key Features of R Recursion

- The use of recursion, often, makes the code shorter and it also looks clean.
- It is a simple solution for a few cases.
- It expresses itself in a function that calls itself.

Applications of Recursion in R

Recursive functions are used in many efficient programming techniques like dynamic programming language(DSL) or divide-and-conquer algorithms. In dynamic programming, for both top-down as well as bottom-up approaches, recursion is vital for performance. In divide-and-conquer algorithms, we divide a problem into smaller subproblems that are easier to solve. The output is then built back up to the top. Recursion has a similar process, which is why it is used to implement such algorithms. In its essence, recursion is the process of breaking down a problem into many smaller problems, these smaller problems are further broken down until the problem left is trivial. The solution is then built back up piece by piece.

Types of Recursion in R

- **Direct Recursion:** The recursion that is direct involves a function calling itself directly. This kind of recursion is the easiest to understand.
- **Indirect Recursion:** An indirect recursion is a series of function calls in which one function calls another, which in turn calls the original function.
- **Mutual Recursion:** Multiple functions that call each other repeatedly make up mutual recursion. To complete a task, each function depends on the others.
- **Nested Recursion:** Nested recursion happens when one recursive function calls another recursively while passing the output of the first call as an argument. The arguments of one recursion are nested inside of this one.
- **Structural Recursion:** Recursion that is based on the structure of the data is known as structural recursion. It entails segmenting a complicated data structure into smaller pieces and processing each piece separately.

REPLACEMENT FUNCTIONS

Replacement functions modify their arguments in place(modifying an R object usually creates a copy). The name of replacement functions are always succeeded by <. They must have arguments named x and value, and return the modified object. In case of a replacement, a function needs additional arguments, the additional arguments should be placed between x and value, and must be called with additional arguments on the left. The name of the function has to be quoted as it is a syntactically valid but non-standard name and the parser would interpret <- as the operator not as part of the function name if it weren't quoted.

Syntax:

```
"function_name<-" <- function(x, additional arguments, value)
{
function body
}
```

Example:

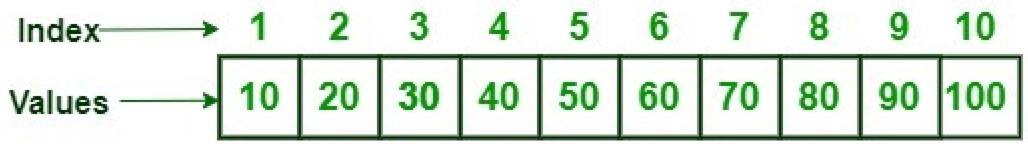
```
# R program to illustrate Replacement function
"replace<-" <- function(x, value)
{
  x[1] = value
  x
}
x = rep.int(5, 7)
replace(x) = 0L
print(x)
```

Output:

```
[1] 0 5 5 5 5 5 5
```

R DATA STRUCTURES: VECTORS

The fundamental data type in R is the vector. R Vectors are the same as the arrays in R language which are used to hold multiple data values of the same type. One major key point is that in R Programming Language the indexing of the vector will start from '1' and not from '0'. We can create numeric vectors and character vectors as well.



Creating a vector

A vector is a basic data structure that represents a one-dimensional array. To create an array we use the "c" function which is the most common method used in R Programming Language.

```
X<- c(61, 4, 21, 67, 89, 2)
cat('using c function', X, '\n')
```

```
# seq() function for creating
# a sequence of continuous values.
# length.out defines the length of the vector.
Y<- seq(1, 10, length.out = 5)
cat('using seq() function', Y, '\n')
# use':' to create a vector
# of continuous values.
Z<- 2:7
cat('using colon', Z)
```

Output:

```
using c function 61 4 21 67 89 2
using seq() function 1 3.25 5.5 7.75 10
using colon 2 3 4 5 6 7
```

Adding and Deleting Vector Elements

Vectors are stored like arrays in C, contiguously, and thus you cannot insert or delete elements The size of a vector is determined at its creation, so if you wish to add or delete elements, we need to reassign the vector.

```
> x <- c(88,5,12,13)
> x <- c(x[1:3],168,x[4])
# insert 168 before the 13
> x
[1] 88 5 12 168 13
```

Obtaining the Length of a Vector

We obtain the length of a vector by using the length() function:

```
> x <- c(1,2,4)
> length(x)
[1] 3
first1 <- function(x) {
  for (i in 1:length(x)) {
    if (x[i] == 1) break # break out of loop
  }
  return(i)
}
x <- c(88,1,12,13)
print(first1(x))
```

Output:

```
[1] 2
```

Matrices and Arrays as Vectors

Arrays and matrices are actually vectors. They merely have extra class attributes. For example, matrices have the number of rows and columns.

```
> m
[,1] [,2]
[1,] 1 2
[2,] 3 4
> m + 10:13
[,1] [,2]
[1,] 11 14
[2,] 14 17
```

The 2-by-2 matrix m is stored as a four-element vector, column-wise, as (1,3,2,4). We then added (10,11,12,13) to it, yielding (11,14,14,17), but R remembered that we were working with matrices and thus gave the 2-by-2 result

Declarations

As with most scripting languages (such as Python and Perl), we do not declare variables in R. For instance, consider this code:

```
z <- 3
```

This code, with no previous reference to z, is perfectly legal (and common-place). However, if you reference specific elements of a vector, you must warn R. For instance, say we wish y to be a two-component vector with values 5 and 12.

```
> y <- vector(length=2)
> y[1] <- 5
> y[2] <- 12
```

The following will also work:

```
> y <- c(5,12)
```

This approach is all right because on the right-hand side we are creating a new vector, to which we then bind y. The reason we cannot suddenly spring an expression like y[2] on R stems from R's functional language nature. The reading and writing of individual vector elements are actually handled by functions. If R doesn't already know that y is a vector, these functions have nothing on which to act. Speaking of binding, just as variables are not declared, they are not constrained in terms of mode. The following sequence of events is perfectly valid:

```
> x <- c(1,5)
> x
[1] 1 5
> x <- "abc"
```

First, x is associated with a numeric vector, then with a string.

Recycling

When applying an operation to two vectors that requires them to be the same length, R automatically recycles, or repeats, the shorter one, until it is long enough to match the longer one. Here is an example:

```
> c(1,2,4) + c(6,0,9,20,22)
[1] 7 2 13 21 24
```

longer object length is not a multiple of shorter object length in: c(1, 2, 4) + c(6,0, 9, 20, 22). The shorter vector was recycled, so the operation was taken to be as follows:

```
> c(1,2,4,1,2) + c(6,0,9,20,22)
[1] 7 2 13 21 24
> x
[,1] [,2]
[1,] 1 4
[2,] 2 5
[3,] 3 6
> x+c(1,2)
[,1] [,2]
[1,] 2 6
[2,] 4 6
[3,] 4 8
```

Again, keep in mind that matrices are actually long vectors. Here, x, as a 3-by-2 matrix, is also a six-element vector, which in R is stored column by column. In other words, in terms of storage, x is the same as c(1,2,3,4,5,6). We added a two-element vector to this six-element one, so our added vector needed to be repeated twice to make six elements. In other words, we were essentially doing this: x + c(1,2,1,2,1,2)

Common Vector Operations

Vector Arithmetic and Logical Operations

R is a functional language. Every operator, including + in the following example, is actually a function.

```
> x <- c(1,2,4)
> x + c(5,0,-1)
[1] 6 2 3
```

When we multiply two vectors.

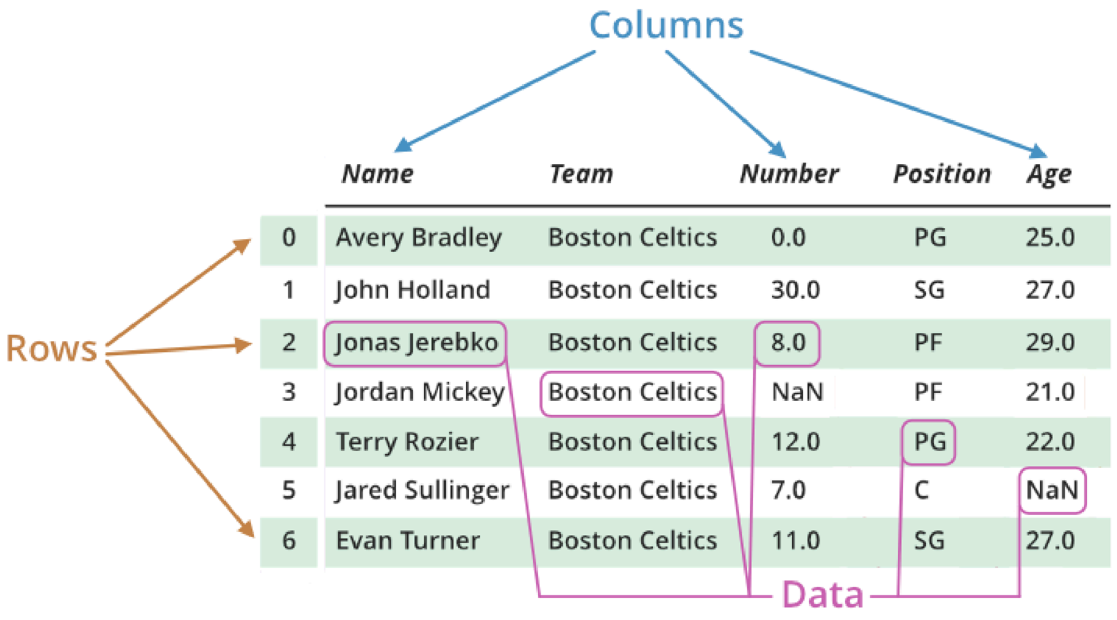
```
> x * c(5,0,-1)
[1] 5 0 -4
```

DATA FRAMES

Data Frames in R Language are generic data objects of R that are used to store tabular data. Data frames can also be interpreted as matrices where each column of a matrix can be of different data types. R DataFrame is made up of three principal components, the data, rows, and columns.

R Data Frames Structure

As you can see in the image below, this is how a data frame is structured. The data is presented in tabular form, which makes it easier to operate and understand.



	Name	*Team*	*Number*	*Position*	*Age*
0	Avery Bradley	Boston Celtics	0.0	PG	25.0
1	John Holland	Boston Celtics	30.0	SG	27.0
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN
6	Evan Turner	Boston Celtics	11.0	SG	27.0

Create Dataframe

To create an R data frame use **data.frame()** function and then pass each of the vectors we have created as arguments to the function.

```
friend.data <- data.frame(
   friend_id = c(1:5),
   friend_name = c("Sachin", "Sourav",
               "Dravid", "Sehwag",
               "Dhoni"),
   stringsAsFactors = FALSE
)
# print the data frame
print(friend.data)
```

Output:

```
friend_id friend_name
1        1      Sachin
```

```
2       2     Sourav
3       3     Dravid
4       4     Sehwag
5       5      Dhoni
```

If the named argument stringsAsFactors is not specified, then by default, stringsAsFactors will be TRUE. (You can also use options() to arrange the opposite default.) This means that if we create a data frame from a character vector—in this case, friend_name. R will convert that vector to a factor. Because our work with character data will typically be with vectors rather than factors, we'll set stringsAsFactors to FALSE.

Accessing Data Frames

Since d is a list, we can access it as such via component index values or component names:

```
> kids <- c("Jack","Jill")
> ages <- c(12,10)
> d <- data.frame(kids,ages,stringsAsFactors=FALSE)
> d # matrix-like viewpoint
kids ages
1 Jack 12
2 Jill 10
> d[[1]]
[1] "Jack" "Jill"
> d$kids
[1] "Jack" "Jill"
```

But we can treat it in a matrix-like fashion as well. For example, we can view column 1:

```
> d[,1]
[1] "Jack" "Jill"
```

This matrix-like quality is also seen when we take d apart using str():

```
> str(d)
'data.frame': 2 obs. of 2 variables:
$ kids: chr "Jack" "Jill"
$ ages: num 12 10
```

Add Rows in R Data Frame

To add rows in a Data Frame, you can use a built-in function rbind().

```
Products <- data.frame(
  Product_ID = c(101, 102, 103),
  Product_Name = c("T-Shirt", "Jeans", "Shoes"),
  Price = c(15.99, 29.99, 49.99),
  Stock = c(50, 30, 25)
)

# Print the existing dataframe
cat("Existing dataframe (Products):\n")
print(Products)
# Adding a new row for a new product
New_Product <- c(104, "Sunglasses", 39.99, 40)
Products <- rbind(Products, New_Product)
# Print the updated dataframe after adding the new product
cat("\nUpdated data frame after adding a new product:\n")
print(Products)
```

Output:

```
Existing dataframe (Products):
Product_ID Product_Name Price Stock
1        101      T-Shirt 15.99    50
2        102        Jeans 29.99    30
3        103        Shoes 49.99    25
Updated dataframe after adding a new product:
Product_ID Product_Name Price Stock
1        101      T-Shirt 15.99    50
2        102        Jeans 29.99    30
3        103        Shoes 49.99    25
4        104   Sunglasses 39.99    40
```

Add Columns in R Data Frame

To add columns in a Data Frame, you can use a built-in function cbind().

```
Products <- data.frame(
  Product_ID = c(101, 102, 103),
  Product_Name = c("T-Shirt", "Jeans", "Shoes"),
  Price = c(15.99, 29.99, 49.99),
  Stock = c(50, 30, 25)
)
# Print the existing dataframe
cat("Existing dataframe (Products):\n")
```

```
print(Products)
# Adding a new column for 'Discount' to the dataframe
Discount <- c(5, 10, 8)  # New column values for discount
Products <- cbind(Products, Discount)
# Rename the added column
colnames(Products)[ncol(Products)] <- "Offer"  # Renaming the last column
# Print the updated dataframe after adding the new column
cat("\nUpdated dataframe after adding a new column 'Discount':\n")
print(Products)
```

Output:

```
Existing dataframe (Products):

 Product_ID Product_Name Price Stock
1        101      T-Shirt 15.99    50
2        102        Jeans 29.99    30
3        103        Shoes 49.99    25
Updated dataframe after adding a new column 'Discount':
Product_ID Product_Name Price Stock Discount
1        101      T-Shirt 15.99    50        5
2        102        Jeans 29.99    30       10
3        103        Shoes 49.99    25        8
Renaming the last column
Product_ID Product_Name Price Stock Offer
1        101      T-Shirt 15.99    50        5
2        102        Jeans 29.99    30       10
3        103        Shoes 49.99    25        8
```

MATRICES

A matrix is a vector with two additional attributes: the number of rows and the number of columns. Since matrices are vectors, they also have modes, such as numeric and character.

Creating Matrices

To create a matrix in R you need to use the function called matrix(). The arguments to this matrix() are the set of elements in the vector. You have to pass how many numbers of rows and how many numbers of columns you want to have in your matrix.

Note: By default, matrices are in column-wise order.

Syntax to Create R-Matrix

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

Parameters:

data – values you want to enter
nrow – no. of rows
ncol – no. of columns
byrow – logical clue, if 'true' value will be assigned by rows
dimnames – names of rows and columns

Example:

```
A = matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, ncol = 3, byrow = TRUE)
rownames(A) = c("a", "b", "c")
colnames(A) = c("c", "d", "e")
cat("The 3x3 matrix:\n")
print(A)
```

Output

```
The 3x3 matrix:
  c d e
a 1 2 3
b 4 5 6
c 7 8 9
```

Creating Special Matrices in R

R allows the creation of various different types of matrices with the use of arguments passed to the matrix() function.

1. Matrix where all rows and columns are filled by a single constant 'k':

To create such a R matrix the syntax is given below:

Syntax: matrix(k, m, n)

Parameters:

k: the constant
m: no of rows
n: no of columns

Example:

```
print(matrix(5, 3, 3))
```

Output

```
     [,1] [,2] [,3]
[1,]    5    5    5
```

```
[2,]    5    5    5
[3,]    5    5    5
```
2. Diagonal matrix:

A diagonal matrix is a matrix in which the entries outside the main diagonal are all zero. To create such a R matrix the syntax is given below:

Syntax: diag(k, m, n)

Parameters:

k: the constants/array

m: no of rows

n: no of columns

Example:

```
print(diag(c(5, 3, 3), 3, 3))
```
Output

```
     [,1] [,2] [,3]
[1,]    5    0    0
[2,]    0    3    0
[3,]    0    0    3
```
3. Identity matrix:

An identity matrix in which all the elements of the principal diagonal are ones and all other elements are zeros. To create such a R matrix the syntax is given below:

Syntax: diag(k, m, n)

Parameters:

k: 1 m: no of rows n: no of columns

Example:

```
print(diag(1, 3, 3))
```
Output

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Matrix Metrics

Matrix metrics tell you about the Matrix you created. You might want to know the number of rows, number of columns, dimensions of a Matrix. It describes about How you can know the dimension of the matrix? How can you know how many rows are there in the matrix? How many columns are in the matrix? How many elements are there in the matrix?

Example:

```
A = matrix( c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, ncol = 3, byrow = TRUE)
cat("The 3x3 matrix:\n")
print(A)
cat("Dimension of the matrix:\n")
print(dim(A))
cat("Number of rows:\n")
print(nrow(A))
cat("Number of columns:\n")
print(ncol(A))
cat("Number of elements:\n")
print(length(A))
print(prod(dim(A)))
```

Output

```
The 3x3 matrix:
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
Dimension of the matrix:
[1] 3 3
Number of rows:
[1] 3
Number of columns:
[1] 3
Number of elements:
[1] 9
```

Accessing Elements of a R-Matrix

We can access elements in the R matrices using the same convention that is followed in data frames. So, you will have a matrix and followed by a square bracket with a comma in between the arrays. Value before the comma is used to access rows and value that is after the comma is used to access columns. Let's illustrate this by taking a simple R code.

Accessing rows:

```
A = matrix(6 c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, ncol = 3, byrow = TRUE)
cat("The 3x3 matrix:\n")
print(A)
```

```
cat("Accessing first and second row\n")
print(A[1:2, ])
```
Output
```
The 3x3 matrix:
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
Accessing first and second row
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```
Accessing columns:
```
A = matrix( c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, ncol = 3, byrow = TRUE)
cat("The 3x3 matrix:\n")
print(A)
cat("Accessing first and second column\n")
print(A[, 1:2])
```
Output
```
The 3x3 matrix:
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
Accessing first and second column
     [,1] [,2]
[1,]    1    2
[2,]    4    5
[3,]    7    8
```
Adding Rows and Columns in a R-Matrix

To add a row in R-matrix you can use rbind() function and to add a column to R-matrix you can use cbind() function.

Adding Row

Let's see below example on how to add row in R-matrix?

Example:
```
number <- matrix( c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, ncol = 3,  byrow = TRUE)
cat("Before inserting a new row:\n")
```

```
print(number)
new_row <- c(10, 11, 12)
A <- rbind(number[1, ], new_row, number[-1, ])
cat("\nAfter inserting a new row:\n")
print(number)
```

Output

```
Before inserting a new row:
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
After inserting a new row:
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]...
```

Adding Column

Let’s see below example on how to add column in R-matrix?

```
number <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, ncol = 3, byrow = TRUE)
cat("Before adding a new column:\n")
print(number)
new_column <- c(10, 11, 12)  # Define the new column
number <- cbind(number, new_column)
cat("\nAfter adding a new column:\n")
print(number)
```

Output

```
Before adding a new column:
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
After adding a new column:
             new_column
[1,] 1 2 3          10
[2,] 4 5 6          1...
```

Deleting Rows and Columns of a R-Matrix

To delete a row or a column, first of all, you need to access that row or column and then insert a negative sign before that row or column. It indicates that you had to delete that row or column.

Row deletion:

```
A = matrix( c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, ncol = 3, byrow = TRUE)
cat("Before deleting the 2nd row\n")
print(A)
A = A[-2, ]
cat("After deleted the 2nd row\n")
print(A)
```

Output

```
Before deleting the 2nd row
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
After deleted the 2nd row
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    7    8    9
```

Column deletion:

```
A = matrix( c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, ncol = 3, byrow = TRUE)
cat("Before deleting the 2nd column\n")
print(A)
A = A[, -2]
cat("After deleted the 2nd column\n")
print(A)
```

Output

```
Before deleting the 2nd column
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
After deleted the 2nd column
     [,1] [,2]
[1,]    1    3
[2,]    4    6
```

```
[3,]    7    9
```