Unit-3: Data Structures in R

Data structures in R refer to the ways in which data is organized and stored in the memory of a computer.

They provide a means to efficiently manage, access, and manipulate data.

Each data structure has its own characteristics and is suited for specific types of tasks.

Usage of Data Structures with R:

- **Efficient Data Representation:**
  - Data structures in R help represent different types of data efficiently. For example, vectors and matrices are suitable for numerical data, while data frames are used for tabular data with mixed types.
- **Data Cleaning and Transformation:**
  - Manipulating and transforming data is a common task in data science. Data structures like lists and data frames allow for easy cleaning, reshaping, and merging of datasets.
- **Statistical Analysis:**
  - R is widely used for statistical analysis. Factors in R are particularly useful for representing categorical variables, making it easier to perform statistical modeling and analysis.
- **Data Exploration:**
  - Lists and data frames enable the exploration of complex data structures. Nested lists, for example, can represent hierarchical data, and data frames provide a convenient way to view and analyze tabular data.
- **Data Visualization:**
  - R has powerful visualization libraries. Properly structured data using data frames or matrices facilitates the creation of meaningful plots and graphs for effective data visualization.

Some Data Structures in R are:

- Vectors
- Matrices
- Arrays
- Factors
- Data Frames
- Lists

Vectors

- A Vector is a one-dimensional array that can hold elements of the same data type.

```r
#vector of numerical values
numbers <- c(1, 2, 3)

#vector of strings
fruits <- c("banana", "apple", "orange")

#vector of logical values
answers <- c(TRUE, FALSE, TRUE)

#printing vectors
print(numbers)
print(fruits)
print(answers)
```

```
[1] 1 2 3
[1] "banana" "apple"  "orange"
[1]  TRUE FALSE  TRUE
```

Creating and naming vectors:

- A vector is created using the c( ) function. (“c” stands for “combine”)
- c( ) function is used to create a vector by combining the provided comma (,) list of elements together.

Syntax: vector <- c(vector_values)

- Each value of a vector can be named in two ways:
  - Naming while defining
  - Naming after defining by using names( ) function

- In the first way, names are provided inside c( ) for each value, by using "=" operator.
- In the second way, a vector with same length is created with names for the respective values and assigned to names( ) function with the vector of values as argument.

Syntax: names(vector) <- c(names_for_vector_value)

Ex:

```
# naming while defining
fruits_prices <- c(apple = 10, banana = 20, orange = 30)
print(fruits_prices)

# naming after defining, using names()
fruits_prices <- c(10, 20, 30)
names(fruits_prices) <- c("apple", "banana", "orange")
print(fruits_prices)
```

```
apple banana orange
   10     20     30
apple banana orange
   10     20     30
```

Vector Arithmetic:

All basic Arithmetic operation can be performed of vectors, such as:

- Adding vectors
- Subtracting vectors
- Multiplying vectors
- Dividing vectors
- Scalar operations

- In add, sub, mul and div operations, the operation is performed b/w consecutive numeric values of two/more vectors. Hence the length of vectors must be the same.
- In scalar operations, a single number (instead of a vector) is used to perform the specified arithmetic operation on all the numeric values of a vector.

Ex:

```r
vec1 <- c(1, 2, 3)
vec2 <- c(4, 5, 6)

# (1) Addition
# Adding two vector
add <- vec1 + vec2
print(add)
# Scalar Addition
scalar_add <- vec1 + 5
print(scalar_add)

# (2) Subtraction
# Subtracting two vector
sub <- vec1 - vec2
print(sub)
# Scalar Subtraction
scalar_sub <- vec1 - 5
print(scalar_sub)

# (3) Multiplication
# Mulitplying two vector
mul <- vec1 * vec2
print(mul)
# Scalar Multiplication
scalar_mul <- vec1 * 5
print(scalar_mul)

# (4) Division
# Dividing two vector
div <- vec1 / vec2
print(div)
# Scalar Division
scalar_div <- vec1 / 5
print(scalar_div)
```

```
[1] 5 7 9
[1] 6 7 8
[1] -3 -3 -3
[1] -4 -3 -2
[1]  4 10 18
[1]  5 10 15
[1] 0.25 0.40 0.50
[1] 0.2 0.4 0.6
```

Vector Sub-setting:

Vector Sub-setting is a process of creating a sub-vector (vector with length less than the actual vector length) by selecting specific elements from a vector, based on certain conditions.

Syntax: sub_vector <- vector[condition]

Sub-setting is always performed based on some certain conditions, such as:

- Sub-setting by Index:

The indices of required values of a vector are combined and specified as a condition.

```
vector <- c("apple", "banana", "cherry", "orange")
sub_vector <- vector[c(1, 3)]
print(sub_vector)
```

```
[1] "apple"  "cherry"
```

- Sub-setting by Logical Condition
  A logical expression is specified as a condition. The values for a sub_vector are only considered when the expression returns TRUE.

```
vector <- c(10, 20, 30, 40, 50)
sub_vector <- vector[vector > 25]
print(sub_vector)
```

```
[1] 30 40 50
```

- Sub-setting by Named Elements
  A named vector can be sub-setted by specifying the combination required names of values as a condition.

```
vector <- c(apple=10, banana=20, orange=30, grape=40)
sub_vector <- vector[c("apple", "grape")]
print(sub_vector)
```

```
apple grape
   10    40
```

- Sub-setting by a Logical Vector
  A set(logical_vector) with same length of vector and with logical values is used as condition. Only the corresponding values of TRUE are considered.

```
vector <- c("apples", "oranges", "bananas", "pears", "grapes")
sub_vector <- vector[c(FALSE, TRUE, TRUE, FALSE, TRUE)]
print(sub_vector)
```

```
[1] "oranges" "bananas" "grapes"
```

- Sub-setting with Negative Index
  The indices of values which are NOT required, are combined and specified as a condition, with a minus(-) symbol.

```r
vector <- c("apple", "banana", "cherry", "orange")
sub_vector <- vector[-c(1, 3)]
print(sub_vector)
```

```
[1] "banana" "orange"
```

Matrices:

A Matrix is a two-dimensional array that can hold values of the same data type, in rows and columns.

Creating and naming Matrices:

- A matrix is created by using matrix( ) function, which takes a vector as first argument, no.of rows (nrow) as second argument and no.of columns (ncol) as third argument.

Syntax: my_matrix <- matrix(c(values), nrow=r , ncol = c)

- The nrow and ncol are predefined keyword to specify no,of rows and columns respectively.
- The length of vector(first argument) must be equal to the product of nrow and ncol

i.e., length(c(values)) == nrow*ncol

Ex:

```r
my_matrix <- matrix(c(1,2,3,4,5,6), nrow = 2, ncol = 3)
print(my_matrix)
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

- Each row and column of a matric can be named in two ways:
  - While defining (dimnames parameter)
  - After defining, using rownames( ) and colnames( )
- In first way, “dimnames”(stands for “dimension-names”) parameter is specified as the fourth argument to the matrix( ) function, which is a list of row_names_vector and column_names_vector.

- In second way, the row_names and column_names vectors are assigned rownames( ) and colnames( ) functions respectively, which takes the matrix as argument.

Ex:

```r
# While defining
my_matrix_1 <- matrix(c(1,2,3,4,5,6), nrow = 2, ncol = 3, dimnames = list(c("row1", "row2"), c("col1", "col2", "col3")))
print(my_matrix_1)

# After defining, using rownames() and colnames()
my_matrix_2 <- matrix(c(7,8,9,10,11,12), nrow = 2, ncol = 3)
rownames(my_matrix_2) <- c("row1", "row2")
colnames(my_matrix_2) <- c("col1", "col2", "col3")
print(my_matrix_2)
```

```
     col1 col2 col3
row1    1    3    5
row2    2    4    6
     col1 col2 col3
row1    7    9   11
row2    8   10   12
```

Matrix Sub-setting:

It is a process of creating a sub matrix by specifying a certain condition such as:

- Sub-setting by Row and Column Indices
- Sub-setting Entire Rows or Columns
- Sub-setting with Logical Condition

```r
my_matrix <- matrix(c(1,2,3,4,5,6), nrow = 2, ncol = 3)

# Sub-setting by Row and Column Indices:
sub_matrix <- my_matrix[c(1,2), c(2,3)]
# condition => (rows_index_vector, column_index_vector)
print(sub_matrix)

# Sub-setting Entire Rows or Columns:
sub_row <- my_matrix[1,]  #first row
sub_col <- my_matrix[,2]  #second column
print(sub_row)
print(sub_col)

# Sub-setting with Logical Condition:
sub_matrix_cond <- my_matrix[my_matrix > 2]
print(sub_matrix_cond)
```

```
     [,1] [,2]
[1,]    3    5
[2,]    4    6
[1] 1 3 5
[1] 3 4
[1] 3 4 5 6
```

Arrays:

An Array is simply a list of elements with the same type of elements.
It can have values stored in one/two/more dimensions.
A one-dimensional array is simply a vector (and) a two-dimensional array is simply a Matrix.
An array is created using array( ) function, which takes a vector containing all values of array as the first argument and a vector specifying the dimensions (dim) as the second argument.
Ex:

```r
my_array <- array(c(1:24), dim = c(2, 3, 4))
# (two-rows, three-columns, four-layers/faces)
print(my_array)
```

```
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18

, , 4

     [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24
```

Class:

Factors:

- A factor is a data structure which categorizes the provided list of data based on shared characteristics.
- Categorization is basically a process of grouping the similar or related data and then labeling the group with a name to identify the group uniquely from other groups.
- It is used in data analysis techniques to classify/categorize the similar data points in a given dataset.
- To create a factor, we use factor( ) function, which takes a vector of items as argument and returns the same list with distinct names, known as labels.

- A label is a distinct name given to a similar set of data.
- In R, the labels are referred as levels.

Ex:

```
temeratures <- factor(c("low", "low", "high", "high", "low", "high"))
print(temeratures)

gender <- factor(c("male", "female", "female", "male", "male", "female"))
print(gender)
```

```
[1] low  low  high high low  high
Levels: high low
[1] male   female female male   male   female
Levels: female male
```

The list of levels of a factor can be stored into a variable, using levels( ) function, which takes the factor as an argument.

```
temeratures <- factor(c("low", "low", "high", "high", "low", "high"))
temp_levels <- levels(temeratures)
print(temp_levels)

gender <- factor(c("male", "female", "female", "male", "male", "female"))
gender_levels <- levels(gender)
print(gender_levels)
```

```
[1] "high" "low"
[1] "female" "male"
```

- To get the counts of each level of a factor, we can use summary( ) function, which takes the factor as an argument and returns a names vector, in which the names represent a level, and its corresponding values represents the counts of the specific level in the given dataset.

```r
temeratures <- factor(c("low", "low", "high", "high", "low", "high"))
temp_counts <- summary(temeratures)
print(temp_counts)

gender <- factor(c("male", "female", "female", "male", "male", "female"))
gender_counts <- summary(gender)
print(gender_counts)
```

```
high  low
   3    3
female   male
     3      3
```

- The Levels can be Ordered (each level has a fixed position) or Unordered (no fixed position).
- It can be specified by “ordered” parameter which is taken as the second argument in the factor( ) function.
- By default, “ordered” is set to “FALSE”.
- We can specify it as TRUE explicitly and provide the “levels” vector as the required order.
- Factor with “ordered” parameter as “TRUE” is known as Ordered Factor.

Ex:

```r
temeratures_1 <- factor(c("low", "medium", "high", "high", "low", "medium", "high"))
# ordered = FALSE
unordered_temp_levels <- levels(temeratures_1)
print(unordered_temp_levels)

temperatures_2 <- factor(c("low", "medium", "high", "high", "low", "medium", "high"), order =
TRUE, levels = c("low", "medium", "high"))
ordered_temp_levels <- levels(temperatures_2)
print(ordered_temp_levels)
```

```
[1] "high"   "low"    "medium"
[1] "low"    "medium" "high"
```

- Comparision operator can be applied on the levels of an Ordered Factor.
- When we specify levels for an ordered factor, the levels are prioritized in ascending order.

I.e., first level gets lowest priority/value and last level gets highest priority/value.

Ex:

```
temperatures_2 <- factor(c("low", "medium", "high", "high", "low", "medium", "high"), order =
TRUE, levels = c("low", "medium", "high"))
ordered_temp_levels <- levels(temperatures_2)
comparision_1 <- ordered_temp_levels[1] < ordered_temp_levels[2]
comparision_2 <- ordered_temp_levels[2] > ordered_temp_levels[3]
print(comparision_1)  TRUE
print(comparision_2)   FALSE
```

Data Frames:

- Just like an SQL table, a Data Frame is also a table where the data is stored in rows and columns.
- Each Column has a unique name, and each row represents a record.
- In R, a data frame can be created by using data.frame( ) function, which takes the vectors (column values) with the column name, as arguments and, returns a data-frame(table).

The vector values represent the column values and name of the vector argument represents the column name.

I.e, each argument is a column.

Ex:

```
students <- data.frame(
  Name = c("Alex", "Ajay", "Amanda"),
  Age = c(15, 14, 20),
  Gender = c("Male", "Male", "Female")
)
print(students)
```

```
    Name Age Gender
1   Alex  15   Male
2   Ajay  14   Male
3 Amanda  20 Female
```

Sub-setting Data-Frames:

- A subset of a data-frame can be created in many ways, one of the ways is by using subset( ) function.
- subset( ) function takes the data-frame as first argument, row/column specific conditions as second argument and, selection of specific columns as third argument.
- The condition is to select based on some specific condition of a column value, from which only required columns can be fetched by specifying the vector of column names as third argument with “select” parameter name.
- Another way to select a column of a data-frame is by using “$” symbol.

(syntax: table$columnName)

Ex:

```
students <- data.frame(
  Name = c("Alex", "Ajay", "Amanda"),
  Age = c(15, 14, 20),
  Gender = c("Male", "Male", "Female")
)
# subsetting by column
col_subset <- subset(students, select = c("Name", "Age"))
print(col_subset)

# subsetting by row
row_subset <- subset(students, Age > 14)
print(row_subset)

# subsetting by row and column
row_col_subset <- subset(students, Age > 14, select = c("Name", "Age"))
print(row_col_subset)
```

```
    Name Age
1   Alex  15
2   Ajay  14
3 Amanda  20
    Name Age Gender
1   Alex  15   Male
3 Amanda  20 Female
    Name Age
1   Alex  15
3 Amanda  20
```

Extending Data-Frames:

Extending a data-frame means, adding more rows/columns.

A New Column can be added by using rbind( ) function, which takes data-frame as first argument and new row vector as second.

A New Column can be added by using cbind( ) function, which takes data-frame as first argument and new column vector with column name as second.

Ex:

```r
students <- data.frame(
  Name = c("Alex", "Ajay", "Amanda"),
  Age = c(15, 14, 20),
  Gender = c("Male", "Male", "Female")
)
print(students)

cat("\n")

# adding a Row
students <- rbind(students, c("James", 21, "Male"))
print(students)

cat("\n")

# addiing a column
students <- cbind(students, Height = c(180, 170, 160, 210))
print(students)
```

```
    Name Age Gender
1   Alex  15   Male
2   Ajay  14   Male
3 Amanda  20 Female

    Name Age Gender
1   Alex  15   Male
2   Ajay  14   Male
3 Amanda  20 Female
4  James  21   Male

    Name Age Gender Height
1   Alex  15   Male    180
2   Ajay  14   Male    170
3 Amanda  20 Female    160
4  James  21   Male    210
```

Sorting Data-Frames:

Sorting is always performed based on some condition of a column value, or a specific sequence (ascending/descending).

The order( ) function lets us to perform sorting on a data frame.

Ex:

```r
students <- data.frame(
  Name = c("Alex", "Ajay", "Amanda"),
  Age = c(15, 14, 20),
  Gender = c("Male", "Male", "Female")
)
print(students)

cat("\n")

# sorting students table by ascenidng order of age
students_asc <- students[order(students$Age), ]
print(students_asc)

cat("\n")

# sorting students table by descending order of age
students_dcs <- students[order(-students$Age), ]
print(students_dcs)
```

```
    Name Age Gender
1   Alex  15   Male
2   Ajay  14   Male
3 Amanda  20 Female

    Name Age Gender
2   Ajay  14   Male
1   Alex  15   Male
3 Amanda  20 Female

    Name Age Gender
3 Amanda  20 Female
1   Alex  15   Male
2   Ajay  14   Male
```

Lists:

- A List is a versatile data structure that can hold elements of different data types including vectors, matrices, data-frames, and even other lists.
- List provides a flexible way to organize and store heterogenous data.

Creating a list:

- A List is created by using the list( ) function, which takes any number and any type of elements as arguments.

Ex:

```
students <- data.frame(
  Name = c("Alex", "Ajay", "Amanda"),
  Age = c(15, 14, 20),
  Gender = c("Male", "Male", "Female")
)
my_matrix <- matrix(c(1,2,3,4,5,6), nrow=2, ncol=3)

# Creating a list
my_list <- list("John", 25, c(1, 2, 3), TRUE, list(1, "Hi", FALSE), students, my_matrix)
print(my_list)
```

```
[1] "John"

[[2]]
[1] 25

[[3]]
[1] 1 2 3

[[4]]
[1] TRUE

[[5]]
[[5]][[1]]
[1] 1

[[5]][[2]]
[1] "Hi"

[[5]][[3]]
[1] FALSE


[[6]]
    Name Age Gender
1   Alex  15   Male
2   Ajay  14   Male
3 Amanda  20 Female

[[7]]
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Creating a Names List:

- Every Element of a list can be named uniquely.

Ex:

```
named_list <- list(Name = "Alice", Age = 30, Scores = c(90, 85, 92), Married = FALSE)
print(named_list)
```

```
$Name
[1] "Alice"

$Age
[1] 30

$Scores
[1] 90 85 92

$Married
[1] FALSE
```

Accessing List Elements:

- List elements can be accessed using indices or names.

Ex:

```
named_list <- list(Name = "Alice", Age = 30, Scores = c(90, 85, 92), Married = FALSE)

# accessing by index
print(named_list[[1]])

# accessing by name
print(named_list$Age)
```

```
[1] "Alice"
[1] 30
```

Manipulating List Elements:

- Lists are mutable, hence we can:

- Modify an element.
- Remove an element.
- add new element.

Ex:

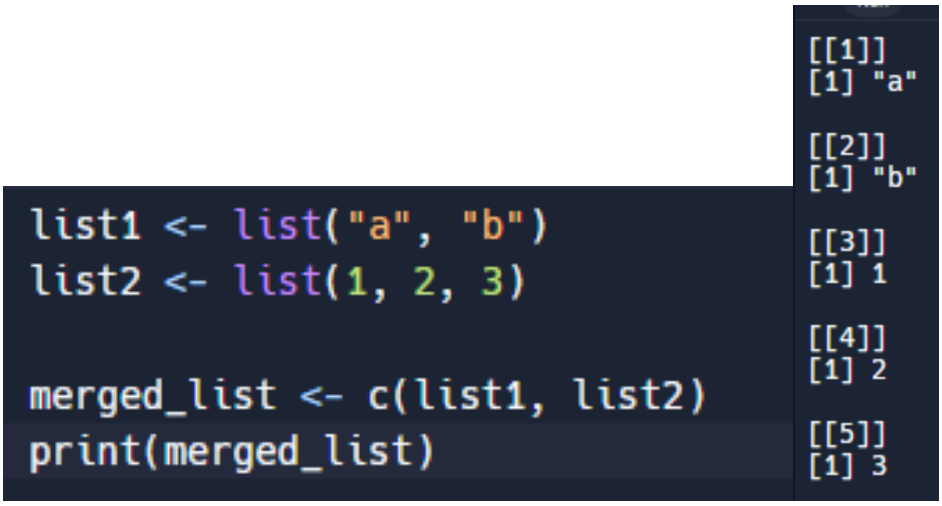
```r
named_list <- list(Name = "Alice", Age = 30, Scores = c(90, 85, 92), Married = FALSE)
print(named_list)
cat("\n")

# modiying an element
named_list$Age <- 31

# adding an element
named_list$City <- "New York"

# removing an element
named_list$Scores <- NULL

print(named_list)
```

```
$Name
[1] "Alice"

$Age
[1] 30

$Scores
[1] 90 85 92

$Married
[1] FALSE


$Name
[1] "Alice"

$Age
[1] 31

$Married
[1] FALSE

$City
[1] "New York"
```

Merging Lists:

- Two/more lists can be merged into a single list, using c( ) function.

Ex:



Converting Lists to Vectors:

- A List with homogenous/heterogeneous elements can be converted into a vector, by using unlist( ) function.
- numeric types will be converted to string types and multi-dimensional data structures (such as, data-frame, matrix, etc) will be converted to single rowed vector.

Ex:

```r
frame <- data.frame(Name = c("Alice", "Bob"), Age=c(25, 30))
print(frame)
my_matrix <- matrix(c(1,2,3,4), nrow = 2, ncol = 2)
print(my_matrix)
cat("\n")

my_list <- list("a", 2, c(1,2), frame, my_matrix)

# converting to vector
vector <- unlist(my_list)

print(vector)
```

```
   Name Age
1 Alice  25
2   Bob  30
     [,1] [,2]
[1,]    1    3
[2,]    2    4

                              Name1   Name2    Age1    Age2

   "a"     "2"     "1"     "2" "Alice"   "Bob"    "25"    "30"     "1"     "2"


   "3"     "4"
```