

R Programming

UNIT – III

Lists

Creating Lists:

A **list** is a collection of elements of *different types*. Lists are particularly useful when you need to store heterogeneous data.

In other words, A **list** is a flexible data structure that can store **elements of different types**, including numbers, characters, vectors, matrices, other lists, and even functions.

In R, To create a list, we use **list()** function and in this, the elements are separated by a comma(,).

Syntax:

```
list_name<- list(e1,e2,e3,..)
```

Examples:

- **Creating a basic List**

```
>std_info<- list(521, "Madhu", c(82,56,74),TRUE)
> print(std_info)
# Output:
[[1]]
[1] 521

[[2]]
[1] "Madhu"

[[3]]
[1] 82 56 74

[[4]]
[1] TRUE
```

- **Creating aNamed List**

```
>std_info<- list(rno=521, name="Madhu", scores=c(82,56,74))
> print(std_info)
# Output:
$rno
[1] 521

$name
[1] "Madhu"

$scores
[1] 82 56 74
```

General List Operations:

Lists in R are flexible data structures that allow storing heterogeneous data types. Here are common list operations:

List Indexing:

In R, Indexing allows the user to access elements from the list. For this purpose, there are three operators. those are,

- [] Operator
- [[]] Operator
- \$ Operator

[] Operator:

[] operator is used to extract element(s) from a list. This allows us to extract multiple elements of a list.

Example:

```
>std_ls<- list(rno=521, name="Madhu", scores=c(82,56,74))
>std_ls[1]
# Output:
$rno
[1] 521

>std_ls[2:3]
# Output:
$name
[1] "Madhu"

$scores
[1] 82 56 74
```

[[]] Operator:

[[]] operator is used for extracting elements from list-objects. This is the same as [] operator but the only difference is that [[]] selects only one element whereas [] operator can select more than 1 element in a single command.

Example:

```
>std_ls<- list(rno=521, name="Madhu", scores=c(82,56,74))
>std_ls[[1]]
# Output : 521
>std_ls[[2]]
# Output: "Madhu"
>std_ls[[3]]
# Output: 82 56 74
```

- `[[ ]]` operator is mainly used to extract nested element of a list.

Example:

```
>std_ls<- list(rno=521, name="Madhu", scores=c(82,56,74))
>std_ls[[3]][[2]] #To access second element of third element
# Output : 56
```

\$ Operator:

\$ operator can be used for lists and data frames in R. Unlike `[ ]` operator, it selects only a single element at a time. It can be used to access an element in named list

Example:

```
>std_ls<- list(rno=521, name="Madhu", scores=c(82,56,74))
>std_ls$rno
# Output: 521
>std_ls$name
# Output: "Madhu"
>std_ls$scores
# Output: 82 56 74
>std_ls$scores[2]
# Output: 56
>std_ls$scores[3]
# Output: 74
```

Adding and Deleting List Elements:

In R, you can add and delete elements from a list dynamically.

Adding Elements to List:

There are different ways to add elements to the list, those are,

- *Using \$ Operator (for Named Elements):*

By using this operator, you can add elements to the named list.

Example:

```
> users <- list(name="Naveen",age=32)
># adding an element to list
>users$city<- "Hyd"

>print(users)
# Output:
$name
[1] "Naveen"
$age
[1] 32
$city
[1] "Hyd"
```

- *Using `[[ ]]` (for Named or Indexed Elements)*

By using this operator, you can add elements to the named list or Indexed List.

Example:

```
> users[["country"]] <- "India"
> print(users)
# Output:
$name
[1] "Naveen"

$age
[1] 32

$city
[1] "Hyd"

$country
[1] "India"
```

- *Using `c()` Function (For Appending Elements)*

By using `c()` function, you can append elements to list.

Example:

```
> users <- c(users, list(gender="Male"))
> print(users)
# Output:
$name
[1] "Naveen"

$age
[1] 32

$city
[1] "Hyd"

$country
[1] "India"

$gender
[1] "Male"
```

Deleting Elements from List:

There are different ways to delete elements from the list, those are,

- *Assign **NULL** to Remove an Element:*

You can remove element from the list by assigning NULL value to it.

Example:

```
> users$city <- NULL
> print(users)
# Output:
  $name
[1] "Naveen"

  $age
[1] 32

  $country
[1] "India"

  $gender
[1] "Male"

> users[["country"]] <- NULL
> print(users)
# Output:
  $name
[1] "Naveen"

  $age
[1] 32

  $gender
[1] "Male"
```

- *Using **[-index]** or Negative Indexing:*

Example:

```
> users <- users[-2] # removing 2nd element(age)
> print(users)
# Output:
  $name
[1] "Naveen"

  $gender
[1] "Male"
```

Getting the Size of a List:

In R, we can obtain the **size(number of elements)** of a list using the ***length()*** function.

Syntax:

```
length(list)
```

Example:

```
>number_ls<- list(10,20,30,40,50,60)
>print(length(number_ls))
# Output: 6

>std_ls<- list(rno=521, name="Madhu", scores=c(82,56,74))
>print(length(std_ls))
# Output: 3
```

Extended Example: Text Concordance Using Lists in R

A **text concordance** is an index of words in a text, along with the positions where they appear. In other words, A text concordance is a tool used in text analysis to display the occurrences of a particular word in a text.

In R, you can create a simple text concordance using lists and basic string manipulation functions. Below is an extended example of how to create a text concordance using lists in R.

Example:

```
># Sample text
> text <- "R is a programming language. R is great for data analysis.
Data science uses R."

># Split the text into words
> words <- unlist(strsplit(tolower(text), "\\W+"))

>print(words)
# Output:
      "r"      "is"      "a"      "programming" "language"      "r"      "is"
"great""for"  "data" "analysis"      "data"      "science"
"uses"          "r"

># Initialize an empty list to store the concordance
>wl<- list()

># Loop through the words to find occurrences of the each word
>for (iin 1:length(words)) {
  wrd<- words[i]
  wl[[wrd]] <- c(wl[[wrd]],i)
}
```

```
># print the concordance
```

```
>print(w1)
```

```
# Output:
```

```
$r
```

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

```
$is
```

```
[1] 2 7
```

```
$a
```

```
[1] 3
```

```
$programming
```

```
[1] 4
```

```
$language
```

```
[1] 5
```

```
$great
```

```
[1] 8
```

```
$`for`
```

```
[1] 9
```

```
$data
```

```
[1] 10 12
```

```
$analysis
```

```
[1] 11
```

```
$science
```

```
[1] 13
```

```
$uses
```

```
[1] 14
```

In the above example,

- ***tolower(text)***: This converts the entire string of text to lowercase.
- ***strsplit(tolower(text), "\\W+")***: The `strsplit()` function splits the text into words. Here, `\\W+` is a regular expression that matches one or more non-word characters (i.e., anything other than letters, numbers, and underscores).
- ***unlist()***: This converts the list produced by `strsplit()` into a vector.

Accessing List Components and Values:

In R, you can access list components and values using functions like **names()** and **unlist()**.

names() function:

The **names()** function returns or sets the names of the components in a list. You can use this function to access or assign names to list elements.

- *Access names of List elements:*

The **names ()** function returns a vector of the names assigned to the components of the list.

Example:

```
> # Create a named list
> std <- list(name = "Mohan", age = 20, scores = c(85, 90, 95))
> # Print the names of the components
> names(std)
# Output: "name"      "age"      "scores"
```

- *Assign names to List elements:*

You can also assign names to the components of a list using the **names()** function.

Example:

```
> # Assign new names to the components of the list
> names(std) <- c("first_name", "years_old", "grades")
# Print the modified list with new names
> print(std)
$first_name
[1] "Mohan"

$years_old
[1] 20

$grades
[1] 85 90 95
```

- *Accessing values from List using names:*

You can also access values from the list using names.

Example:

```
> print(std["name"])
# Output:
$name
"Mohan"

> print(std["scores"])
# Output:
$scores
85 90 95
```

unlist() function:

The **unlist()** function is used to flatten a list. This function combines all the components of a list into a single vector, ignoring the structure of the list.

Example:

```
> std <- list(name = "Mohan", age = 20, scores = c(85, 90, 95))
# flatten the list using unlist()
> std_u <- unlist(std)
> print(std_u)
# Output:
      name      age scores1 scores2 scores3
"Mohan"    "20"    "85"    "90"    "95"
```

- *Extracting Values After unlist()*

Once you have flattened the list using **unlist()**, you can access individual values as you would in a normal vector.

Example:

```
> print(std_u["name"])
# Output:
name
"Mohan"
> print(std_u["age"])
# Output:
age
"20"
> print(std_u["scores2"])
# Output:
scores2
"90"
```

unname() function:

The **unname()** function is used to remove names from the named list.

Example:

```
> std <- list(name = "Mohan", age = 20, scores = c(85, 90, 95))
> stdun <- unname(std) # To remove names
> print(stdun)
# Output:
[[1]]
[1] "Mohan"

[[2]]
[1] 20

[[3]]
[1] 85 90 95
```

Applying Functions to Lists:

In R, functions can be applied to **lists** in various ways using functions like `lapply()`, `sapply()` and `Map()`.

lapply() function:

The `lapply()` function applies a function to each element of a list and **returns a list**.

Syntax:

```
lapply(list, function)
```

Example:

```
> my_list <- list(1:3, 4:6, 7:9)
> res_ls <- lapply(my_list, sum)
> print(res_ls)
# Output:
[[1]]
[1] 6

[[2]]
[1] 15

[[3]]
[1] 24

> class(res_ls)
# Output:
[1] "list"
```

sapply() function:

The `sapply()` function is similar to `lapply()`, but it simplifies the output to a vector or matrix if possible.

Syntax:

```
sapply(list, function)
```

Example:

```
> my_list <- list(1:3, 4:6, 7:9)
> result <- sapply(my_list, sum)
> print(result)
# Output:
[1] 6 15 24

> class(result)
# Output:
[1] "integer"
```

Map() function:

The Map() function applies a function to **multiple lists at the same time**.

Syntax:

```
Map (function, list1, list2)
```

Example:

```
> list1 <- list(a = 1:3, b = 4:6)
> list2 <- list(c = 7:9, d = 10:12)
> result <- Map(`+`, list1, list2)
> print(result)
# Output:
$a
[1]  8 10 12

$b
[1] 14 16 18
```

Data Frame

In R, A **data frame** is a two-dimensional array-like structure, or we can say it is a **table** in which each column contains the value of one variable, and row contains the set of value from each column. Data Frames are two-dimensional, heterogeneous data structures.

Data Frames can have different types of data inside it. While the first column can be character, the second and third can be numeric or logical. However, each column should have the same type of data.

Creating a Data Frame:

In R, To create a data frame we use the **data.frame()** function.

Syntax:

```
df <- data.frame(vector1, vector2, ..)
```

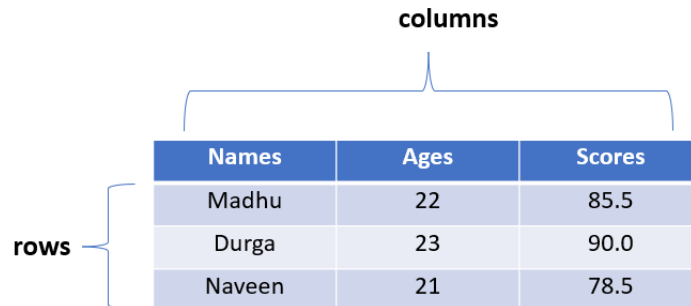
Example:

```
> # Create a data frame
> students <- data.frame(
  Names = c("Madhu", "Durga", "Naveen"),
  Ages = c(22, 23, 21),
  Scores = c(85.5, 90.0, 78.5)
)
# print data frame
> print(students)
  Names Ages Scores
1 Madhu   22  85.5
2 Durga   23  90.0
3 Naveen  21  78.5
```

Here, the data frame `students` has three columns: **Names** (character), **Ages** (numeric), and **Scores** (numeric).

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.



The diagram illustrates the structure of a data frame. A bracket labeled "columns" spans the top of the table, and a bracket labeled "rows" spans the left side of the table.

Names	Ages	Scores
Madhu	22	85.5
Durga	23	90.0
Naveen	21	78.5

In R, Once a data frame is created, you can examine its structure using:

```
> str(students)
'data.frame':  3 obs. of  3 variables:
 $ Names : chr  "Madhu" "Durga" "Naveen"
 $ Ages  : num  22 23 21
 $ Scores: num  85.5 90 78.5
```

Accessing Data Frames:

Once you create a data frame in R, you can access specific elements, columns, or rows using various methods. Below are different ways to access data from a data frame.

Accessing a Specific Column:

There are multiple ways to extract a column from a data frame.

- Using \$ Operator:

Example:

```
> print(students$Names)
# Output:
[1] "Madhu" "Durga" "Naveen"

> print(students$Ages)
# Output:
[1] 22 23 21
```

Note: This returns a **vector** instead of a data frame.

- Using [] Notation:

Example:

```
> print(students["Scores"])
# Output:
  Scores
1   85.5
2   90.0
3   78.5

> print(students["Names"])
# Output:
  Names
1 Madhu
2 Durga
3 Naveen
```

Note: This returns a data frame.

- Using [] Notation:

Example:

```
> print(students[["Scores"]])
# Output:
[1] 85.5 90.0 78.5

> print(students[["Names"]])
# Output:
[1] "Madhu" "Durga" "Naveen"
```

Note: This returns a **vector** instead of a data frame.

- Using **column** indexing:

Example:

```
> print(students[,2]) # To access 2nd column
# Output:
[1] 22 23 21

> print(students[,1]) # To access 1st column

# Output:
[1] "Madhu" "Durga" "Naveen"
# To access Multiple columns
> print(students[,c(1,3)])
# Output:
  Names Scores
1 Madhu   85.5
2 Durga   90.0
3 Naveen   78.5
```

Accessing a Specific Row:

To access a particular row, use **row indexing**:

Example:

```
> print(students[2,]) # To access 2nd row
# Output:
  Names Ages Scores
2 Durga   23     90

> print(students[3,]) # To access 3rd row
# Output:
  Names Ages Scores
3 Naveen  21    78.5

# To access multiple rows
> print(students[c(1,3),])
# Output:
  Names Ages Scores
1  Madhu   22    85.5
3 Naveen  21    78.5
```

Accessing a Specific Element (Row & Column):

You can extract a single value by specifying **row and column index**.

Example:

```
> print(students[1,2])
# Output:
[1] 22

> print(students[1,1])
# Output:
[1] "Madhu"

> print(students[2,3])
# Output:
[1] 90

> print(students[3,"Scores"])
# Output:
[1] 78.5
```

Extended Example: Exam Grades

First, you can create a file called "ExamsQuiz.txt" along with the following data.

"Exam 1"	"Exam 2"	Quiz
2.0	3.3	4.0
3.3	2.0	3.7
4.0	4.0	4.0
2.3	0.0	3.3
2.3	1.0	3.3
3.3	3.7	4.0

As you can see, each line contains the three test scores for one student. This is the classic two-dimensional file notion.

Notice that we have separated the fields here by spaces (use ***read.table()*** function). Other delimiters may be specified, notably commas for comma-separated value (CSV) files (use ***read.csv()*** function).

The variable names specified in the first record must be separated by the same delimiter as used for the data, which is spaces in this case. If the names themselves contain embedded spaces, as we have here, they must be quoted.

Next, you can read the above text file using ***read.table()*** function.

```
> examsquiz <- read.table("ExamsQuiz.txt",header=TRUE)
> class(examsquiz)
# Output:
[1] "data.frame"
> examsquiz
# Output:
  Exam.1 Exam.2 Quiz
1    2.0    3.3  4.0
2    3.3    2.0  3.7
3    4.0    4.0  4.0
4    2.3    0.0  3.3
5    2.3    1.0  3.3
6    3.3    3.7  4.0
```

Note: In the above the column names appears with periods by replacing blanks.

To access few records from first and last:

```
> head(examsquiz,2)
# Output:
  Exam.1 Exam.2 Quiz
1    2.0    3.3  4.0
2    3.3    2.0  3.7
> tail(examsquiz,2)
# Output:
  Exam.1 Exam.2 Quiz
5    2.3    1.0  3.3
6    3.3    3.7  4.0
```

Other Matrix-Like Operations:

In R, **data frames** are similar to **matrices**, but they can hold **different data types** (numeric, character, etc.). However, you can still perform **matrix-like operations** on data frames.

Extracting Sub data Frames:

You can extract **subsets** of a data frame in R using **indexing, conditions, and functions like subset()**. Here are different ways to extract sub data frames.

- *Extracting Specific Rows and Columns*

You can extract a subset using **row and column indexing**.

Example:

```
# Student data frame
df <- data.frame(
  Name = c("Madhu", "Durga", "Naveen", "Kiran", "David"),
  Age = c(22, 24, 23, 21, 25),
  Score = c(85.5, 90.0, 78.5, 88.0, 92.5)
)
# Extract rows 1 to 3
sub_df <- df[1:3, ]
print(sub_df)
# Output:
  Name Age Score
1 Madhu  22  85.5
2 Durga  24  90.0
3 Naveen 23  78.5

# Extract specific columns (e.g., "Name" and "Score")
sub_df2 <- df[, c("Name", "Score")]
print(sub_df2)
# Output:
  Name Score
1 Madhu  85.5
2 Durga  90.0
3 Naveen  78.5
4 Kiran  88.0
5 David  92.5
```

- *Extracting Data Based on Conditions*

You can extract rows **where a condition is met**.

Example:

```
> high_scores <- df[df$Score >= 90, ]
> print(high_scores)
# Output:
  Name Age Score
2 Durga  24  90.0
5 David  25  92.5
```

- *Using the subset() Function*

The subset() function allows **easier filtering** of rows.

Example:

```
> young_students <- subset(df, Age < 23)
> print(young_students)
# Output:
  Name Age Score
1 Madhu  22  85.5
4 Kiran  21  88.0
```

Using the rbind() and cbind() Functions:

In R, rbind() and cbind() are used to combine data frames, matrices, or vectors by rows and columns, respectively.

rbind():

This function used to combine data frames or add a new row to existing data frame.

Example:

```
> new_row <- data.frame(Name="Kishore",Age=20,Score=86.45)
> df <- rbind(df,new_row)
> print(df)
# Output:
  Name Age Score
1 Madhu  22  85.50
2 Durga  24  90.00
3 Naveen 23  78.50
4 Kiran  21  88.00
5 David  25  92.50
6 Kishore 20  86.45
```

cbind():

This function used to add a new column to existing data frame.

Example:

```
> City <- c("Hyd","Sec","Del","Mum","Hyd","Sec")
> df <- cbind(df,City)
> print(df)
# Output:
  Name Age Score City
1 Madhu  22  85.50 Hyd
2 Durga  24  90.00 Sec
3 Naveen 23  78.50 Del
4 Kiran  21  88.00 Mum
5 David  25  92.50 Hyd
6 Kishore 20  86.45 Sec
```