

CSE1006

Foundations of Data Analytics

Adla Padma
Assistant Professor
School of Computer Science & Engineering
VIT-AP University

Module - 2

Introduction to R

- ❖ Introduction to R
- ❖ R installation
- ❖ Basic operations in R using command line
- ❖ use of IDE R Studio
- ❖ 'R help' feature in R
- ❖ Data types and function
- ❖ Variables in R
- ❖ Scalars, Vectors, Matrices, List, Data frames
- ❖ functions in R, Factors

Introduction to R

R is a popular programming language used for statistical computing and graphical presentation.

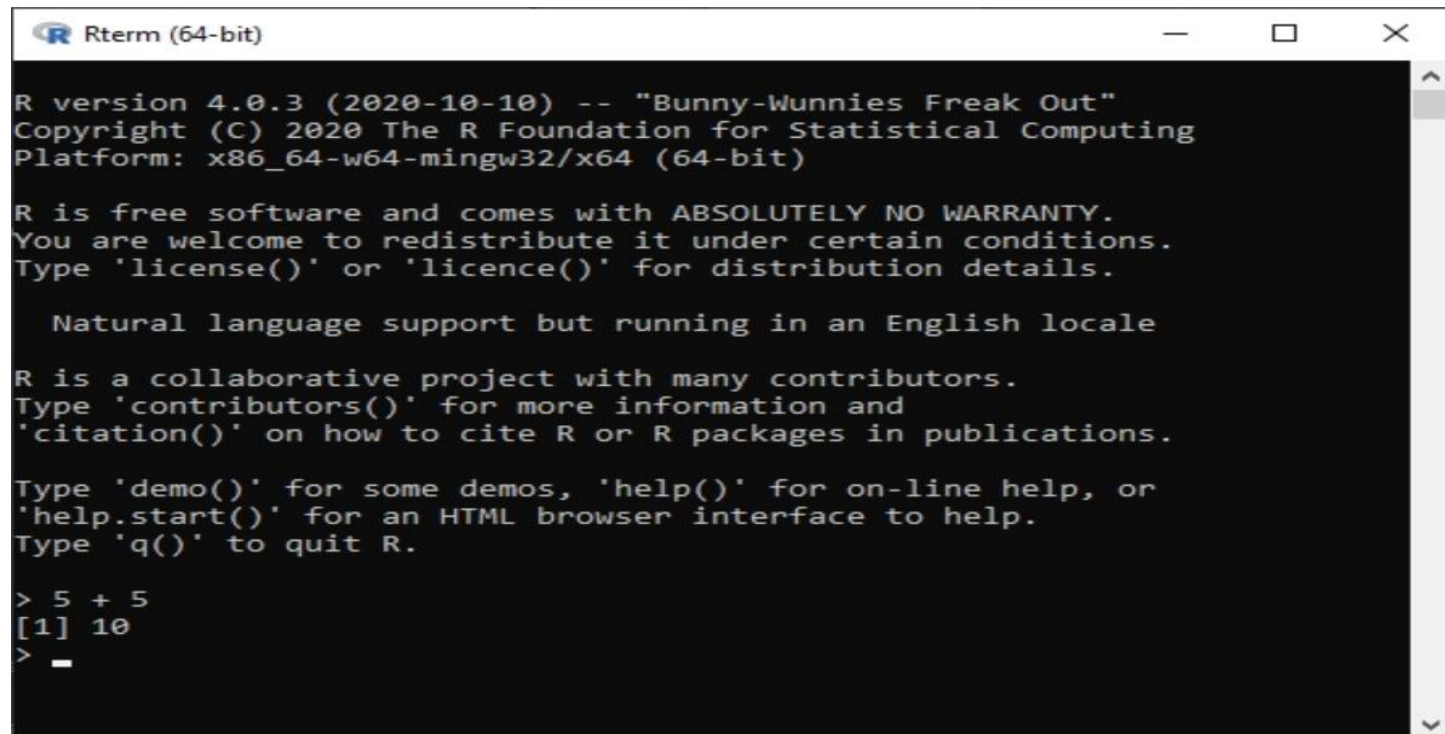
Its most common use is to analyze and visualize data.

Features of R:

- It is a great resource for data analysis, data visualization, data science and machine learning
- It provides many statistical techniques (such as statistical tests, classification, clustering and data reduction)
- It is easy to draw graphs in R, like pie charts, histograms, box plot, scatter plot, etc.
- It works on different platforms (Windows, Mac, Linux)
- It is open-source and free
- It has many packages (libraries of functions) that can be used to solve different problems.

Installation of R

- To install R, go to <https://cloud.r-project.org/> and download the latest version of R for Windows, Mac or Linux.
- When you have downloaded and installed R, you can run R on your PC.
- The screenshot below shows how it may look like when you run R on a Windows PC:



```
Rterm (64-bit)

R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

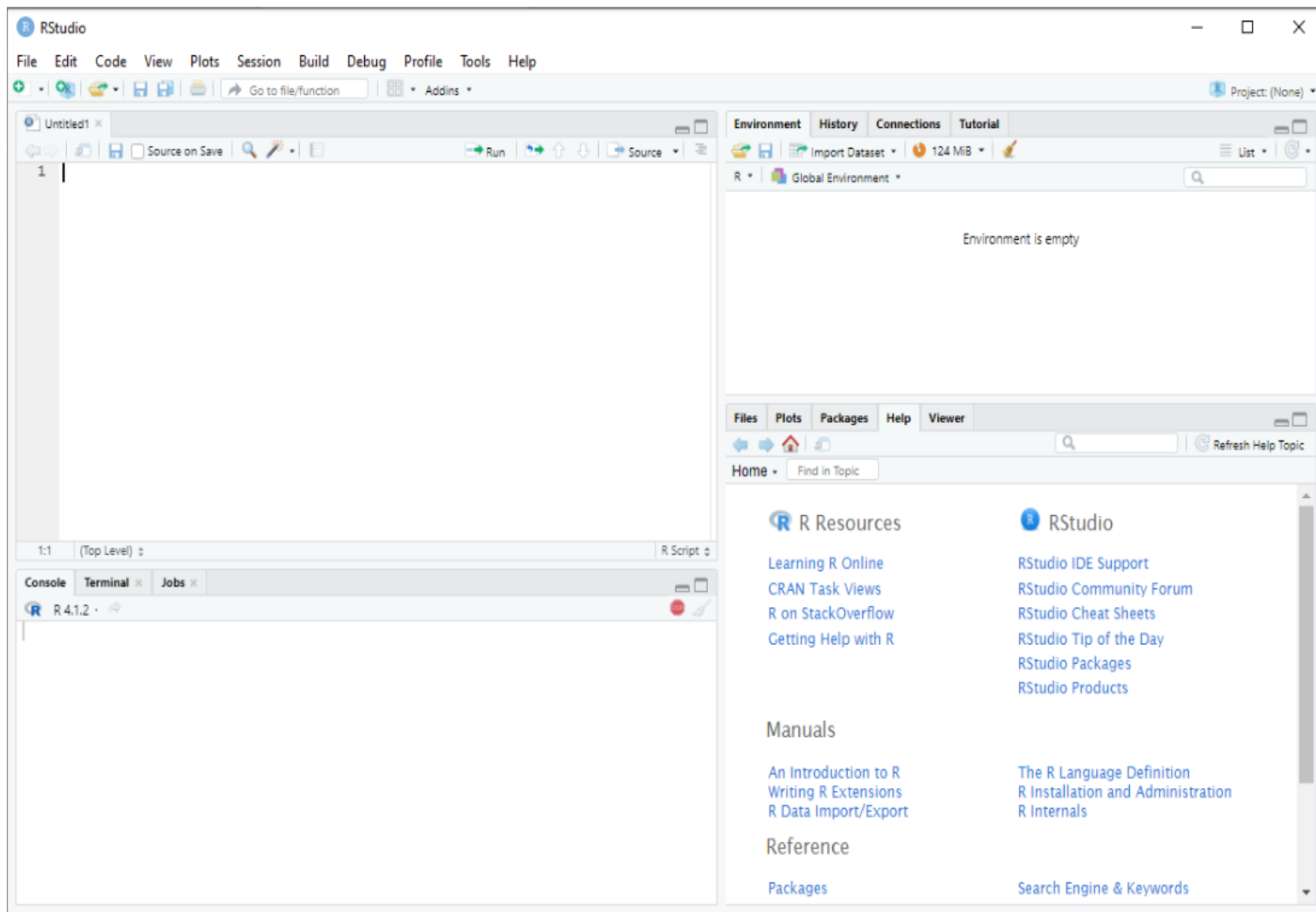
> 5 + 5
[1] 10
> -
```

Installing RStudio Desktop

To install RStudio Desktop on your computer, do the following:

1. Go to the [RStudio](https://www.rstudio.com/) website.
2. Click on "**DOWNLOAD**" in the top-right corner.
3. Click on "**DOWNLOAD**" under the "**RStudio Open Source License**".
4. Download RStudio Desktop recommended for your computer.
5. Run the RStudio Executable file (.exe) for Windows OS or the Apple Image Disk file (.dmg) for macOS X.
6. Follow the installation instructions to complete RStudio Desktop installation.

RStudio is now successfully installed on your computer. The RStudio Desktop IDE interface is shown in the figure below:

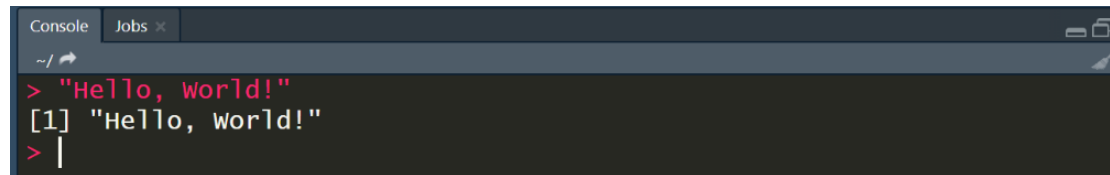


Running R code

We will be using RStudio but we can also use R command prompt by typing the following command in the command line.

`$ R`

This will launch the interpreter and now let's write a basic Hello World program to get started

A screenshot of an R console window. The window has a title bar with 'Console' and 'Jobs x'. Below the title bar is a dark grey area with a prompt character '>'. The console shows the command '> "Hello, world!"' in red text, followed by the output '[1] "Hello, world!"' in white text. The prompt character '>' is followed by a vertical bar '|' indicating the cursor is ready for the next command.

```
> "Hello, world!"  
[1] "Hello, world!"  
> |
```

Usually, we will write our code inside scripts which are called **RScripts** in R. To create one, write the below given code in a file and save it as **myFile.R** then run it in console by writing:

`Rscript myFile.R`

Print Statement:

Unlike many other programming languages, you can output code in R without using a print function.

To output text in R, use single or double quotes.

Ex1: "Hello World!"

Output: [1] "Hello World!"

Ex2: 'Hello World!'

Output: [1] 'Hello World!'

To output numbers, just type the number (without quotes)

Ex3: 5

Output: [1] 5

To do simple calculations, add numbers together

Ex3: 5+3

Output: [1] 8

R does have a print() function to output the code.

Ex: print("Hello World!")

Output: [1] "Hello World!"

R Variables:

- Variables are containers for storing data values.
- R does not have a command for declaring a variable. A variable is created the moment you first assign a value to it. To assign a value to a variable, use the Assignment Operator.
- To output (or print) the variable value, just type the variable name.

Example:

```
name <- "VIT AP"
```

```
YOE <- 2018
```

```
name # output "VIT AP"
```

```
YOE # output 2018
```

Print() function also can be used to output variable value

```
Print (name) # output "VIT AP"
```

```
Print (name,YOE) # output "VIT AP 2018"
```

```
Print (name,"Amaravati") # output "VIT AP Amaravati"
```

```
Print ("Name=",name) # output "Name = VIT AP"
```

Rules to naming a R Variable:

- A variable name must start with a letter and can be a combination of letters, digits, period(.) and underscore(_). If it starts with period(.), it cannot be followed by a digit.
- A variable name cannot start with a number or underscore (_)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- Reserved words cannot be used as variables (TRUE, FALSE, NULL, if...)

Legal variable names:

Myvar, my_var, myVar, MYVAR, Myvar2, .myvar

Illegal variable names:

2myvar, my-var, my var, _my_var, my_v@ar, TRUE

R Data Types

- Variables can store data of different types, and different types can do different things.
- In R, variables do not need to be declared with any particular type and can even change type after they have been set.

Basic data types in R can be divided into the following types:

numeric - (10.5, 55, 787)

integer - (1L, 55L, 100L, where the letter "L" declares this as an integer)

complex - (9 + 3i, where "i" is the imaginary part)

character (string) - ("k", "R is exciting", "FALSE", "11.5")

logical (boolean) - (TRUE or FALSE)

The **class** function is used to check the data type of a variable.

```
x <- 10.5
```

```
class(x)    #Numeric
```

```
x <- "R is exciting"
```

```
class(x)    #Character
```

```
x <- 1000L
```

```
class(x)    #Integer
```

```
x <- TRUE
```

```
class(x)    #Boolean
```

```
x <- 9i + 3
```

```
class(x)    #complex
```

Operators in R

Operators are the symbols directing the compiler to perform various kinds of operations between the operands.

Types of the operator in R language

1. Arithmetic Operators
2. Logical Operators
3. Relational Operators
4. Assignment Operators
5. Miscellaneous Operator

Arithmetic Operators:

Arithmetic operations in R simulate various math operations, like addition, subtraction, multiplication, division, and modulo using the specified operator between operands.

Operands may be either scalar values, complex numbers, or vectors.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponent
%%	Modulus (Remainder from division). Give the remainder of the first vector with the second
%/%	Integer Division. The result of division of first vector with second (quotient)

Example : Arithmetic.R

1. # R Program to demonstrate Arithmetic operators
2. a=10
3. b=20
4. cat("SUM=",a+b)
5. cat("Subtration=",a-b)
6. cat("Multipliation=",a*b)
7. cat("Division=",a/b)
8. cat("reminder=",a%%b)
9. cat("Integer division=",a%/%b)

> Rscript Arithmetic.R

SUM= 30Subtration= -10Multipliation= 200Division=
0.5reminder= 10Integer reminder= 0

Assignment Operators:

Assignment operators are used to assign values to variables.

In R, we can use `=`, `->`, `->>`, `<-`, `<<-` as an Assignment operator.

Example: `A=10` (or) `A<-10` (or) `A<<-10` (or) `10->>A` (or) `10->A`

It is possible to assign same value to multiple variables at a time.

Example: `A<-B<-C<-10`

Relational Operators:

Relational operators are used to compare two values.

Returns a boolean TRUE value if the first operand satisfies the relation compared to the second and Returns FALSE otherwise.

A TRUE value is always considered to be greater than the FALSE.

R supports these relational operators `<`, `>`, `<=`, `>=`, `==`, `!=`

Logical Operators:

- Logical operators are used to combine conditional statements.
- Logical operations 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.

Operator	Description
&	Element-wise Logical AND operator. It returns TRUE if both elements are TRUE
&&	Logical AND operator - Returns TRUE if both statements are TRUE
	Elementwise- Logical OR operator. It returns TRUE if one of the statement is TRUE
	Logical OR operator. It returns TRUE if one of the statement is TRUE.
!	Logical NOT - returns FALSE if statement is TRUE

&- Returns True if both the operands are True.

|- Returns True if either of the operands is True.

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

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

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

Example:	<code>list1 <- c(TRUE, 0.1)</code>	
	<code>list2 <- c(0,4+3i)</code>	
True	<code>print(list1 & list2)</code>	<code>#Output : False</code>
True	<code>print(list1 list2)</code>	<code>#Output : True</code>
False	<code>print(!list1)</code>	<code>#Output : False</code>
	<code>print(list1 && list2)</code>	<code>#Output : False</code>
	<code>print(list1 list2)</code>	<code>#Output : True</code>

R Miscellaneous Operators:

Miscellaneous operators are used to manipulate data

Operator	Description	Example
:	Creates a series of numbers in a sequence	<code>x <- 1:10</code>
<code>%in%</code>	Find out if an element belongs to a vector	<code>x %in% y</code>
<code>%*%</code>	Matrix Multiplication	<code>x <- Matrix1 %*% Matrix2</code>

Taking Input from User in R

In R, **readline()** method allows user to provide input to the code interactively. **readline()** method takes input in string format.

Example: `var = readline();`

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

Example:

```
print("enter some data");
```

```
var=readline();
```

```
var=as.integer(var);
```

```
print(var);
```

```
var = readline(prompt = "Enter any number : ");
```

```
x = scan() # scan() is input function
```

```
x = "R programming"
```

```
cat(x, "is best\n") # cat() is Output function
```

Control statements

Control statements are expressions used to control the execution and flow of the program based on the conditions provided in the statements.

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

Syntax: if(expression){
 statements

 }

2. **if – else condition:** It is similar to if condition but when the test expression in if condition fails, then statements in else condition are execute.

Syntax: if(expression){
 statements

 }
 else{
 statements

 }

Control statements

3. if-else-if ladder:

Syntax:

```
if(condition 1 is true) {  
    execute this statement  
} else if(condition 2 is true) {  
    execute this statement  
}  
.....  
else {
```

execute this statement #This block

is executed when none of the
above conditions are true.

4. Nested if – else condition:

Syntax:

```
if(expression)  
{
```

#Parent Condition

if-else statement

#Child

condition

```
}  
else{
```

if-else statement

#Child

condition

```
}
```

Control statements

Example: # R Nested if else statement Example

```
if(a > b){  
    if(a > c){  
        print("a is big")  
    }  
    else{  
        print("c is big")  
    }  
}  
else{  
    if(b > c){  
        print("b is big")  
    }  
    else{  
        print("c is big")  
    }  
}
```

Control statements

5. Switch Statement:

- In this switch function expression is matched to list of cases.
- If a match is found, then it prints that case's value.
- No default case is available here.
- If no case is matched it outputs NULL.

Syntax: switch (expression, case1, case2, case3,...,case n)

Eg1 :

```
# Expression in terms of the index value
x <- switch(2,                                # Expression
            "VIT AP",                          # case 1
            "University",                      # case 2
            "Amaravati"                       # case 3
        )
print(x)                                     # University
```

Control statements

5. Switch Statement:

Eg2 :

```
# Expression in terms of the string value
y <- switch(
    "place",
    "name"="VIT AP",
    "title"="University",
    "place"="Amaravati"
) print(y)
```

Expression
case 1
case 2
case 3
Amaravati

Eg3 :

```
# Expression in terms of the string value
z <- switch(
    "place",
    "name"="VIT AP",
    "title"="University"
) print(z)
```

NULL

LOOPS

- Loops can execute a block of code as long as a specified condition is reached.
- Loops save time, reduce errors, and they make code more readable.
- R has two loop commands:
 - while loops
 - for loops

While Loop:

Syntax:

```
while (test_expression)
{
    statement(s)
    update expression
}
```


LOOPS

For Loop:

For loop in R is useful to iterate over the elements of a list, data frame, vector, matrix, or any other object.

Syntax:

```
for (Var in vector){  
    statement(s)  
}
```

Repeat Loop:

Repeat loop, unlike other loops, doesn't use a condition to exit the loop instead it looks for a break statement that executes if a condition within the loop body results to be true.

Syntax:

```
repeat {  
    commands  
    if (condition) {  
        break  
    }  
}
```

Break and Next statements

The basic function of the Break and Next statement is to alter the running loop in the program and flow the control outside of the loop.

break statement:

The break Statement in R is a jump statement that is used to terminate the loop at a particular iteration.

Syntax: break

Example:

```
N=1:10
for(val in N)
{
    if(val==5)
    {
        break
    }
    print(val)
    cat("\n");
}
```

Output:

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

Break and Next statements

next statement:

The next statement in R is used to skip the current iteration in the loop and move to the next iteration without exiting from the loop itself.

This is similar to continue statement in C language.

Syntax: next

Example:

```
N=1:7
for(val in N)
{
    if(val==5)
    {
        next
    }
    print(val)
    cat("\n");
}
```

Output:

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

Functions

A function is a block of code which only runs when it is called. Functions are useful when you want to perform a certain task multiple times. A function accepts input arguments and returns the output by executing valid R commands that are inside the function.

Creating a function:

Functions are created in R by using the command **function()**.

Syntax:

```
F=function(arguments)           #Here, F is a function  
name.  
{  
    Statements  
}
```

Statements in function() runs when it is called.

To call a function, use the function name followed by parenthesis, like F()

Types of Function

Built-in Function: Built-in functions in R are pre-defined functions that are available in R programming language to perform common tasks or operations.

Examples:

Find sum of numbers 4 to 6.

```
print(sum(4:6))
```

#output : 15

Find max of numbers 4 and 6.

```
print(max(4:6))
```

#output : 6

Find min of numbers 4 and 6.

```
print(min(4:6))
```

#output : 4

User-defined Function: R language allow us to write our own function.

Example:

A simple R function to check whether x is even or odd

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

```
print(evenOdd(4))
print(evenOdd(3))
```

Output:

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

function to add 2 numbers

```
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)
```

Example:

A simple R function to calculate area and perimeter of a rectangle

```
Rectangle = function(length, width)
```

```
{
```

```
    area = length * width
```

```
    perimeter = 2 * (length + width)
```

creating an object called result which is a list of area and perimeter

```
    result = list("Area" = area, "Perimeter" = perimeter)
```

```
    return(result)
```

```
}
```

```
resultList = Rectangle(2, 3) # function call
```

```
print(resultList["Area"])
```

```
print(resultList["Perimeter"])
```

Output:

```
[1] 6
```

```
[1] 10
```

Recursive function

```
my_function <- function(fname) {  
  paste(fname, "princy")  
}
```

```
my_function("krithi")  
my_function("vinu")  
my_function("priya")
```

Recursive: It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

```
tri_recursion <- function(k)  
{  
  if (k > 0) {  
    result <- k +  
    tri_recursion(k - 1)  
    print(result)  
  } else {  
    result = 0  
    return(result)  
  }  
}  
tri_recursion(6)
```

1. Factorial using recursion
2. Fibonacci sequence using recursion

Statistical functions

Ex. Mean(), median(), cor(), var(), sd()

Create a data frame

```
data <- data.frame(  
  Age = c(25, 30, 35, 40, 45),  
  Salary = c(50000, 60000, 70000, 80000, 90000)  
)
```

Statistical functions

```
mean_age <- mean(data$Age) # Calculate mean of Age  
median_salary <- median(data$Salary) # Calculate median of Salary  
# Calculate correlation between Age and Salary  
correlation <- cor(data$Age, data$Salary)  
variance_age <- var(data$Age) # Calculate variance of Age  
std_dev <- sd(data$Age) #standard deviation  
cat("Mean Age:", mean_age, "\n")  
cat("Median Salary:", median_salary, "\n")  
cat("Correlation between Age and Salary:", correlation, "\n")  
cat("Variance of Age:", variance_age, "\n")  
cat("standard deviation of Age:", std_dev, "\n")
```

unique(x)

subset(x, condition, select)

aggregate(formula, data, FUN)

formula: specifying the grouping and the column to summarize (e.g., Salary ~ Department).

FUN: Function to apply (e.g., mean, sum).

order(..., decreasing = FALSE)

...: Vector(s) or column(s) to sort by.

decreasing: Logical, whether to sort in descending order

Data Manipulation Functions

Ex. Unique(), subset(), aggregate(), order()

```
# Create a data frame
```

```
data <- data.frame(  
  ID = c(1, 2, 3, 4, 5, 6),  
  Department = c("HR", "IT", "HR", "Finance", "IT", "Finance"),  
  Salary = c(50000, 60000, 55000, 70000, 65000, 75000)  
)
```

```
# Data manipulation
```

```
unique_departments <- unique(data$Department) # Get unique department  
subset_data <- subset(data, Salary > 60000) # Filter rows with Salary > 60000  
average_salary <- aggregate(Salary ~ Department, data, mean)  
ordered_data <- data[order(data$Salary), ] # Salary in ascending order
```

```
# Print results
```

```
cat("Unique Departments:\n", unique_departments, "\n")  
cat("\nSubset of Data (Salary > 60000):\n")  
print(subset_data)  
cat("\nAverage Salary by Department:\n")  
print(average_salary)  
cat("\nData Ordered by Salary:\n")  
print(ordered_data)
```

Functions with Arguments

function with arguments

```
calculate_square <- function(x) {  
  result <- x^2  
  return(result)  
}  
value1 <- 5  
square1 <- calculate_square(value1)  
print(square1)  
  
value2 <- -2.5  
square2 <- calculate_square(value2)  
print(square2)
```

```
#To check n is divisible by 5 or not  
divisibleBy5 <- function(n){  
  if(n %% 5 == 0){  
    return("number is divisible by 5")  
  }else{  
    return("number is not divisible by  
5")  
  }  
}  
# Function call  
divisibleBy5(100)  
divisibleBy5(4)  
divisibleBy5(20.0)
```

Data Structures in R

- A data structure is a particular way of organizing data in a computer so that it can be used effectively.
- The idea is to reduce the space and time complexities of different tasks. Data structures in R programming are tools for holding multiple values.
- R's base data structures are often organized by their dimensionality (1D, 2D, or nD) and whether they're homogeneous (all elements must be of the identical type) or heterogeneous (the elements are often of various types).
- The most essential data structures used in R include:

Vectors

Lists

Dataframes

Matrices

Arrays

Factors

Vectors

A vector is simply a list of items that are of the same type.

Vectors are one-dimensional data structures.

Example:

Vector of strings

```
fruits <- c("banana", "apple", "orange")
```

Print fruits

```
#output: [1] "banana" "apple"
```

```
"orange"
```

To create a vector with numerical values in a sequence, use the `:` operator.

Example:

Vector with numerical values in a sequence

```
numbers <- 1:10
```

```
numbers          #output: [1] 1 2 3 4 5 6 7 8 9 10
```

Vectors

You can also create numerical values with decimals in a sequence, but note that if the last element does not belong to the sequence, it is not used. Vectors are one-dimensional data structures.

Example:

Vector with numerical decimals in a sequence

```
numbers1 <- 1.5:6.5
```

```
numbers1          #output: [1] 1.5 2.5 3.5 4.5 5.5 6.5
```

Vector with numerical decimals in a sequence where the last element is not used

```
numbers2 <- 1.5:6.3
```

```
numbers2          #output: [1] 1.5 2.5 3.5 4.5 5.5
```

It is also possible to create a vector with Boolean values

Example:

Vector of logical values

```
log_values <- c(TRUE, FALSE, TRUE, FALSE)
```

```
log_values  
FALSE
```

```
#output: [1] TRUE FALSE TRUE
```

Vectors

To find out how many items a vector has, use length() function.

Example:

```
#finding number of items in a vector
```

```
fruits <- c("banana", "apple", "orange")
```

```
length(fruits)
```

```
#output: [1] 3
```

To sort items in a vector alphabetically / numerically, use the sort() function.

Example:

```
#sorting a vector
```

```
fruits <- c("banana", "apple", "orange")
```

```
numbers <- c(13, 3, 5, 7)
```

```
sort(fruits) # Sort a string #output: [1] "apple" "banana" "orange"
```

```
sort(numbers) # Sort numbers #output: [1] 3 5 7 13
```


Vectors

You can access the vector items by referring to its index number inside brackets []. The first item has index 1, the second item has index 2.

Example:

```
fruits <- c("banana", "apple", "orange")
```

```
# Access the first item (banana)
```

```
fruits[1]
```

```
#output: [1] "banana"
```

```
fruits[c(1, 3)]
```

```
#output: [1] "banana" "orange"
```

You can also use negative index numbers to access all items except the ones specified.

Example:

```
#to access all items except specified
```

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")
```

```
# Access all items except for the first item
```

```
fruits[c(-1)]
```

```
#output: [1] "apple" "orange" "mango"
```

```
"lemon")
```

Vectors

To change the value of a specific item, refer to the index number

Example:

```
fruits <- c("banana", "apple", "orange")
```

Change "banana" to "pear"

```
fruits[1] <- "pear"
```

Print fruits

```
fruits #output: [1] "pear" "apple" "orange"
```

To repeat vectors, use the `rep()` function.

Example:

```
repeat_each <- rep(c(1,2,3), each = 3)
```

```
repeat_each          #output:[1] 1 1 1 2 2 2 3 3 3
```

```
repeat_times <- rep(c(1,2,3), times = 3)
```

```
repeat_times      #output:[1] 1 2 3 1 2 3 1 2 3
```

Vectors

Use the `seq()` function to make bigger or smaller steps in a sequence.

The `seq()` function has three parameters: “**from**” is where the sequence starts, “**to**” is where the sequence stops and “**by**” is the interval of the sequence.

Example:

```
numbers <- seq(from = 0, to = 100, by = 20)
```

```
numbers
```

```
#output: [1] 0 20 40 60 80 100
```

Lists

A list in R is a generic object consisting of an ordered collection of objects. Lists are one-dimensional, heterogeneous data structures.

The list can be a **list of vectors**, a **list of matrices**, a **list of characters** and a **list of functions**, and so on.

To create a List in R you need to use the function called “**list()**”.

Example:

```
# List of strings
```

```
thislist <- list("apple", "banana", "cherry")
```

```
# Print the list
```

```
thislist
```

You can access the list items by referring to its index number, inside brackets. The first item has index 1, the second item has index 2 and so on.

Example:

```
thislist <- list("apple", "banana", "cherry")
```

```
thislist[1]
```

#Output:

```
[[1]]
```

```
[1] apple
```

Lists

To change the value of a specific item, refer to the index number.

Example:

```
thislist <- list("apple", "banana", "cherry")  
thislist[1] <- "blackcurrant"  
# Print the updated list  
thislist
```

```
[[1]]  
[1] "blackcurrant"  
  
[[2]]  
[1] "banana"  
  
[[3]]  
[1] "cherry"
```

To find out how many items a list has, use length() function

Example:

```
thislist <- list("apple", "banana", "cherry")  
length(thislist)
```

#Output: [1] 3

To find out if a specified item is present in a list, use the **%in%** operator

```
thislist <- list("apple", "banana", "cherry")
```

"apple" %in% thislist

#Output: [1] TRUE

Lists

To add an item to the end of the list, use the `append()` function. After appending it print the newlist by default.

Example:

```
thislist <- list("apple", "banana", "cherry")  
append(thislist, "orange")
```

```
[[1]]  
[1] "apple"  
  
[[2]]  
[1] "banana"  
  
[[3]]  
[1] "cherry"  
  
[[4]]  
[1] "orange"
```

To add an item to the right of a specified index, add "after=index number" in the `append()` function

Example:

```
thislist <- list("apple", "banana", "cherry")
```

```
append(thislist, "orange", after = 2)
```

#Output :

```
"apple" "banana" "orange" "cherry"
```

Lists

You can also remove list items. The following example creates a new, updated list without an "apple" item.

Example:

```
thislist <- list("apple", "banana", "cherry")
```

```
newlist <- thislist[-1]
```

```
newlist                                #Output : "banana" "cherry"
```

Note that there is no change in thislist

You can specify a range of indexes by specifying where to start and where to end the range, by using the : operator

Example:

```
thislist <-
```

```
list("apple", "banana", "cherry", "orange", "kiwi", "melon")
```

```
(thislist)[2:5]      #Output : "banana" "cherry" "orange",
```

Lists

You can also use a list in loops to access items of loop on after another.

Example:

```
thislist <- list("apple", "banana", "cherry")  
for (x in thislist)  
{  
  print(x)  
}
```

You can join two loops using **c()** function.

Example:

```
list1 <- list("a", "b", "c")  
list2 <- list(1,2,3)  
list3 <- c(list1,list2)  
list3
```


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. In R programming, matrices are two-dimensional, homogeneous data structures.

Specify the **nrow** and **ncol** parameters to get the amount of rows and columns.

By default, matrices are in column-wise order. To order row-wise set `byrow = TRUE`.

Example 1:

```
# Create a matrix
```

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

```
# Print the matrix
```

```
thismatrix
```

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

Matrices

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

Example 2:

```
# Create a matrix
```

```
thismatrix <- matrix(c(1,2,3,4,5,6), nrow = 3, ncol = 2, byrow =  
TRUE)
```

```
# Print the matrix
```

```
thismatrix
```

You can also create a matrix with strings

Example 3:

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow  
= 2)
```

```
thismatrix
```

You can access the items by using [] brackets. The first number "1" in the bracket specifies the row-position, while the second number "2" specifies the column-position.

```
thismatrix[1,2]
```

```
#[1] "cherry"
```

Matrices

The whole row can be accessed if you specify a comma **after** the number in the bracket.

The whole column can be accessed if you specify a comma **before** the number in the bracket.

Example 2:

Create a matrix

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow  
= 2, ncol = 2)
```

```
thismatrix[2,]          # Accessing 2nd row elements of a matrix
```

```
thismatrix[,2]          # Accessing 2nd col elements of a matrix
```

```
thismatrix[c(1,2),]      # Accessing 1st and 2nd row elements of a  
matrix
```

```
thismatrix[,c(1,2)]      # Accessing 1st and 2nd column elements  
of a matrix
```

Use the c() function with negative(–) sign to remove rows and columns in a Matrix

Matrices

The function **rbind()** is used to add additional rows in a Matrix and **cbind()** is used to add additional columns in a Matrix.

Example:

```
matrixA=matrix(c(1,2,3,4,5,6,7,8,9),nrow=3,ncol=3)
```

```
matrixA
```

```
matrixB=rbind(matrixA,c(10,11,12))
```

```
matrixB
```

```
matrixA #No change in matrixA
```

```
matrixC=cbind(matrixA,c(10,11,12))
```

```
matrixC
```

```
matrixD = matrixC[-c(1),-c(1)]
```

```
matrixD #1st row and 1st col deleted
```

	[,1]	[,2]	[,3]
[1,]	5	8	11
[2,]	6	9	12

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

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

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

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

Matrices

Use the `dim()` function to find the number of rows and columns in a Matrix.

Example:

```
matrixA=matrix(c(1,2,3,4,5,6,7,8,9),nrow=3,ncol=3)
```

```
matrixA
```

```
dim(matrixA)
```

```
cat("number of rows = ",nrow(matrixA))
```

```
cat("\nnumber of rows = ",ncol(matrixA))
```

```
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
[1] 3 3
number of rows = 3
number of rows = 3
```

Use the `length()` function to find the number of elements in a Matrix.

Example:

```
length(matrixA)
```

```
#output : 9
```

Matrices

You can use for loop to access elements in a Matrix.

Example:

```
matrixA=matrix(c(1,2,3,4,5,6,7,8,9),nrow=3,ncol=3)
```

```
matrixA
```

```
#accessing matrix elements
```

```
for(rows in 1:nrow(matrixA))
```

```
{
```

```
  for(cols in 1:ncol(matrixA))
```

```
  {
```

```
    print(matrixA[rows,cols])
```

```
  }
```

```
}
```

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

Matrices

You can use `rbind()` and `cbind()` to combine 2 matrices.

Example:

```
matrixA=matrix(c(1,1,1,1,1,1,1,1,1),nrow=3,ncol=3)
```

```
matrixB=matrix(c(2,2,2,2,2,2,2,2,2),nrow=3,ncol=3)
```

```
newmatrixA=rbind(matrixA,matrixB)
```

```
newmatrixA
```

```
newmatrixA=cbind(matrixA,matrixB)
```

```
newmatrixA
```

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

Arrays

Compared to matrices, arrays can have more than two dimensions. We can use the **array()** function to create an array, and the **dim** parameter to specify the dimensions

Example:

An array with one dimension with values ranging from 1 to 24

```
thisarray <- c(1:24)
```

```
thisarray
```

An array with more than one dimension

```
multiarray <- array(thisarray, dim = c(4, 3, 2))
```

```
multiarray
```

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

, , 2
     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

In the above example, The first and second number in the bracket specifies the amount of rows and columns.

The last number in the bracket specifies how many dimensions we want

Arrays

You can access the array elements by referring to the index position.
You can use the [] brackets to access the desired elements from an array.
Syntax: `array[row position, column position, matrix level]`

Example:

`# An array with one dimension with values ranging from 1 to 24`

`thisarray <- c(1:24)`

`multiarray <- array(thisarray, dim = c(4, 3, 2))`

`multiarray[2, 3, 2]`

In the above example, we are trying to access
the element of multiarray located in

Row-2

Col-2

Dimension-2

Dim→2

, , 1			
	[,1]	[,2]	[,3]
[1,]	1	5	9
[2,]	2	6	10
[3,]	3	7	11
[4,]	4	8	12
, , 2			
	[,1]	[,2]	[,3]
[1,]	13	17	21
[2,]	14	18	22
[3,]	15	19	23
[4,]	16	20	24

Arrays

You can also access the whole row or column from a matrix in an array, by using the `c()` function.

Syntax: `array[row position, column position, matrix level]`

Example:

```
thisarray <- c(1:24)
```

Access all the items from the first row from matrix one

```
multiarray <- array(thisarray, dim = c(4, 3, 2))
```

```
multiarray[c(1),,1]    #output: 1 5 9
```

Access all the items from the first column from matrix one

```
multiarray <- array(thisarray, dim = c(4, 3, 2))
```

```
multiarray[,c(1),2]    #output: 13 14 15 16
```

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

, , 2
      [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

Data Frames

Data Frames are data displayed in a format as a table.

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.

Example

Create a data frame

```
Data_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)
```

Print the data frame

```
Data_Frame
```

	Training	Pulse	Duration
1	Strength	100	60
2	Stamina	150	30
3	Other	120	45

Data Frames

- Use the summary() function to summarize the data from a Data Frame.

Example2:

Create a data frame

```
Student <- data.frame (  
  Name = c("John", "Anne", "Joyce","Kavitha"),  
  Age = c(23, 25, 22,26),  
  Rank = c(2, 3, 1,4)  
)
```

Print the data frame

```
print(Student)
```

```
summary(Student)
```

Training Pulse Duration			
1	Strength	100	60
2	Stamina	150	30
3	Other	120	45
Training		Pulse	Duration
Other	:1	Min. :100.0	Min. :30.0
Stamina	:1	1st Qu.:110.0	1st Qu.:37.5
Strength	:1	Median :120.0	Median :45.0
		Mean :123.3	Mean :45.0
		3rd Qu.:135.0	3rd Qu.:52.5
		Max. :150.0	Max. :60.0

Data Frames

- We can use single brackets [], double brackets [[]] or \$ to access columns from a data frame

Example:

Accessing a data frame

```
Student <- data.frame (  
  Name = c("John", "Anne", "Joyce", "Kavitha"),  
  Age = c(23, 25, 22, 26),  
  Rank = c(2, 3, 1, 4)  
)
```

Print the data frame

```
Student[1]
```

```
Student[["Name"]]
```

```
Student$Name
```

```
      Name  
1      John  
2      Anne  
3     Joyce  
4  Kavitha  
[1] John      Anne      Joyce      Kavitha  
Levels: Anne John Joyce Kavitha  
[1] John      Anne      Joyce      Kavitha  
Levels: Anne John Joyce Kavitha
```

Data Frames

- Use the `rbind()` function to add new rows in a Data Frame
- Use the `cbind()` function to add new column in a Data Frame

Example:

```
Student <- data.frame (  
  Name = c("John", "Anne", "Joyce","Kavitha"),  
  Age = c(23, 25, 22,26),  
  Rank = c(2, 3, 1,4)  
)
```

Adding a row in a data frame

```
Student2 <- rbind(Student, c("Peter", 25, 5))  
Student2
```

Adding a column in a data frame

```
Student3 <- cbind(Student,percentage = c(90.5,81.33,95.43,70.57))  
Student3
```

Output

```
Rscript /tmp/5NG4XvDZxB.r  
Name Age Rank  
1 John 23 2  
2 Anne 25 3  
3 Joyce 22 1  
4 Kavitha 26 4  
5 Peter 25 5  
Name Age Rank percentage  
1 John 23 2 90.50  
2 Anne 25 3 81.33  
3 Joyce 22 1 95.43  
4Kavitha 26 4 70.57
```

Data Frames

- Use **dim()** function to find the number of rows and columns in a Data Frame.
- Use **c()** function with Negative index to remove rows and columns in a Data Frame.

Example:

```
Student <- data.frame (  
  Name = c("John", "Anne", "Joyce","Kavitha"),  
  Age = c(23, 25, 22,26),  
  Rank = c(2, 3, 1,4)  
)
```

```
# Removing a row and a column in a data frame  
dim(Student)
```

```
# Removing a row and a column in a data frame  
Student2 <- Student[-c(1),-c(1)]  
Student2
```

```
[1] 4 3  
    Age Rank  
2   25    3  
3   22    1  
4   26    4
```

Data Frames

- You can also use the **ncol()** function to find the number of columns and **nrow()** to find the number of rows.
- Use the **length()** function to find the number of columns in a Data Frame (**similar to ncol()**)

Example:

```
Student <- data.frame (  
  Name = c("John", "Anne", "Joyce","Kavitha"),  
  Age = c(23, 25, 22,26),  
  Rank = c(2, 3, 1,4)  
)
```

finding number of rows and columns

<code>ncol(Student)</code>	<code>#3</code>
<code>nrow(Student)</code>	<code>#3</code>
<code>length(Student)</code>	<code>#3</code>

Data Frames

Use the `rbind()` function to combine two or more data frames in R vertically.

Use the `cbind()` function to combine two or more data frames in R horizontally.

Example:

```
Student1 <- data.frame (  
  Name = c("John", "Anne", "Joyce","Kavitha"),  
  Age = c(23, 25, 22,26),  
  Rank = c(2, 3, 1,4)  
)  
Student2 <- data.frame (  
  Name = c("John1", "Anne1", "Joyce1","Kavitha1"),  
  Age = c(23, 25, 22,26),  
  Rank = c(2, 3, 1,4)  
)  
Student3 <- rbind(Student1, Student2)  
Student3
```

Output

Rscript /tmp/5NG4XvDZxB.r

	Name	Age	Rank
1	John	23	2
2	Anne	25	3
3	Joyce	22	1
4	Kavitha	26	4
5	John1	23	2
6	Anne1	25	3
7	Joyce1	22	1
8	Kavitha1	26	4

#R Program to demonstrate the use of function

```
>subtract_two_nums <- function(x, y)
{
    return (x - y)
}
```

```
> print(subtract_two_nums(3, 1))
```

Output:

```
[1] 2
```

#R Program to demonstrate the use of function

```
>circumference<- function(r)
{
    return 2*pi*r
}
```

```
> print(circumference(5))
```

Output:

```
[1] 31.41593
```

Examples:

#R Program to demonstrate the use of nested functions

```
radius_from_diameter <- function(d)
{
    d/2
}
circumference <- function(r)
{
    2*pi*r
}
print(circumference(radius_from_diameter(4)))
```

Output:

```
[1] 12.56637
```

Examples:

#R Program to demonstrate the use of nested functions

```
sum_circle_ares <- function(r1, r2, r3)
{
  circle_area <- function(r)
  {
    pi*r^2
  }
  circle_area(r1) + circle_area(r2) + circle_area(r3)
}
print(sum_circle_ares(1, 2, 3))
```

Output:

```
[1] 43.9823
```

Examples:

Lazy Evaluation

R functions perform “lazy” evaluation in which arguments are only evaluated if required in the body of the function.

the y argument is not used so not included it causes no harm

```
lazy <- function(x, y){
```

```
  x*2
```

Output: [1] 8

```
}
```

```
lazy(4)
```

however, if both arguments are required in the body an error will result if an argument is missing

```
lazy2 <- function(x, y){
```

```
  (x+y)*2
```

```
}
```

```
lazy2(4)
```

Output: #Error in lazy2(4): argument "y" is missing, with no default

Examples:

#R Program to demonstrate the use of functions that returns multiple values

```
bad <- function(x, y)
```

```
{
```

```
  2*x + y
```

```
  x + 2*y
```

```
  2*x + 2*y
```

```
  x/y
```

```
}
```

```
bad(1, 2)
```

Output:

```
[1] 0.5
```

Examples:

#R Program to demonstrate the use of functions that returns multiple values

```
good <- function(x, y)
{
  output1 <- 2*x + y
  output2 <- x + 2*y
  output3 <- 2*x + 2*y
  output4 <- x/y
  c(output1, output2, output3, output4)
}
good(1, 2)
```

Output:

```
[1] 4.0 5.0 6.0 0.5
```

Examples:

#R Program to demonstrate the use of functions with default argument

Create a function with arguments.

```
new.function <- function(a = 3, b = 6)
{
  result <- a * b
  print(result)
}
```

Call the function without giving any argument.

```
new.function()
```

Call the function with giving new values of the argument.

```
new.function(9,5)
```

Output:

```
18      45
```


Factors

Factors are used to categorize data. Examples of factors are:

- Demography: Male/Female, Music: Rock, Pop, Classic, Jazz Training: Strength, Stamina

use the **factor()** function and add a vector as argument

```
music_cat <- factor(c("Jazz", "Rock", "Classic",  
"Classic", "Pop", "Jazz", "Rock", "Jazz"))
```

```
music_cat #levels(music_cat), length()
```

```
music_cat[3] # Access Factor
```

```
music_cat[3] <- "Pop" # Change the Item value
```

```
gender <- c("Male","Female","Female","Male","Female")
```

```
gender.factor <- factor(gender)
```

```
gender.factor
```

```
levels(gender.factor)
```

```
length(gender.factor)
```

```
gender.factor[3] <- "Tansgender"
```