R Basic Syntax

Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/R

Data Types Overview

- Generally, while doing programming in any programming language, need to use various variables to store various information.
- Variables are nothing but reserved memory locations to store values.
- This means that, when you create a variable you reserve some space in memory.
- You may like to store information of various data types like character, wide character, integer, floating point, double floating point, Boolean etc.
- Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.
- In contrast to other programming languages like C and java in R, the variables are not declared as some data type.

Data Types Overview (Cont.)

- **The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable.**
- **There are many types of R-objects.**
- **The frequently used ones are:**
  - Vectors
  - Lists
  - Matrices
  - Arrays
  - Factors
  - Data Frames

Data Types Overview (Cont.)

- **The simplest of these objects is the vector object.**
- **There are six data types of these atomic vectors, also termed as six classes of vectors.**
- **The other R-Objects are built upon the atomic vectors.**
  - Logical
  - Numeric
  - Integer
  - Complex
  - Character
  - Raw

Data Types Overview (Cont.)

Data Type	Example	Verify

Logical	TRUE, FALSE	`> v <- TRUE`
		`> print(class(v))`
		`[1] "logical"`

Numeric	12.3, 5, 999	`> v <- 23.5`
		`> print(class(v))`
		`[1] "numeric"`

Integer	2L, 34L, 0L	`> v <- 2L`
		`> print(class(v))`
		`[1] "integer"`

Data Types Overview (Cont.)

Data Type	Example	Verify

Complex	3 + 2i	`> v <- 2 + 5i`
		`> print(class(v))`
		`[1] "complex"`

Character	‘a’, “good”, “TRUE”, ‘23.4’	`> v <- "TRUE"`
		`> print(class(v))`
		`[1] "character"`

Raw	“Hello” is stored as 48 65 6c 6c 6f	`> v <- charToRaw("Hello")`
		`> print(class(v))`
		`[1] "raw"`

Vectors

- When you want to create vector with more than one element, you should use `c()` function which means to combine the elements into a vector.

```
Console C:/R Home/
> # Create a vector.
> apple <- c('red', 'green', 'yellow')
> print(apple)
[1] "red"    "green"  "yellow"
>
> # Get the class of the vector.
> print(class(apple))
[1] "character"
```

Lists

- A list is an R-object which can contain many different types of elements inside it like vectors, functions and even another list inside it.

```
Console C:/R Home/
> # Create a list.
> list1 <- list(c(2,5,3), 21.3, sin)
>
> # Print the list.
> print(list1)
[[1]]
[1] 2 5 3

[[2]]
[1] 21.3

[[3]]
function (x)  .Primitive("sin")
```

Matrices

- A matrix is a two-dimensional rectangular data set.
- It can be created using a vector input to the matrix function.

Console C:/R Home/

```
> # Create a matrix.
>
> M = matrix(c('a','a','b','c','b','a'), nrow=2, ncol=3, byrow=TRUE)
> print(M)
     [,1] [,2] [,3]
[1,] "a"  "a"  "b"
[2,] "c"  "b"  "a"
>
```

Arrays

- While matrices are confined to two dimensions, arrays can be of any number of dimensions.
- The **`array`** function takes a **`dim`** attribute which creates the required number of dimension.
- In the below example we create an array with two elements which are 3x3 matrices each.

```
Console C:/R Home/
> # Create an array.
> a <- array(c('green', 'yellow'), dim = c(3,3,2))
> print(a)
, , 1

     [,1]     [,2]     [,3]
[1,] "green"  "yellow" "green"
[2,] "yellow" "green"  "yellow"
[3,] "green"  "yellow" "green"

, , 2

     [,1]     [,2]     [,3]
[1,] "yellow" "green"  "yellow"
[2,] "green"  "yellow" "green"
[3,] "yellow" "green"  "yellow"
```

Factors

- Are the r-objects which are created using a vector.
- It stores the vector along with the distinct values of the elements in the vector as labels.
- The labels are always character irrespective of whether it is numeric or character or Boolean etc. in the input vector.
- They are useful in statistical modeling.

Factors (Cont.)

- Are created using the `factor()` function.
- The `nlevels` functions gives the count of levels.

```
Console C:/R Home/
> # Create a vector.
> apple_colors <- c('green', 'green', 'yellow', 'red', 'red', 'red', 'green')
>
> # Create a factor object.
> factor_apple <- factor(apple_colors)
>
> # Print the factor.
> print(factor_apple)
[1] green  green  yellow red    red    red    green
Levels: green red yellow
> print(nlevels(factor_apple))
[1] 3
```

Data Frames

- Are tabular data objects.
- Unlike a matrix in data frame each column can contain different modes of data.
- The first column can be numeric while the second column can be character and third column can be logical.
- It is a list of vectors of equal length.

Data Frames (Cont.)

- Are created using the `data.frame()` function.

```
Console C:/R Home/
> # Create the data frame.
>
> BMI <- data.frame(
+     gender = c('Male', 'Male', 'Female'),
+     height = c(152, 171.5, 165),
+     weight = c(81, 93, 26),
+     Age = c(42, 38, 26)
+ )
>
> print(BMI)
  gender height weight Age
1   Male  152.0     81  42
2   Male  171.5     93  38
3 Female  165.0     26  26
```

Variables

- Provides us with named storage that our programs can manipulate.
- In R can store an atomic vector, group of atomic vectors or a combination of many `R Objects`.
- A valid variable name consists of letters, numbers and the dot or underline characters.
- The variable name starts with a letter or the dot not followed by a number.

Variables (Cont.)

Variable Name	Validity	Reason
var_name2.	Valid	Has letters, numbers, dot and underscore
var_name%	Invalid	Has the character '%'. Only dot(.) and underscore allowed.
2var_name	Invalid	Starts with a number
.var_name, var.name	Valid	Can start with a dot(.) but the dot(.) should not be followed by a number.
.2var_name	Invalid	The starting dot is followed by a number making it invalid.
_var_name	Invalid	Starts with _ which is not valid

Variable Assignment

- The variables can be assigned values using leftward(`<-`), rightward(`->`) and equal(`=`) to operator.
- The values of the variables can be printed using `print()` or `cat()` function.
- The `cat()` function combines multiple items into a continuous print output.

Variable Assignment

Console C:/R Home/

```
> # Assignment using equal operator.
> var.1 = c(0, 1, 2, 3)
>
> # Assignment using leftward operator.
> var.2 <- c("Learn", "R")
>
> # Assignment using rightward operator.
> c(TRUE, 1) -> var.3
>
> print(var.1)
[1] 0 1 2 3
> cat("var.1 is ", var.1, "\n")
var.1 is  0 1 2 3
> cat("var.2 is ", var.2, "\n")
var.2 is  Learn R
> cat("var.3 is ", var.3, "\n")
var.3 is  1 1
```

Data Type of a Variable

- In R, a variable itself is not declared of any data type.
- So R is called a dynamically typed language, which means that we can change a variable's data type of the same variable again and again when using it in a program.

```
Console C:/R Home/
> var_x <- "Hello"
> cat("The class of var_x is ", class(var_x), "\n")
The class of var_x is  character
>
> var_x <- 34.5
> cat("Now the class of var_x is ", class(var_x), "\n")
Now the class of var_x is  numeric
>
> var_x <- 27L
> cat("Next the class of var_x becomes ", class(var_x), "\n")
Next the class of var_x becomes  integer
>
```

Finding Variables

- To know all the variables currently available in the workspace we use the `ls()` function.
- Also the `ls()` function can use patterns to match the variable names.

```
Console C:/R Home/
> print(ls())
 [1] "a"           "apple"         "apple_colors" "BMI"
 [5] "f"           "factor_apple"  "list1"        "M"
 [9] "myString"    "oddcount"      "v"            "var.1"
[13] "var.2"       "var.3"         "var_x"        "y"
[17] "z"
```

Finding Variables (Cont.)

- The `ls()` function can use patterns to match the variable names.

```
> print(ls(pattern="var"))
[1] "var.1" "var.2" "var.3" "var_x"
>
```

- The variables starting with `dot(.)` are hidden, they can be listed using `"all.names = TRUE"` argument to `ls()` function.

```
> print(ls(all.name = TRUE))
 [1] ".Random.seed" "a"            "apple"        "apple_colors"
 [5] "BMI"          "f"            "factor_apple" "list1"
 [9] "M"            "myString"     "oddcount"     "v"
[13] "var.1"        "var.2"        "var.3"        "var_x"
[17] "y"            "z"
>
```

Deleting Variables

- Variables can be deleted by using the `rm()` function.
- Below we delete the variable `var.3`.
- On printing the value of the variable error is thrown.

```
Console C:/R Home/
> rm(var.3)
> print(var.3)
Error in print(var.3) : object 'var.3' not found
>
```

Deleting Variables

- All the variables can be deleted by using the **`rm()`** and **`ls()`** function together.

```
> rm(list = ls())
> print(ls())
character(0)
>
```

Operators

Operators

- **Is a symbol that tells the compiler to perform specific mathematical or logical manipulations.**
- **R language is rich in built-in operators and provides following types of operators.**
- **Types of Operators**
  - Arithmetic Operators
  - Relational Operators
  - Logical Operators
  - Assignment Operators
  - Miscellaneous Operators

Arithmetic Operators

Operators	Operation	Example
+	Addition	15 + 5 = 20
–	Subtraction	15 – 5 = 10
*	Multiplication	15 * 5 = 75
/	Division	15 /5 = 3
%/%	Integer Division – Same as Division but it will return the integer value by flooring the extra decimals	16 %/% 3 = 5. If you divide 16 with 3 you get 5.333 but the Integer division operator will trim the decimal values and outputs the integer.
^	Exponent – It returns the Power of One variable against the other	15 ^ 3 = 3375 (It means 15 Power 3 or 15^3).
%%	Modulus – It returns the remainder after the division	15 %% 5 = 0 (Here remainder is zero). If it is 17 %% 4 then it will be 1.

- **All these operators are binary operators.**

Lab : Arithmetic Operators

Console C:/R Home/

```
> a <- 16
> b <- 3
> add <- a + b
> sub = a - b
> multi = a * b
> division = a / b
>
> print(paste("Addition : ", add))
[1] "Addition :  19"
> print(paste("Subtraction : ", sub))
[1] "Subtraction :  13"
> print(paste("Multiplication : ", multi))
[1] "Multiplication :  48"
> print(paste("Division : ", division))
[1] "Division :  5.3333333333333"
>
```

Lab : Arithmetic Operators

```
Console C:/R Home/
> a <- 16
> b <- 3
> Integer_Division = a %/% b
> exponent = a ^ b
> modulus = a %% b
>
> print(paste("Integer Division : ", Integer_Division))
[1] "Integer Division :  5"
> print(paste("Exponent : ", exponent))
[1] "Exponent :  4096"
> print(paste("Modulus : ", modulus))
[1] "Modulus :  1"
>
```

Relational Operators

Operators	Usage	Description	Example
>	i > j	i is greater than j	25 > 14 returns True
<	i < j	i is less than j	25 < 14 returns False
>=	i >= j	i is greater than or equal to j	25 >= 14 returns True
<=	i <= j	i is less than or equal to j	25 <= 14 return False
==	i == j	i is equal to j	25 == 14 returns False
!=	i != j	i is not equal to j	25 != 14 returns True

- **Are mostly used either in If Conditions or Loops.**
- **Are commonly used to check the relationship between two variables.**
  - If the relation is true then it will return Boolean **`True`**.
  - If the relation is false then it will return Boolean **`False`**.

Lab : Relational Operators

Console C:/R Home/

```
> # Example for Comparision in R Programming
> a <- 15
> b <- 12
>
> print(paste("Output of 15 > 12 is : ", a > b))
[1] "Output of 15 > 12 is :  TRUE"
> print(paste("Output of 15 < 12 is : ", a < b))
[1] "Output of 15 < 12 is :  FALSE"
> print(paste("Output of 15 >= 12 is : ", a >= b))
[1] "Output of 15 >= 12 is :  TRUE"
> print(paste("Output of 15 <= 12 is : ", a <= b))
[1] "Output of 15 <= 12 is :  FALSE"
> print(paste("Output of 15 Equal to 12 is : ", a == b))
[1] "Output of 15 Equal to 12 is :  FALSE"
> print(paste("Output of 15 Not Equal to 12 is : ", a != b))
[1] "Output of 15 Not Equal to 12 is :  TRUE"
>
```

Logical Operators

Operators	Name	Description	Example
&	Logical AND	It will return true when both conditions are true	c(20, 30) & c(30, 10)
&&	Logical AND	Same as above but, It will work on single element	If (age > 18 && age <= 25)
|	Logical OR	It will returns true when at-least one of the condition is true	c(20, 30) | c(30, 10)
||	Logical OR	Same as above but, It will work on single element	If (age == 35 || age < 60)
!	Logical NOT	If the condition is true, logical NOT operator returns as false	If age = 18 then !(age = 18) returns false.

- **Are used to combine two or more conditions, and perform the logical operations using & (Logical AND), | (Logical OR) and ! (Logical NOT).**

Logical AND Truth table

Condition 1	Condition 2	Condition 1 && Condition 2
TRUE	TRUE	**TRUE**
TRUE	FALSE	**FALSE**
FALSE	TRUE	**FALSE**
FALSE	FALSE	**FALSE**

Logical OR Truth table

Condition 1	Condition 2	Condition 1 || Condition 2
TRUE	TRUE	**TRUE**
TRUE	FALSE	**TRUE**
FALSE	TRUE	**TRUE**
FALSE	FALSE	**FALSE**

Lab : Comparison Operators

Console C:/R Home/

```
> # Logical Operators in R example
>
> age <- 16
> if(!(age > 18)){
+     print("You are too young.")
+ } else if(age > 18 && age <= 35){
+     print("Young guy")
+ } else if(age == 36 || age <= 60){
+     print("You are Middle Age person")
+ } else {
+     print("You are too old.")
+ }
[1] "You are too young."
>
```

Lab : Comparison Operators

```
Console C:/R Home/
> num1 <- c(TRUE, FALSE, 0, 23)
> num2 <- c(FALSE, FALSE, TRUE, TRUE)
>
> num1 & num2
[1] FALSE FALSE FALSE  TRUE
>
> num1 && num2
[1] FALSE
>
> num1 | num2
[1]  TRUE FALSE  TRUE  TRUE
>
> num1 || num2
[1] TRUE
>
> !num1
[1] FALSE  TRUE  TRUE FALSE
>
> !num2
[1]  TRUE  TRUE FALSE FALSE
```

Assignment Operators

Operator	Description	Example
<- or = or <<-	Called Left Assignment	> v1 <- c(3, 1, TRUE, 2 + 3i) > v2 <<- c(3, 1, TRUE, 2 + 3i) > v3 = c(3, 1, TRUE, 2 + 3i) > > print(v1) [1] 3+0i 1+0i 1+0i 2+3i > print(v2) [1] 3+0i 1+0i 1+0i 2+3i > print(v3) [1] 3+0i 1+0i 1+0i 2+3i
-> or ->>	Called Right Assignment	> c(3, 1, TRUE, 2 + 3i) -> v1 > c(3, 1, TRUE, 2 + 3i) ->> v2 > > print(v1) [1] 3+0i 1+0i 1+0i 2+3i > print(v2) [1] 3+0i 1+0i 1+0i 2+3i

Miscellaneous Operators

Operator	Description	Example

:	-Colon operator.
	-It creates the series of numbers in sequence for a vector.
		`> v <- 2:8`
		`> print(v)`
		`[1] 2 3 4 5 6 7 8`

%in%	This operator is used to identify if an element belongs to a vector.	`> v1 <- 8`
		`> v2 <- 12`
		`> t <- 1:10`
		`> print(v1 %in% t)`
		`[1] TRUE`
		`> print(v2 %in% t)`
		`[1] FALSE`

%*%	This operator is used to multiply a matrix with its transpose.	

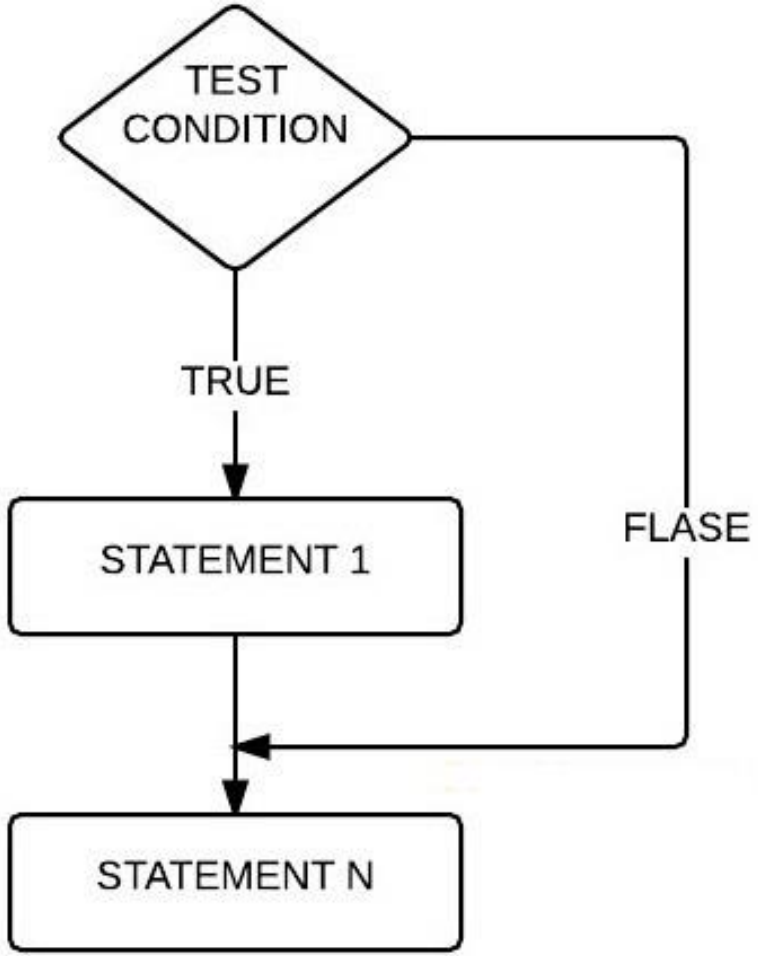
```
> M = matrix(c(2,6,5,1,10,4), nrow=2, ncol=3, byrow=TRUE)
> t = M %*% t(M)
> print(t)
     [,1] [,2]
[1,]   65   82
[2,]   82  117
```

Condition Statements

If Statement and Flow Chart

- Consists of a Boolean expression followed by one or more statements.
- Syntax

```
if(Boolean expression){
        statement 1
        statement 2
}
```



Lab : If Statement

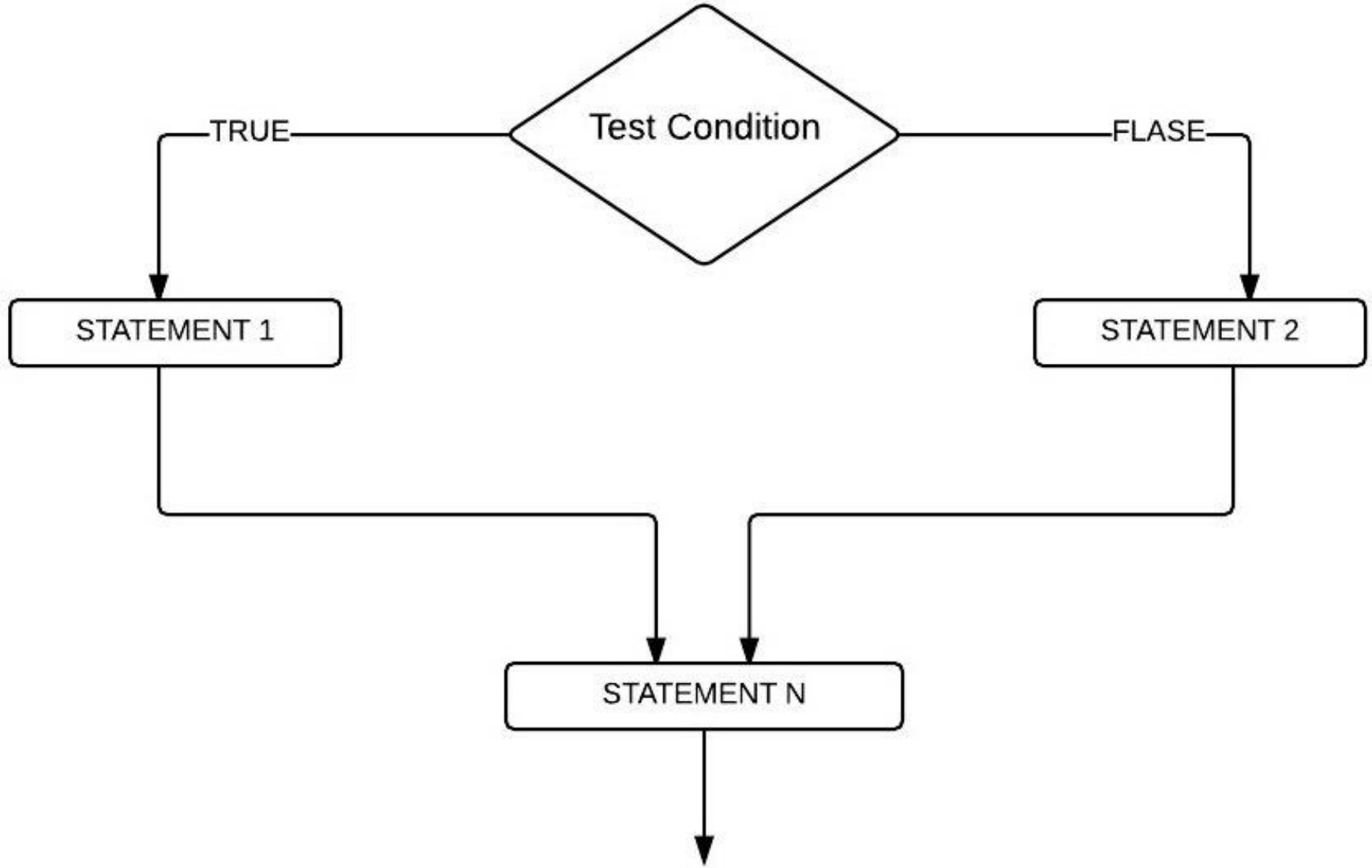
```
1  # R IF Statement Example
2
3  number <- as.integer(readline(prompt="Please Enter any integer Value: "))
4
5  if (number > 1) {
6    print("You have entered POSITIVE Number")
7  }
8
```

If Else Statement

- An `if` statement can be followed by an optional `else` statement which executes when the boolean expression is `false`.
- Syntax

```
if(Boolean expression) {
   True statements
} else {
   False statements
}
```

If Else Statement Flow Chart



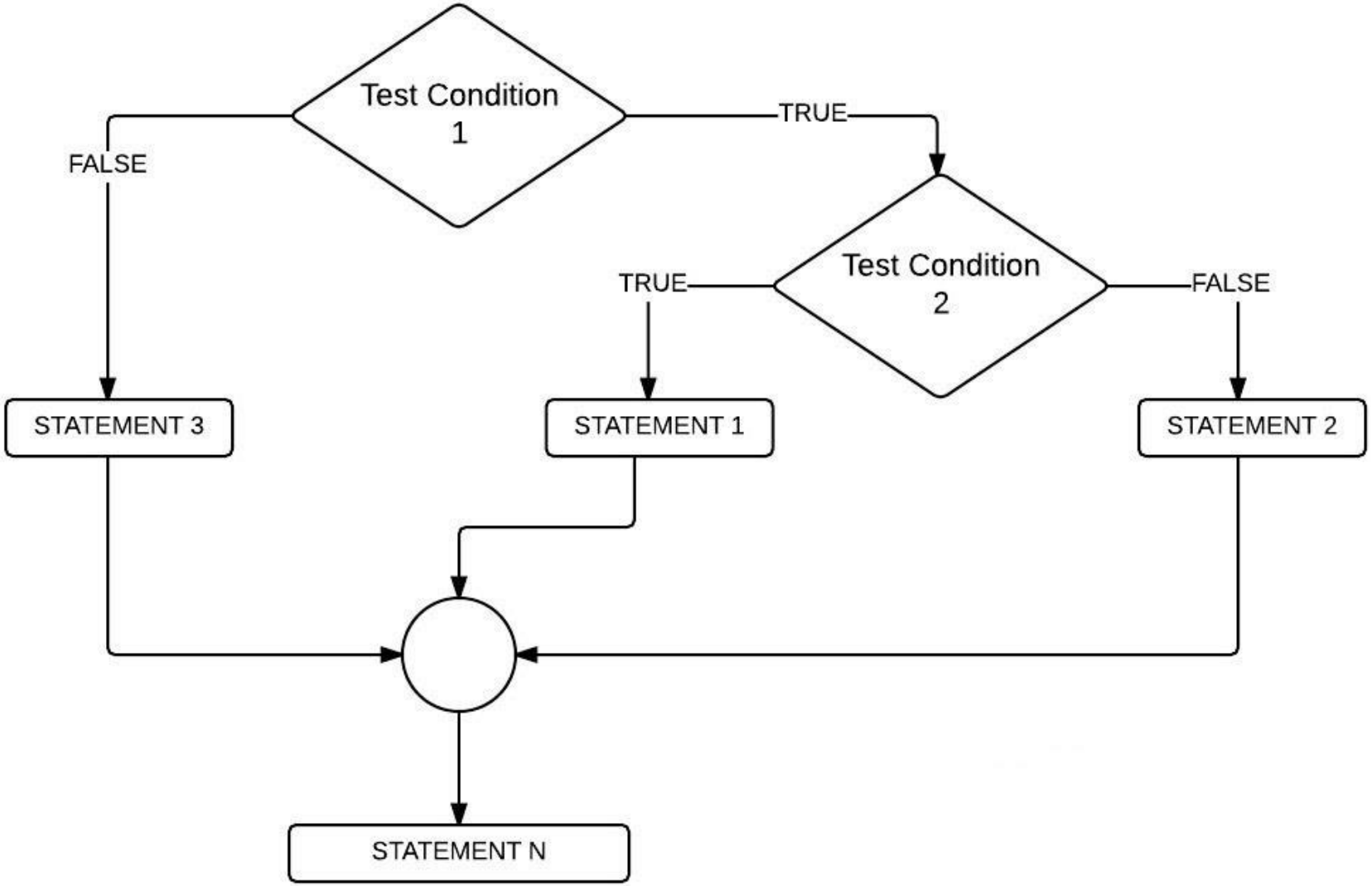
Lab : If Else Statement

```r
# R IF Else Statement Example

my.age <- as.integer(readline(prompt="Please Enter your Age: "))

if (my.age > 18) {
  print("You are eligible to Vote.")
  print("Don't forget to carry Your Voter ID's to Polling booth.")
} else {
  print("You are NOT eligible to Vote.")
  print("We are Sorry")
}
print("This Message is from Outside the IF ELSE STATEMENT")
```

Nested If Else in R

- Placing one If Statement inside another If Statement is called as Nested If Else in R Programming.

Nested If Else in R Flow Chart



Lab : Nested If Else in R

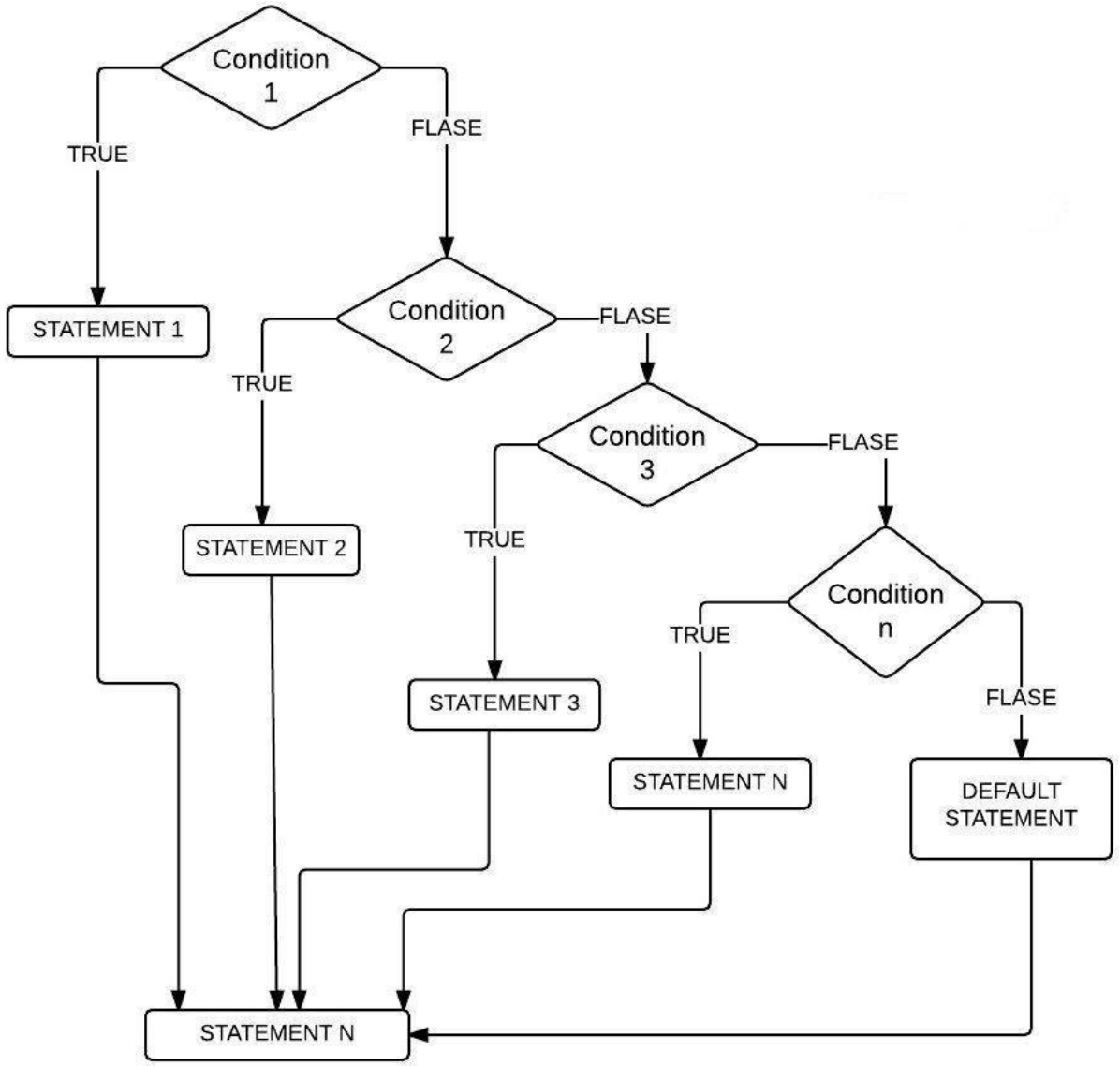
```r
# Nested IF Else in R Programming Example

my.age <- as.integer(readline(prompt="Please Enter your Age: "))

if (my.age < 18) {
  print("You are Not a Major.")
  print("You are Not Eligible to Work")
} else {
  if (my.age >= 18 && my.age <= 60 )  {
    print("You are Eligible to Work")
    print("Please fill the Application Form and Email to us")
  } else {
    print("As per the Government Rules, You are too Old to Work")
    print("Please Collect your pension!")
  }
}
print("This Message is from Outside the Nested IF Else Statement")
```

The If...Else If...Else Statement

- An `if` statement can be followed by an optional `else if...else` statement, which is very useful to test various conditions using single `if...else if` statement.
- Syntax

```
if (Boolean_Expression 1)  {
      Statement 1
} else if (Boolean_Expression 2)  {
      Statement 2
.........
} else if (Boolean_Expression N)  {
      Statement N
} else  {
      Default statements
}
```

Flow Chart of a Else If Statement



Lab : Else If Statement

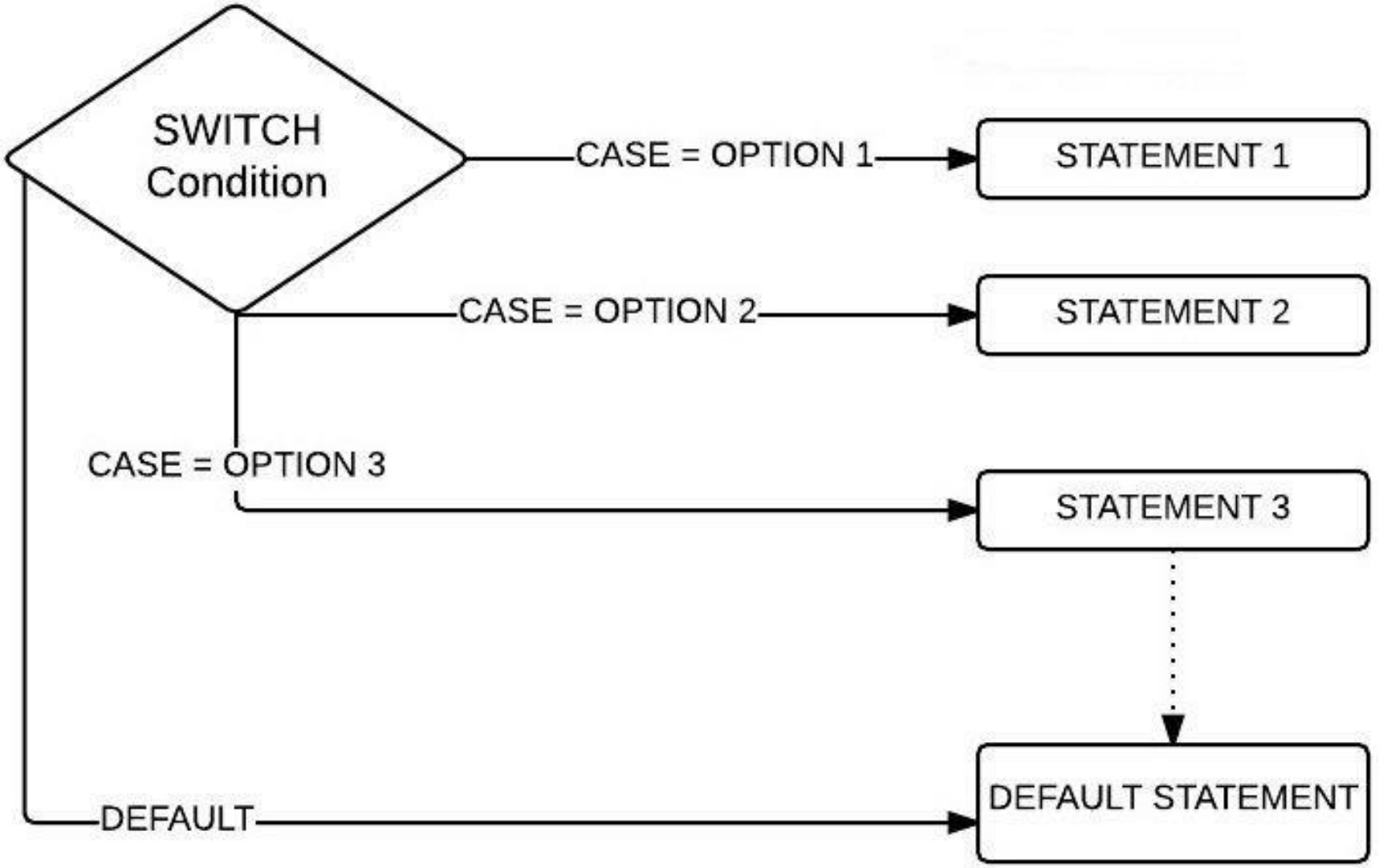
```r
# R Else If Statement Example

my.marks <- as.integer(readline(prompt="Please Enter your Total Marks: "))

if (my.marks >= 550) {
  print("Congratulations!!")
  print("You are eligible for Full Scholarship")
} else if (my.marks >= 490)  {
  print("Congratulations!!")
  print("You are eligible for 50% Scholarship")
} else if (my.marks >= 400)  {
  print("Congratulations!!")
  print("You are eligible for 10% Scholarship")
} else  {
  print("You are NOT eligible for Scholarship")
  print("We are really Sorry for You")
}
```

Switch Statement

- Allows a variable to be tested for equality against a list of values.
- Each value is called a `case`, and the variable being switched on is checked for each `case`.
- Syntax

```
switch (Expression, case1, case2, case3, …)
```

Switch Statement Flow Chart



Lab : Switch Statement

```r
# R Switch Case Example

switch(3,
       "Learn",
       "R Programming",
       "Tutorial",
       "Gateway"
)
```

Lab : Switch Statement

```
1  number1 <- 30
2  number2 <- 20
3  operator <- readline(prompt="Please enter any ARITHMETIC OPERATOR You wish!: ")
4
5  switch(operator,
6         "+" = print(paste("Addition of two numbers is: ", number1 + number2)),
7         "-" = print(paste("Subtraction of two numbers is: ", number1 - number2))
          ,
8         "*" = print(paste("Multiplication of two numbers is: ", number1 *
          number2)),
9         "^" = print(paste("Exponent of two numbers is: ", number1 ^ number2)),
10        "/" = print(paste("Division of two numbers is: ", number1 / number2)),
11        "%/%" = print(paste("Integer Division of two numbers is: ", number1 %/%
          number2)),
12        "%%" = print(paste("Division of two numbers is: ", number1 %% number2))
13 )
```

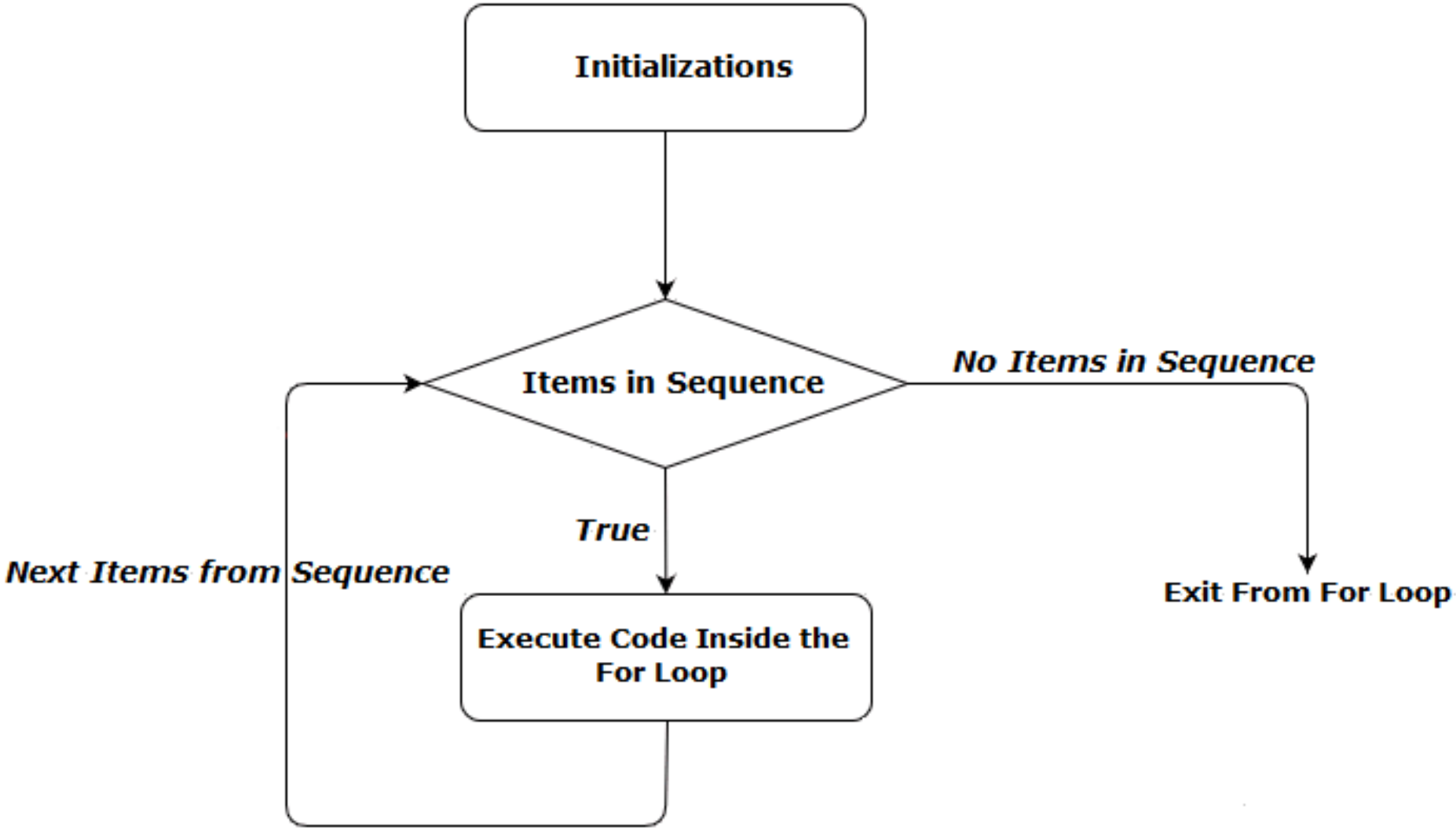
Loop Statements

For Loop

- Is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.
- Syntax

```
for (val in vector)  {
      Statement 1
      Statement 2
      ……
      Statement N
}
```

For Loop Flow Chart



Lab : For Loop

```
1  # For Loop in R Programming Example
2
3  countries <- c('India', 'U K', 'Japan', 'U S A', 'Australia', 'China')
4
5  for (str in countries) {
6    print(paste("Countries are:  ", str))
7  }
8  print("----This is Coming from Outside the For Loop---")
9
```

Lab : For Loop

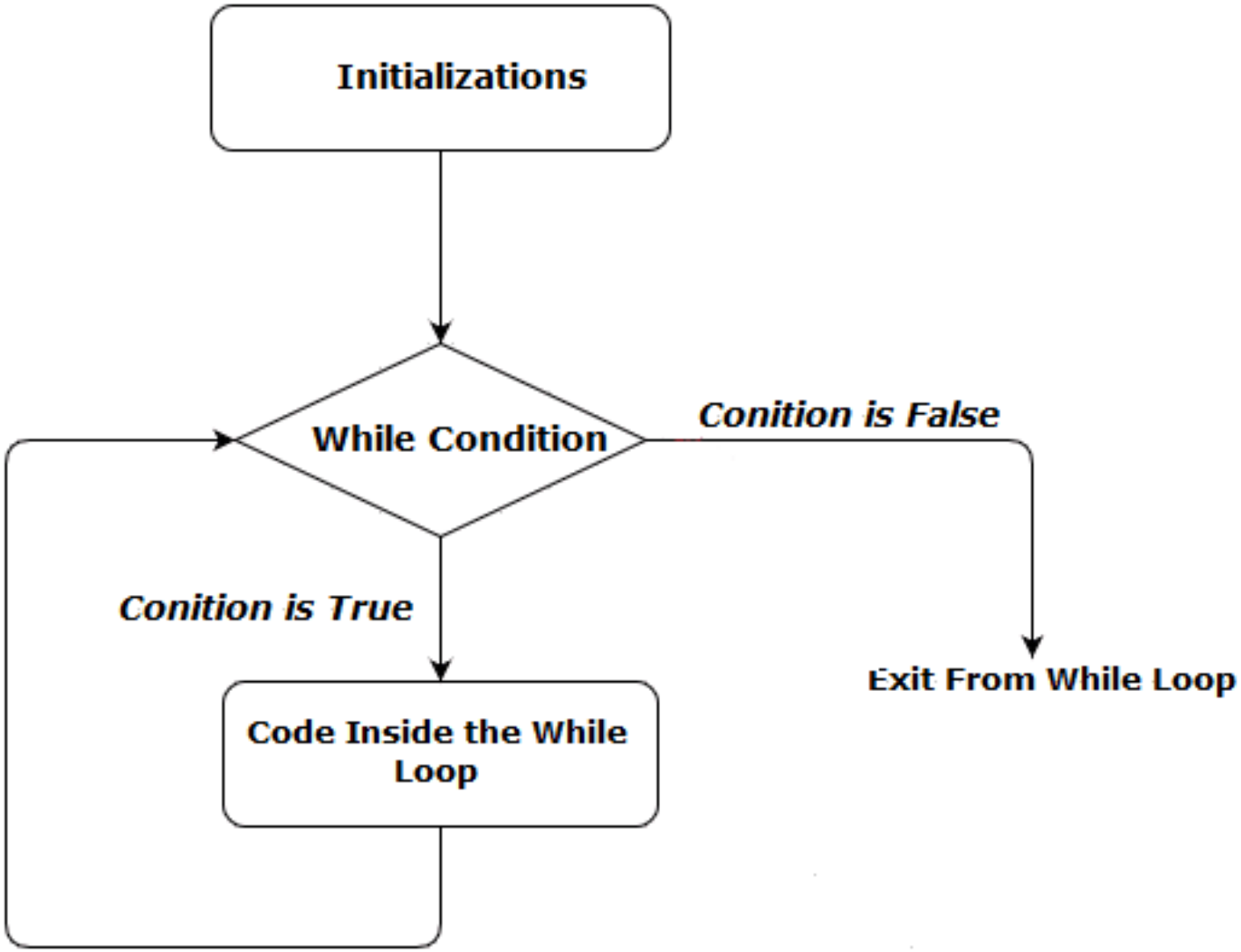
```r
# R For Loop Example

numbers <- c(1:10)

for (num in numbers) {
  print(9 * num)
}
print("---- This is Coming from Outside the For Loop ---")

```

While Loop in R

- The While loop executes the same code again and again until a stop condition is met.
- Syntax

```
while ( Expression )  {
     statement 1
     statement 2
     ……….
     statement N;
     # Increment or Decrements the Values
}
#This statement is from Outside the While Loop
```

Flow Chart of a While loop in R



Lab : While Loop

```r
# R While Loop Example

total = 0
number <- as.integer(readline(prompt="Please Enter any integer Value below 10:
 "))

while (number <= 10)  {
  total = total + number
  number = number + 1
}

print(paste("The total Sum of Numbers From the While Loop is:  ", total))
```

Lab : While Loop

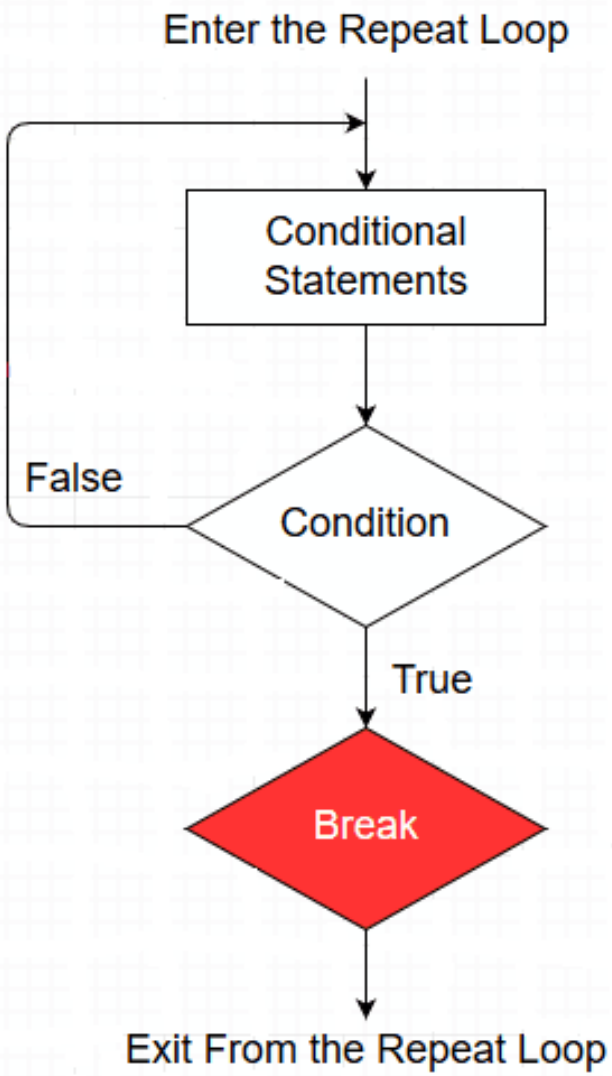
```r
# Infinite While Loop in R Programming Example

number <- 1

while (number < 10)  {
  print(paste("Number from the While Loop is:  ", number))
  number = number + 1
}
```

Repeat Loop

- The **`Repeat loop`** executes the same code again and again until a stop condition is met.
- Syntax

```
repeat  {
     statement 1
     statement 2
     ……….
     statement N
}
```

Repeat Flow Chart



Lab : Repeat

```r
# R Repeat Loop Example

total <- 0
number <- as.integer(readline(prompt="Please Enter any integer Value below 10:  "))

repeat  {
  total = total + number
  number = number + 1
  if (number > 10)  {
    break
  }
}

print(paste("The total Sum of Numbers From the Repeat Loop is:  ", total))
```

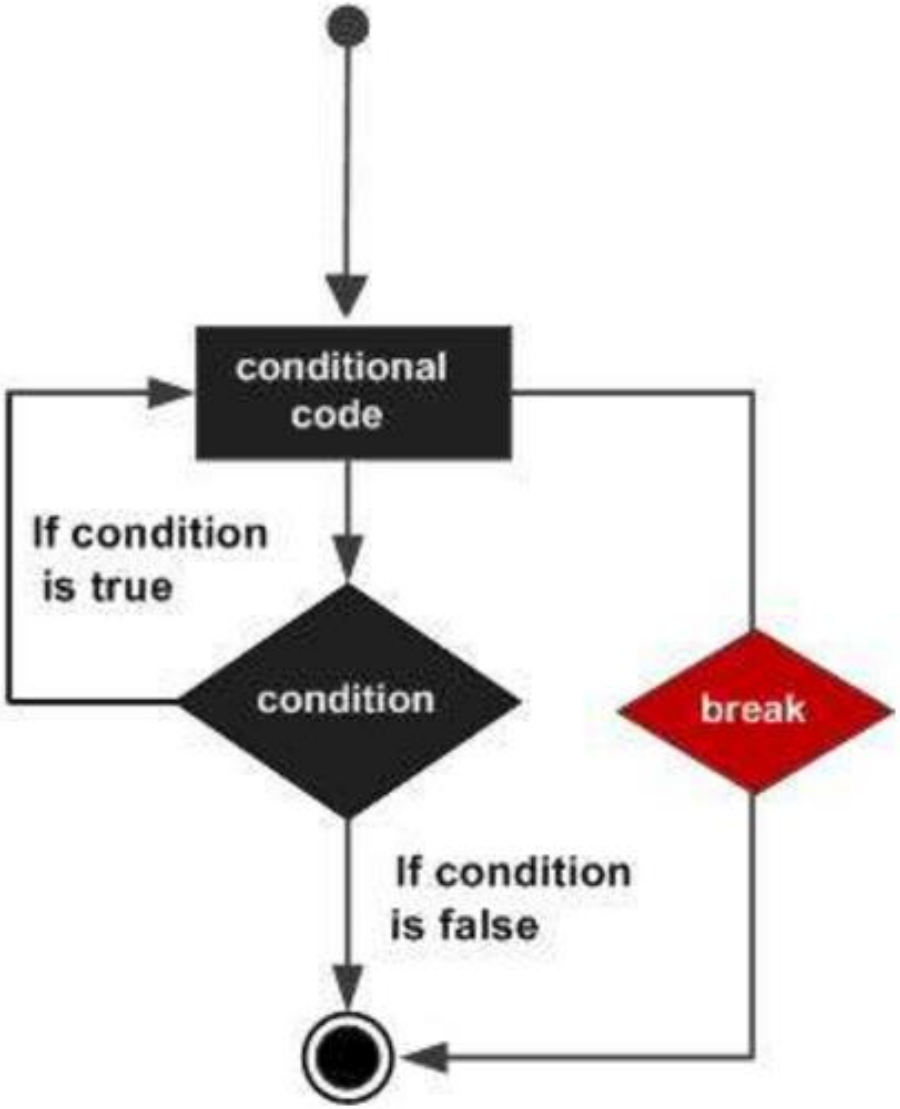
Loop Control Statements

Break Statement

- Is very useful to exit from any loop such as For Loop, While Loop and Repeat Loop.
- While executing these loops, if compiler finds the break statement inside them, it will stop executing the statements and immediately exit from the loop.
- Syntax

```
break
```

Break Flow Chart



Lab : For Loop Break Statement

```
1   # R Break Statement Example
2
3   number <- 1:10
4
5   for (val in number)  {
6     if (val == 7)  {
7       print(paste("Coming out from for loop Where i =  ", val))
8       break
9     }
10    print(paste("Values are :  ", val))
11  }
12
```

Lab : While Loop Break Statement

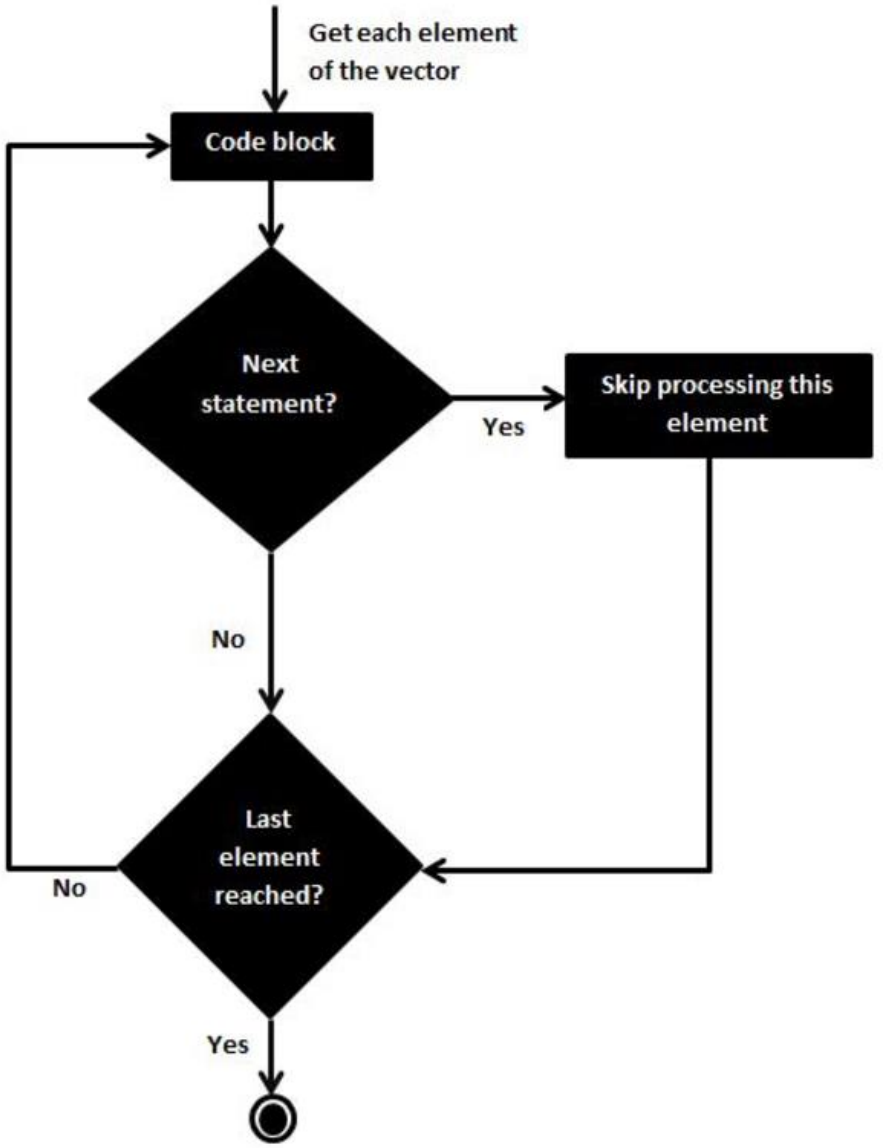
```r
# R Break Statement Example

number <- 10

while (number > 0)  {
  if (number == 3)  {
    print(paste("Coming out from While loop Where number =  ", number))
    break
  }
  print(paste("Values are :  ", number))
  number = number - 1
}
```

Next Statement

- Is one of the most useful statement to controls the flow of R loops.
- We generally use this statement inside the For Loop and While Loop.
- While executing these loops, if compiler find the Next statement inside them, it will stop the current loop iteration and starts the new iteration from the beginning.
- Syntax

```
next
```

Next Flow Chart



Lab : Next Statement in For Loop

```r
# R Next Statement Example

number <- 1:20

for (val in number) {
  if (val %% 2 != 0)  {
    print(paste("ODD Number =  ", val, "(Skipped by Next Statement)"))
    next
  }
  print(paste("EVEN Number =  ", val))
}
```

Lab : Next Statement in R While Loop Example

```r
# R Next Statement Example

number <- 0

while (number <= 10)  {
  if (number == 4 || number == 7)  {
    print(paste("Skipped by the Next Statement =  ", number))
    number = number + 1
    next
  }
  print(paste("Values are :  ", number))
  number = number + 1
}
```

Functions

Functions

- Is a set of statements organized together to perform a specific task.
- R has a large number of in-built functions and the user can create their own functions.
- In R, a function is an object.
- The R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.
- The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

Function Definition

- Is created by using the keyword `function`.
- Syntax

```
function_name <- function(arg_1, arg_2, ...){
        Function body
}
```

Function Components

- **Function Name**
  - This is the actual name of the function.
  - It is stored in R environment as an object with this name.
- **Arguments**
  - An argument is a placeholder.
  - When a function is invoked, you pass a value to the argument.
  - Arguments are optional.
  - Function may contain no arguments.
  - Also arguments can have default values.

Function Components (Cont.)

- **Function Body**
  - The function body contains a collection of statements that defines what the function does.
- **Return Value**
  - The return value of a function is the last expression in the function body to be evaluated.

Built-in Function

- Simple examples of in-built functions are `seq()`, `mean()`, `max()`, `sum(x)` and `paste(...)` etc.
- They are directly called by user written programs.
- You can refer most widely used R functions.

```
Console C:/R Home/
> # Create a sequence of numbers from 32 to 44.
> print(seq(32, 44))
 [1] 32 33 34 35 36 37 38 39 40 41 42 43 44
>
> # Find mean of numbers from 25 to 82.
> print(mean(25:82))
[1] 53.5
>
> # Find sum of numbers from 41 to 68.
> print(sum(41:68))
[1] 1526
```

User-defined Function

- We can create user-defined functions in R.
- They are specific to what a user wants and once created they can be used like the built-in functions.
- Below is an example of how a function is created and used.

Console C:/R Home/

```
> # Create a function to print squares of numbers in sequence.
> new.function <- function(a){
+     for(i in 1 : a){
+         b <- i ^ 2
+         print(b)
+     }
+ }
```

Calling a Function

```
Console C:/R Home/
> # Create a function to print squares of numbers in sequence.
> new.function <- function(a){
+     for(i in 1 : a){
+         b <- i ^ 2
+         print(b)
+     }
+ }
>
> # Call the function new.function supplying 6 as an argument.
> new.function(6)
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

Calling a Function without an Argument

```
Console C:/R Home/
> # Create a function without an argument.
> new.function <- function(){
+     for(i in 1:5){
+         print(i ^ 2)
+     }
+ }
>
>
> # Call the function without supplying an argument.
> new.function()
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
>
```

Calling a Function with Argument Values (by position and by name)

- The arguments to a function call can be supplied in the same sequence as defined in the function.
- Or
- The arguments can be supplied in a different sequence but assigned to the names of the arguments.

```
> # Create a function with arguments.
> new.function <- function(a, b, c){
+     result <- a * b + c
+     print(result)
+ }
> # Call the function by position of arguments.
> new.function(5, 3, 11)
[1] 26
> # Call the function by names of the arguments.
> new.function(a = 11, b = 5, c = 3)
[1] 58
```

Calling a Function with Default Argument

- Can define the value of the arguments in the function definition and call the function without supplying any argument.
- Can also call such functions by supplying new values of the argument and get non default result.

```
> # 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()
[1] 18
>
> # Call the function with giving new values of the argument.
> new.function(9, 5)
[1] 45
```

Lazy Evaluation of Function

- **Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.**

```
> # Create a function with arguments.
> new.function <- function(a, b){
+     print(a ^ 2)
+     print(a)
+     print(b)
+ }
>
> # Evaluate the function without supplying one of the arguments.
> new.function(6)
[1] 36
[1] 6
Error in print(b) : argument "b" is missing, with no default
>
```

Strings

- Any value written within a pair of single quote(‘ ’) or double quotes(“ ”) in R is treated as a string.
- Internally R stores every string within double quotes, even when you create them with single quote.

Rules Applied in String Construction

- **The quotes at the beginning and end of a string should be both double quotes or both single quote.**
  - They can not be mixed.
- **Double quotes can be inserted into a string starting and ending with single quote.**
- **Single quote can be inserted into a string starting and ending with double quotes.**
- **Double quotes can not be inserted into a string starting and ending with double quotes.**
- **Single quote can not be inserted into a string starting and ending with single quote.**

Examples of Valid Strings

Console C:/R Home/

```
>
> a <- 'Start and end with single quote'
> print(a)
[1] "Start and end with single quote"
>
> b <- "Start and end with double quotes"
> print(b)
[1] "Start and end with double quotes"
>
> c <- "Single quote ' in between double quotes"
> print(c)
[1] "Single quote ' in between double quotes"
>
> d <- 'Double quotes " in between single quote'
> print(d)
[1] "Double quotes \" in between single quote"
```

String Manipulation

Concatenating Strings - paste() function

- **Many strings in R are combined using the `paste()` function.**
- **It can take any number of arguments to be combined together.**
- **Syntax**

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

  - `...` represents any number of arguments to be combined.
  - `sep` represents any separator between the arguments.
  - It is optional.
  - `collapse` is used to eliminate the space in between two strings.
  - But not the space within two words of one string.

Concatenating Strings - paste() function(Cont.)

Console C:/R Home/

```
>
> a <- "Hello"
> b <- 'How'
> c <- "are you? "
>
> print(paste(a, b, c))
[1] "Hello How are you? "
>
> print(paste(a, b, c, sep = "-"))
[1] "Hello-How-are you? "
>
> print(paste(a, b, c, seq = "", collapse = ""))
[1] "Hello How are you?  "
>
```

Formatting numbers & strings - format() function

- **Format an R object for pretty printing.**
- **Syntax**

```
format(x, digits, nsmall, scientific, width, justify = c("left", "right", "centre", "none"))
```

  - **`x`** is the vector input.
  - **`digits`** is the total number of digits displayed.
  - **`nsmall`** is the minimum number of digits to the right of the decimal point.
  - **`scientific`** is set to **`TRUE`** to display scientific notation.
  - **`width`** indicates the minimum width to be displayed by padding blanks in the beginning.
  - **`justify`** is the display of the string to left, right or center.

Formatting numbers & strings - format() function (Cont.)

Console C:/R Home/

```
> # Total number of digits displayed. Last digit rounded off.
> result <- format(23.123456789, digits = 9)
> print(result)
[1] "23.1234568"
>
> # Display numbers in scientific notation.
> result <- format(c(6, 13.14521), scientific = TRUE)
> print(result)
[1] "6.000000e+00" "1.314521e+01"
>
> # The minimum number of digits to the right of the decimal point.
> result <- format(23.47, nsmall = 5)
> print(result)
[1] "23.47000"
>
> # Format treats everything as a string.
> result <- format(6)
> print(result)
[1] "6"
```

Formatting numbers & strings - format() function (Cont.)

Console C:/R Home/

```
>
> # Numbers are padded with blank in the beginning for width.
> result <- format(13.7, width = 6)
> print(result)
[1] "  13.7"
>
> # Left justify strings.
> result <- format("Hello", width = 8, justify = "l")
> print(result)
[1] "Hello   "
>
> # Justify string with center.
> result <- format("Hello", width = 8, justify = "c")
> print(result)
[1] " Hello  "
```

Counting number of characters in a string - nchar() function

- This function counts the number of characters including spaces in a string.
- Syntax

  `nchar(x)`

  - **`x`** is the vector input.

```
Console C:/R Home/
> result <- nchar("Count the number of characters")
> print(result)
[1] 30
```

Changing the case - toupper() & tolower() functions

- These functions change the case of characters of a string.
- Syntax

  `toupper(x)`

  `tolower(x)`

  - **`x`** is the vector input.

```
Console C:/R Home/
> # Changing the Upper case.
> result <- toupper("Changing To Upper")
> print(result)
[1] "CHANGING TO UPPER"
>
> # Changing the lower case.
> result <- tolower("Changing To Lower")
> print(result)
[1] "changing to lower"
```

Extracting parts of a string - substring() function

- **This function extracts parts of a String.**
- **Syntax**

```
substring(x,first,last)
```

  - **x** is the character vector input.
  - **first** is the position of the first character to be extracted.
  - **last** is the position of the last character to be extracted.

```
> # Extract characters from 3th to 8th position.
> result <- substring("Hello, World", 3, 8)
> print(result)
[1] "llo, W"
>
```