
# BHARATHIDASAN UNIVERSITY

## Tiruchirappalli- 620024

## Tamil Nadu, India.

## Programme: M.Sc. Statistics

## Course Title: R Programming

## Course Code: 23ST05CC

## Unit-II

## Variable Types

**Dr. T. Jai Sankar**
**Associate Professor and Head**
**Department of Statistics**

**Ms. I. Angel Agnes Mary**
**Guest Faculty**
**Department of Statistics**

## UNIT – II

### Variables of R

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.

| 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.

**# 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)
cat ("var.1 is ", var.1 ,"\n")
cat ("var.2 is ", var.2 ,"\n")
cat ("var.3 is ", var.3 ,"\n")
```

When we execute the above code, it produces the following result:

```
[1] 0 1 2 3
var.1 is 0 1 2 3
var.2 is learn R
var.3 is 1 1
```

**Note:** The vector c(TRUE,1) has a mix of logical and numeric class. So logical class is coerced to numeric class making TRUE as 1.

**Operators of R**

An operator 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**

We have the following types of operators in R programming:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Miscellaneous Operators

**Arithmetic Operators**

Following table shows the arithmetic operators supported by R language. The operators act on each element of the vector.

| Operator | Description | Example |
|---|---|---|
| + (plus) | Adds two vectors | v <- c( 2,5.5,6)<br>t <- c(8, 3, 4)<br>print(v+t)<br>it produces the following result:<br>[1] 10.0 8.5 10.0 |
| − (Minus) | Subtracts second vector from the first | v <- c( 2,5.5,6)<br>t <- c(8, 3, 4)<br>print(v-t)<br>it produces the following result:<br>[1] -6.0 2.5 2.0 |
| * (Multiplication) | Multiplies both vectors | v <- c( 2,5.5,6)<br>t <- c(8, 3, 4)<br>print(v*t)<br>it produces the following result:<br>[1] 16.0 16.5 24.0 |
| / (Division) | Divide the first vector with the second | v <- c( 2,5.5,6)<br>t <- c(8, 3, 4)<br>print(v/t)<br>When we execute the above code, it produces the following result:<br>[1] 0.250000 1.833333 1.500000 |
| %% | Give the remainder of the first vector with the second | v <- c( 2,5.5,6)<br>t <- c(8, 3, 4)<br>print(v%%t)<br>it produces the following result:<br>[1] 2.0 2.5 2.0 |
| %/% | The result of division of first vector with second (quotient) | v <- c( 2,5.5,6)<br>t <- c(8, 3, 4)<br>print(v%/%t)<br>it produces the following result:<br>[1] 0 1 1 |

| | | |
|---|---|---|
| ^ (Cap) | The first vector raised to the exponent of second vector<br>The first vector raised to the exponent of second vector | v <- c( 2,5.5,6)<br>t <- c(8, 3, 4)<br>print(v^t)<br>it produces the following result:<br>[1] 256.000 166.375 1296.000 |

**Relational Operators**

Following table shows the relational operators supported by R language. Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

| Operator | Description | Example |
|---|---|---|
| > | Checks if each element of the first vector is greater than the corresponding element of the second vector. | v <- c(2,5.5,6,9)<br>t <- c(8,2.5,14,9)<br>print(v>t)<br>it produces the following result:<br>[1] FALSE TRUE FALSE FALSE |
| < | Checks if each element of the first vector is less than the corresponding element of the second vector. | v <- c(2,5.5,6,9)<br>t <- c(8,2.5,14,9)<br>print(v < t)<br>it produces the following result:<br>[1] TRUE FALSE TRUE FALSE |
| = = | Checks if each element of the first vector is equal to the corresponding element of the second vector. | v <- c(2,5.5,6,9)<br>t <- c(8,2.5,14,9)<br>print(v==t)<br>it produces the following result:<br>[1] FALSE FALSE FALSE TRUE |
| <= | Checks if each element of the first vector is less than or equal to the corresponding element of the second vector. | v <- c(2,5.5,6,9)<br>t <- c(8,2.5,14,9)<br>print(v<=t)<br>it produces the following result:<br>[1] TRUE FALSE TRUE TRUE |
| >= | Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector. | v <- c(2,5.5,6,9)<br>t <- c(8,2.5,14,9)<br>print(v>=t)<br>it produces the following result:<br>[1] FALSE TRUE FALSE TRUE |

| | | |
|---|---|---|
| != | Checks if each element of the first vector is unequal to the corresponding element of the second vector. | v <- c(2,5.5,6,9)<br>t <- c(8,2.5,14,9)<br>print(v!=t)<br>it produces the following result:<br>[1] TRUE TRUE TRUE FALSE |

**Logical Operators**

Following table shows the logical operators supported by R language. It is applicable only to vectors of type logical, numeric or complex. All numbers greater than 1 are considered as logical value TRUE.

Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

| Operator | Description | Example |
|---|---|---|
| & | It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE. | v <- c(3,1,TRUE,2+3i)<br>t <- c(4,1,FALSE,2+3i)<br>print(v&t)<br>it produces the following result:<br>[1] TRUE TRUE FALSE TRUE |
| \| | It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one the elements is TRUE. | v <- c(3,0,TRUE,2+2i)<br>t <- c(4,0,FALSE,2+3i)<br>print(v\|t)<br>it produces the following result:<br>[1] TRUE FALSE TRUE TRUE |
| ! | It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value. | v <- c(3,0,TRUE,2+2i)<br>print(!v)<br>it produces the following result:<br>[1] FALSE TRUE FALSE FALSE |

The logical operator && and || considers only the first element of the vectors and give a vector of single element as output.

| Operator | Description | Example |
|---|---|---|
| && | Called Logical AND operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE. | v <- c(3,0,TRUE,2+2i)<br>t <- c(1,3,TRUE,2+3i)<br>print(v&&t)<br>it produces the following result:<br>[1] TRUE |
| \|\| | Called Logical OR operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE. | v <- c(0,0,TRUE,2+2i)<br>t <- c(0,3,TRUE,2+3i)<br>print(v\|\|t)<br>it produces the following result:<br>[1] FALSE |

**Assignment Operators**

These operators are used to assign values to vectors.

| Operator | Description | Example |
|---|---|---|
| <-<br>or<br>=<br>or<br><<- | Called Left Assignment | v1 <- c(3,1,TRUE,2+3i)<br>v2 <<- c(3,1,TRUE,2+3i)<br>v3 = c(3,1,TRUE,2+3i)<br>print(v1)<br>print(v2)<br>print(v3)<br>it produces the following result:<br>[1] 3+0i 1+0i 1+0i 2+3i<br>[1] 3+0i 1+0i 1+0i 2+3i<br>[1] 3+0i 1+0i 1+0i 2+3i |
| -><br>or<br>->> | Called Right Assignment | c(3,1,TRUE,2+3i) -> v1<br>c(3,1,TRUE,2+3i) ->> v2<br>print(v1)<br>print(v2)<br>it produces the following result:<br>[1] 3+0i 1+0i 1+0i 2+3i<br>[1] 3+0i 1+0i 1+0i 2+3i |

**Miscellaneous Operators**

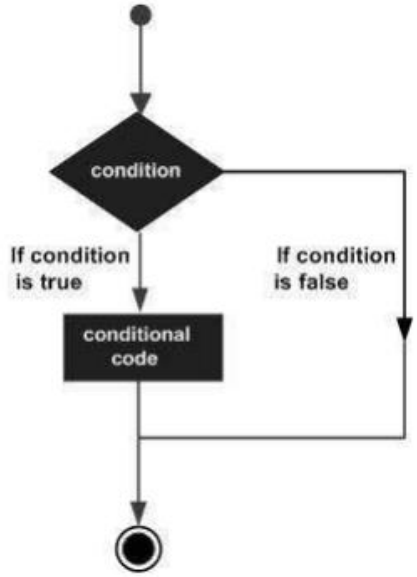
These operators are used to for specific purpose and not general mathematical or logical computation.

| Operator | Description | Example |
|---|---|---|
| : | Colon operator. It creates the series of numbers in sequence for a vector. | v <- 2:8<br>print(v)<br>it produces the following result:<br>[1] 2 3 4 5 6 7 8 |
| %in% | This operator is used to identify if an element belongs to a vector. | v1 <- 8<br>v2 <- 12<br>t <- 1:10<br>print(v1 %in% t)<br>print(v2 %in% t)<br>it produces the following result:<br>[1] TRUE<br>[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)<br>t = M %*% t(M)<br>print(t)<br>it produces the following result:<br>[,1] [,2]<br>[1,] 65 82<br>[2,] 82 117 |

**R – Decision making**

Decision making structures require the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:




R provides the following types of decision making statements. Click the following links to check their detail.

| Statement | Description |
|---|---|
| If Statement | An if statement consists of a Boolean expression followed by one or more statements. |
| If . . . else Statement | An if statement can be followed by an optional else statement, which executes when the Boolean expression is false. |
| Switch Statement | A switch statement allows a variable to be tested for equality against a list of values. |

**R -If Statement**

An if statement consists of a Boolean expression followed by one or more statements.

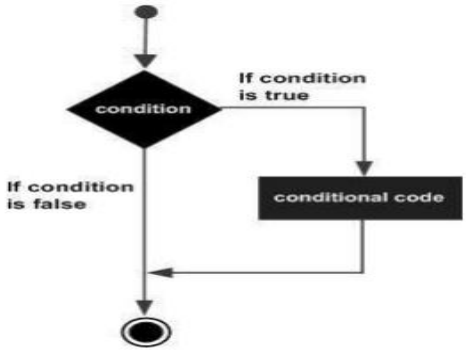
**Syntax**

The basic syntax for creating if statement in R is:

```
if(boolean_expression) {
// statement(s) will execute if the boolean expression is true.
}
```

If the Boolean expression evaluates to be true, then the block of code inside the if statement will be executed. If Boolean expression evaluates to be false, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

**Flow Diagram**




**Example**

```
x <- 30L

if(is.integer(x)){

        print("X is an Integer")

}
```

When the above code is compiled and executed, it produces the following result:

```
[1] "X is an Integer"
```

**R –If...Else Statement**

An if statement can be followed by an optional else statement which executes when the Boolean expression is false.

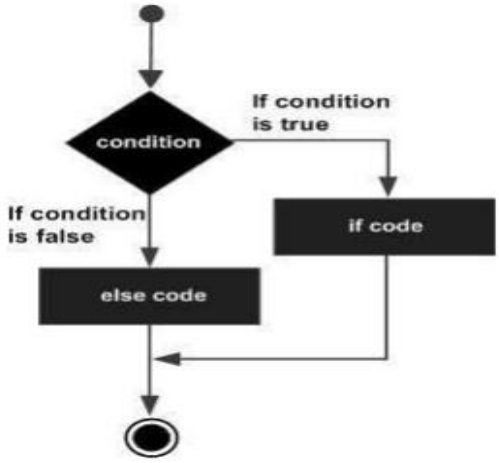
**Syntax**

The basic syntax for creating an if...else statement in R is:

```
if(boolean_expression) {

// statement(s) will execute if the boolean expression is true.

} else {

// statement(s) will execute if the boolean expression is false.

}
```

If the Boolean expression evaluates to be true, then the if block of code will be executed, otherwise else block of code will be executed.

**Flow Diagram**




**Example**

```
x <- c("what","is","truth")

if("Truth" %in% x){

        print("Truth is found")

} else {

        print("Truth is not found") }
```

When the above code is compiled and executed, it produces the following result:

```
[1] "Truth is not found"
```

Here "Truth" and "truth" are two different strings.

**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.

When using if, else if, else statements there are few points to keep in mind.

- An if can have zero or one else and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

**Syntax**

The basic syntax for creating an if...else if...else statement in R is:

```
if(boolean_expression 1) {
        // Executes when the boolean expression 1 is true.
}else if( boolean_expression 2) {
        // Executes when the boolean expression 2 is true.
}else if( boolean_expression 3) {
        // Executes when the boolean expression 3 is true.
}else {
        // executes when none of the above condition is true.
}
```

**Example**

```
x <- c("what","is","truth")
if("Truth" %in% x){
        print("Truth is found the first time")
} else if ("truth" %in% x) {
        print("truth is found the second time")
} else {
        print("No truth found")
}
```

When the above code is compiled and executed, it produces the following result:

```
[1] "truth is found the second time"
```

**R –Switch Statement**

A 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**

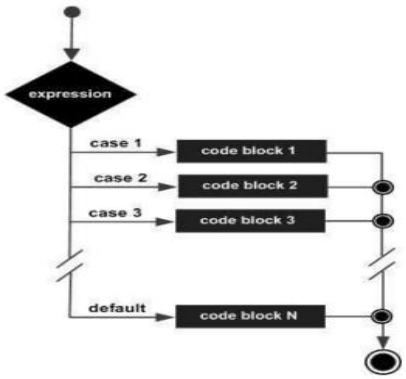
The basic syntax for creating a switch statement in R is:

```
switch (expression, case1, case2, case3....)
```

The following rules apply to a switch statement:

- If the value of expression is not a character string it is coerced to integer.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- If the value of the integer is between 1 and nargs()-1 (The max number of arguments)then the corresponding element of case condition is evaluated and the result returned.
- If expression evaluates to a character string then that string is matched (exactly) to the names of the elements.
- If there is more than one match, the first matching element is returned.
- No Default argument is available.
- In the case of no match, if there is a unnamed element of ... its value is returned. (If there is more than one such argument an error is returned.)

**Flow Diagram**




**Example**

```
x <- switch( 3, "first", "second", "third", "fourth" )
print(x)
```

When the above code is compiled and executed, it produces the following result:

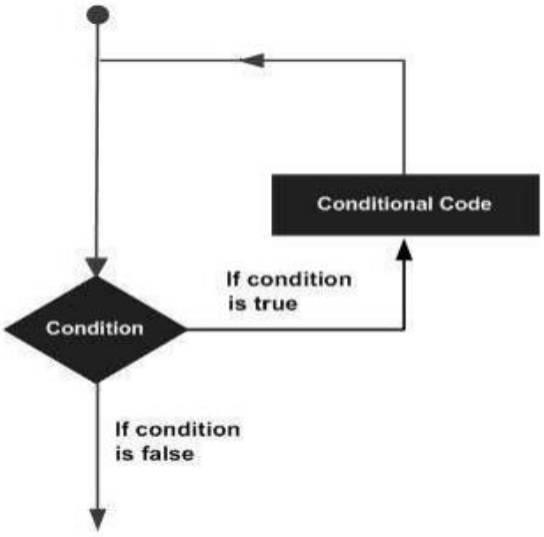
```
[1] "third"
```

**R – Loops**

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and the following is the general form of a loop statement in most of the programming languages:




R programming language provides the following kinds of loop to handle looping requirements. Click the following links to check their detail.

| Statement | Description |
|---|---|
| Repeat Loop | Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| While Loop | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| For Loop | Like a while statement, except that it tests the condition at the end of the loop body. |

**R -Repeat Loop**

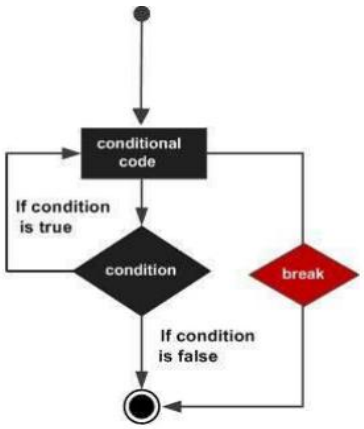
The Repeat loop executes the same code again and again until a stop condition is met.

**Syntax**

The basic syntax for creating a repeat loop in R is:

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

**Flow Diagram**




**Example**

```
v <- c("Hello","loop")
cnt <- 2
repeat {
        print(v)
        cnt <- cnt+1
        if(cnt > 5){
                break
}
}
```

When the above code is compiled and executed, it produces the following result:

[1] "Hello" "loop"

[1] "Hello" "loop"

[1] "Hello" "loop"

[1] "Hello" "loop"

**R -While Loop**

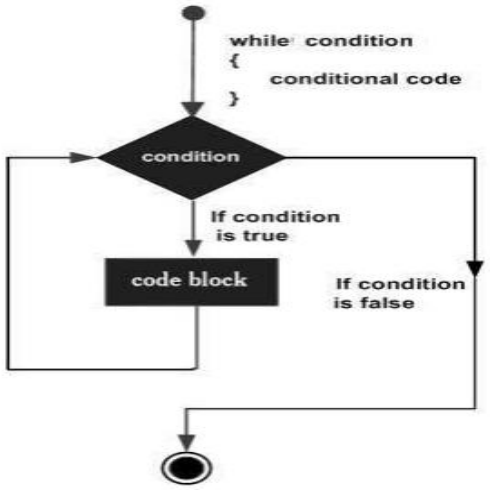
The While loop executes the same code again and again until a stop condition is met.

**Syntax**

The basic syntax for creating a while loop in R is :

```
while (test_expression) {
        statement
}
```

**Flow Diagram**




Here key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

**Example**

```
v <- c("Hello","while loop")
cnt <- 2
while (cnt < 7){
    print(v)
    cnt = cnt + 1
}
```

When the above code is compiled and executed, it produces the following result :

```
[1] "Hello" "while loop"
[1] "Hello" "while loop"
[1] "Hello" "while loop"
[1] "Hello" "while loop"
[1] "Hello" "while loop"
```

**R – For Loop**

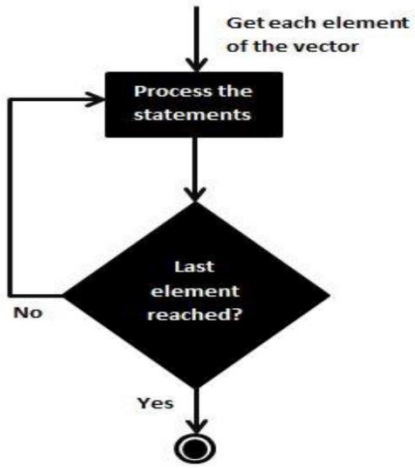
A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

**Syntax**

The basic syntax for creating a **for** loop statement in R is:

```
for (value in vector) {
    statements
}
```

**Flow Diagram**




R's for loops are particularly flexible in that they are not limited to integers, or even numbers in the input. We can pass character vectors, logical vectors, lists or expressions.

**Example**

```
v <- LETTERS[1:4]

for ( i in v) {

        print(i)

}
```

When the above code is compiled and executed, it produces the following result:

```
[1] "A"

[1] "B"

[1] "C"

[1] "D"
```

**Loop Control Statements**

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

R supports the following control statements. Click the following links to check their detail.

| Control Statement | Description |
|---|---|
| break statement | Terminates the loop statement and transfers execution to the statement immediately following the loop. |
| Next statement | The next statement simulates the behavior of R switch. |

**R – Break Statement**

The break statement in R programming language has the following two usages:

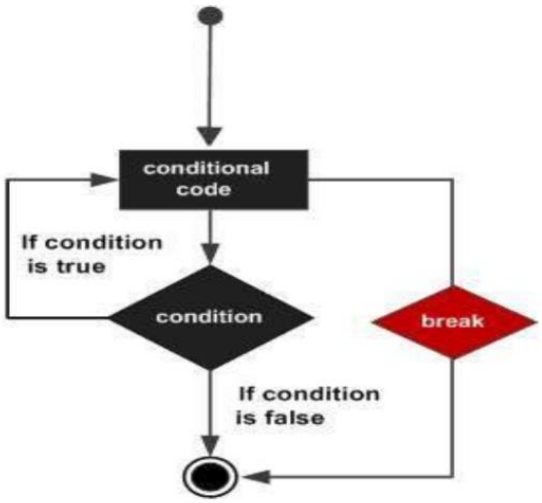
- When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
- It can be used to terminate a case in the switch statement (covered in the next chapter).

**Syntax**

The basic syntax for creating a break statement in R is:

```
break
```

**Flow Diagram**



**Example**

```
v <- c("Hello","loop")
cnt <- 2
repeat{
        print(v)
        cnt <- cnt+1
        if(cnt > 5){
                break
        }
}
```

When the above code is compiled and executed, it produces the following result:

```
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
```

**R – Next Statement**

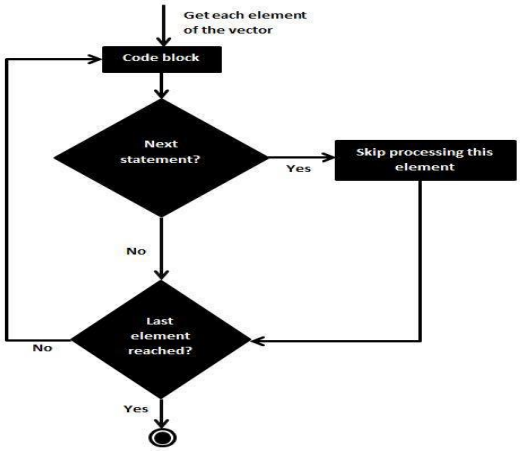
The **next** statement in R programming language is useful when we want to skip the current iteration of a loop without terminating it. On encountering next, the R parser skips further evaluation and starts next iteration of the loop.

**Syntax**

The basic syntax for creating a next statement in R is:

```
Next
```

**Flow Diagram**




**Example**

```
v <- LETTERS[1:6]

for ( i in v){

    if (i == "D"){

        next

    }

    print(i)

}
```

When the above code is compiled and executed, it produces the following result:

```
[1] "A"

[1] "B"

[1] "C"

[1] "E"

[1] "F"
```

**R – Function**

**Function Definition**

An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows:

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

**Function Components**

The different parts of a function are:

- **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; that is, a function may contain no arguments. Also arguments can have default values.
- **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.

R has many **in-built** functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as **user defined** functions.

**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.

```
# Create a sequence of numbers from 32 to 44

print(seq(32,44))

# Find mean of numbers from 25 to 82

print(mean(25:82))

# Find sum of numbers frm 41 to 68

print(sum(41:68))
```

When we execute the above code, it produces the following result:

```
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44

[1] 53.5

[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.

```
# 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**

```
# 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)
```

When we execute the above code, it produces the following result:

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

**Calling a Function without an Argument**

```
# 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()
```

When we execute the above code, it produces the following result:

```
[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 they 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)

# Call the function by names of the arguments

new.function(a=11,b=5,c=3)
```

When we execute the above code, it produces the following result:

```
[1] 26

[1] 58
```

**Calling a Function with Default Argument**

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we 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()

# Call the function with giving new values of the argument

new.function(9,5)
```

When we execute the above code, it produces the following result:

```
[1] 18

[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)
```

When we execute the above code, it produces the following result:

```
[1] 36

[1] 6
```

Error in print(b) : argument "b" is missing, with no default.