# R Reserved Words

Reserved words in R programming are a set of words that have special meaning and cannot be used as an identifier (variable name, function name etc.).

Here is a list of reserved words in the R’s parser.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reserved words in R | | | | |
| if | else | Repeat | while | function |
| for | in | Next | break | TRUE |
| FALSE | NULL | Inf | NaN | NA |
| NA\_integer\_ | NA\_real\_ | NA\_complex\_ | NA\_character\_ | … |

This list can be viewed by typing help(reserved) or ?reserved at the R command prompt as follows.

> ?reserved

Among these words, if, else, repeat, while, function, for, in, next and break are used for conditions, loops and user defined functions.

# R Variables and Constants

## Variables in R

Variables are used to store data, whose value can be changed according to our need. Unique name given to variable (function and objects as well) is identifier.

### Rules for writing Identifiers in R

1. Identifiers can be a combination of letters, digits, period (.) and underscore (\_).
2. It must start with a letter or a period. If it starts with a period, it cannot be followed by a digit.
3. Reserved words in R cannot be used as identifiers.

### Valid identifiers in R

total, Sum, .fine.with.dot, this\_is\_acceptable, Number5

### Invalid identifiers in R

tot@l, 5um, \_fine, TRUE, .0ne

### Best Practices

Earlier versions of R used underscore (\_) as an assignment operator. So, the period (.) was used extensively in variable names having multiple words.

Current versions of R support underscore as a valid identifier but it is good practice to use period as word separators.

For example, a.variable.name is preferred over a\_variable\_name or alternatively we could use camel case as aVariableName

## Constants in R

Constants, as the name suggests, are entities whose value cannot be altered. Basic types of constant are numeric constants and character constants.

### Numeric Constants

All numbers fall under this category. They can be of type integer, double or complex.

It can be checked with the typeof() function.

Numeric constants followed by L are regarded as integer and those followed by i are regarded as complex.

> typeof(5)

[1] "double"

> typeof(5L)

[1] "integer"

> typeof(5i)

[1] "complex"

### Character Constants

Character constants can be represented using either single quotes (') or double quotes (") as delimiters.

> 'example'

[1] "example"

> typeof("5")

[1] "character"

### Built-in Constants

Some of the built-in constants defined in R along with their values is shown below.

> LETTERS

[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"

[20] "T" "U" "V" "W" "X" "Y" "Z"

> letters

[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"

[20] "t" "u" "v" "w" "x" "y" "z"

> pi

[1] 3.141593

> month.name

[1] "January" "February" "March" "April" "May" "June"

[7] "July" "August" "September" "October" "November" "December"

> month.abb

[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"

But it is not good to rely on these, as they are implemented as variables whose values can be changed.

> pi

[1] 3.141593

> pi <- 56

> pi

[1] 56

# R Operators

## R Arithmetic Operators

These operators are used to carry out mathematical operations like addition and multiplication. Here is a list of arithmetic operators available in R.

|  |  |
| --- | --- |
| Arithmetic Operators in R | |
| Operator | Description |
| + | Addition |
| – | Subtraction |
| \* | Multiplication |
| / | Division |
| ^ | Exponent 2^3 = 8 |
| %% | Modulus (Remainder from division) 32.1 |
| %/% | Integer Division 32 |

## R Relational Operators

Relational operators are used to compare between values. Here is a list of relational operators available in R.

|  |  |
| --- | --- |
| Relational Operators in R | |
| Operator | Description |
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |
| == | Equal to |
| != | Not equal to |

## R Logical Operators

Logical operators are used to carry out Boolean operations like AND, OR etc.

|  |  |
| --- | --- |
| Logical Operators in R | |
| Operator | Description |
| ! | Logical NOT |
| & | Element-wise logical AND |
| && | Logical AND |
| | | Element-wise logical OR |
| || | Logical OR |

## R Assignment Operators

These operators are used to assign values to variables.

|  |  |
| --- | --- |
| Assignment Operators in R | |
| Operator | Description |
| <-, <<-, = | Leftwards assignment |
| ->, ->> | Rightwards assignment |

The operators <- and = can be used, almost interchangeably, to assign to variable in the same environment.

The <<- operator is used for assigning to variables in the parent environments (more like global assignments). The rightward assignments, although available are rarely used.

> x <- 5

> x

[1] 5

> x = 9

> x

[1] 9

> 10 -> x

> x

[1] 10

# R if…else Statement

## R if statement

The syntax of if statement is:

if (test\_expression) {

statement

}

If the test\_expression is TRUE, the statement gets executed. But if it’s FALSE, nothing happens.

Here, test\_expression can be a logical or numeric vector, but only the first element is taken into consideration.

In the case of numeric vector, zero is taken as FALSE, rest as TRUE.

## if…else statement

The syntax of if…else statement is:

if (test\_expression) {

statement1

} else {

statement2

}

## if…else Ladder

The if…else ladder (if…else…if) statement allows you execute a block of code among more than 2 alternatives

The syntax of if…else statement is:

if ( test\_expression1) {

statement1

} else if ( test\_expression2) {

statement2

} else if ( test\_expression3) {

statement3

} else {

statement4

}

# R ifelse() Function

## Syntax of ifelse() function

ifelse(test\_expression, x, y)

Here, test\_expression must be a logical vector (or an [object](https://www.datamentor.io/r-programming/object-class-introduction) that can be coerced to logical). The return value is a vector with the same length as test\_expression.

## Example: ifelse() function

> a = c(5,7,2,9)

> ifelse(a %% 2 == 0,"even","odd")

[1] "odd" "odd" "even" "odd"

# R for Loop

x <- c(2,5,3,9,8,11,6)

count <- 0

for (val in x) {

if(val %% 2 == 0) count = count+1

}

print(count)

# R while Loop

## Example of while Loop

i <- 1

while (i < 6) {

print(i)

i = i+1

}

**R break and next Statement**

## break statement

A break statement is used inside a loop ([repeat](https://www.datamentor.io/r-programming/repeat-loop), [for](https://www.datamentor.io/r-programming/for-loop), [while](https://www.datamentor.io/r-programming/while-loop)) to stop the iterations and flow the control outside of the loop.

In a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated.

The syntax of break statement is:

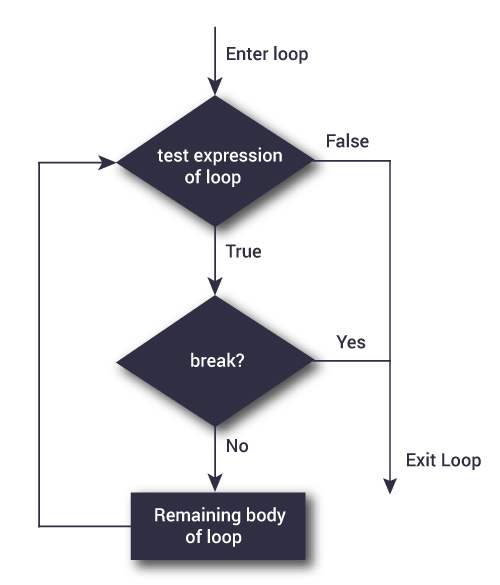
if (test\_expression) {

break

}

**Note:** the break statement can also be used inside the  else branch of if...else statement.

### Flowchart of break statement



### Example 1: break statement

x <- 1:5

for (val in x) {

if (val == 3){

break

}

print(val)

}

**Output**

[1] 1

[1] 2

## next statement

A next statement 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.

The syntax of next statement is:

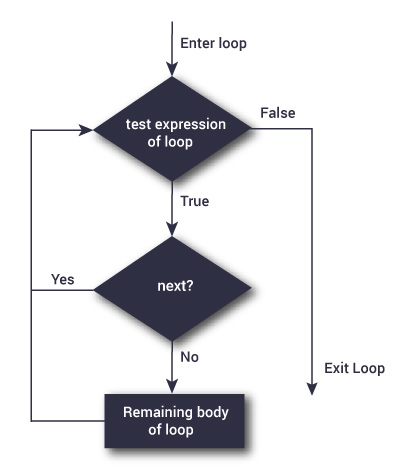
if (test\_condition) {

next

}

**Note:** the next statement can also be used inside the  else branch of if...else statement.

### Flowchart of next statement



### Example 2: Next statement

x <- 1:5

for (val in x) {

if (val == 3){

next

}

print(val)

}

**Output**

[1] 1

[1] 2

[1] 4

[1] 5

# R repeat loop

A repeat loop is used to iterate over a block of code multiple number of times.

There is no condition check in repeat loop to exit the loop.

We must ourselves put a condition explicitly inside the body of the loop and use the [break statement](https://www.datamentor.io/r-programming/break-next) to exit the loop. Failing to do so will result into an infinite loop.

## Syntax of repeat loop

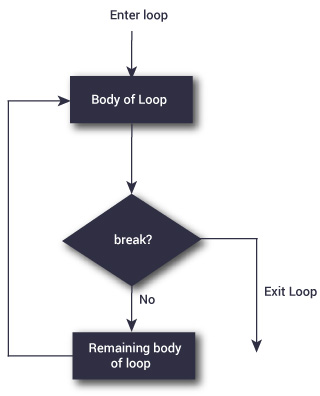
repeat {

statement

}

In the statement block, we must use the break statement to exit the loop.

## Flowchart of repeat loop



## Example: repeat loop

x <- 1

repeat {

print(x)

x = x+1

if (x == 6){

break

}

}

**Output**

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

# R Functions

## Example of a Function

pow <- function(x, y) {

# function to print x raised to the power y

result <- x^y

print(paste(x,"raised to the power", y, "is", result))

}

## How to call a function?

We can call the above function as follows.

>pow(8, 2)

[1] "8 raised to the power 2 is 64"

> pow(2, 8)

[1] "2 raised to the power 8 is 256"

## Example: return()

Let us look at an example which will return whether a given number is positive, negative or zero.

check <- function(x) {

if (x > 0) {

result <- "Positive"

}

else if (x < 0) {

result <- "Negative"

}

else {

result <- "Zero"

}

return(result)

}

# R Environment and Scope

## R Programming Environment

Environment can be thought of as a collection of [objects](https://www.datamentor.io/r-programming/object-class-introduction) (functions, variables etc.).

An environment is created when we first fire up the R interpreter. Any variable we define, is now in this environment.

The top level environment available to us at the R command prompt is the global environment called R\_GlobalEnv.

**Global environment** can be referred to as .GlobalEnv in R codes as well.

We can use the ls() function to show what variables and functions are defined in the current environment.

Moreover, we can use the environment() function to get the current environment.

> a <- 2

> b <- 5

> f <- function(x) x<-0

> ls()

[1] "a" "b" "f"

> environment()

<environment: R\_GlobalEnv>

> .GlobalEnv

<environment: R\_GlobalEnv>

In the above example, we can see that a, b and f are in the R\_GlobalEnv environment.

Notice that x (in the argument of the function) is not in this global environment. When we define a function, a new environment is created.

In the above example, the function f creates a new environment inside the global environment.

Actually an environment has a frame, which has all the objects defined, and a pointer to the enclosing (parent) environment.

Hence, x is in the frame of the new environment created by the function f. This environment will also have a pointer to R\_GlobalEnv.

### Example: Cascading of environments

f <- function(f\_x){

g <- function(g\_x){

print("Inside g")

print(environment())

print(ls())

}

g(5)

print("Inside f")

print(environment())

print(ls())

}

Now when we run it from the command prompt, we get.

> f(6)

[1] "Inside g"

<environment: 0x0000000010c2bdc8>

[1] "g\_x"

[1] "Inside f"

<environment: 0x0000000010c2a870>

[1] "f\_x" "g"

> environment()

<environment: R\_GlobalEnv>

> ls()

[1] "f"

Here, we defined function g inside f and it is clear that they both have different environments with different objects within their respective frames.

## R Programming Scope

Let us consider the following example.

outer\_func <- function(){

b <- 20

inner\_func <- function(){

c <- 30

}

}

a <- 10

### Global variables

Global variables are those variables which exists throughout the execution of a program. It can be changed and accessed from any part of the program.

### Local variables

On the other hand, Local variables are those variables which exist only within a certain part of a program like a function, and is released when the function call ends.

In the above program the variable c is called a local variable.

If we assign a value to a variable with the function inner\_func(), the change will only be local and cannot be accessed outside the function.

This is also the same even if names of both global variable and local variables matches.

For example, if we have a function as below.

outer\_func <- function(){

a <- 20

inner\_func <- function(){

a <- 30

print(a)

}

inner\_func()

print(a)

}

When we call it,

> a <- 10

> outer\_func()

[1] 30

[1] 20

> print(a)

[1] 10

We see that the variable a is created locally within the environment frame of both the functions and is different to that of the global environment frame.

### Accessing global variables

Global variables can be read but when we try to assign to it, a new local variable is created instead.

To make assignments to global variables, superassignment operator, <<-, is used.

When using this operator within a function, it searches for the variable in the parent environment frame, if not found it keeps on searching the next level until it reaches the global environment.

If the variable is still not found, it is created and assigned at the global level.

outer\_func <- function(){

inner\_func <- function(){

a <<- 30

print(a)

}

inner\_func()

print(a)

}

On running this function,

> outer\_func()

[1] 30

[1] 30

> print(a)

[1] 30

When the statement a <<- 30 is encountered within inner\_func(), it looks for the variable a in outer\_func() environment.

# R Recursive Function

A [function](https://www.datamentor.io/r-programming/function) that calls itself is called a recursive function and this technique is known as recursion.

This special programming technique can be used to solve problems by breaking them into smaller and simpler sub-problems.

An example can help clarify this concept:

Let us take the example of finding the factorial of a number. Factorial of a positive integer number is defined as the product of all the integers from 1 to that number. For example, the factorial of 5 (denoted as 5!) will be

5! = 1\*2\*3\*4\*5 = 120

This problem of finding factorial of 5 can be broken down into a sub-problem of multiplying the factorial of 4 with 5.

5! = 5\*4!

Or more generally,

n! = n\*(n-1)!

Now we can continue this until we reach 0! which is 1.

The implementation of this is provided below.

## Example: Recursive Function in R

# Recursive function to find factorial

recursive.factorial <- function(x) {

if (x == 0) return (1)

else return (x \* recursive.factorial(x-1))

}

Here, we have a function which will call itself. Something like recursive.factorial(x) will turn into x \* recursive.factorial(x) until x becomes equal to 0.

# R Infix Operator

## Example: How infix operators work in R?

> 5+3

[1] 8

> `+`(5,3,1,2,2)

[1] 8

> 5-3

[1] 2

> `-`(5,3)

[1] 2

> 5\*3-1

[1] 14

> `-`(`\*`(5,3),1)

[1] 14

# R switch() Function

## Example: switch() function

If the value evaluated is a number, that item of the list is returned.

> switch(2,"red","green","blue")

[1] "green"

> switch(1,"red","green","blue")

[1] "red"

## Example: switch() Function with as String Expression

The expression used in the switch () function can be a string as well. In this case, the matching named item’s value is returned.

> switch("color", "color" = "red", "shape" = "square", "length" = 5)

[1] "red"

Here, "color" is a string  which matches with the first item of the list. Hence, we are getting “red” as an output.

> switch("length", "color" = "red", "shape" = "square", "length" = 5)

[1] 5

Similarly, "length" expression matches with the last item of the list. Hence, we are getting 5 as an output.