**Types of Functions in R Programming**

A [function](https://www.geeksforgeeks.org/functions-in-r-programming/) is a set of statements orchestrated together to perform a specific operation. A function is an object so the 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 the task and returns control to the interpreter as well as any return values that may be stored in other objects.

### How to Define a Function?

In [R programming](https://www.geeksforgeeks.org/introduction-to-r-programming-language/), a function can be defined using the keyword function. The syntax to define a function in R is as follows:

***Syntax:***

*function\_name = function(arg\_1, arg\_2, …)  
{  
Function body  
}*

The various components/parts of a function are:

* **Function name:** It 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. Whenever a function is invoked, a value if passed to the argument.They are optional; that is, a function may contain no arguments. Also arguments can have default values.
* **Function Body:** It contains all the set of statements that defines what actually the function does.
* **Return Values:** It is the values that function returns after the successful execution of the tasks.In more general,it is the last expression in the function body to be evaluated.

### ****Calling a Function****

It is nothing but calling the original function with a valid number of arguments. A function can be called with an argument, without an argument and with a default value as well.

**Example:**Calling a function without an argument

|  |
| --- |
| # create a function cube  # without an argument  cube <- function()  {    for(i in 1:10)    {      print(i^3)    }  }  # calling function cube without an argument  cube() |

**Output:**

[1] 1

[1] 8

[1] 27

[1] 64

[1] 125

[1] 216

[1] 343

[1] 512

[1] 729

[1] 1000

**Example:**Calling a function with an argument.

|  |
| --- |
| # create a function factorial  # with a numeric argument n  factorial <- function(n)  {    if(n==0)    {      return(1)    }    else    {      return(n \* factorial(n - 2))    }  }    # calling function cube with an argument  factorial(7) |

**Output:**

[1] 5040

**Example:**Calling a function with default argument.

|  |
| --- |
| # create a function def\_arg  # without an argument  def\_arg <- function(a = 23, b = 35)  {    output <- (a + b) \* a + (a - b) \* b    print(output)  }    # calling function def\_arg without an argument  def\_arg()    # call the function with giving new values of the argument.  def\_arg(16, 22) |

**Output:**

[1] 914

[1] 476

### Types of Function

There are mainly three types of function in R programming:

1. Primitive Functions
2. Infix Functions
3. Replacement Functions

#### Primitive Functions

Generally, a function comprises of three parts:

* The **formals ()**, the list of arguments that control how you call the function.
* The **body ()**, the code inside the function.
* The **environment ()**, the data structure that determines how the function finds the values associated with the names.

The formals and body are defined explicitly whenever one creates a function, but the environment is specified implicitly, based on where you define the function. But there is an exception to the rule that a function has three components, some functions call C code directly. These functions are known as primitive functions. Primitive functions exist primarily in C, not R, so their **formals()**, **body()**,and **environment()** are NULL. These functions are only found in the base package. Primitive functions are harder to write but are highly efficient. They are of two types, either type builtin or type special.

|  |
| --- |
| typeof(sum)  typeof('[') |

[1] "builtin" #> typeof(sum)

[1] "character" #> typeof('[')

**Example:** To print the names of available primitive functions in your R console run the following code.

|  |
| --- |
| names(methods:::.BasicFunsList) |

**Output:**

[1] "$" "$<-" "[" "[<-" "[[" "[[=" "cosh" "cummax" "dimnames<-"

[22] "as.raw" "log2" "tan" "dim" "as.logical" "^" "is.finite"

[29] "sinh" "log10" "as.numeric" "dim<-" "is.array" "tanpi" "gamma"

[36] "atan" "as.integer" "Arg" "signif" "cumprod" "cos" "length"

[43] "!=" "digamma" "exp" "floor" "acos" "seq.int" "abs"

[50] "length<-" "sqrt" "!" "acosh" "is.nan" "Re" "tanh"

[57] "names" "cospi" "&" "anyNA" "trunc" "cummin" "levels<-"

[64] "\*" "Mod" "|" "names<-" "+" "log" "lgamma"

[71] "as.complex" "asinh" "-" "sin" "/" "as.environment" "<="

[78] "as.double" "is.infinite" "is.numeric" "rep" "round" "sinpi" "dimnames"

[85] "asin" "as.character" "%/%" "is.na" "" "Im"

[92] "%%" "trigamma" "==" "cumsum" "atanh" "sign" "ceiling"

[99] "Conj" "as.call" "log1p" "expm1" "(" ":" "="

[106] "@" "{" "~" "&&" ".C" "baseenv" "quote"

[113] "<-" "is.name" "if" "||" "attr<-" "untracemem" ".cache\_class"

[120] "substitute" "interactive" "is.call" "switch" "function" "is.single" "is.null"

[127] "is.language" "is.pairlist" ".External.graphics" "globalenv" "class<-" ".Primitive" "is.logical"

[134] "enc2utf8" "UseMethod" ".subset" "proc.time" "enc2native" "repeat" "<<-"

[141] "@<-" "missing" "nargs" "isS4" ".isMethodsDispatchOn" "forceAndCall" ".primTrace"

[148] "storage.mode<-" ".Call" "unclass" "gc.time" ".subset2" "environment<-" "emptyenv"

[155] "seq\_len" ".External2" "is.symbol" "class" "on.exit" "is.raw" "for"

[162] "is.complex" "list" "invisible" "is.character" "oldClass<-" "is.environment" "attributes"

[169] "break" "return" "attr" "tracemem" "next" ".Call.graphics" "standardGeneric"

[176] "is.atomic" "retracemem" "expression" "is.expression" "call" "is.object" "pos.to.env"

[183] "attributes<-" ".primUntrace" "...length" ".External" "oldClass" ".Internal" ".Fortran"

[190] "browser" "is.double" ".class2" "while" "nzchar" "is.list" "lazyLoadDBfetch"

[197] "...elt" "is.integer" "is.function" "is.recursive" "seq\_along" "unlist" "as.vector"

[204] "lengths"

#### Infix Functions

Infix functions are those functions in which the function name comes in between its arguments, and hence have two arguments. R comes with a number of built-in infix operators such as **:, ::, :::, $, @, ^, \*, /, +, -, >, >=, <, <=, ==, !=, !, &, &&, |, ||, ~, <-, and <<-**. One can create his own infix functions that start and end with %. The name of an infix function is more flexible as it can contain any sequence of characters except %. There are some predefined infix operators in R programming.

| Operators | Description |
| --- | --- |
| %% | Remainder operator |
| %/% | Integer Division |
| %\*% | Matrix multiplication |
| %o% | Outer Product |
| %x% | Kronecker product |
| %in% | Matching Operator |

**Example:** Create a two argument function that gives greater of two numbers and bind it to a name that starts and ends with %.

|  |
| --- |
| # R program to illustrate  # Infix function    '%Greater%' <- function(a, b)  {    if(a > b) print(a)    else if(b > a) print(b)    else print("equal")  }  5 %Greater% 7  2300 %Greater% 67 |

**Output:**

[1] 7

[1] 2300

#### Replacement Functions

Replacement functions modify their arguments in place(modifying an R object usually creates a copy). The name of replacement functions are always succeeded by <. They must have arguments named x and value, and return the modified object. In case of a replacement, a function needs additional arguments, the additional arguments should be placed between x and value, and must be called with additional arguments on the left. The name of the function has to be quoted as it is a syntactically valid but non-standard name and the parser would interpret <- as the operator not as part of the function name if it weren't quoted.

***Syntax:***

*“function\_name<-" <- function(x, additional arguments, value)  
{  
function body  
}*

**Example:**

|  |
| --- |
| # R program to illustrate  # Replacement function    "replace<-" <- function(x, value)  {    x[1] = value    x  }  x = rep.int(5, 7)  replace(x) = 0L  print(x) |

**Output:**

[1] 0 5 5 5 5 5 5