R Functions

In this tutorial, you'll learn about R functions and how to create them with the help of examples.

Introduction to R Functions

A function is just a block of code that you can call and run from any part of your program. They are used to break our code in simple parts and avoid repeatable codes.

You can pass data into functions with the help of parameters and return some other data as a result. You can use the `function()` reserve keyword to create a function in R. The syntax is:

```
func_name <- function (parameters) {
statement
}
```

Here, `func_name` is the name of the function. For example,

```
# define a function to compute power
power <- function(a, b) {
    print(paste("a raised to the power b is: ", a^b))
}
```

Here, we have defined a function called power which takes two parameters - `a` and `b`. Inside the function, we have included a code to print the value of `a` raised to the power `b`.

Call the Function

After you have defined the function, you can call the function using the function name and arguments. For example,

```
# define a function to compute power
power <- function(a, b) {
    print(paste("a raised to the power b is: ", a^b))
}

# call the power function with arguments
power(2, 3)
```

Output

```
[1] "a raised to the power b is:  8"
```

Here, we have called the function with two arguments - `2` and `3`. This will print the value of `2` raised to the power `3` which is `8`.

The arguments used in the actual function are called formal arguments. They are also called parameters. The values passed to the function while calling the function are called actual arguments.

Named Arguments

In the above function call of the `power()` function, the arguments passed during the function call must be of the same order as the parameters passed during function declaration.

This means that when we call `power(2, 3)`, the value `2` is assigned to `a` and `3` is assigned to `b`. If you want to change the order of arguments to be passed, you can use named arguments. For example,

```
# define a function to compute power
power <- function(a, b) {
    print(paste("a raised to the power b is: ", a^b))
}

# call the power function with arguments
power(b=3, a=2)
```

Output

```
[1] "a raised to the power b is:  8"
```

Here, the result is the same irrespective of the order of arguments that you pass during the function call.

You can also use a mix of named and unnamed arguments. For example,

```
# define a function to compute power
power <- function(a, b) {
    print(paste("a raised to the power b is: ", a^b))
}

# call the power function with arguments
power(b=3, 2)
```

Output

```
[1] "a raised to the power b is:  8"
```

Default Parameters Values

You can assign default parameter values to functions. To do so, you can specify an appropriate value to the function parameters during function definition.

When you call a function without an argument, the default value is used. For example,

```
# define a function to compute power
power <- function(a = 2, b) {
    print(paste("a raised to the power b is: ", a^b))
}

# call the power function with arguments
```

```
power(2, 3)

# call function with default arguments
power(b=3)
```

Output

```
[1] "a raised to the power b is:  8"
[1] "a raised to the power b is:  8"
```

Here, in the second call to `power()` function, we have only specified the `b` argument as a named argument. In such a case, it uses the default value for `a` provided in the function definition.

Return Values

You can use the `return()` keyword to return values from a function. For example,

```
# define a function to compute power
power <- function(a, b) {
    return (a^b)
}

# call the power function with arguments
print(paste("a raised to the power b is: ", power(2, 3)))
```

Output

```
[1] "a raised to the power b is:  8"
```

Here, instead of printing the result inside the function, we have returned `a^b`. When we call the `power()` function with arguments, the result is returned which can be printed during the call.

Nested Function

In R, you can create a nested function in 2 different ways.

- By calling a function inside another function call.
- By writing a function inside another function.

Example 1: Call a Function Inside Another Function Call

Consider the example below to create a function to add two numbers.

```
# define a function to compute addition
add <- function(a, b) {
    return (a + b)
}

# nested call of the add function
print(add(add(1, 2), add(3, 4)))
```

Output

```
[1] 10
```

Here, we have created a function called `add()` to add two numbers. But during the function call, the arguments are calls to the `add()` function.
First, `add(1, 2)` and `add(3, 4)` are computed and the results are passed as arguments to the outer `add()` function. Hence, the result is the sum of all four numbers.

Example 1:

```
# R program to calculate factorial value

# Using factorial() method
answer1 <- factorial(4)
```

```
answer2 <- factorial(-3)
answer3 <- factorial(0)

print(answer1)
print(answer2)
print(answer3)
```

Output:

```
24
NaN
1
```

Example 2:

```
# R program to calculate factorial value

# Using factorial() method
answer1 <- factorial(c(0, 1, 2, 3, 4))

print(answer1)
```

Output:

```
   1   1   2   6  24
```

Types of Function in R Language

1. **Built-in Function:** Built-in functions in R are pre-defined functions that are available in R programming languages to perform common tasks or operations**.**
2. **User-defined Function:** R language allow us to write our own function.

Built-in Function in R Programming Language

Here we will use built-in functions like sum(), max() and min().

```
# Find sum of numbers 4 to 6.
print(sum(4:6))

# Find max of numbers 4 and 6.
print(max(4:6))

# Find min of numbers 4 and 6.
print(min(4:6))
```

Output

```
[1] 15
[1] 6
[1] 4
```

User-defined Functions in R Programming Language

R provides built-in functions like **print()**, **cat()**, etc. but we can also create our own functions. These functions are called user-defined functions.

Example

R

```
# 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"
```

R Function Example – Single Input Single Output

Now create a function in R that will take a single input and gives us a single output.

Following is an example to create a function that calculates the area of a circle which takes in the arguments the radius. So, to create a function, name the function as “areaOfCircle” and the arguments that are needed to be passed are the “radius” of the circle.

R

```
# A simple R function to calculate
# area of a circle

areaOfCircle = function(radius){
 area = pi*radius^2
 return(area)
}

print(areaOfCircle(2))
```

Output

```
12.56637
```

R Function Example – Multiple Input Multiple Output

Now create a function in R Language that will take multiple inputs and gives us multiple outputs using a list.

The functions in R Language take multiple input objects but returned only one object as output, this is, however, not a limitation because you can create lists of all the outputs which you want to create and once the list is created you can access them into the elements of the list and get the answers which you want.

Let us consider this example to create a function “Rectangle” which takes “length” and “width” of the rectangle and returns area and perimeter of that rectangle. Since R Language can return only one object. Hence, create one object which is a list that contains “area” and “perimeter” and return the list.

R

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

Rectangle = function(length, width){
 area = length * width
 perimeter = 2 * (length + width)

 # create an object called result which is
 # a list of area and perimeter
 result = list("Area" = area, "Perimeter" = perimeter)
 return(result)
}

resultList = Rectangle(2, 3)
print(resultList["Area"])
print(resultList["Perimeter"])
```

Output

```
$Area
[1] 6

$Perimeter
[1] 10
```

Inline Functions in R Programming Language

Sometimes creating an R script file, loading it, executing it is a lot of work when you want to just create a very small function. So, what we can do in this kind of situation is an inline function.

To create an inline function you have to use the function command with the argument x and then the expression of the function.

Example

R

```
# A simple R program to
# demonstrate the inline function


f = function(x) x^2*4+x/3

print(f(4))
print(f(-2))
print(0)
```

Output

```
65.33333
15.33333
0
```

Passing Arguments to Functions in R Programming Language

There are several ways you can pass the arguments to the function:
Case 1: Generally in R, the arguments are passed to the function in the same order as in the function definition.
Case 2: If you do not want to follow any order what you can do is you can pass the arguments using the names of the arguments in any order.
Case 3: If the arguments are not passed the default values are used to execute the function.
Now, let us see the examples for each of these cases in the following R code:
R

```
# A simple R program to demonstrate
# passing arguments to a function

Rectangle = function(length=5, width=4){
  area = length * width
  return(area)
}

# Case 1:
print(Rectangle(2, 3))

# Case 2:
print(Rectangle(width = 8, length = 4))

# Case 3:
print(Rectangle())
```

Output

```
6
32
20
```

Lazy Evaluations of Functions in R Programming Language

In R the functions are executed in a lazy fashion. When we say lazy what it means is if some arguments are missing the function is still executed as long as the execution does not involve those arguments.
Example
In the function “Cylinder” given below. There are defined three-argument “diameter”, “length” and “radius” in the function and the volume calculation does not involve this argument “radius” in this calculation. Now, when you pass this argument “diameter” and “length” even though you are not passing

this “radius” the function will still execute because this radius is not used in the calculations inside the function.
Let’s illustrate this in an R code given below:
R

```
# A simple R program to demonstrate
# Lazy evaluations of functions

Cylinder = function(diameter, length, radius ){
  volume = pi*diameter^2*length/4
  return(volume)
}

# This'll execute because this
# radius is not used in the
# calculations inside the function.
print(Cylinder(5, 10))
```

Output

```
196.3495
```

If you do not pass the argument and then use it in the definition of the function it will throw an error that this “radius” is not passed and it is being used in the function definition.

Example

R

```
# A simple R program to demonstrate
# Lazy evaluations of functions

Cylinder = function(diameter, length, radius ){
  volume = pi*diameter^2*length/4
  print(radius)
  return(volume)
}

# This'll throw an error
print(Cylinder(5, 10))
```

Output

```
Error in print(radius) : argument "radius" is missing, with no
default
```

```
# 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.

r

```
# 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
```