Functions in R (User-defined)

Pavan Kumar A
Senior Project Engineer
Big Data Analytics Team
CDAC-KP

FUNCTIONS

- Functions refer to smaller groups/blocks of statements which a large program is divided.

- R offers a range of predefined functions, such as round() and sqrt()

  Predefined functions are categorized as per their functionalities and grouped into containers known as packages.

  All mathematical packages are included in **`Math`** package

- R also allows users to define functions according to their requirements

Functions

- Using functions in R provides following advantages:
  - They can be simply called as and when required, without actually writing the code again.
  - They can work with variables inputs, i.e. with different values (arguments can be given different values each time it was called)
  - They return the result as an object. This result can further be used as an input for another function or program (Workflows in R)

Transforming An R Script into Function

- First need to create Rscript (In Reditor), R function and then transform the script into function.
- R script can be read by source() function.
  - Suppose you create a file named **`Myfile.R`** in Notepad.
  - To read this file in R, type the following command in the R console

`>source("Myfile.R")`

```
z<-c(1:10)
table<-round(z*2)
res<-paste(table, sep="")
print("Table of 2:")
print(res)
```

```
> source("C:/Users/CDAC/Documents/table.R")
[1] "Table of 2:"
 [1] "2"  "4"  "6"  "8"  "10" "12" "14" "16" "18" "20"
>
```

Creating a Function

- Define a function with an appropriate name
- Declare the function keyword using parentheses
- Provide arguments to the functions
- Declare return statement if required

Syntax for defining a function in R

```
<functionname> <-function(arg1, arg2..)
{
        statement 1
        statement 2
        ……..
return(output)
}
```

Creates a simple function with the name "MyFirstFunc"

```
MyFirstFunc<-function()
{
        print("hello")
}
```

Creating a Function

- Creating the User-Defined Function

```
function.name <- function(arguments)
{
        computations on the arguments
        some other code
 }
```

Some arguments used as input to the function, within the () following the keyword 'function';
A body, which is the code within the curly braces {}, where we carry out the computation

```
showtable<-function()
{
z<-c(1:10)
table<-round(z*2)
res<-paste(table, sep="")
print("Table of 2:")
return(res)
}
```

Returns the result of the showTable() function

The **`return`** statement is optional in case of functions, because by default, R will always return the value of the last line of code written in the function.

Calling a Function

- After creating function, we can call or invoke the function by specifying the source of the script file and writing the name of the function

```
> source("C:/Users/CDAC/Documents/Function.R")
> MyFirstFunc()
[1] "Hello"
>
```

```
> source("C:/Users/CDAC/Documents/table.R")
> showtable
function()
{
z<-c(1:10)
table<-round(z*2)
res<-paste(table, sep="")
print("Table of 2:")
return(res)
}
> showtable()
[1] "Table of 2:"
 [1] "2"  "4"  "6"  "8"  "10" "12" "14" "16" "18" "20"
>
```

Accessing the Function Objects

- R considers functions as objects, and allows you to assign the functionality of an object to another object.

`assignTable<-showTable`

Here we are assigning the functions of **`showTable`** function to **`assignTable`** object

```
> assignTable<-showtable
> showtable
function()
{
z<-c(1:10)
table<-round(z*2)
res<-paste(table, sep="")
print("Table of 2:")
return(res)
}
> assignTable
function()
{
z<-c(1:10)
table<-round(z*2)
res<-paste(table, sep="")
print("Table of 2:")
return(res)
}
> assignTable()
[1] "Table of 2:"
 [1] "2"  "4"  "6"  "8"  "10" "12" "14" "16" "18" "20"
> showtable()
[1] "Table of 2:"
 [1] "2"  "4"  "6"  "8"  "10" "12" "14" "16" "18" "20"
>
```

Writing Functions without Braces

- If a function contains only a single line of code, then you can define the function without the parentheses.
- For example: Calculating the square of a number

```
calSquare<-function(y) y*y
```

Creates the function calSquare() without using braces

RETURN AND NESTED FUNCTION CALLS IN R

- The **`return`** statement is optional in case of functions, because by default, R will always return the value of the last line of code written in the function.

- But it is advisable to use
  - When the function performs several computations
  - When we want the value to be accessible outside of the function body

- Arguments : Arguments in the function can be of any type

RETURN AND NESTED FUNCTION CALLS IN R

- **Example**

```
myFirstFun<-function(n)
{ n*n # compute the square of integer n
}
# call the function with argument n
u<-myFirstFun(n)
```

```
# define a numeric vector v1 of 4 elements
v<-c(1, 3, 0.2, 1.5, 1.7)
# define a matrix M
M<-cbind( c(0.2, 0.9, 1), c(1.0, 5.1, 1), c(6, 0.2, 1),
c(2.0, 9, 1))
```

```
> source("C:/Users/CDAC/Documents/NestedFunction.R")
> result
[1] 1.00 9.00 0.04 2.25 2.89
> source("C:/Users/CDAC/Documents/NestedFunction.R")
> result
NULL
>
```

```
#passing only 1 argument, nested call and return

mySecFun<-function(v)
{

# compute the square of each element of v into u

u=c(0,0,0,0)
for(i in 1:length(v))
            {
            u[i]=myFirstFun(v[i]);
            }
return(u)
}
Sqv=mySecFun(v)
Sqv
```

Arguments and their default

- **Examples**
  - Let's see a sequence of examples to compute some power of a value n passed as an argument, with few variations on arguments management

```
# we define the function and specify the exponent, second argument directly
MyFourthFun <- function(n, y = 2) # sets default of exponent to 2 (we just square)
{
    n^y   # compute the power of n to the y
}
MyFourthFun(2,3) # specify both args
MyFourthFun(2) # or just first'
# MyFourthFun() # or none: error!
```

```
> MyFourthFun<-function(n, y=2)
+ {
+ n^y
+ }
> MyFourthFun(2,3)
[1] 8
> MyFourthFun(2)
[1] 4
> MyFourthFun()
Error in MyFourthFun() : argument "n" is missing, with no default
>
```

Arguments and their default

- We can also specify the second argument, our exponent as a list of values, so to compute the powers of the given n with exponent less or equal to 1

```
# with variable exponent from 0.05 to 1 in steps of 0.01
MyFourthFun <- function(n, y = seq(0.05, 1, by = 0.01))
{
        n^y # compute the power of n to the y
}
MyFourthFun(2,3) # as before
MyFourthFun(2) # computes ALL possible according to given default
# MyFourthFun() # or none: error!
```

```
> MyFourthFun<-function(n, y=seq(1, 3,by=1)){n^y}
> MyFourthFun(2,3)
[1] 8
> MyFourthFun(2)
[1] 2 4 8
> MyFourthFun()
Error in MyFourthFun() : argument "n" is missing, with no default
>
```

Using Dot Argument in Function

- R allows you to add an additional value in the function without declaring an additional argument in it.
- This can be done by adding … (3 dots) argument
- **Example**

```
## Defining the function with the dot argument
dotArg<-function(a,b,…)
{
        res=a+b+…
        return(res)        # calculates the sum of arguments
}
```

```
> dotArg<-function(a,b,...){a+b+...}
> dotArg(2,3)
[1] 5
> dotArg(2,3,4)
[1] 9
> dotArg(2,3,6)
[1] 11
>
```

Passing Functions as Arguments

- R allows you to pass functions as one of the arguments in the user defined function.

  Example

  Here we are passing the **`round()`** function as an argument for rounding the area of circle

```
> x2<-3.12*2^2
> x2
[1] 12.48
> round(x2)
[1] 12
> funArg<-function(r,pi,FUN=round){pi=3.12
+ area=FUN(pi*r^2)
+ return(area)
+ }
> funArg(1)
[1] 3
> funArg(2)
[1] 12
>
```

Anonymous Functions

- Functions created without any names are called Anonymous Functions
- These functions are used to write one-line codes
- Any anonymous functions can be written like argument in a function

Syntax

Example

```
function(argument list) expression
```

```
# Anonymous function syntax
(function(x)  x * 10) (10)
```

```
# equivalent (normal) way
fun<-function(x) x * 10 fun(10)
```

THANK YOU !!!