R Introduction and Features

R is a free software programming language and software environment for statistical computing and graphics. The R language is widely used among statisticians and data miners for developing statistical software and data analysis and visualization. Let us now look at some key capabilities of R:

- R is easily extensible through functions and extensions.
- It provides effective data handling and manipulation components.
- It provides tools specific to a wide variety of data analysis.
- R has several operators for calculations on arrays and other types of ordered data.
- It provides advanced visualization capabilities through different tools available.

Moving from Scripts to Functions in R

Before starting with R functions, let us revise R scripting. R Functions provide 2 major advantages over script:

- Functions can work with variable input, so we can use it with different data.
- It returns the output as an object, so you can work with the result of that function.

a. How to create a Script in R?

Let us now learn how to create an R script.

As we know, R supports several editors. So the script can be created in any of the editors like Notepad, MS Word or Word Pad and saved with .R extension in the currently working directory.

Now to read the file into R, source function can be used.

For examples, if we want to read the sample.R script in R, we need to provide below command:

```
> source(“sample.R”)
```

This will read the file sample in R.

To create any script in R, you need to open a new script file in an editor and type the code.

Let us see how to create a script that presents fractional numbers as percentages rounded to one decimal digit.

```
x <- c(0.458, 1.6653, 0.83112)
  //Represents the values you want to
present as percentage
percent <- round(x * 100, digits = 1)
  //Round the result to 1 decimal place
result <- paste(percent, “%”, sep = “”)
 //Pastes a percentage sign after the
rounded number
print(result)                    // Prints the
desired result
```

i.e, you need to give values you want to convert to a percentage as input and then convert it into percentage and round to required places. Then put % sign and display the answer.

Save the above script as script file with any name for example pastePercent.R

Now you can call this script on the console with the help of source command which we have already seen.

```
> source(‘pastePercent.R’)
```

The output gets displayed as below:

```
[1] “45.8%” “166.5%” “83.1%”
```

This is the how a script is written in R and how to execute R Script.

b. Transforming the Script into Function

Now when we have seen how to write a script and run a script in R, we are going to see how to convert R script into the function in R.

Firstly define a function with a name so that it becomes easier to call a function and pass arguments to it as input.

The function should be followed by parentheses that act as a front gate for your function and between the parentheses, arguments for the function are provided.

Use the return() statement that acts as a back gate of your function.

The return() statement provides the final result of the function that is returned to your workspace.

We will see this with the example below.

Let us now see how we can convert the script we had written earlier to convert values to percentage and round off into a function.

```
addPercent <- function(x){
named x.
percent <- round(x * 100, digits = 1)
result <- paste(percent, “%”, sep = “”)
return(result)
}
```

The keyword **function** defines the starting of function. The **parentheses** after the function form the front gate, or argument list, of the function. Between the parentheses are the arguments to the function. In this case, there is only one argument.

The **return** statement defines the end of the function and returns the result. The object put between the parentheses is returned from inside the function to the workspace. Only one object can be placed between the parentheses.

The braces, {}, are the walls of the function. Everything between the braces is part of the assembly line or the body of the function. This is how functions are created in R.

4. Using Functions in R

After transforming the script into a function, you need to save it and you can use it in R again if required.

As R does not let you know by itself that it loaded the function but it is present in the workspace, if you want you can check it by using **ls()** command as below:

```
> ls()
```

It will display complete list as below:

```
[1] “addPercent” “percent” “result” “x”
```

Now as we know what all functions in R are present in the memory, we can use it when required.

For example, if you want to create percentage from values again, you can use add percent function for the same as below:

```
> new.numbers <- c(0.8223, 0.02487, 1.62,
0.4)    // Inserts new numbers

```

```
> addPercent(new.numbers)      // Uses the
addPercent() function
```

This will give output as:

```
[1] "82.2%" "2.5%" "162%" "40%"      //Shows
the output of the code
```

5. Using the Function Objects in R

In R, A function is also an object and you can manipulate it as you do other objects.

You can assign a function to new object using below command:

```
> ppaste <- addPercent
```

Now **ppaste** is a function as well that does exactly the same as addPercent. Note that you do not add after parentheses addPercent in this case. If you add the parentheses, you call the function and put the result of that call in ppaste. If you do not add the parentheses, you refer to the function object itself without calling it.

Let us see how we can assign a function to a new object with the help of an example:

```
> ppaste           //Copies the function code of
addPercent into a new object as we have
already assigned addpercent function to
ppaste above.
> addPercent(new.numbers)      //Uses the
addPercent() function for new values
percent <- round(x * 100, digits = 1)
    //Rounds the result to one decimal place
result <- paste(percent, "%", sep = "")
  //Pastes a percentage sign after the
rounded number
return(result)                          //Returns the
result
}
```

6. Reducing the Number of Lines in R

As of now, we have seen how to convert the script into a function and how to assign a function to the new object. All these include a large number of lines to be written.

Let us now understand how we can reduce the number of lines in R?

There are basically 2 ways of doing it:

- Returning values by default
- Dropping {}

Let us see the above 2 ways in detail below:

a. Returning Values by Default in R

Till now in all above code, we have written return() function to return output. But in R this can be skipped as by default R returns the value of the last line of code in the function body.

Now the above code will become as:

```
addPercent <- function(x){          // Defines
the function
percent <- round(x * 100, digits = 1)
   //Rounds the result to one decimal place
paste(percent, “%”, sep = “”)}          //Pastes
the function into the console. There is no
need to use the return statement.
```

You need return if you want to exit the function before the end of the code in the body.

For example, you could add a line to the addPercent function that checks whether x is numeric, and if not, returns NULL, as shown in the following table:

```
addPercent <- function(x){      //Defines the
function
if( !is.numeric(x) ) return(NULL)   //Checks
whether x is numeric, and if not, returns
NULL
percent <- round(x * 100, digits = 1)
   //Rounds the result to one decimal place
paste(percent, “%”, sep = “”)}         //Pastes
the result into console
```

b. Dropping the {}

You can drop braces also in some cases though they form a **proverbial wall** around the function.

If a function consists of only one line of code, you can just add that line after the argument list without enclosing it in braces. R will see the code after the argument list as the body of the function.

Suppose, you want to calculate the odds from a proportion. You can write a function without using braces, as shown below:

```
> odds <- function(x) x / (1-x)
```

Here no braces are used to write the function.

7. Scope of R Functions

Every object you create ends up in this environment, which is also called the **global environment.** The workspace—or global environment—is the universe of the R user where everything happens.

There are 2 types of functions in R as explained below:

a. External Functions in R

If you use a function, the function first creates a temporary local environment. This local environment is nested within the global environment, which means that, from that local environment, you also can access any object from the global environment. As soon as the function ends, the local environment is destroyed along with all objects in it.

This is what external functions are.

If R sees any object name, it first searches the local environment. If it finds the object there, it uses that one else it searches in the global environment for that object.

b. Internal Functions in R

Using global variables in a function is not considered as good practice. Writing your functions in such a way that they need objects in the global environment is not efficient because you use functions to avoid dependency on objects in the global environment in the first place.

The whole concept behind R strongly opposes using global variables used in different functions. As a **functional programming language**, one of the main ideas of R is that the outcome of a function should not be dependent on anything but the values for the arguments of that function. If you give the arguments the same values, you will always get the same results.

This characteristic of R may strike you as odd, but it has its merits. Sometimes you need to repeat some calculations a few times within a function, but these calculations only make sense inside that function.

Below example shows using internal function:

```
calculate.eff <- function(x, y, control){
  //Defines the calculate.eff() function that
create a new local environment
min.base <- function(z) z—
mean(control)      //Defines the min.base()
function inside the local environment of the
calculate.eff() function to create a new local
environment
```

Inside the calculate.eff() function, there is another function definition for a min.base() function.

Exactly as in the case of other objects, this function is created in the local environment of calculate.

eff() and destroyed again when the function is done. You will not find min.base() back in the workspace.

```
min.base(x) / min.base(y)}
```

The code for calculate.eff function is shown below:

```
> half <- c(2.23, 3.23, 1.48)
> full <- c(4.85, 4.95, 4.12)
> nothing <- c(0.14, 0.18, 0.56, 0.23)
> calculate.eff(half, full, nothing)
```

This gives output as below:

```
[1] 0.4270093 0.6318887 0.3129473
```

A closer look at the function definition of min.base() shows that it uses an object control but does not have an argument with that name.

8. Finding the Methods behind the Function

It is easy to find out the function you used in R. You can just look at the function code of print() by typing its name at the command line.

The command for displaying info code of the print() function is as below:

```
> print              //Command to access the
code of the print function.
function (x, ...)         //Shows the used
method that is print in this case.
UseMethod(“print”)
<bytecode: 0x0464f9e4>    //Shows
additional information about the print method.
<environment: namespace:base>
```

a. The UseMethod() Function

How can that one line of code in the print() function do so many complex things, such as printing vectors, data frames, and lists, all in a different way? The answer is contained in **UseMethod()**, which is the central function in the generic function system of R.

The UseMethod() function moves along and looks for a function that can deal with the type of object that is given as the argument x.

Suppose you have a data frame you want to print. R will look up the function **print.data.frame()** and use that function to print the object you passed as an argument.

R does that by looking through the complete set of functions in search of another function that starts with print followed by a dot and then the name of the object type.

You can also call the function print.data.frame() inside your code as well. This is explained in coming section.

b. Calling Functions Inside Code

You can also call the function print.data.frame() yourself.

Below is the example for the same:

```
> small.one <- data.frame(a = 1:2, b = 2:1)
  //Defines the small.one data frame.
> print.data.frame(small.one)    //Defines the
print.data.frame() function to print the
small.one object.
a b                          //Displays the result as
below:
1 1 2
2 2 1
```

9. Using Default Methods in R

R provides the feature to create an object with the names that are already used by R. It is possible with the use of default keyword.

R will ignore the type of the object in that case and just look for a default method if you use the default keyword with the name of an object.

Below example explains it:

```
> print.default(small.one)           //specifies
the default print method
$a                              // Prints data frame as a
list
[1] 1 2
$b
[1] 2 1
attr( ,"class")
[1]  "data.frame"
```

If you like this post and have any query about the functions in R, so, do let me know by leaving a comment