

# TMC 204

## Statistical Data Analysis with R

### Unit 5

### Functions in R

### (numeric, character, statistical)

Presented By : Aditya Joshi

Asst. Professor

Department of Computer Application

Graphic Era Deemed to be University

16-04-2020

# Functions in R:

## Moving from Scripts to R Function

R function provides two major advantages over the script:

1. Functions can work with any input. You can provide diverse input data to the functions.
2. The output of the function is an object that allows you to work with the result.

## How to Create a Script in R?

We will now learn the methodology of creating a script in R.

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 can be saved with R extension in the current working directory.

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

For example, 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.

In order to create a script, first, open a script file in the editor mode and type the required code.

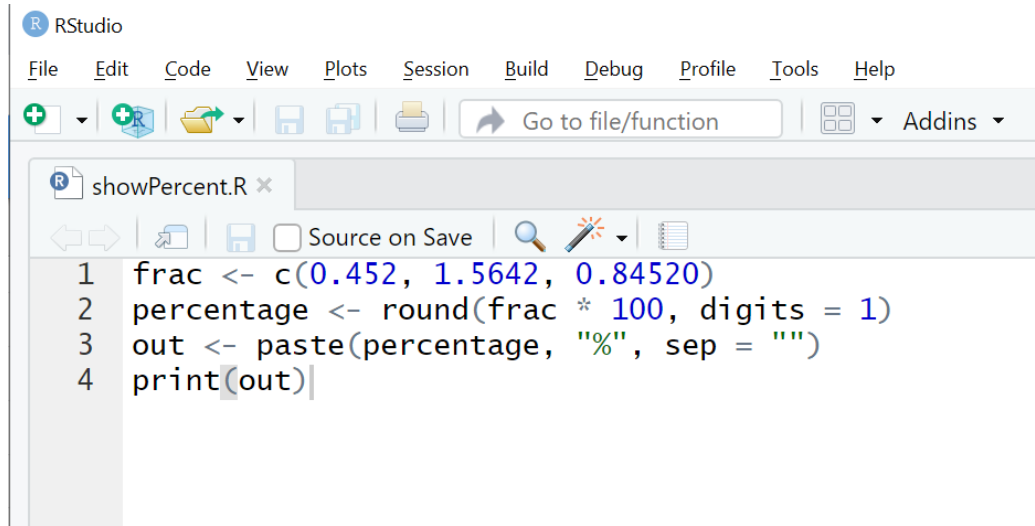
We will create a script that takes in input in the form of fractions and converts it into a percentage by further rounding it to one decimal digit

```
> frac <- c(0.452, 1.5642, 0.84520)
> percentage <- round(frac * 100, digits = 1)
> out <- paste(percentage, "%", sep = "")
> print(out)
[1] "45.2%" "156.4%" "84.5%"
```

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

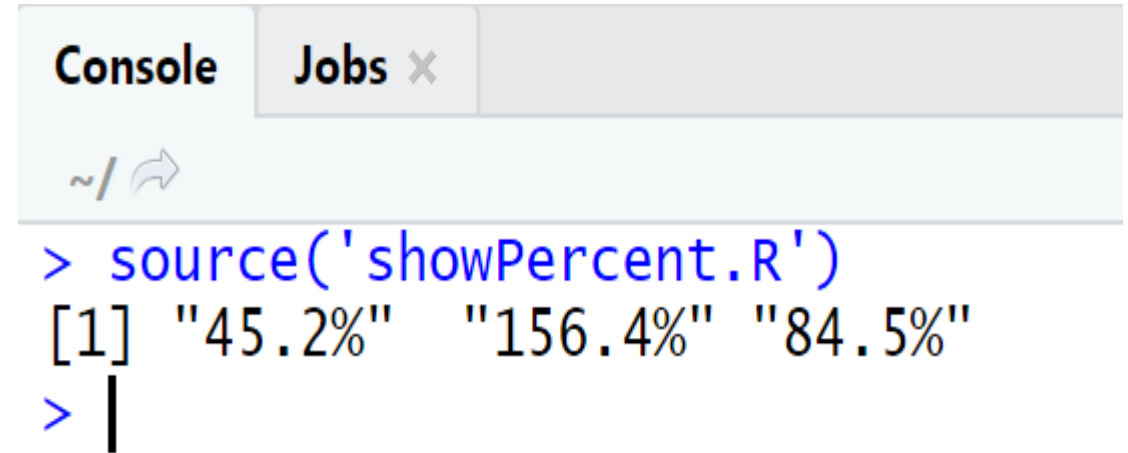
Save the above script as script file with any name for example showPercent.R.

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



The image shows the RStudio interface with a script editor open. The script is named 'showPercent.R' and contains the following R code:

```
1 frac <- c(0.452, 1.5642, 0.84520)
2 percentage <- round(frac * 100, digits = 1)
3 out <- paste(percentage, "%", sep = "")
4 print(out)
```



The image shows the RStudio console with the following commands and output:

```
> source('showPercent.R')
[1] "45.2%" "156.4%" "84.5%"
> |
```

**You may click on source to execute or use source ('scriptname.R') This is how a script is written and executed in R.**

## **Transforming the Script into R Function:**

Now, 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 an R function and pass arguments to it as input.

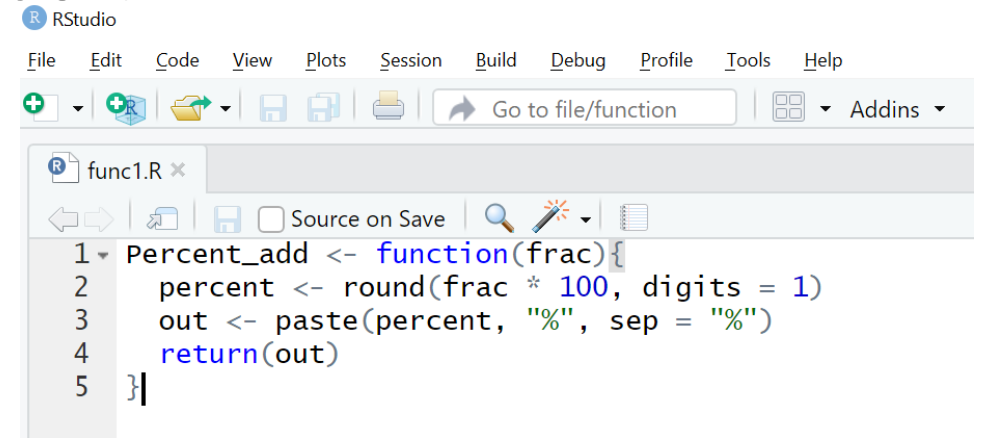
The R function should be followed by parentheses that act as a front gate for your function and between the parentheses, arguments for the function are provided.

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.

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

```
Percent_add <- function(frac){  
  percent <- round(frac * 100, digits = 1)  
  out <- paste(percent, "%", sep = "%")  
  return(out)  
}
```

A screenshot of the RStudio IDE interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for adding files, saving, and other standard IDE functions. The main editor window is titled 'func1.R' and contains the following R code:

```
1 Percent_add <- function(frac){  
2   percent <- round(frac * 100, digits = 1)  
3   out <- paste(percent, "%", sep = "%")  
4   return(out)  
5 }
```

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.

## Using R Function:

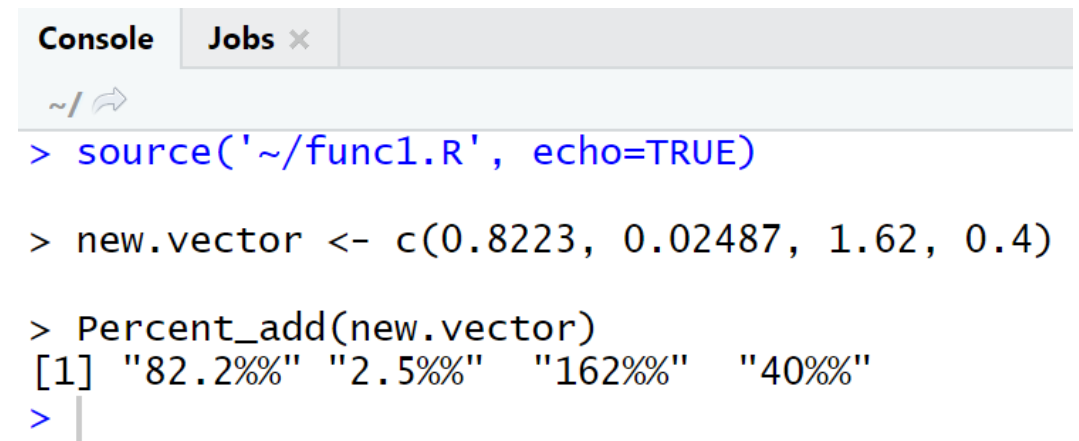
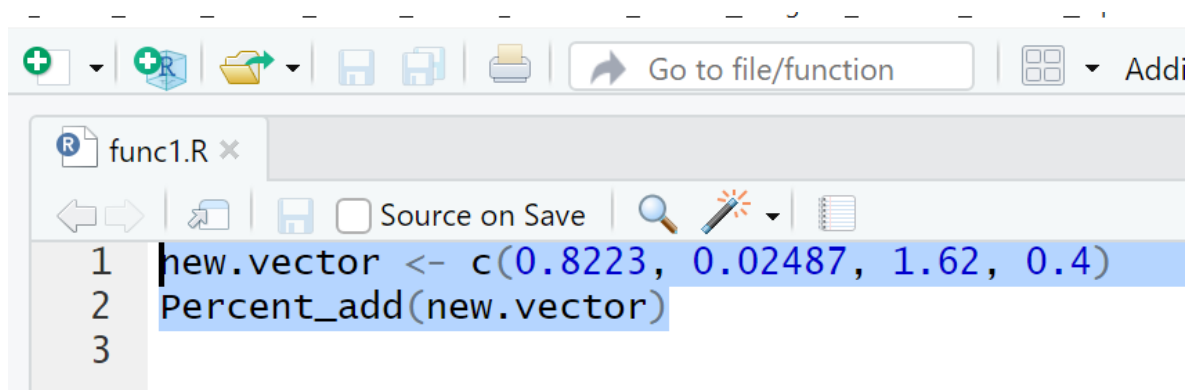
After transforming the script into an R function, you need to save it and you can use the function in R again if required.

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.

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

```
new.vector <- c(0.8223, 0.02487, 1.62, 0.4)
```

```
Percent_add(new.vector)
```



## Using the Function Objects in R:

In R, a function is also an object and you can manipulate it as you do for other objects. You can assign a function to the new object using below command:

```
percent_paste <- Percent_add
```

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

```
percent_paste
```

```
Percent_add <- function(frac){  
  percentage <- round(frac * 100, digits = 1)  
  out <- paste(percentage, "%", sep = "")  
  return(out)  
}  
print(Percent_add(new.vector))
```



RStudio

File Edit Code View Plots Session Build Debug Profile Tools Help

+ +R 📁 Save Print Go to file/function Addins

Func2.R x

← → 📄 Source on Save 🔍 🛠️ 📝

```
1 percent_paste <- Percent_add
2 percent_paste
3 Percent_add <- function(frac){
4   percentage <- round(frac * 100, digits = 1)
5   out <- paste(percentage, "%", sep = "")
6   return(out)
7 }
8 print(Percent_add(new.vector))
```

RStudio

File Edit Code View Plots Session Build Debug Profile Tools Help

+ +R 📁 Save Print Go to file/function Addins

Source

Console Jobs x

~/ ➡

```
> source('~ / Func2.R', echo=TRUE)

> percent_paste <- Percent_add

> percent_paste
function(frac){
  percentage <- round(frac * 100, digits = 1)
  out <- paste(percentage, "%", sep = "")
  return(out)
}

> Percent_add <- function(frac){
+   percentage <- round(frac * 100, digits = 1)
+   out <- paste(percentage, "%", sep = "")
+   return(out)
+ }

> print(Percent_add(new.vector))
[1] "82.2%" "2.5%" "162%" "40%"
>
```

## 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.

So, let us now understand how we can reduce the number of lines in R.

There are basically two ways of doing it:

1. Returning values by default
2. Dropping {}

### 1. Returning Values by Default in R

Till now, in all the 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 R function body.

```
Percent_add <- function(fac){  
percentage <- round(fac * 100, digits = 1)  
paste(percentage, "%", sep = "")}
```

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 Percent\_add function that checks whether fac is numeric, and if not, returns NULL

```
Percent_add <- function(frac){  
  if( !is.numeric(frac) ) return(NULL)  
  percentage <- round(frac * 100, digits = 1)  
  paste(percentage, "%", sep = "")}
```

## 2. Dropping the {}

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

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

## Scope of R Function

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 two types of R functions as explained below:

### 1. External R Function

If you use an R 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 the objects in it.

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.

### 2. Internal R Function

Using global variables in an R function is not considered a 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 for 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.

```
calculate_func <- function(data1, data2, data3){  
  base_min <- function(z) z - mean(data3)  
  base_min(data1) / base_min(data2)  
}
```

CODE FOR CALCULATING calculate\_func is

```
> d1 <- c(3.24, 2.21, 1.45)  
> d2 <- c(4.65, 5.12, 4.23)  
> d3 <- c(0.11, 0.20, 0.49, 0.28)  
> calculate_func(d1,d2,d3)
```

## Function in R:

- A function is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.
- In R, a function is an object so the R 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 its task and returns control to the interpreter as well as any result which may be stored in other objects.

## Function Definition

An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows –

```
function_name <- function(arg_1, arg_2, ...) {  
    Function body  
}
```

# Function Components

The different parts of a function are –

**Function Name** – This 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. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.

**Function Body** – The function body contains a collection of statements that defines what the function does.

**Return Value** – The return value of a function is the last expression in the function body to be evaluated.

R has many **in-built** functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as **user defined** functions.

## Built-in Function

Simple examples of in-built functions are **seq()**, **mean()**, **max()**, **sum(x)** and **paste(...)** etc. They are directly called by user written programs.

# Create a sequence of numbers from 32 to 44.

```
print(seq(32,44))
```

# Find mean of numbers from 25 to 82.

```
print(mean(25:82))
```

# Find sum of numbers from 41 to 68.

```
print(sum(41:68))
```

Source

Console

Jobs x

~/ ↗

```
> print(seq(32,44))
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44
>
> print(mean(25:82))
[1] 53.5
> print(sum(41:68))
[1] 1526
> |
```



## User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used.



# Create a function to print squares of numbers in sequence.

```
new.function <- function(a) {  
  for(i in 1:a) {  
    b <- i^2  
    print(b)  
  }  
}
```





## Calling a Function

# Call the function new.function supplying 6 as an argument.



```
new.function(6)
```

 Go to file/function  Addins



squarenoseq.R x

 Source on Save   

```
1 # Create a function to print squares of numbers in sequence.
2 new.function <- function(a) {
3   for(i in 1:a) {
4     b <- i^2
5     print(b)
6   }
7 }
8
9 # Call the function new.function supplying 6 as an argument.
10 new.function(6)
```

10:16  (Top Level) 

Console Jobs x

```
> # Call the function new.function supplying 6 as an argument.
> new.function(6)
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
>
```

```
> new.function(6)
```

```
[1] 1
```

```
[1] 4
```

```
[1] 9
```

```
[1] 16
```

```
[1] 25
```

```
[1] 36
```

```
> |
```

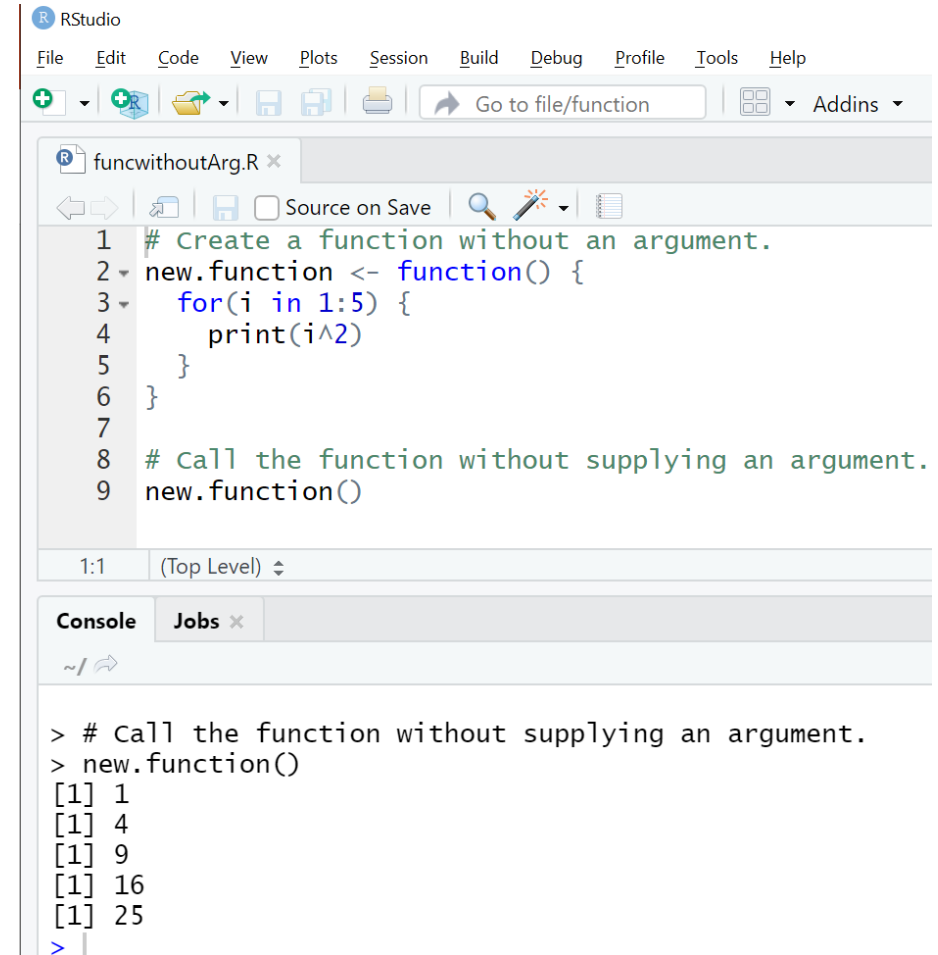
# Calling a Function without an Argument

# Create a function without an argument.

```
new.function <- function() {  
  for(i in 1:5) {  
    print(i^2)  
  }  
}
```

# Call the function without supplying an argument.

```
new.function()
```



The screenshot shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The main editor window, titled 'funcwithoutArg.R', contains the following R code:

```
1 # Create a function without an argument.  
2 new.function <- function() {  
3   for(i in 1:5) {  
4     print(i^2)  
5   }  
6 }  
7  
8 # Call the function without supplying an argument.  
9 new.function()
```

Below the editor is a console window with the following output:

```
> # Call the function without supplying an argument.  
> new.function()  
[1] 1  
[1] 4  
[1] 9  
[1] 16  
[1] 25  
>
```

## Calling a Function with Argument Values (by position and by name)

The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.

## # Create a function with arguments.

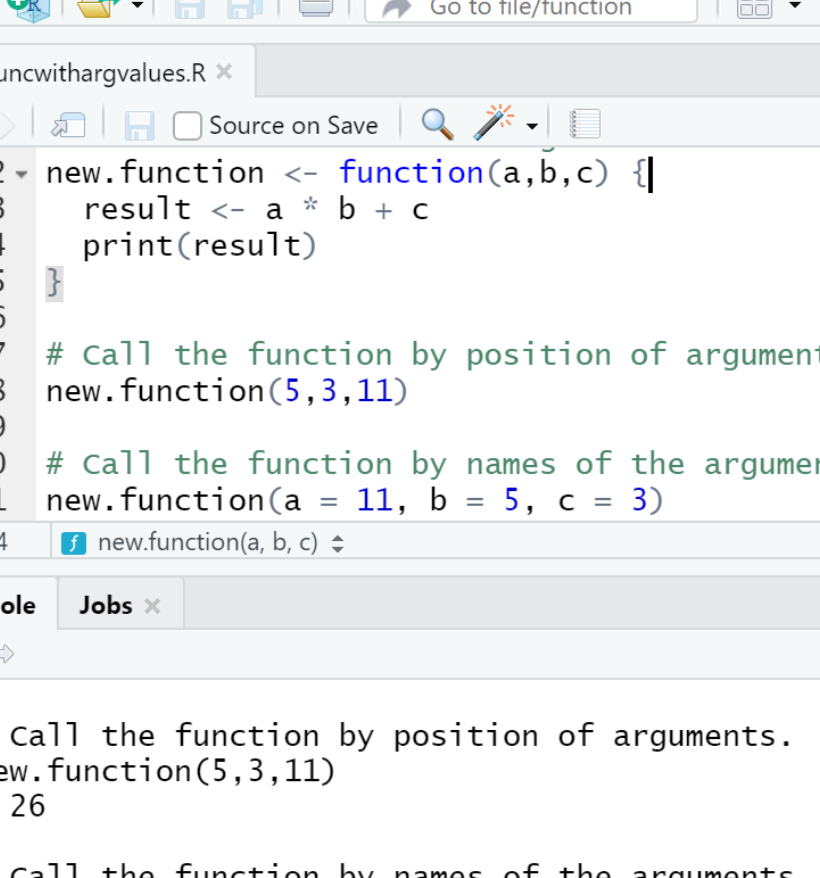
```
new.function <- function(a,b,c) {  
  result <- a * b + c  
  print(result)  
}
```

## # Call the function by position of arguments.

```
new.function(5,3,11)
```

## # Call the function by names of the arguments.

```
new.function(a = 11, b = 5, c = 3)
```



The screenshot displays the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for adding files, saving, printing, and navigating. The main editor window shows a script named 'funcwithargvalues.R' containing the following R code:

```
2 new.function <- function(a,b,c) {  
3   result <- a * b + c  
4   print(result)  
5 }  
6  
7 # Call the function by position of arguments.  
8 new.function(5,3,11)  
9  
10 # Call the function by names of the arguments.  
11 new.function(a = 11, b = 5, c = 3)
```

The status bar at the bottom of the editor shows the cursor is at line 2, column 34, and the function being called is 'new.function(a, b, c)'.

The Console window at the bottom shows the output of the function calls:

```
> # Call the function by position of arguments.  
> new.function(5,3,11)  
[1] 26  
  
> # Call the function by names of the arguments.  
> new.function(a = 11, b = 5, c = 3)  
[1] 58  
>
```

# Calling a Function with Default Argument

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.

# Create a function with arguments.

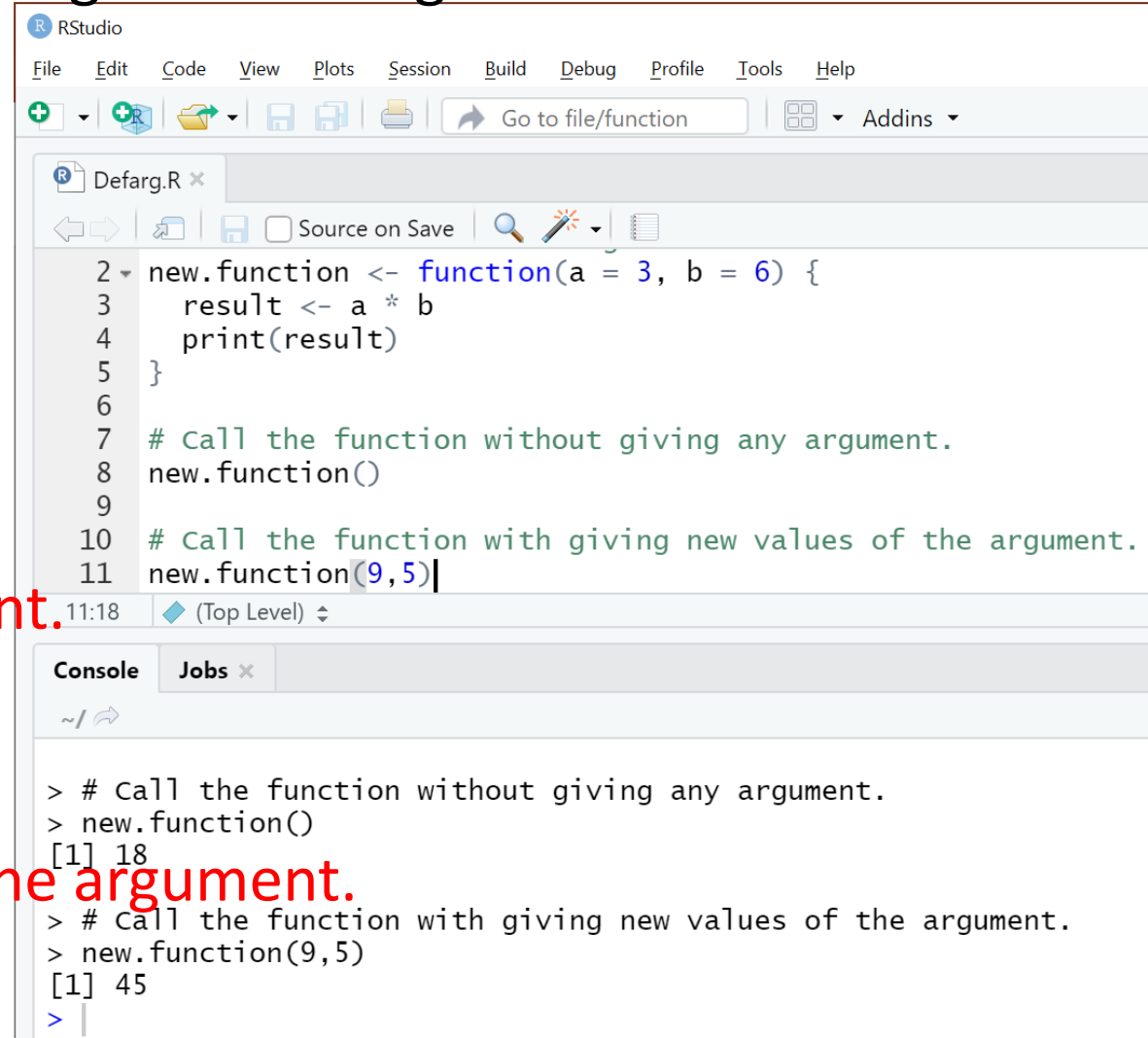
```
new.function <- function(a = 3, b = 6) {  
  result <- a * b  
  print(result)  
}
```

# Call the function without giving any argument.

```
new.function()
```

# Call the function with giving new values of the argument.

```
new.function(9,5)
```



The screenshot shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The main editor window displays the R script 'Defarg.R' with the following code:

```
2 new.function <- function(a = 3, b = 6) {  
3   result <- a * b  
4   print(result)  
5 }  
6  
7 # Call the function without giving any argument.  
8 new.function()  
9  
10 # Call the function with giving new values of the argument.  
11 new.function(9,5)
```

The console window at the bottom shows the output of the function calls:

```
> # Call the function without giving any argument.  
> new.function()  
[1] 18  
> # Call the function with giving new values of the argument.  
> new.function(9,5)  
[1] 45  
>
```

# Lazy Evaluation of Function

Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.

# Create a function with arguments.

```
new.function <- function(a, b) {
```

```
  print(a^2)
```

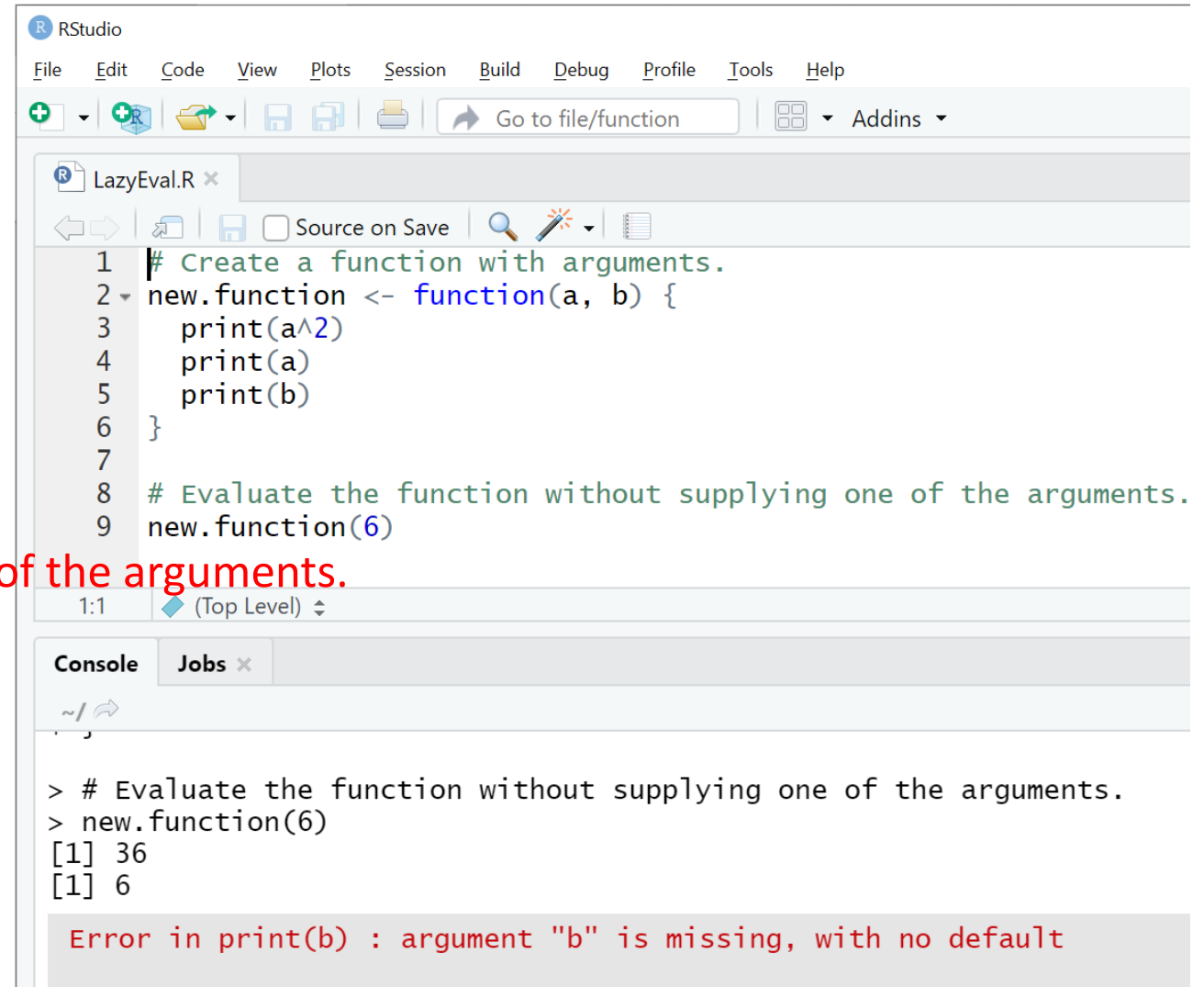
```
  print(a)
```

```
  print(b)
```

```
}
```

# Evaluate the function without supplying one of the arguments.

```
new.function(6)
```



The screenshot shows the RStudio interface. The script editor displays the following code:

```
1 # Create a function with arguments.
2 new.function <- function(a, b) {
3   print(a^2)
4   print(a)
5   print(b)
6 }
7
8 # Evaluate the function without supplying one of the arguments.
9 new.function(6)
```

The console output shows the execution of the function:

```
> # Evaluate the function without supplying one of the arguments.
> new.function(6)
[1] 36
[1] 6
```

An error message is displayed at the bottom of the console:

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