R Environment and Scope (With Examples)

R Environment and Scope (With Examples)

In this article, you'll learn about the environment (global environment, cascading of environments and so on) in R programming. You will also learn about scope of variables with the help of examples.

In order to write functions in a proper way and avoid unusual errors, we need to know the concept of environment and scope in R.

R Programming Environment

Environment can be thought of as a collection of objects (functions, variables etc.). An environment is created when we first fire up the R interpreter. Any variable we define, is now in this environment.

The top level environment available to us at the R command prompt is the global environment called `R_GlobalEnv`. **Global environment** can be referred to as `.GlobalEnv` in R codes as well.

We can use the `ls()` function to show what variables and functions are defined in the current environment. Moreover, we can use the `environment()` function to get the current environment.

```
> a <- 2
> b <- 5
> f <- function(x) x<-0
> ls()
[1] "a" "b" "f"
> environment()
<environment: R_GlobalEnv>
> .GlobalEnv
<environment: R_GlobalEnv>
```

In the above example, we can see that `a`, `b` and `f` are in the `R_GlobalEnv` environment.

Notice that `x` (in the argument of the function) is not in this global environment. When we define a function, a new environment is created.

In the above example, the function `f` creates a new environment inside the global environment.

Actually an environment has a frame, which has all the objects defined, and a pointer to the enclosing (parent) environment.

Hence, `x` is in the frame of the new environment created by the function `f`. This environment will also have a pointer to `R_GlobalEnv`.

Example: Cascading of environments

```
f <- function(f_x){
g <- function(g_x){
print("Inside g")
print(environment())
print(ls())
}
g(5)
print("Inside f")
print(environment())
print(ls())
}
```

Now when we run it from the command prompt, we get.

```
> f(6)
[1] "Inside g"
<environment: 0x0000000010c2bdc8>
[1] "g_x"
[1] "Inside f"
<environment: 0x0000000010c2a870>
[1] "f_x" "g"
> environment()
<environment: R_GlobalEnv>
> ls()
[1] "f"
```

Here, we defined function `g` inside `f` and it is clear that they both have different environments with different objects within their respective frames.

R Programming Scope

Let us consider the following example.

```
outer_func <- function(){
b <- 20
inner_func <- function(){
c <- 30
}
}
a <- 10
```

Global variables

Global variables are those variables which exists throughout the execution of a program. It can be changed and accessed from any part of the program.

However, global variables also depend upon the perspective of a function.

For example, in the above example, from the perspective of `inner_func()`, both `a` and `b` are **global** variables.

However, from the perspective of `outer_func()`, `b` is a local variable and only `a` is global variable. The variable `c` is completely invisible to `outer_func()`.

Local variables

On the other hand, Local variables are those variables which exist only within a certain part of a program like a function, and is released when the function call ends.

In the above program the variable `c` is called a local variable.

If we assign a value to a variable with the function `inner_func()`, the change will only be local and cannot be accessed outside the function.

This is also the same even if names of both global variable and local variables matches.

For example, if we have a function as below.

```
outer_func <- function(){
a <- 20
inner_func <- function(){
a <- 30
print(a)
}
inner_func()
print(a)
}
```

When we call it,

```
> a <- 10
> outer_func()
[1] 30
[1] 20
> print(a)
[1] 10
```

We see that the variable `a` is created locally within the environment frame of both the functions and is different to that of the global environment frame.

Accessing global variables

Global variables can be read but when we try to assign to it, a new local variable is created instead.

To make assignments to global variables, superassignment operator, <<-, is used.

When using this operator within a function, it searches for the variable in the parent environment frame, if not found it keeps on searching the next level until it reaches the global environment.

If the variable is still not found, it is created and assigned at the global level.

```
outer_func <- function(){
inner_func <- function(){
a <<- 30
print(a)
}
inner_func()
print(a)
}
```

On running this function,

```
> outer_func()
[1] 30
[1] 30
> print(a)
[1] 30
```

When the statement `a <<- 30` is encountered within `inner_func()`, it looks for the variable `a` in `outer_func()` environment.

When the search fails, it searches in `R_GlobalEnv`.

Since, `a` is not defined in this global environment as well, it is created and assigned there which is now referenced and printed from within `inner_func()` as well as `outer_func()`.