Basic Course on R: Programming Structures 2

Elizabeth Ribble*

28 Oct - 1 Nov 2019

Contents

*emcclel3@msudenver.edu

1 Environment and Scope Issues

- Each function, whether built-in or user-defined, has an associated ***environment***, which can be thought of as a container that holds all of the objects present at the time the function is created.

1.1 The Top-Level (or Global) Environment

- When a function is created on the command line, it's environment is the so-called ***Global Environment*** (or ***Workspace***):

```
w <- 2
```

```
f <- function(y) {
  d <- 3
  return(d * (w + y))
}
```

```
environment(f)

## <environment: R_GlobalEnv>
```

- The function `objects()` (or `ls()`), when called from the command line, lists the objects in the Global Environment:

```
objects()

## [1] "f" "w"
```

1.2 Global and Local Variables

- In the function `f()` defined above, the variable `w` is said to be ***global*** to `f()` and the variable `d`, because it's created *within* `f()`, is said to be ***local***.

- Global variables (like `w`) are visible from within a function, but local variables (like `d`) aren't visible from outside the function. In fact, local variables are *temporary*, and disappear when the function call is completed:

```
f(y = 1)

## [1] 9

d

## Error in eval(expr, envir, enclos):  object 'd' not found
```

- When a global and local variable share the same name, the local variable is used:

```
w <- 2
d <- 4
f <- function(y) {
  d <- 3
  return(d * (w + y))
}
f(y = 1)

## [1] 9
```

- Note also that when an assignment takes place within a function, and the local variable shares its name with an existing global variable, only the local variable is affected:

```
w <- 2
d <- 4                  # This value of d will remain unchanged.
f <- function(y) {
  d <- 3                # This doesn't affect the value of d in the
  return(d * (w + y))   # global environment (Workspace)
}
f(y = 1)

## [1] 9

d

## [1] 4
```

1.3 Nested Functions and the Scope Hierarchy

- For user-defined functions created on the command line, the global variables for that function are those in the Workspace, or ***global environment***. They're listed by typing `ls()` (or `objects()`) on the command line.
- When a function is *created inside another function*, its global variables are the local variables of the outer function *plus* the outer function's global variables.
- Regardless of whether a function is created on the command line or inside another function, its local variables are the variables created inside of it *plus* its formal arguments to which values have been passed.
- For example:

```
w <- 2            # w is global to f() and therefore also to h()
f <- function(y) {
  d <- 3
  h <- function() {
    b <- 5              # b is local to h()
    return(d * (w + y))
    }
  return(h())
}
```

  Above,

  - `w` is global to `f()` and therefore also to `h()`.
  - `y` and `d` are local to `f()`, but global to `h()`.
  - `b` is local to `h()`.

- This ***scope hierarchy*** continues when multiple function definitions are ***nested*** inside of each other.

- We can use a `print(ls())` statement to see which objects are local to `f()`:

```
w <- 2                  # w is global to f() and therefore also to h()
f <- function(y) {
  d <- 3                # y and d are local to f() but global to h()
  h <- function() {
    b <- 5              # b is local to h()
    return(d * (w + y))
    }
  print(ls())
  return(h())
}
f(y = 2)

## [1] "d" "h" "y"
## [1] 12
```

- Likewise we can use a `print(environment(h))` statement to view the environment of `h()`:

```
w <- 2                  # w is global to f() and therefore also to h()
f <- function(y) {
  d <- 3                # y and d are local to f() but global to h()
  h <- function() {
    b <- 5              # b is local to h()
    return(d * (w + y))
    }
  print(environment(h))
  return(h())
}
f(y = 2)

## <environment: 0x48d6a00>
## [1] 12
```

In the output above, the environment of `h()` is referred to by its memory location. The *environment* of `h()` is the "container" that contains `h()` as well as the objects `d` and `y`.

1.4 Writing "Upstairs" in the Scope Hierarchy

- Sometimes we need to assign a value to a variable in the global environment from within a function. We can do so using either of the following:

```
<<-         # Assign a value to a variable in the global environment
            # (Workspace).
assign()    # Assign a value to a variable in the global environment
            # (Workspace).
```

- Here's an example using the so-called ***superassignment operator*** `<<-`:

```
w <- 2
d <- 4                  # This value of d will be replaced by 3.
f <- function(y) {
  d <<- 3               # This replaces the value 4 of d in the global
  return(d * (w + y)) # environment (Workspace) by 3.
}
f(y = 2)

## [1] 12

d

## [1] 3
```

Above, the assignment of `3` to `d` is done in the global environment, or Workspace, overwriting the previous value (`4`).

- Here's how to accomplish the same thing using `assign()`:

```
w <- 2
d <- 4                  # This value of d will be replaced by 3.
f <- function(y) {
  assign("d", 3, envir = .GlobalEnv)
  return(d * (w + y))
}
f(y = 2)

## [1] 12

d

## [1] 3
```

(Note that when `assign()` is used, `d` is written as a character string (i.e. in quotes as `"d"`) and the global environment is written as `.GlobalEnv`.)

- Be aware that assignment to the variable `d` using `<<-` actually results in a search up the environment hierarchy, stopping at the first level at which the name `d` is encountered. If it's not encountered, then assignment is done in the global environment. For example:

```
w <- 2
d <- 4                      # This value of d will remain unchanged.
f <- function(y) {
    d <- 5                  # This value of d will be replaced by 3.
    h <- function() {
        d <<- 3             # This replaces the value 5 of d in the
        return(d * (w + y)) # h() and f() environments by 3.
    }
    return(h())
}
f(y = 2)

## [1] 12

d

## [1] 4
```

1.5 When Should You Use Global Variables?

- Here are suggestions about using global variables. They're especially important when your code will be shared with other R users:
  - Assignment to the global environment using `<<-` or `assign()` should be used very sparingly (i.e. only when necessary) because it can accidentally overwrite existing variables.
  - The use of a global variable can be justified when that variable needs to be accessed by several different functions (that aren't nested).
  - It's generally preferable to pass variables as arguments to functions rather than accessing them from the global environment.

2 Printing a Warning Message or Terminating a Function Call Using `warning()`, `return()`, or `stop()`

- The following functions are useful for terminating a function call or just printing a warning message:

```
return()      # Terminate a function call and return a value.
stop()        # Terminate a function call and print an error message.
warning()     # Print a warning message (without terminating the
              # function call).
```

2.1 Terminating a Function Call Using `if()` and `return()`

- One way to terminate a function call is with `return()` which, when encountered, immediately terminates the call and returns a value. For example:

```
mySign <- function(x) {
  if(x < 0) return("Negative")
  if(x > 0) return("Positive")
  return("Zero")
}
```

Passing `my.sign()` the value `x = 13` produces the following:

```
mySign(x = 13)

## [1] "Positive"
```

(Note that the last line, `return("Zero")`, was never encountered during the call to `my.sign()`.)

2.2 Terminating a Function Call and Printing an Error Message Using `if()` and `stop()`

- Another way to terminate a function call is with `stop()`, which then prints an error message without returning a value. Here's an example:

```
myRatio <- function(x, y) {
  if(y == 0) stop("Cannot divide by 0")
  return(x/y)
}
```

An attempt to pass the value `0` for `y` now results in the following:

```
myRatio(x = 3, y = 0)

## Error in myRatio(x = 3, y = 0):  Cannot divide by 0
```

(Note that the last line, `return(x/y)`, was never encountered during the call to `myRatio()`.)

2.3 Printing a Warning Message Using `if()` and `warning()`

- `warning()` just prints a warning message to the screen without terminating the function call. Here's an example:

```
myRatio <- function(x, y) {
  if(y == 0) warning("Attempt made to divide by 0")
  return(x/y)
}
```

Now when we pass the value `0` for `y` the function call isn't terminated (the value `Inf` is returned), but we get the warning message:

```
myRatio(x = 3, y = 0)

## Warning in myRatio(x = 3, y = 0):  Attempt made to divide by 0

## [1] Inf
```

3 Recursion

- ***Recursion*** is a programming technique in which a function calls itself.
- Here's an example in which the function `f()` takes a non-negative integer `x` and returns the factorial of `x`, denoted $x!$ and defined as

$$x! = \begin{cases} 1 & \text{if } x = 0 \\ x(x-1)(x-2)\cdots(2)(1) & \text{if } x > 0 \end{cases}$$

- Notice that we can write

$$x! = \begin{cases} 1 & \text{if } x = 0 \\ x(x-1)! & \text{if } x > 0 \end{cases}$$

```
f <- function(x) {
  if(x == 0) {
    return(1)
    } else {
      return(x * f(x - 1))
      }
}
f(0)

## [1] 1

f(1)

## [1] 1

f(5)

## [1] 120
```

- In general, to solve a problem of type X by writing a recursive function `f()`:
  1. Break the original problem of type X into one or more smaller problems of type X.
  2. Within `f()`, call `f()` on each of the smaller problems.
  3. Within `f()`, piece together the results of Step 2 to solve the original problem.

4 Replacement Functions

- Some of R's built-in functions can be used both to *return* a value and to *replace* a value. For example using the data frame:

```
var1 <- c(1, 2, 3)
var2 <- c(19, 20, 16)
var3 <- c("small", "medium", "large")
x <- data.frame(var1, var2, var3)
x
```

```
##   var1 var2   var3
## 1    1   19  small
## 2    2   20 medium
## 3    3   16  large
```

`names()` will both return the names of the variables in `x`:

```
names(x)
```

```
## [1] "var1" "var2" "var3"
```

and replace them:

```
names(x) <- c("IDNumber", "Weight", "Size")
names(x)
```

```
## [1] "IDNumber" "Weight"   "Size"
```

- Such functions are called ***replacement functions***.
- It's possible to create user-defined replacement functions.