Advanced R - Functions

Anh Le

August 13, 2014

1 Function components

```
f <- function(x) x^2
f

## function(x) x^2

formals(f)

## $x

body(f)

## x^2

environment(f)

## <environment: R_GlobalEnv>
```

2 Dynamic lookup

R looks for values when the function is run, not when it's created.

To check whether a function depends on global variables, use `codetools::findGlobals(f)`.

3 Function argument

3.1 Calling a function given a list of arguments

Using `do.call()`

```
args <- list(1:10, na.rm=TRUE)
args
```

```
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $na.rm
## [1] TRUE
```

```
do.call(mean, args)
```

```
## [1] 5.5
```

```
mean(1:10, na.rm=TRUE) # Equivalent
```

```
## [1] 5.5
```

3.2 Lazy evaluation

By default, R function arguments are lazy – they are only evaluated if they are actually used.

The following happens because in each `adder`, `x` is not evaluated before we first call the first adder. When we do call it, the adder search its environment, see no `x`, then search its parent frame (where it is defined, i.e. the `lapply` environment), in which the loop has ended and `x = 10`. Therefore, for all the adders, `x = 10`.

Notice that each adder has a unique environment, but they all share one parent frame, the `lapply` environment.

```
x = 100
add <- function(x) {
  print(parent.frame())
  print(environment())
  function(y) x + y
}
adders <- lapply(1:10, add)
```

```
## <environment: 0x28020c8>
## <environment: 0x2803240>
## <environment: 0x28020c8>
## <environment: 0x2804ed8>
## <environment: 0x28020c8>
## <environment: 0x281d4c0>
## <environment: 0x28020c8>
## <environment: 0x2822fd8>
## <environment: 0x28020c8>
## <environment: 0x2829258>
## <environment: 0x28020c8>
```

```
## <environment: 0x282a3b0>
## <environment: 0x28020c8>
## <environment: 0x282d3f0>
## <environment: 0x28020c8>
## <environment: 0x282f440>
## <environment: 0x28020c8>
## <environment: 0x2832818>
## <environment: 0x28020c8>
## <environment: 0x2834830>

adders[[1]](10)

## [1] 20

adders[[10]](10)

## [1] 20
```

In the following, by forcing the evaluation of `x` in the adder's own environment, we make `x` available there (with the correct value) so the adder does not have to look up to the parent frame, where `x = 10`.

```
x = 100
add <- function(x) {
  force(x) # Equivalent to just x
  function(y) x + y
}
adders <- lapply(1:10, add)
adders[[1]](10)

## [1] 11

adders[[10]](10)

## [1] 20
```

3.3 Functions search for arguments and variables in its environment first. If they are not there, functions evaluate arguments in the parent frame (where it's called), but other variables in the parent environment (where it's defined)

Uncomment the assignment within `p()` to see that functions search for values within its environment first.

```r
p <- function(x) {
  # x <- "arg defined within p envir"
  # y <- "var defined within p envir"
  cat("Argument =", x, "\n")
  cat("Other variable = ", y, "\n")
}
p.outer <- function(x) {
  p(x)
}

y <- "var defined in the global envir"
p.outer("arg defined in the p.outer")

## Argument = arg defined in the p.outer
## Other variable =  var defined in the global envir
```

3.4 Exercises

1. Clarify the following list of odd function calls (basically just argument matching based on position, partial name, etc.)

2. What does this function return? Why? Which principle does it illustrate?

   It will return 3. When x is evaluated, it created both x and y in the environment. Thus the default value is never used.

   ```r
   f1 <- function(x = {y <- 1; 2}, y = 0) {
     x + y
   }
   f1()


   ## [1] 3
   ```

3. What does this function return? Why? Which principle does it illustrate?

   It will return 100. Lazy evaluation means x is not evaluated until it's called, by what time z already exists. Thus, `x = z` does not raise an error.

   ```r
   f2 <- function(x = z) {
     z <- 100
     x
   }
   f2()


   ## [1] 100
   ```

4 Special calls

4.1 Infix function

Taking a page from Ruby

```
`%||%` <- function(a, b) if (!is.null(a)) a else b
function_that_might_return_null() %||% default_value

## Error:  could not find function "function_that_might_return_null"
```

4.2 Replacement function

Replacement function typically has 2 arguments (x and value) and MUST return the modified object. Following is a function that modify the second element of a vector:

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:10
second(x) <- 5
x

##  [1]  1  5  3  4  5  6  7  8  9 10
```

There can be more than 2 arguments

```
`modify<-` <- function(x, position, value) {
  x[position] <- value
  x
}
x <- 1:10
modify(x, 1) <- 10
x

##  [1] 10  2  3  4  5  6  7  8  9 10
```

Behind the scene, `modify(x, 1) <- 10` gets translated to `x <- `modify<-`(x, 1, 10)`. Therefore, we can't do `modify(get("x"), 1) <- 10` because it gets translated to the invalid `get("x") <- `modify<-`(get("x"), 1, 10)`.

4.3 Exercises

1. Create a list of all the replacement functions found in the base package. Which ones are primitive functions?

2. What are valid names for user created infix functions?

3. Create an infix `xor()` operator

```
`%xor%` <- function(s1, s2) {
  result <- c()
  for (el in union(s1, s2)) {
    if (!(el %in% s1 & el %in% s2)) {
      result <- c(result, el)
    }
  }
  return(result)
}
c(1, 2, 6) %xor% c(2, 4, 7)

## [1] 1 6 4 7
```

4.4 Return values

Function can return `invisible` values, which are not printed out by default when you call the function.

You can force an invisible value to be displayed by wrapping it in parentheses

```
f1 <- function() 1
f2 <- function() invisible(1)
f1()

## [1] 1

f2()
(f2())

## [1] 1
```

4.5 On exit

The code in `on.exit()` is run regardless of how the function exits, whether with an explicit (early) return, an error, or simply reaching the end of the function body.

An example:

```
in_dir <- function(dir) {
  old <- getwd()
  on.exit(setwd(old))
```

```
  setwd(dir)
  print(getwd())
}
getwd()

## [1] "/home/anh/projects/learning/R/advanced_R"

in_dir("~")

## [1] "/home/anh"

getwd()

## [1] "/home/anh/projects/learning/R/advanced_R"
```