Biostatistics 615 - Statistical Computing

Lecture 10 Advanced R – Functions

Jian Kang

Oct 27, 2015

Function components

All R functions have three parts

- The code inside the function: `body()`
- The list of arguments which controls how you can call the function: `formals()`
- The "map" of the location of the function's variables: `environment()`

```
> f = function(x) x^2
> f
function(x) x^2
> formals(f)
$x



> body(f)
x^2
> environment(f)
<environment: R_GlobalEnv>
```

There is one exception to the rule that functions have three components `Primitive functions`, like `sum()`, call `C` code directly with `.Primitive()` and contain no `R` code.

Lexical scoping

Scoping is the set of rule that govern how R look up the value of a symbol.
There are four basic principles behind R's implementation of lexical scoping

- Name masking
- Functions versus. variables
- A fresh start
- Dynamic lookup

Name masking

- Look up within function first

```
f = function(){
   x = 1
   y = 2
   c(x,y)
}
f()
rm(f)
```

- If a name is not defined inside a function, R will look one level up

```
x = 2
f = function(){
   y = 1
   c(x,y)
}
g()
rm(x,g)
```

Functions versus Variables

- The same principles apply regardless of the type of associated value: finding functions works exactly the same way as finding varaibles:

```
l = function(x) x + 1
m = function(){
   l = function(x) x*2
   l(10)
}
m()
```

- For functions, there is one small tweak to the rule. If you are using a name in a context where it's obvious that you want a function (e.g. $f(3)$), R will ignore objects that are not functions while it is searching.

```
n = function(x) x/2
o = function(){
     n = 10
     n(n)
}
o()
rm(n,n)
```

- However, using the same name for functions and other objects will make for confusing code, and is generally best avoided.

A fresh start

- What happens to the values in between invocations of a function?

```
j = function(){
   if(!exists("a")){
      a = 1
   } else {
      a = a + 1
   }
   print(a)
}
j()
rm(j)
```

- A function has no way to tell what happened the last time it was run;

Dynamic lookup

Lexical scoping determines where to look for values, not when to look for them. R looks for values when the function is run, not when it's created.

```
f = function() x+2
x = 10
f()
x = 2
f()
```

Bad example

Since all of the standard operators in R are functions, you can override them with your own alternatives. If you ever are feeling particularly evil, run the following code while your friend is away from their computer

```
function (e1, e2)  .Primitive("+")
> `(` = function(e1){
+     if( is.numeric(e1) && runif(1)<0.1){
+         e1 + 1
+     }
+     else{
+         e1
+     }
+ }
> replicate(50, (1+2))
 [1] 3 3 3 3 3 3 3 4 4 3 3 3 4 3 3 3 3 3 3 3 3 3 3 3 4 3
[27] 3 3 3 3 3 3 3 3 3 3 3 3 3 4 3 3 4 3 4 3 3 4 3 3
> rm("(")
```

Every operation is a function call

```
> x = 10; y  = 5
> x + y
[1] 15
> `+`(x,y)
[1] 15

> for(i in 1: 2) print(i)
[1] 1
[1] 2
> `for`(i, 1:2, print(i))
[1] 1
[1] 2

> x[3]
[1] NA
> `[`(x,3)
[1] NA
```

Useful applications

```
> x = list(1:3, 4:9,10:12)
> sapply(x, "[", 2)
[1]  2  5 11
> sapply(x, function(x) x[2])
[1]  2  5 11
```

Function arguments

- How calling arguments are mapped to formal arguments
- How you can call a function given a list of arguments
- How default arguments work
- The impact of lazy evaluation

Calling functions

When calling a function you can specify arguments by position, by complete name, or by partial name. Arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position

```
> f = function(abcdef, bcde1, bcde2){
+     list(a = abcdef, b1 = bcde1, b2 = bcde2)
+ }
>
> str(f(1,2,3))
> str(f(2,3, abcdef = 1))
> str(f(2,3,a = 1))
List of 3
 $ a : num 1
 $ b1: num 2
 $ b2: num 3
>
> str(f(1,3,b = 1))
Error in f(1, 3, b = 1) : argument 3 matches multiple formal arguments
```

Calling a function given a list of argments

Suppose you had a list of function arguments

```
> args = list(1:100, na.rm=TRUE)
> do.call(mean, args)
[1] 50.5
```

Default arguments

Function arguments in R can have default values

```
> f = function(a = 1, b = 2){
+     c(a, b)
+ }
> f()
[1] 1 2
```

Since arguments in R are evaluated lazily, the default value can be defined in terms of other arguments

```
> f = function(a = 1, b = 2*a){
+     c(a, b)
+ }
> f()
[1] 1 2
> h = function(a = 1, b = d){
+     d = (a + 1)^2
+     c(a, b)
+ }
> h()
[1] 1 4
```

Missing arguements

You can determine if an argument was supplied or not with `missing()` function

```
> i  = function(a, b){
+     c(missing(a), missing(b))
+ }
> i()
[1] TRUE TRUE
> i(a = 1)
[1] FALSE  TRUE
> i(b = 2)
[1]  TRUE FALSE
> i(1, 2)
[1] FALSE FALSE
```

Sometimes you want to add a non-trival default value, which might take several lines of code to compute. Instead of inserting that code in the function definition, you could use `missing()` to conditionally compute if needed.

Alternatively, you can set the default value to NULL and use `is.null()` to check if the argument was supplied.

Lazy evaluation

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

```
> f = function(x){
+     10
+ }
> f(stop("This is an error !!"))
[1] 10

> f = function(x){
+     force(x)
+     10
+ }
> f(stop("This is an error !!"))
Error in force(x) : This is an error !!
```