

Biostatistics 615 - Statistical Computing

Lecture 11 Advanced R – Functions II

Jian Kang

Oct 29, 2015

Infix functions

- Most functions in R are “prefix” operators: the name of the function comes before the arguments.
- You can also create infix functions where the function name comes in between its arguments, like `+` or `-`.
- All user-created infix functions must start and end with `%`. R comes with the following infix functions predefined:

`%\%`, `%*%`, `%/%`, `%in%`, `%o%`, `%x%`.

- The complete list of built-in infix operators that do not need `%` is

`:`, `::`, `:::`, `$`, `@`, `^`, `*`, `/`, `+`, `-`, `>`, `>=`, `<`, `<=`, `==`, `!=`,
`!`, `&`, `&&`, `|`, `||`, `~`, `<-`, `<<-`

```
> `%+%` = function(a, b) paste0(a, b)
> "new" +% " string"
[1] "new string"
```

Infix functions

The names of infix functions are more flexible than regular R functions: they can contain any sequence of characters (except `%`, of course). You will need to escape any special characters in the string used to define the function, but not when you call it:

```
> `% %` = function(a, b) paste(a, b)
> `%'` = function(a, b) paste(a, b)
> `%/\\%` = function(a, b) paste(a, b)
>
> "a" % % "b"
[1] "a b"
> "a" %' "b"
[1] "a b"
> "a" %/\\% "b"
[1] "a b"
```

R's default precedence rules mean that infix operators are composed from left to right:

```
> `%-` = function(a, b) paste0("(", a, " %-", b, ")")
> "a" %- "b" %- "c"
[1] "((a %- b) %- c)"
```

Replacement functions

- Replacement functions act like they modify their arguments in place, and have the special name `xxx<-`.
- They typically have two arguments (`x` and `value`), although they can have more, and they must return the modified object.
- For example, the following function allows you to modify the second element of a vector:

```
> `second<-` = function(x, value) {  
+   x[2] = value  
+   x  
+ }  
> x = 1:10  
> second(x) = 5L  
> x  
[1] 1 5 3 4 5 6 7 8 9 10
```

When R evaluates the assignment `second(x) <- 5`, it notices that the left hand side of the `<-` is not a simple name, so it looks for a function named `second<-` to do the replacement.

Replacement functions

- They actually create a modified copy. We can see that by using `pryr::address()` to find the memory address of the underlying object.

```
> x = 1:100
> address(x)
[1] "0x10c9b8ce0"
> second(x) = 100L
> address(x)
[1] "0x10bf11de0"
```

- It is often useful to combine replacement and subsetting.

```
> x = c(a = 1, b = 2, d = 3)
> names(x)
[1] "a" "b" "d"
> names(x)[2] = "e"
> names(x)
[1] "a" "e" "d"
```

Anonymous functions

R doesn't have a special syntax for creating a named function: when you create a function, you use the regular assignment operator to give it a name. If you choose not to give the function a name, you get an anonymous function.

- Like all functions in R, anonymous functions have `formals()`, a `body()`, and a parent `environment()`:

```
formals(function(x = 4) g(x) + h(x))
body(function(x = 4) g(x) + h(x))
environment(function(x = 4) g(x) + h(x))
```

- You can call an anonymous function without giving it a name,

```
> function(x) 3
function(x) 3
> (function(x) 3)()
[1] 3
> (function(x) x + 3)(10)
[1] 13
```

Closure

- Another important use of anonymous functions is to create **closures**, functions written by functions.
- Closures get their name because they enclose the environment of the parent function and can access all its variables.
- This is useful because it allows us to have two levels of parameters: a parent level that controls operation and a child level that does the work.
- The following example uses this idea to generate a family of power functions in which a parent function (**power()**) creates two child functions (**square()** and **cube()**).

```
power = function(exponent) {  
  function(x) {  
    x ^ exponent  
  }  
}  
  
square = power(2)  
cube = power(3)  
> square(4)  
[1] 16  
> cube(4)  
[1] 64
```

Function factories

A function factory is a factory for making new functions.

Function factories are most useful when:

- The different levels are more complex, with multiple arguments and complicated bodies.
- Some work only needs to be done once, when the function is generated.

Example: find the maximum likelihood estimate

- find the MLE for intensity parameter λ , if our data come from a Poisson distribution. First, we create a function factory that, given a dataset, returns a function that computes the negative log likelihood (NLL) for parameter lambda.

```
poisson_nll <- function(x) {  
  n <- length(x)  
  sum_x <- sum(x)  
  function(lambda) {  
    n * lambda - sum_x * log(lambda) # + terms not involving lambda  
  }  
}
```

- Note how the closure allows us to precompute values that are constant with respect to the data.
- We can use this function factory to generate specific NLL functions for input data. Then `optimise()` allows us to find the best values (MLE), given a generous starting range.

Example: MLE for Poisson Distribution

```
> x1 = c(41, 30, 31, 38, 29, 24, 30, 29, 31, 38)
> x2 = c(6, 4, 7, 3, 3, 7, 5, 2, 2, 7, 5, 4, 12, 6, 9)
> nll1 = poisson_nll(x1)
> nll2 = poisson_nll(x2)
> optimise(nll1, c(0, 100))$minimum
[1] 32.09999
> optimise(nll2, c(0, 100))$minimum
[1] 5.466681
> mean(x1)
[1] 32.1
> mean(x2)
[1] 5.466667
```

Mutable state

The key to managing variables at different levels is the double arrow assignment operator (`<<-`). Unlike the usual single arrow assignment (`<-` or `=`) that always assigns in the current environment, the double arrow operator will keep looking up the chain of parent environments until it finds a matching name

The following example shows a counter that records how many times a function has been called. Each time `new_counter` is run, it creates an environment, initializes the counter `i` in this environment, and then creates a new function.

```
new_counter = function() {  
  i = 0  
  function() {  
    i <<- i + 1  
    i  
  }  
}
```

Mutable state

The new function is a closure, and its enclosing environment is the environment created when `new_counter()` is run. Ordinarily, function execution environments are temporary, but a closure maintains access to the environment in which it was created. In the example below, closures `counter_one()` and `counter_two()` each get their own enclosing environments when run, so they can maintain different counts.

The following example shows a counter that records how many times a function has been called. Each time `new_counter` is run, it creates an environment, initializes the counter `i` in this environment, and then creates a new function.

```
> counter_one <- new_counter()
> counter_two <- new_counter()
> counter_one()
[1] 1
> counter_one()
[1] 2
> counter_two()
[1] 1
> counter_two()
[1] 2
```

What happened? if we have

```
i = 0
new_counter2 = function() {
  i <- i + 1
  i
}
new_counter3 = function() {
  i = 0
  function() {
    i = i + 1
    i
  }
}
```

Apply Functions Over List or Vector: lapply()

Imagine you have loaded a data file, like the one below, that uses -99 to represent missing values.

You want to replace all the -99s with NAs.

```
> set.seed(2020)
> df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
> names(df) <- letters[1:6]
> df
```

	a	b	c	d	e	f
1	8	2	10	7	10	-99
2	5	5	5	3	2	1
3	7	1	5	3	4	-99
4	6	7	6	1	9	-99
5	2	9	-99	10	5	1
6	1	9	8	-99	7	5

`lapply()` will be useful to solve this problem.

Apply Functions Over List or Vector: lapply()

`lapply()` takes three inputs:

- `x`, a vector or a list;
- `f`, a function;
- `...`, other arguments to pass to `f()`.

It applies the function to each element of the list and returns a new list.

`lapply(x, f, ...)` is equivalent to the following for loop:

```
out <- vector("list", length(x))
for (i in seq_along(x)) {
  out[[i]] <- f(x[[i]], ...)
}
```

不会改变数据结构

Apply Functions Over List or Vector: lapply()

```
fix_missing = function(x) {  
  x[x == -99] = NA  
  x  
}  
df[] = lapply(df, fix_missing)
```

This code has five advantages over copy and paste:

- It's more compact.
- If the code for a missing value changes, it only needs to be updated in one place.
- It works for any number of columns. There is no way to accidentally miss a column.
- There is no way to accidentally treat one column differently than another.
- It is easy to generalize this technique to a subset of columns:

Apply Functions Over Array Margins: `apply()`

`apply()` takes three inputs:

- `x`, an array, including a matrix;
- `MARGIN`, a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, `c(1, 2)` indicates rows and columns. Where `X` has named dimnames, it can be a character vector selecting dimension names
- `f`, a function;
- `...`, other arguments to pass to `f()`.

It returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

```
> x = cbind(x1 = 3, x2 = c(4:1, 2:5))
> apply(x, 2, mean)
x1 x2
3  3
```