Functional programming in R

L. Gatto

May 2, 2013

Outline

Terminology

- *First-class functions* – a function is a value just like any other variable. Functions can thus be used as arguments to other functions. Functions are considered *first-class citizens*.
- *Higher-order functions* – refers to functions that take functions as parameters (input) or return functions (output).

- Illustrate cases where we
  - assign functions to variables or storing them in data structures
  - pass functions as arguments to other functions
  - return function as the values from other functions

Outline

An OO implementation

```
L <- list(data = rnorm(5), fun = mean, res = NULL)
L
```

```
$data
[1]  0.1565569 -0.2507388 -0.2940839  1.4638425 -0.4700434

$fun
function (x, ...)
UseMethod("mean")
<bytecode: 0x2349d68>
<environment: namespace:base>

$res
NULL
```

An OO implementation

```
L$res <- L$fun(L$data)
L
```

```
$data
[1]  0.1565569 -0.2507388 -0.2940839  1.4638425 -0.4700434

$fun
function (x, ...)
UseMethod("mean")
<bytecode: 0x2349d68>
<environment: namespace:base>

$res
[1] 0.1211067
```

Outline

Functions and function arguments

```
10^(1:5)
`^`(10, seq(1, 5, 1))
```

```
 [1] 1e+01 1e+02 1e+03 1e+04 1e+05
```

```
 [1] 1e+01 1e+02 1e+03 1e+04 1e+05
```

```
(v <- rnorm(6))
v[v > 0]
`[`(v, `>`(v, 0))
```

```
 [1]  0.1433087  1.0431326  0.7330517 -1.0431603  0.5226738  0.
```

```
 [1] 0.1433087 1.0431326 0.7330517 0.5226738 0.2518947
```

```
 [1] 0.1433087 1.0431326 0.7330517 0.5226738 0.2518947
```

Functions and function arguments (recursion)

```
fact <- function(x)
  ifelse (x == 0 | x == 1,
          1,
          fact(x - 1) * x)

fact(3)
fact(6)
fact(fact(3))
```

```
 [1] 6

 [1] 720

 [1] 720
```

Function creating functions (1)

```
make.power <- function(n) {
  function(x) x^n
}

cube <- make.power(3)
square <- make.power(2)
cube(2)
square(2)
```

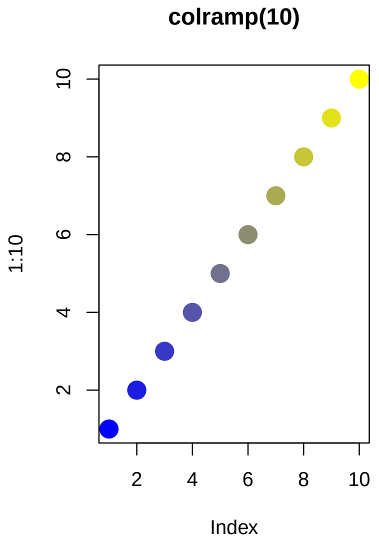
```
 [1] 8
 [1] 4
```

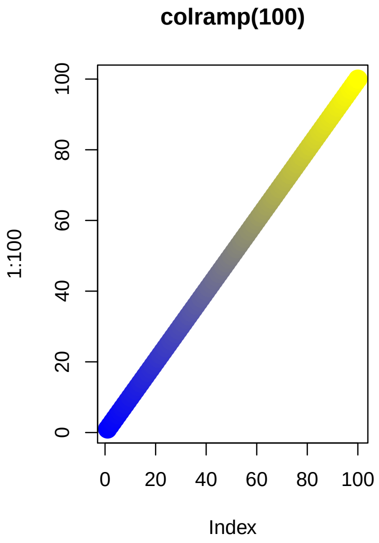
Function creating functions (2)

```
colramp <- colorRampPalette(c("blue", "yellow"))
colramp

 function (n)
 {
     x <- ramp(seq.int(0, 1, length.out = n))
     rgb(x[, 1L], x[, 2L], x[, 3L], maxColorValue = 255)
 }
 <bytecode: 0x1da0320>
 <environment: 0x1da1948>
```

Function creating functions (3)





Outline

Reduce

`Reduce(f, x)` uses a binary function to successively combine the elements of a given vector and a possibly given initial value.

```
L <- replicate(3, matrix(rnorm(9), 3), simplify = FALSE)
Reduce("+", L)
try(sum(L))
```

```
           [,1]       [,2]       [,3]
[1,] -0.9375172  0.1751917  2.4015698
[2,] -2.0221494 -0.7916472  0.8566429
[3,]  0.6414585  2.2553825 -4.3977670

Error in sum(L) : invalid 'type' (list) of argument
```

Reduce (2)

```
## Using a vector to save space
Reduce("+", list(1, 2, 3), init = 10)
Reduce("+", list(1, 2, 3), accumulate = TRUE)
Reduce("+", list(1, 2, 3), right = TRUE, accumulate = TRUE)
```

```
 [1] 16
 [1] 1 3 6
 [1] 6 5 3
```

Filter and Negate

`Filter(f, x)` extracts the elements of a vector for which a predicate (logical) function gives true.

`Negate(f)` creates the negation of a given function.

```
even <- function(x) x %% 2 == 0
(y <- sample(100, 10))
Filter(even, y)
Filter(Negate(even), y)
```

```
  [1] 78 21 28 88 81 92 99 16 77  9
```

```
 [1] 78 28 88 92 16
```

```
 [1] 21 81 99 77  9
```

Map

`Map(f, ...)` applies a function to the corresponding elements of given vectors. Similar to `mapply` without any attempt to simplify.

```
Map(even, 1:3)
```

```
 [[1]]
 [1] FALSE

 [[2]]
 [1] TRUE

 [[3]]
 [1] FALSE
```

Find and Position

`Find(f, x)` and `Position(f, x)` give the first (or last elements) and its position in the vector, for which a predicate (logical) function gives true.

```
Find(even, 10:15)
Find(even, 10:15, right = TRUE)
Position(Negate(even), 10:15)
Position(Negate(even), 10:15, right = TRUE)
```

```
 [1] 10


 [1] 14


 [1] 2


 [1] 6
```

Outline

A note on efficiency

Although these higher order functions are arguably elegant and allow powerful constructs (see references), they come at a slight speed cost compared to `mapply`, `[` and vectorised functions.

Note: Hadoop's *MapReduce* model is a programming model for processing large data sets, typically used to do distributed computing on clusters of computers. The model is inspired by the map and reduce functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as their original forms. (Wikipedia, MapReduce)

References

- R Gentleman, *R Programming for Bioinformatics*, CRC Press, 2008
- `?Map`, or any other of the higher order functions
- Blog post, *Higher Order Functions in R*, John Myles White http://www.johnmyleswhite.com/notebook/2010/09/23/higher-order-functions-in-r/