Functional Programming in R

A Pragmatic Introduction

Soumendra Prasad Dhanee

Data Scientist
embibe.com

18^{th} October, 2014
8^{th} November, 2014
(delivered at R Meetup, Mumbai)

Overview

- We'll start by reviewing basics
- We'll see the ideas at action first (by solving problems) and abstractions later
- We'll digress a lot

Next ...

Writing a function in R

```
functionname <- function(arg1, arg2, ...) {
  <<expressions>>
   [return(<<expression>>)]
}
```

- return() is optional
- last statement of the function body is returned by default

Example of a Function

```
square.of <- function(x) {
  x*x
}
```

Call the function

```
square.of(4)
y <- square.of(x=4)
square.of(z=4)
```

More on Functions

- Arguments of R functions can be either required or optional, indicated by syntax
- Required arguments have no default value
- Optional arguments are defined to have a default value

More on Functions - Example

```
add <- function(x, y = 2) {
  x+y
}
```

```
add(3)
add(3,2)
add(x=3)
add(x=3, y=2)
add(y=2)
add( , 3)
```

Example - Computing Power I

Write a function that computes the kth power of an argument x, using one default argument, and one optional argument, i.e. an argument that has a default value.

We'll be coming back to this function later on.

Example - Computing Power II

```
power <- function(x, k=2) x^k
```

The apply family of functions

- lapply(X, FUN, ...)
  returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X
- sapply(X, FUN, ...)
  a user-friendly version of lapply by default returning a vector or matrix if appropriate

Read the Documentation

Vectorized Functions

... functions that operate on vectors or matrices or dataframes.

- Often much faster than looping over a vector, often use `.Internal` or `.Primitive`
- Higher abstraction - less code to write, less to debug

.Internal and .Primitive - I

```
log

## function (x, base = exp(1))  .Primitive("log")
```

```
paste

## function (..., sep = " ", collapse = NULL)
## .Internal(paste(list(...), sep, collapse))
## <bytecode: 0x1f8a2c8>
## <environment: namespace:base>
```

.Internal and .Primitive - II

```
colMeans

## function (x, na.rm = FALSE, dims = 1L)
## {
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     if (!is.array(x) || length(dn <- dim(x)) < 2L)
##         stop("'x' must be an array of at least two dimen
##     if (dims < 1L || dims > length(dn) - 1L)
##         stop("invalid 'dims'")
##     n <- prod(dn[1L:dims])
##     dn <- dn[-(1L:dims)]
##     z <- if (is.complex(x))
##         .Internal(colMeans(Re(x), n, prod(dn), na.rm)) +
```

Examples of Vectorized Functions

- ifelse()
- is.na()

- log()
- sqrt()
- rnorm()

- colMeans()
- rowSums()

- $x > y$
- x == y
- !x

Primality Testing I

Given a number, decide if it is a prime or not.

Hint 1 : min(x) gives the lowest element of vector x

```
min(c(2, 1, 3))

## [1] 1
```

Primality Testing II

```
min( x%%( 2:(x-1) ) )>0
```

The Deconstruction

```
2:(x-1)
x%%( 2:(x-1) )
min( x%%( 2:(x-1) ) )>0
```

Lists of functions

```
x <- 1:10
funs <- list(
  sum = sum,
  mean = mean,
  median = median
)
sapply(funs, function(f) f(x))

##     sum   mean median
##    55.0    5.5    5.5
```

Let's get started

Problem 1

- Input

  A numeric vector (1:200000)

- Output

  A vector containing the even numbers in the input vector, listed in the order they appeared in the input vector

Solution - for loop I

Solution using for loop

```
x <- 1:200000
ans <- logical(200000)
for(i in x)
{
  if(i%%2==0) { ans[i] = TRUE } else {
    ans[i] = FALSE }
}
x[ans]
```

```
##      user  system elapsed
##     0.542   0.005   0.546
```

Solution - for loop II

Solution using for loop

```
x <- 1:200000
ans <- c()
for(i in x)
{
  if(i%%2==0) { ans[i] = TRUE } else {
    ans[i] = FALSE }
}
x[ans]
```

```
##    user  system elapsed
##  56.511   2.631  59.333
```

Solution - vectorized functions

Solution using vectorized functions

```
x <- 1:200000
ans <- x%%2==0
x[ans]
```

```
##     user  system elapsed
##    0.013   0.000   0.013
```

Solution - higher-order functions

Solution using higher-order functions

```
x <- 1:200000
ans <- sapply(x, function(i) i%%2==0)
x[ans]
```

```
##    user  system elapsed
##   0.554   0.011   0.566
```

What are higher-order functions?

Higher-order Functions

A higher-order function (also functional form, functional or functor) is a function that does at least one of the following:

- takes one or more functions as an input
- outputs a function

-source - Wikipedia

An example

sapply

sapply(X, FUN, ...)

What are first-class functions?

First-class Functions

First-class functions are functions that can be treated like any other piece of data ...

- can be stored in a variable
- can be stored in a list
- can be stored in an object
- can be passed to other functions
- can be returned from other functions

... just like any other piece of data.

Next ...

Rounding off

Problem 2a

- Input

  A numeric vector (`rnorm(100, 0, 1)`)
- Output

  A vector containing all the numbers in the input vector rounded to the nearest integer

Solution 2a - Rounding off

Solution using sapply

```
sapply(rnorm(100, 0, 1), round)
```

Rounding off

Problem 2b

- Input

  A numeric vector (`rnorm(100, 0, 1)`)

- Output

  A vector containing all the numbers in the input vector rounded to two decimal places

Solution 2b - Rounding off

Solution using sapply

```
sapply(rnorm(100, 0, 1), function(x) round(x, 2))
```

Anonymous Functions

Anonymous Functions

Anonymous function is a function that is not bound to an identifier.
source - Wikipedia

Acting on Matrices

Problem 3

- Input
  A 3-by-3 matrix (matrix(1:9, ncol=3))
- Output
  Add 1 to row 1, 4 to row 2, 7 to row 3

Solution 3 - Matrix Addition

Solution using sweep

```
m <- matrix(1:9, ncol=3)
sweep(m, 1, c(1, 4, 7), "+")
```

```
##      [,1] [,2] [,3]
## [1,]    2    5    8
## [2,]    6    9   12
## [3,]   10   13   16
```

Next ...

Applicative Programming

Calling by function B of function A, where function A was originally supplied to function B as an argument.

Examples

- map
- reduce (fold)
- filter

Map

Map(f, x, ...)

```
function (f, ...)
{
    f <- match.fun(f)
    mapply(FUN = f, ..., SIMPLIFY = FALSE)
}
<bytecode: 0x402ad90>
<environment: namespace:base>
```

mapply

mapply is a multivariate version of sapply.

```
mapply(rep, 1:4, 4:1)

mapply(rep, times = 1:4, x = 4:1)

mapply(rep, x = 1:4, times = 4:1)
```

Filter I

Choose all the even numbers.

```
Filter(function(x) x%%2, 1:200000)
```

```
##    user  system elapsed
##   0.419   0.004   0.424
```

Filter II

The previous solution was wrong.

Now what is the output of the following -

```
Filter(c(T, F), 1:200000)
```

Filter III

```
function (f, x)
{
    ind <- as.logical(unlist(lapply(x, f)))
    x[!is.na(ind) & ind]
}
<bytecode: 0x4f87f00>
<environment: namespace:base>
```

Negate I

What is the outcome?

```
Negate(c(T, F), 1:10)
```

Negate II

What is the outcome?

```
Filter(Negate(function(x) x%%2), 1:10)
```

Negate III

```
function (f)
{
    f <- match.fun(f)
    function(...) !f(...)
}
<bytecode: 0x4e9c250>
<environment: namespace:base>
```

Next ...

Computing Power Revisited

Write a function that computes the kth power of an argument x, using one default argument, and one optional argument, i.e. an argument that has a default value.

```
power <- function(x, k=2) {
  x^k
}
```

Function Factory with Closures

A function returning a function:

```r
power <- function(exponent) {
  function(x) {
    x ^ exponent
  }
}

square <- power(2)
square(2)
square(4)

cube <- power(3)
cube(2)
cube(4)
```

What are Closures?

An object is data with functions. A closure is a function with data.
- John D. Cook

... an abstraction binding a function to its scope.
- Wikipedia

Closures get their name because they enclose the environment of the parent function and can access all its variables.
- Hadley Wickham

Closer look at closures I

```
square

## function(x) {
##      x ^ exponent
##   }
## <environment: 0x217dcf8>

cube

## function(x) {
##      x ^ exponent
##   }
## <environment: 0x26cb1d0>
```

Closer look at closures II

```
as.list(environment(square))

## $exponent
## [1] 2

as.list(environment(cube))

## $exponent
## [1] 3
```

Closer look at closures III

```
library(pryr)
unenclose(square)
#> function (x)
#> {
#>     x^2
#> }
unenclose(cube)
#> function (x)
#> {
#>     x^3
#> }
```