Introduction to Statistical Modelling in R

BCAM - Basque Center for Applied Mathematics, Applied Statistics

Dae-Jin Lee < dlee@bcamath.org >

CHAPTER 3. Introduction to basic programming in R

Contents

---

1 Introduction to basic programming in R

1.1 Operators

R's binary and logical operators will look very familiar to programmers. Note that binary operators work on vectors and matrices as well as scalars.

1.1.1 Arithmetic Operators

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation
x %% y	modulus (x mod y) 5%%2 is 1
x %/% y	integer division 5%/%2 is 2

1.1.2 Logical Operators

Operator	Description
<	less than
>	greater than
<=	less or equal to
>=	greater or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x|y	x OR y
x&y	x AND y
isTRUE(x)	test if x is TRUE

```
# An example
x <- c(1:10)
x[(x>8) | (x<5)]
```

```
## [1]  1  2  3  4  9 10
```

```
# yields 1 2 3 4 9 10

# How it works
x <- c(1:10)
x
```

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

```
x > 8
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
```

```
x < 5
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
x > 8 | x < 5
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE
```

```
x[c(T,T,T,T,F,F,F,F,T,T)]
```

```
## [1]  1  2  3  4  9 10
```

1.2 Control Structures

These allow you to control the flow of execution of a script typically inside of a function. Common ones include:

- if, else
- for
- while
- repeat
- break
- next
- return

1.2.1 Conditional Executions

if

```
if (condition) {
    # do something
} else {
    # do something else
}
```

e.g.:

```
x <- 1:15
if (sample(x, 1) <= 10) { # ?sample
    print("x is less than 10")
} else {
    print("x is greater than 10")
}
```

```
## [1] "x is greater than 10"
```

Vectorization with `ifelse`

```
ifelse(x <= 10, "x less than 10", "x greater than 10")
```

Other valid ways of writing if/else

```
if (sample(x, 1) < 10) {
    y <- 5
} else {
    y <- 0
}
```

```
y <- if (sample(x, 1) < 10) {
    5
} else {
    0
}
```

for

A `for` loop works on an iterable variable and assigns successive values till the end of a sequence.

```
for (i in 1:10) {
    print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
```

```
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

```
x <- c("apples", "oranges", "bananas", "strawberries")

for (i in x) {
    print(x[i])
}
```

```
## [1] NA
## [1] NA
## [1] NA
## [1] NA
```

```
for (i in 1:4) {
    print(x[i])
}
```

```
## [1] "apples"
## [1] "oranges"
## [1] "bananas"
## [1] "strawberries"
```

```
for (i in seq(x)) {
    print(x[i])
}
```

```
## [1] "apples"
## [1] "oranges"
## [1] "bananas"
## [1] "strawberries"
```

```
for (i in 1:4) print(x[i])
```

```
## [1] "apples"
## [1] "oranges"
## [1] "bananas"
## [1] "strawberries"
```

nested lopps

```
m <- matrix(1:10, 2)
for (i in seq(nrow(m))) {
    for (j in seq(ncol(m))) {
        print(m[i, j])
    }
}
```

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

while

```
i <- 1
while (i < 10) {
    print(i)
    i <- i + 1
}
```

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

Be sure there is a way to exit out of a `while` loop.

Repeat and break

```
repeat {
    # simulations; generate some value have an expectation if within some range,
    # then exit the loop
    if ((value - expectation) <= threshold) {
```

```
        break
    }
}
```

Next

```
for (i in 1:20) {
    if (i%%2 == 1) {  # %% is the modulus
        next
    } else {
        print(i)
    }
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
## [1] 12
## [1] 14
## [1] 16
## [1] 18
## [1] 20
```

1.2.1.1 Comparison Operators

- equal: ==

```
  "hola" == "hola"
```

```
## [1] TRUE
```

```
  "hola" == "Hola"
```

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

```
   1 == 2-1
```

```
## [1] TRUE
```

- not equal: !=

```
    a <- c(1,2,4,5)
    b <- c(1,2,3,5)
    a == b
```

```
## [1]  TRUE  TRUE FALSE  TRUE
```

```
    a != b
```

```
## [1] FALSE FALSE  TRUE FALSE
```

- greater/less than: `>` `<`

```
set.seed(1)
a <- rnorm(10)
b <- rnorm(10)
a<b
```

```
##  [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
```

- greater/less than or equal: `>=` `<=`

```
set.seed(2)
a <- rnorm(10)
b <- rnorm(10)
a >= b
```

```
##  [1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
```

- `which`

```
set.seed(3)
which(a>b)
```

```
## [1] 3 6 9
```

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
which(LETTERS=="R")
```

```
## [1] 18
```

- which.min or which.max

```
set.seed(4)
a <- rnorm(10)
a
```

```
##  [1]  0.2167549 -0.5424926  0.8911446  0.5959806  1.6356180  0.6892754
##  [7] -1.2812466 -0.2131445  1.8965399  1.7768632
```

```
which.min(a)
```

```
## [1] 7
```

```
which.max(a)
```

```
## [1] 9
```

- is.na

```
 a[2] <- NA
is.na(a)
```

```
##  [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
which(is.na(a))
```

```
## [1] 2
```

1.2.2 Logical Operators

- and: &

```
z = 1:6
which(2 < z & z > 3)
```

```
## [1] 4 5 6
```

- or: |

```
z = 1:6
(z > 2) & (z < 5)
```

```
## [1] FALSE FALSE  TRUE  TRUE FALSE FALSE
```

```
which((z > 2) & (z < 5))
```

```
## [1] 3 4
```

- not: `!`

```
x <- c(TRUE,FALSE,0,6)
y <- c(FALSE,TRUE,FALSE,TRUE)

!x
```

```
## [1] FALSE  TRUE  TRUE FALSE
```

Operators `&` and `|` perform element-wise operation producing the result having the length of the longer operand. But `&&` and `||` examines only the first element of the operands resulting into a single length logical vector. Zero is considered `FALSE` and non-zero numbers are taken as `TRUE`.

Example: - `&&` vs `&`

```
x&y
```

```
## [1] FALSE FALSE FALSE  TRUE
```

```
x&&y
```

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

- `||` vs `|`

```
x||y
```

```
## [1] TRUE
```

```
x|y
```

```
## [1]  TRUE  TRUE FALSE  TRUE
```

1.3 `if` statements

```
if(cond1=true) { cmd1 } else { cmd2 }
```

```
if(1==0) {
    print(1)
} else {
    print(2)
}
```

```
## [1] 2
```

1.4 `ifelse` statement

```
ifelse(test, true_value, false_value)
```

```
x <- 1:10 # Creates sample data
ifelse(x<5 | x>8, x, 0)
```

```
##  [1]  1  2  3  4  0  0  0  0  9 10
```

1.5 `while` statement

1.6 Loops

The most commonly used loop structures in `R` are `for`, `while` and `apply` loops. Less common are `repeat` loops. The `break` function is used to break out of loops, and next halts the processing of the current iteration and advances the looping index.

1.6.1 Writing a simple for loop in `R`

Suppose you want to do several printouts of the following form: The year is [year] where [year] is equal to 2010, 2011, up to 2015. You can do this as follows:

```
print(paste("The year is", 2010))
```

```
## [1] "The year is 2010"
```

1.6.2 for

For loops are controlled by a looping vector. In every iteration of the loop one value in the looping vector is assigned to a variable that can be used in the statements of the body of the loop. Usually, the number of loop iterations is defined by the number of values stored in the looping vector and they are processed in the same order as they are stored in the looping vector.

Syntax

```
for(variable in sequence) {
    statements
}
```

```
for (j in 1:5)
{
  print(j^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

Repeat the loop saving the resuls in a vector `x`.

```
n = 5
x = NULL  # creates a NULL object
for (j in 1:n)
{
  x[j] = j^2
}
x
```

```
## [1]  1  4  9 16 25
```

Let's use a for loop to estimate the average of squaring the result of a roll of a dice.

```
nsides = 6
ntrials = 1000
trials = NULL
for (j in 1:ntrials)
{
  trials[j] = sample(1:nsides,1)  # We get one sample at a time
}
mean(trials^2)
```

```
## [1] 14.563
```

Example: stop on condition and print error message

```
x <- 1:10
z <- NULL
for(i in seq(along=x)) {
    if (x[i]<5) {
        z <- c(z,x[i]-1)
    } else {
        stop("values need to be <5")
    }
}
## Error: values need to be <5
z
## [1] 0 1 2 3
```

1.7 while

Similar to `for` loop, but the iterations are controlled by a conditional statement.

```
z <- 0
while(z < 5) {
    z <- z + 2
    print(z)
}
```

```
## [1] 2
## [1] 4
## [1] 6
```

1.8 apply loop family

For Two-Dimensional Data Sets: apply

Syntax:

```
apply(X, MARGIN, FUN, ARGs)
```

`X`: `array`, `matrix` or `data.frame`; `MARGIN`: 1 for rows, 2 for columns, `c(1,2)` for both; `FUN`: one or more functions; `ARGs`: possible arguments for function.

```
## Example for applying predefined mean function
apply(mtcars[,1:3], 1, mean)

## With custom function
x <- 1:10
test <- function(x) { # Defines some custom function
    if(x < 5) {
        x-1
    } else {
        x / x
    }
}

apply(as.matrix(x), 1, test)

## Same as above but with a single line of code
apply(as.matrix(x), 1, function(x) { if (x<5) { x-1 } else { x/x } })
```

For Ragged Arrays: `tapply`

Apply a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors.

```
## Computes mean values of vector agregates defined by factor
tapply(as.vector(mtcars$mpg), factor(mtcars$cyl), mean)
```

```
##        4        6        8
## 26.66364 19.74286 15.10000
```

```
## The aggregate function provides related utilities
aggregate(mtcars[,c(1,3,4)], list(mtcars$cyl), mean)
```

```
##   Group.1      mpg     disp        hp
## 1       4 26.66364 105.1364  82.63636
## 2       6 19.74286 183.3143 122.28571
## 3       8 15.10000 353.1000 209.21429
```

For Vectors and Lists: `lapply` and `sapply`

Both apply a function to vector or list objects. The function `lapply` returns a list, while `sapply` attempts to return the simplest data object, such as `vector` or `matrix` instead of `list`.

Syntax

```
lapply(X,FUN)
sapply(X,FUN)
```

```
## Creates a sample list
mylist <- as.list(mtcars[,c(1,4,6)])
mylist
```

```
## $mpg
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
##
## $hp
##  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230
## [18]  66  52  65  97 150 150 245 175  66  91 113 264 175 335 109
##
## $wt
##  [1] 2.620 2.875 2.320 3.215 3.440 3.460 3.570 3.190 3.150 3.440 3.440
## [12] 4.070 3.730 3.780 5.250 5.424 5.345 2.200 1.615 1.835 2.465 3.520
## [23] 3.435 3.840 3.845 1.935 2.140 1.513 3.170 2.770 3.570 2.780
```

Compute sum of each list component and return result as list

```
lapply(mylist, sum)
```

```
## $mpg
## [1] 642.9
##
## $hp
## [1] 4694
##
## $wt
## [1] 102.952
```

Compute sum of each list component and return result as vector

```
sapply(mylist, sum)
```

```
##      mpg       hp       wt
##  642.900 4694.000  102.952
```

1.9 Other Loops

Repeat Loop

Syntax

`repeat` statements

Loop is repeated until a break is specified. This means there needs to be a second statement to test whether or not to break from the loop.

Example:

```
z <- 0
repeat {
    z <- z + 1
    print(z)
    if(z > 100) break()
}
```

1.10 Improving Speed Performance of Loops

Looping over very large data sets can become slow in `R`. However, this limitation can be overcome by eliminating certain operations in loops or avoiding loops over the data intensive dimension in an object altogether. The latter can be achieved by performing mainly vector-to-vector or matrix-to-matrix computations which run often over 100 times faster than the corresponding `for()` or `apply()` loops in `R`. For this purpose, one can make use of the existing speed-optimized R functions (e.g.: `rowSums`, `rowMeans`, `table`, `tabulate`) or one can design custom functions that avoid expensive `R` loops by using vector- or matrix-based approaches. Alternatively, one can write programs that will perform all time consuming computations on the C-level.

1. Speed comparison of `for` loops with an append versus and inject step

```
N <- 1e3
myMA <- matrix(rnorm(N), N, 10, dimnames=list(1:N, paste("C", 1:10, sep="")))
results <- NULL
system.time(for(i in seq(along=myMA[,1]))
```

```
            results <- c(results, mean(myMA[i,])))

results <- numeric(length(myMA[,1]))
system.time(for(i in seq(along=myMA[,1]))
            results[i] <- mean(myMA[i,]))
```

The inject approach is 20-50 times faster than the append version.

2. Speed comparison of `apply` loop versus `rowMeans` for computing the mean for each row in a large matrix:

```
system.time(myMAmean <- apply(myMA, 1, mean))
system.time(myMAmean <- rowMeans(myMA))
```

The `rowMeans` approach is over 200 times faster than the `apply` loop.