Introduction to R Programming

The Apply Family

Pedro Fonseca

30 April 2022

Built in datasets

R provides many built in datasets. For a complete list see:

```r
library(help = "datasets")
```

In this lecture we will the following datasets:

- USArrests
- airquality

For information about a specific dataset see, for example:

```r
?USArrests
```

The USArrests dataset

This data set contains:

- Arrests per 100.000 residents for assault, murder and rape in each of the 50 USA states in 1973.
- There is also a column with the percentage of population living in urban areas.

The USArrests dataset

```
head(USArrests)
```

```
##            Murder Assault UrbanPop Rape
## Alabama      13.2     236       58 21.2
## Alaska       10.0     263       48 44.5
## Arizona       8.1     294       80 31.0
## Arkansas      8.8     190       50 19.5
## California    9.0     276       91 40.6
## Colorado      7.9     204       78 38.7
```

The USArrests dataset

Let's do some calculations with the arrests of four states:

```
USArrests_short <- USArrests[1:4, -3]

USArrests_short
```

```
##          Murder Assault Rape
## Alabama    13.2     236 21.2
## Alaska     10.0     263 44.5
## Arizona     8.1     294 31.0
## Arkansas    8.8     190 19.5
```

Row and column sums

```
colSums(USArrests_short)
```

```
##  Murder Assault    Rape
##    40.1   983.0   116.2
```

```
rowSums(USArrests_short)
```

```
##  Alabama   Alaska  Arizona Arkansas
##    270.4    317.5    333.1    218.3
```

Row and column means

```
colMeans(USArrests_short)
```

```
##  Murder Assault    Rape
##  10.025 245.750  29.050
```

```
rowMeans(USArrests_short)
```

```
##   Alabama    Alaska   Arizona  Arkansas
##  90.13333 105.83333 111.03333  72.76667
```

The `apply()` function

To collapse data frames across rows or columns using functions other than the sum and the mean we can use `apply()`:

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

- X is a data frame
- `MARGIN = 1` for rows, `MARGIN = 2` for columns
- FUN is a function
- `...` are optional arguments to pass to FUN

The apply() function

Apply functions over the columns of USArrests_short:

```
apply(USArrests_short, 2, mean)
```

```
##   Murder Assault    Rape
##   10.025 245.750  29.050
```

```
apply(USArrests_short, 2, median)
```

```
##   Murder Assault    Rape
##      9.4   249.5    26.1
```

```
apply(USArrests_short, 2, sd)
```

```
##     Murder    Assault      Rape
##   2.257395 44.078528 11.479402
```

The apply() function

Apply functions the rows of USArrests_short:

```
apply(USArrests_short, 1, max)
```

```
##  Alabama   Alaska  Arizona Arkansas
##      236      263      294      190
```

```
apply(USArrests_short, 1, min)
```

```
##  Alabama   Alaska  Arizona Arkansas
##     13.2     10.0      8.1      8.8
```

```
apply(USArrests_short, 1, var)
```

```
##  Alabama   Alaska  Arizona Arkansas
## 15973.81 18823.58 25238.70 10336.36
```

apply() vs for loop

Loop over the columns of USArrests_short:

```
res <- vector()

for(i in 1:ncol(USArrests_short)){

  res[i] <- mean(USArrests_short[[i]], na.rm = TRUE)
  names(res)[i] <- names(USArrests_short)[i]
}

res
```

```
##  Murder Assault    Rape
##  10.025 245.750  29.050
```

apply() vs for loop

Loop over the rows of USArrests_short:

```
res <- vector()

for(j in 1:nrow(USArrests_short)){

  res[j] <- max(USArrests_short[j, ],na.rm = TRUE)
  names(res)[j] <- rownames(USArrests_short)[j]
}

res
```

```
##  Alabama   Alaska  Arizona Arkansas
##      236      263      294      190
```

apply() with ...

Now let's see an example that requires using dot-dot-dot (...). Suppose you're working with the airquality the data frame:

```
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

apply() with ...

Try to use `apply()` to compute the means of the first four columns:

```
apply(airquality[, 1:4], 2, mean)
```

```
##     Ozone   Solar.R      Wind      Temp
##        NA        NA  9.957516 77.882353
```

We get `NA` for the first two columns. Why?

apply() with ...

Problem: Some columns have NAs.

```
sum(is.na(airquality$Ozone))
```

```
## [1] 37
```

```
sum(is.na(airquality$Solar.R))
```

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

apply() with ...

Solution: use ... to pass the na.rm argument to mean():

```
apply(airquality[, 1:4], 2, mean, na.rm = TRUE)
```

```
##      Ozone    Solar.R       Wind       Temp
##  42.129310 185.931507   9.957516  77.882353
```

apply() user-defined functions

Checking for NAs one column at a time is not be feasible when we have many columns. How can we check how many NAs there there in each column?

```
count_nas <- function(x){sum(is.na(x))}

apply(airquality, 2, count_nas)
```

```
##   Ozone Solar.R    Wind    Temp   Month     Day 
##      37       7       0       0       0       0
```

The `lapply()` function

Syntax: `lapply(X, FUN, ...)`

`lapply()` is similar to `apply()` but:

- X is a vector (atomic or list)
- There is no MARGINS argument
- Always returns a list

`lapply()` 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.

The `lapply()` function

```r
lapply(c(1:3), log, base = 10)
```

```
## [[1]]
## [1] 0
##
## [[2]]
## [1] 0.30103
##
## [[3]]
## [1] 0.4771213
```

The `lapply()` function

```r
A <- matrix(1:10, ncol = 5)
B <- matrix(c(1, 5, 7, -1), ncol = 4)
C <- matrix(letters[1:4], ncol = 2)

my_list <- list(A, B, C)
```

The `lapply()` function

```
my_list
```

```
## [[1]]
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
##
## [[2]]
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    7   -1
##
## [[3]]
##      [,1] [,2]
## [1,] "a"  "c"
## [2,] "b"  "d"
```

The `lapply()` function

Every element of `my_list`, except the last, contain a numerical matrix. Sum the elements of each of those matrices:

```
lapply(my_list[-3], sum)
```

```
## [[1]]
## [1] 55
##
## [[2]]
## [1] 12
```

The lapply() function

Extract the element in position (1, 2) from each matrix:

```
lapply(my_list,"[", 1, 2)
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] "c"
```

The lapply() function

Extract the first row from each matrix:

```
lapply(my_list,"[", 1 , )
```

```
## [[1]]
## [1] 1 3 5 7 9
##
## [[2]]
## [1]  1  5  7 -1
##
## [[3]]
## [1] "a" "c"
```

The `lapply()` function

Extract the 2nd column from each matrix:

```
lapply(my_list,"[", , 2)
```

```
## [[1]]
## [1] 3 4
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] "c" "d"
```

lapply() vs for loop

Extract the 2nd column from each matrix:

```r
res <- vector(mode = "list")

for(i in seq_along(my_list)){
  res[[i]] <- my_list[[i]][, 2]
}
res
```

```
## [[1]]
## [1] 3 4
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] "c" "d"
```

The `sapply()` function

The sapply() function:

- Works like lapply(), but simplifies the output to the most elementary data structure that is possible.
- Returns vectors or matrices.

The sapply() function

```
sapply(my_list[-3], sum)
```

```
## [1] 55 12
```

```
sapply(my_list,"[", 1, 2)
```

```
## [1] "3" "5" "c"
```

The sapply() function

```
sapply(airquality, function(x) sum(is.na(x)))
```

```
##   Ozone Solar.R    Wind    Temp   Month     Day
##      37       7       0       0       0       0
```

The sapply() function

```
set.seed(123)

our_list <- list(
  w = 1:6,
  x = sample(1:5, 4, replace = TRUE),
  y = matrix(sample(1:100, 9), nrow = 3),
  z = sample(1:10, 3, replace = TRUE)
)

str(our_list)
```

```
## List of 4
##  $ w: int [1:6] 1 2 3 4 5 6
##  $ x: int [1:4] 3 3 2 2
##  $ y: int [1:3, 1:3] 43 14 25 90 91 69 96 57 92
##  $ z: int [1:3] 9 9 3
```

The sapply() function

```
sapply(our_list, max)
```

```
##  w  x  y  z
##  6  3 96  9
```

```
sapply(our_list, min)
```

```
##  w  x  y  z
##  1  2 14  3
```

```
sapply(our_list, class)
```

```
## $w
## [1] "integer"
##
## $x
## [1] "integer"
##
## $y
## [1] "matrix" "array"
```

The `sapply()` function

How many numbers are there inside each element of `our_list`?

```
sapply(our_list, length)
```

```
## w x y z
## 6 4 9 3
```

The sapply() function

```
set.seed(123)

our_list_2 <- list(
  w = 1:6,
  x = sample(1:5, 4, replace = TRUE),
  y = airquality
)
```

```
sapply(our_list_2, class)
```

```
##            w            x            y
##    "integer"    "integer" "data.frame"
```

```
dim(airquality)
```

```
## [1] 153   6
```

How many numbers are there inside each element of our_list_2?

The sapply() function

The length of a data frame is the number of columns, and hence `sapply(our_list_2, length)` won't do the trick.

```
our_fun <- function(x){
  if(class(x) == "data.frame"){
    nrow(x) * ncol(x)
  }else{
      length(x)
  }
}

sapply(our_list_2, our_fun)
```

```
##   w   x   y
##   6   4 918
```

sapply() vs for loop

The same but with a for loop:

```
res <- vector()

for(i in seq_along(our_list_2)){

  res[i] <- our_fun(our_list_2[[i]])
  names(res)[i] <- names(our_list_2)[i]
}

res
```

```
##   w   x   y
##   6   4 918
```

The `mapply()` function

`mapply()` is a generalization of `sapply()`. It applies a multivariate function over multiple vectors of arguments.

The mapply() function

Suppose we want 3 samples of different sizes from a Normal$(0, 1)$ distribution:

```
set.seed(123)


rnorm(n = 1)
```

```
## [1] -0.5604756
```

```
rnorm(n = 2)
```

```
## [1] -0.2301775  1.5587083
```

```
rnorm(n = 3)
```

```
## [1] 0.07050839 0.12928774 1.71506499
```

The mapply() function

The same result can be obtained more compactly with mapply():

```
set.seed(123)

sample_size <- 1:3
mapply(FUN = rnorm, n = sample_size)
```

```
## [[1]]
## [1] -0.5604756
##
## [[2]]
## [1] -0.2301775  1.5587083
##
## [[3]]
## [1] 0.07050839 0.12928774 1.71506499
```

The mapply() function

Since we only iterated over one vector, we could have used sapply():

```r
set.seed(123)

sample_size <- 1:3
sapply(FUN = rnorm, X = sample_size)
```

```
## [[1]]
## [1] -0.5604756
##
## [[2]]
## [1] -0.2301775  1.5587083
##
## [[3]]
## [1] 0.07050839 0.12928774 1.71506499
```

The `mapply()` function

But what if we want to sample from normal distributions with different means, while still having samples of different sizes?

```r
set.seed(123)


rnorm(n = 1, mean = 5)
```

```
## [1] 4.439524
```

```r
rnorm(n = 2, mean = 10)
```

```
## [1]  9.769823 11.558708
```

```r
rnorm(n = 3, mean = -3)
```

```
## [1] -2.929492 -2.870712 -1.284935
```

The mapply() function

In this case we need to iterate over two vectors, one for sample sizes and one for means:

```r
set.seed(123)

sample_size <- 1:3
mu <- c(5, 10, -3)
mapply(rnorm, n = sample_size, mean = mu)
```

```
## [[1]]
## [1] 4.439524
##
## [[2]]
## [1]  9.769823 11.558708
##
## [[3]]
## [1] -2.929492 -2.870712 -1.284935
```

The `mapply()` function

Now suppose we also want each sample to have a different standard deviation:

```r
set.seed(123)


rnorm(n = 1, mean = 5, sd = 1)
```

```
## [1] 4.439524
```

```r
rnorm(n = 2, mean = 10, sd = 3)
```

```
## [1]  9.309468 14.676125
```

```r
rnorm(n = 3, mean = -3, sd = 5)
```

```
## [1] -2.647458 -2.353561  5.575325
```

The mapply() function

```r
set.seed(123)

sample_size <- 1:3
mu <- c(5, 10, -3)
sigma <- c(1, 3, 5)

mapply(rnorm, mean = mu, sd = sigma, n = sample_size)
```

```
## [[1]]
## [1] 4.439524
##
## [[2]]
## [1]  9.309468 14.676125
##
## [[3]]
## [1] -2.647458 -2.353561  5.575325
```

The mapply() function

Now suppose we wanted our results with two decimal places only:

```
set.seed(123)

results <- mapply(rnorm, mean = mu, sd = sigma,
                  n = sample_size)
```

```
sapply(results, FUN = round, 2)
```

```
## [[1]]
## [1] 4.44
##
## [[2]]
## [1]  9.31 14.68
##
## [[3]]
## [1] -2.65 -2.35  5.58
```

mapply() vs for loop

```
set.seed(123)

sample_size <- 1:3
mu <- c(5, 10,-3)
sigma <- c(1, 3, 5)

res <- vector(mode = "list")

for (i in 1:3) {
  res[[i]] <- round(rnorm(mean = mu[i],
                          sd = sigma[i],
                          n = sample_size[i]),
                    2)
}
```

mapply() vs for loop

```r
res
```

```
## [[1]]
## [1] 4.44
##
## [[2]]
## [1]  9.31 14.68
##
## [[3]]
## [1] -2.65 -2.35  5.58
```