SCR Week 4: live coding

apply() and tapply() | Session 1

```r
med_example <- data.frame(
  patient = 1:100,
  age = rnorm(100, mean = 60, sd = 12),
  treatment = factor(
    rep(c(1,2), 50),
    labels = c("treatm", "control")  print: label会显示出来
  ),
  hospital = factor(
    rep(c(1,2), each = 50),
    labels = c("A", "B")
  )
)
set.seed(42)
med_example$QL <- 30 + 10*(med_example$hospital == "A") + 30*(med_example$treatment == "treatm") + rnor
```

apply()

tapply() or aggregate()

```r
#med_example$QL[1] <- NA
output_tapply <- tapply(
  X = med_example$QL,
  INDEX = list(
    med_example$hospital,
    med_example$treatment
  ),
  FUN = mean
)
```

```r
output_tapply
```

```
##     treatm  control
## A 80.59825 48.68832
## B 70.13616 41.87787
```

Instead of tapply(), we could use aggregate() as well:

```r
aggregate(
  QL ~ hospital + treatment,
  FUN = mean,
  data = med_example
)
# aggregate is more generic..
aggregate(weight ~ feed, data = chickwts, mean)
```

apply() function

apply(X, MARGIN, FUN)
Here:
-x: an array or matrix:
-MARGIN=1`: the manipulation is performed on rows
-MARGIN=2`: the manipulation is performed on columns
-MARGIN=c(1,2)` the manipulation is performed on rows and columns
-FUN: mean, median, sum, min, max or user-defined functions

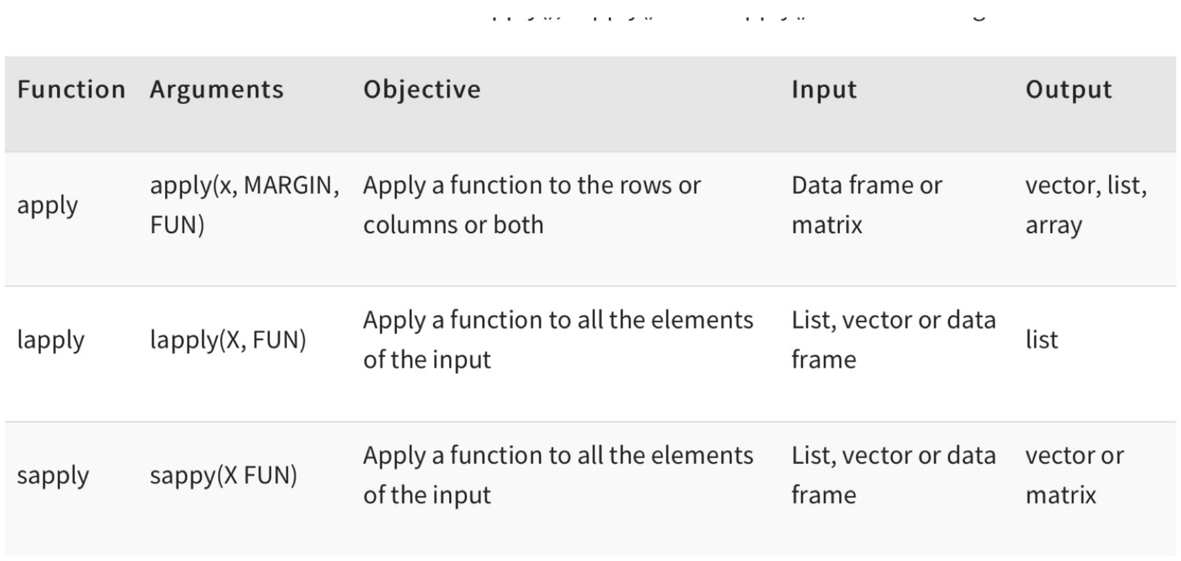
lapply() function

l in lapply() stands for list. The difference between lapply() and apply() lies between the output return. The output of lapply() is a list. lapply() can be used for other objects like data frames and lists.

We can use unlist() to convert the list into a vector.

sapply() function

sapply() function does the same jobs as lapply() function but returns a vector



Function	Arguments	Objective	Input	Output
apply	apply(x, MARGIN, FUN)	Apply a function to the rows or columns or both	Data frame or matrix	vector, list, array
lapply	lapply(X, FUN)	Apply a function to all the elements of the input	List, vector or data frame	list
sapply	sappy(X FUN)	Apply a function to all the elements of the input	List, vector or data frame	vector or matrix

tapply() function

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.

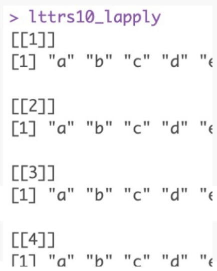
lapply() / sapply() | Session 2

```r
lttrs10_lapply <- lapply(
  X = 1:10,
  FUN = function(i) {
    sample(letters[1:10], i, replace = TRUE)
  }
)
```

```r
lttrs10_sapply <- sapply(
  X = 1:10,
  FUN = function(i) {
    letters[1:10]
  }
)
lttrs10_replicate <- replicate(10, sample(letters[1:10]))
```

Some examples for simplify2array(), unlist(), and do.call().

```r
lttrs10_lapply <- lapply(
  X = 1:10,
  FUN = function(i) {
    letters[1:10]
  }
)
```



```
> lttrs10_lapply
[[1]]
[1] "a" "b" "c" "d" "

[[2]]
[1] "a" "b" "c" "d" "

[[3]]
[1] "a" "b" "c" "d" "

[[4]]
[1] "a" "b" "c" "d" "
```

```r
simplify2array(lttrs10_lapply)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,] "a"  "a"  "a"  "a"  "a"  "a"  "a"  "a"  "a"  "a"
##  [2,] "b"  "b"  "b"  "b"  "b"  "b"  "b"  "b"  "b"  "b"
##  [3,] "c"  "c"  "c"  "c"  "c"  "c"  "c"  "c"  "c"  "c"
##  [4,] "d"  "d"  "d"  "d"  "d"  "d"  "d"  "d"  "d"  "d"
##  [5,] "e"  "e"  "e"  "e"  "e"  "e"  "e"  "e"  "e"  "e"
##  [6,] "f"  "f"  "f"  "f"  "f"  "f"  "f"  "f"  "f"  "f"
##  [7,] "g"  "g"  "g"  "g"  "g"  "g"  "g"  "g"  "g"  "g"
##  [8,] "h"  "h"  "h"  "h"  "h"  "h"  "h"  "h"  "h"  "h"
##  [9,] "i"  "i"  "i"  "i"  "i"  "i"  "i"  "i"  "i"  "i"
## [10,] "j"  "j"  "j"  "j"  "j"  "j"  "j"  "j"  "j"  "j"
```

```r
do.call('rbind', lttrs10_lapply)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,] "a"  "b"  "c"  "d"  "e"  "f"  "g"  "h"  "i"  "j"
##  [2,] "a"  "b"  "c"  "d"  "e"  "f"  "g"  "h"  "i"  "j"
##  [3,] "a"  "b"  "c"  "d"  "e"  "f"  "g"  "h"  "i"  "j"
##  [4,] "a"  "b"  "c"  "d"  "e"  "f"  "g"  "h"  "i"  "j"
##  [5,] "a"  "b"  "c"  "d"  "e"  "f"  "g"  "h"  "i"  "j"
##  [6,] "a"  "b"  "c"  "d"  "e"  "f"  "g"  "h"  "i"  "j"
##  [7,] "a"  "b"  "c"  "d"  "e"  "f"  "g"  "h"  "i"  "j"
##  [8,] "a"  "b"  "c"  "d"  "e"  "f"  "g"  "h"  "i"  "j"
##  [9,] "a"  "b"  "c"  "d"  "e"  "f"  "g"  "h"  "i"  "j"
## [10,] "a"  "b"  "c"  "d"  "e"  "f"  "g"  "h"  "i"  "j"
```

```
matrix_elements <- unlist(lttrs10_lapply)
dim(matrix_elements) <- c(10, 10)
my_matrix <- matrix_elements
```

```
lapply(data.frame(my_matrix), print)
```

```
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
```

```
## $X1
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##
## $X2
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##
## $X3
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##
## $X4
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##
## $X5
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##
## $X6
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
```

```
##
## $X7
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##
## $X8
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##
## $X9
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
##
## $X10
##  [1] a b c d e f g h i j
## Levels: a b c d e f g h i j
```

Explanation of the `for`-loop | Session 3

- `for(var in seq) expr`
- for(item in sequence) {body with expressions}
- for(element in container) {“body with expressions”}

```
seq <- 1:3
for (var in seq) {
  print(var)
}
var    var = 3
```

Note that `for` sets `var` to the last used element of `seq`

```
seq <- 1:3
for (var in seq) {
  print(var)
}; rm(var)
```

We remove `var` in case you might use it in your lexical scope (the environment you are coding in)

Then, something else which is nice with the `for` loop. In `for(var in seq) expr` we can retrieve objects that created in the previous iteration(s) (It has memory!).

```
a <- 3
for (i in  1:3) {
  a <- a + i
  print(a)
}
a; i
rm(list = c("a", 'i'))
```

The `<*>apply` functions do not have these properties. For example,

```
a <- 3
lapply(1:3, function(i) { a <- a + i; print(a) } )
```

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

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

```
a
```

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

```
i
```

```
## Error in eval(expr, envir, enclos): object 'i' not found
```

Avoid extending containers!!!

`if else` | Session 4

My grades for SCR 19-20

```r
my_A  <- 8.5
my_E1  <- 8.5
my_E2  <- 8
```

To get explanation for `if(cond) expr`

```r
?"if"
```

An example of `if(cond) expr`

```r
if(my_E2 >= my_E1) my_E2
if(my_E2 <= my_E1) my_E2
```

```
## [1] 8
```

An Example for `if(cond) cons.expr else alt.expr`

```r
if(my_E2 >= my_E1) my_E2 else (my_E1 + my_E2)/2
```

```
## [1] 8.25
```

the same, but perhaps more readable?

```r
if(my_E2 >= my_E1) {
  my_E2
} else {
  (my_E1 + my_E2)/2
}
```

```
## [1] 8.25
```

How about scoping?

```r
if(my_E2 >= my_E1) {
  char_log <- "the second exam was better or equal"
  my_E <- my_E2
} else {
  char_log <- "the first exam was better"
  my_E <- (my_E1 + my_E2)/2
}
```

using a loop and conditionals `if else`

```r
grades <- trunc(runif(100, 10, 100))/10 # uniform grades

n <- length(grades)
grd_rnd <- numeric(n)

for(i in 1:n) {
  grade <- grades[i]
  tmp <- trunc(grade)
  if(grade - tmp >= 0.5) {
    grd_rnd[i] <- ceiling(grade)
  } else {
    grd_rnd[i] <- floor(grade)
  }
}
```

`while` and `repeat` | Session 5

```
i <- 1
while (i < 3) {
  print(i)
  i <- i + 1
}
```

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

```
i
```

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

Introducing `break` and `repeat expr`:

```
i <- 1
repeat {
  print(i)
  i <- i + 1
  if(i >= 3) break # break out the loop
}
```

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

```
i
```

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

A commonality between `for`,`repeat`,`while`:

```
for(i in 1:3) {
  if(i >= 3) break
  print(i)
}
```

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

```
i
```

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

A superstitious `while(cond) expr`

```r
i <- 1
while (i < 16) {
  if(i == 4 || i == 13) {
    i <- i + 1
    next # go directly to the next iteration
  }
  print(i)
  i <- i + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 14
## [1] 15
```

BAD CODE: introducing extending containers / growing variables

```r
i <- 1
a <- 1
repeat {
  i <- i + 1
  if(i >= 16) break
  print(i)
  a[i] <- a[i - 1] + i # cumulative sum
}
```

```
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
```

```r
a
```

```
##  [1]   1   3   6  10  15  21  28  36  45  55  66  78  91 105 120
```

GOOD PRACTICE:

```
i <- 1
a <- numeric(15);
repeat {
  i <- i + 1
  if(i >= 15) break
  print(i)
  a[i] <- a[i - 1] + i # cumulative sum
}
```

```
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
```

```
a
```

```
##  [1]   0   2   5   9  14  20  27  35  44  54  65  77  90 104   0
```

Vectorization? | Session 6? If we have time...

```
set.seed(42)
a2 <- sample(c(0,1), 5, replace = TRUE)
a2 <- cumsum(a2)
```

Cumulative

```
n <- 5; a <- 0; set.seed(42)
for(i in 1:n) {
  a <- a + sample(c(0,1), 1)
  cat("i = ", i, "and a = ", a, "\n")
}
```

```
n <- 5;
a1 <- numeric(n)
set.seed(42)
for(i in 1:n) {
  if(i == 1) {
    a1[i] <- sample(c(0,1), 1)
  }
  if(i < n) {
    a1[i + 1] <- a1[i] + sample(c(0,1), 1)
  }
  cat("i = ", i, "and a = ", a[i], "\n")
}
a1
```

```
system.time(replicate(1e4, {
  set.seed(42)
  a2 <- sample(c(0,1), 5, replace = TRUE)
  a2 <- cumsum(a2)
}))
```

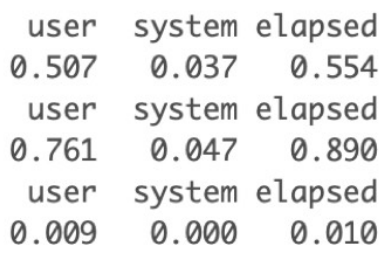
```
system.time(replicate(1e4, {
  n <- 5;
  a1 <- numeric(n)
  set.seed(42)
  for(i in 1:n) {
    if(i == 1) {
      a1[i] <- sample(c(0,1), 1)
    }
    if(i < n) {
      a1[i + 1] <- a1[i] + sample(c(0,1), 1)
    }
  }
  a1
}))
```

```
##    user  system elapsed
##   0.290   0.019   0.311
```

An example to figure out youself in your own time:

```r
x <- matrix(rnorm(1e6), 1e2, 1e4)
x_sds1 <- numeric(1e3)
for(i in 1:ncol(x)) x_sds1[i] <- mean(x[,i])
x_sds2 <- apply(X = x, MARGIN = 2, mean)
x_sds3 <- colMeans(x)
all.equal(x_sds1, x_sds2, x_sds3)
```

```r
system.time(replicate(10, {
  x_sds1 <- numeric(1e3)
  for(i in 1:ncol(x)) x_sds1[i] <- mean(x[,i])
}))
system.time(replicate(10, {
  x_sds2 <- apply(X = x, MARGIN = 2, mean)
}))
system.time(replicate(10, {
  x_sds3 <- colMeans(x)
}))
```



```
 user  system elapsed 
0.507   0.037   0.554 
 user  system elapsed 
0.761   0.047   0.890 
 user  system elapsed 
0.009   0.000   0.010 
```