ISOM 3390: Business Programming in R

Instructor: Bingjie Qian (PhD candidate)

Summer 2020, HKUST

Topic 5: Loop Functions

5.1 An Overview of Loop Functions

An example.

Create a data frame that contains three columns:

```
set.seed(0)
df <- data.frame(a = rnorm(10), b = runif(10, 0, 20), c = rnorm(10, 4, 8))
df
```

```
##                a          b          c
## 1   1.262954285 15.5489044  0.7079133
## 2  -0.326233361 18.6941046  6.0177876
## 3   1.329799263  4.2428504 -3.1353690
## 4   1.272429321 13.0334753  7.4854664
## 5   0.414641434  2.5111019 -5.9003074
## 6  -1.539950042  5.3444134  2.2058569
## 7  -0.928567035  7.7222819  7.0191652
## 8  -0.294720447  0.2678067  5.0666909
## 9  -0.005767173  7.6477591 10.4335161
## 10  2.404653389 17.3938169  3.5431458
```

Suppose we want to compute the mean of each column, we can do this with three lines of code:

```
mean(df$a)
```

```
## [1] 0.358924
```

```
mean(df$b)
```

```
## [1] 9.240651
```

```
mean(df$c)
```

```
## [1] 3.344387
```

What if the data frame has many columns?

- We can use a `for` loop.

```
output <- vector("numeric", ncol(df))
for (i in seq_along(df)) {
  output[i] <- mean(df[[i]])
}
output

## [1] 0.358924 9.240651 3.344387
```

Problems with loops:

- Loops are slow and not very expressive.
- Writing loops is not particularly easy when working interactively with the console.

Recall that we can use **vectorized operations** on vectors:

```
v1 <- c(1, 3, 4, 5)
v2 <- c(2, 4, 8, 9)
v1 + v2

## [1]  3  7 12 14
```

Can we **vectorize** the `mean()` function to make it operate on "whole objects" (e.g., a data frame vs. its individual columns), just as the operator + does?

In R, there is a family of functions that give us the ability to *apply functions to each element of a vector, a list, or an array*.

With them, we can wrap up the `for` loop in a *function*, and call that function instead of using the `for` loop directly.

The `apply` functions

- `lapply()` applies a function over a list or vector.
- `sapply()` produces a result of the simplest type possible.
- `mapply()` applies a function over a set of arguments.
- `apply()` evaluates a function over the margins of an array.
- `tapply()` groups the elements of a vector and applies a function over the resulted subsets.

5.2 List Apply: `lapply()`

Usage:

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

- `X` is a vector or list, `FUN` is a function.

- `lapply(X, FUN, ...)` applies the function `FUN` to each element of the vector or list `X`, and returns a list of the same length as `X`.
- "lapply" is short for "list apply".

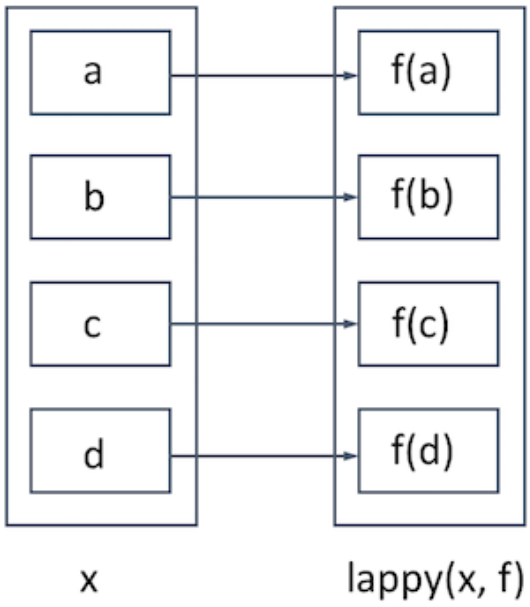

Fig 1. lapply()

Revisit the previous example, using `lapply()`:

```
output <- lapply(df, mean)   # data frame is a list of equal-length vectors
output    # return a list

## $a
## [1] 0.358924
##
## $b
## [1] 9.240651
##
## $c
## [1] 3.344387

mean(df[,1])

## [1] 0.358924

mean(df[,2])

## [1] 9.240651

mean(df[,3])

## [1] 3.344387
```

Specifying Other Arguments

What if the function `FUN` has more than one parameter?

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

- The elements of `X` are always supplied as the 1st argument of `FUN`.
- `...` takes the remaining arguments and passes them down to `FUN`.

In the following example, we use the function `runif()`, which has three parameters:

```
str(runif)

## function (n, min = 0, max = 1)
```

`runif(n, min = 0, max = 1)` draws `n` samples from a uniform distribution on the interval from `min` to `max`.

```
lapply(1:3, runif)

## [[1]]
## [1] 0.6927316
##
## [[2]]
## [1] 0.4776196 0.8612095
##
## [[3]]
## [1] 0.43809711 0.24479728 0.07067905
```

The elements of the output are `runif(1)`, `runif(2)`, `runif(3)`, respectively.

Specify the argument `max` for `runif`:

```
set.seed(0)
lapply(1:3, runif, max = 10)

## [[1]]
## [1] 8.966972
##
## [[2]]
## [1] 2.655087 3.721239
##
## [[3]]
## [1] 5.728534 9.082078 2.016819
```

The elements of the output are `runif(1, max=10)`, `runif(2, max=10)`, `runif(3, max=10)`, respectively.

Use of Anonymous Functions

What if we want to pass the elements of `X` as an argument of `FUN` other than the 1st one?

For example, suppose we want to pass the elements of `1:3` to the parameter `max` of `runif()`.

We can use an **anonymous function**.

```
set.seed(0)
lapply(1:3, function(x, ...) runif(3, max = x, ...), min = 1)
```

```
## [[1]]
## [1] 1 1 1
##
## [[2]]
## [1] 1.896697 1.265509 1.372124
##
## [[3]]
## [1] 2.145707 2.816416 1.403364
```

In the aboe code, `function(x, ...) runif(3, max = x, ...)` is an anonymous function that passes its first argument `x` to the parameter `max` of `runif()`.

The elements of the output are `runif(3, max=1, min=1)`, `runif(3, max=2, min=1)`, `runif(3, max=3, min=1)`, respectively.

`lapply()` and other members in the `apply` family of functions make heavy use of anonymous functions.

Applying a List of Functions

Recall that functions are **first-class** objects in R, in that they can be computed, passed, stored, etc., just like other objects.

So we can use `lapply()` to *apply a list of functions to a data object*.

```
data_summary <- list(mean, sum, median)  # a list of three functions
set.seed(0)
x <- runif(100)          # a data object
lapply(data_summary, function(f) f(x))
```

```
## [[1]]
## [1] 0.5207647
##
## [[2]]
## [1] 52.07647
##
## [[3]]
## [1] 0.4878107
```

In the above code, `function(f) f(x)` is an anonymous function that takes a function `f` as its argument and returns `f(x)`.

The elements of the output are `mean(x)`, `sum(x)`, `median(x)`, respectively.

Variations of `lapply()`

There are some other variations of `lapply()` that simply use different types of input or output:

- `sapply()` produces a result of the simplest type possible.
- `mapply()` applies a function over a set of arguments.
- `apply()` evaluates a function over the margins of an array.
- `tapply()` groups the elements of a vector and applies a function over the resulted subsets.

5.3 Vector Output: `sapply()`

`sapply()` (short for "simplified lapply") behaves similarly to `lapply()` except that it tries to **simplify the results** if possible.

```
lapply(df, mean)          # return a list

## $a
## [1] 0.358924
##
## $b
## [1] 9.240651
##
## $c
## [1] 3.344387

unlist(lapply(df, mean))

##        a        b        c
## 0.358924 9.240651 3.344387

sapply(df, mean)          # return a vector

##        a        b        c
## 0.358924 9.240651 3.344387
```

Essentially, `sapply()` calls `lapply()` on its input and then applies the following simplifying algorithm:

- If the result is a list where every element is of length 1, then a *vector* is returned.
- If the result is a list where every element is a vector of the same length (> 1), a *matrix* is returned.
- If it can't figure things out, a *list* is returned.

5.4 Multiple Inputs: `mapply()`

Usage:

```
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

- `mapply()` (short for "multivariate apply") is a **multivariate version** of `sapply()`.
- It applies the function `FUN` in parallel over a set of arguments in `...`.
- Arguments are *recycled* if necessary.

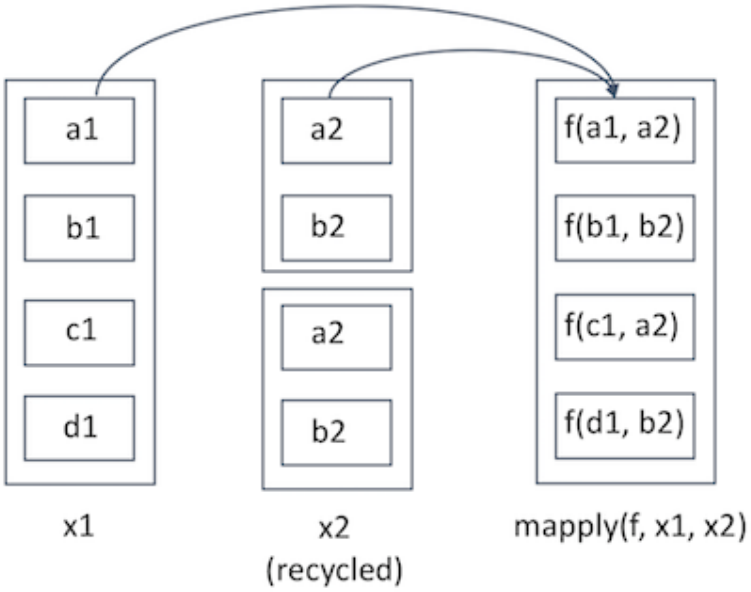

Fig 2. mapply()

Take the function `rnorm()` as an example:

```
str(rnorm)

## function (n, mean = 0, sd = 1)
```

`rnorm(n, mean=0, sd=1)` draws `n` samples from the normal distribution with mean equal to `mean` and standard deviation equal to `sd`.

```
set.seed(0)
mapply(rnorm, 1:4, 1:4, 2)  # 2 is recycled

## [[1]]
## [1] 3.525909
##
## [[2]]
## [1] 1.347533 4.659599
##
## [[3]]
```

```
## [1]  5.54485864  3.82928287 -0.07990008
##
## [[4]]
## [1] 2.142866 3.410559 3.988466 8.809307
```

The elements in the output are `rnorm(1, 1, 2)`, `rnorm(2, 2, 2)`, `rnorm(3, 3, 2)`, `rnorm(4, 4, 2)`, respectively.

Constant Inputs

In many statistical applications, we want to calculate the following value for a vector `x`:

$$\sum_{i=1}^{n} \frac{(x_i-\mu)^2}{\sigma^2}$$

sumsq

Implement it with an R function:

```
set.seed(0)
x <- rnorm(50)   # generate a vector `x`

sumsq <- function(mu, sigma, x) sum(((x - mu)/sigma)^2)
sumsq(1, 1, x)   # mu = 1, sigma = 1
```

```
## [1] 88.77574
```

What if we want to evaluate it for 10 different pairs of `mu` and `sigma`?

For example, suppose we want to calculate `sumsq(1, 1, x)`, `sumsq(2, 2, x)`, ..., `sumsq(10, 10, x)`.

Try the following code:

```
sumsq(1:10, 1:10, x)           # return a single value (pair-wise operation
on 3 vectors)
```

```
## [1] 51.70552
```

```
mapply(sumsq, 1:10, 1:10, x)   # return a vector of 50 values
```

```
##  [1] 0.06914496 1.35284041 0.30995228 0.46497761 0.84102053 1.57919018
##  [7] 1.28290154 1.07503730 1.00128200 0.57689290 0.05588805 1.95861319
## [13] 1.91145096 1.14996754 1.12326723 1.14187420 0.92923446 1.23541033
## [19] 0.90552495 1.26282270 1.49883185 0.65821122 0.91308449 0.63832529
## [25] 1.02297316 0.83917570 0.71383927 1.18019810 1.30583931 0.99067660
## [31] 1.52697069 1.61657017 1.30973553 1.35109916 0.73042637 0.65288777
## [37] 0.73661504 1.11026080 0.74375214 1.05664960 0.57441709 0.51786295
## [43] 1.32463524 1.45929015 1.52106369 1.38673806 1.49670129 0.73176540
## [49] 0.82364758 1.04598252
```

The parameter `MoreArgs` of `mapply()` takes a list of arguments that will be supplied as **constant inputs** to each call.

```
mapply(sumsq, 1:10, 1:10, MoreArgs = list(x))
```

```
##  [1] 88.77574 59.09566 53.77662 51.97478 51.16813 50.74473 50.49831
50.34413
##  [9] 50.24236 50.17238
```

In the above code, the vector `x` will be processed *as a whole*, rather than element-wise.

`Vectorize()`: Vectorize a Function

An alternative way to achieve this is to use **`Vectorize()`**.

Syntax:

```
Vectorize(FUN, vectorize.args = arg.names, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

- `Vectorize` creates a function wrapper that vectorizes the action of `FUN`.
- `vectorize.args` is a character vector of arguments which should be vectorized. Defaults to all arguments of `FUN`.

```
vsumsq <- Vectorize(sumsq, c("mu", "sigma"))    # argument `x` is not
vectorized
vsumsq(1:10, 1:10, x)
```

```
##  [1] 88.77574 59.09566 53.77662 51.97478 51.16813 50.74473 50.49831
50.34413
##  [9] 50.24236 50.17238
```

5.5 Array Input: `apply()`

Create a matrix `m` which has 2 rows and 4 columns:

```
set.seed(0)
m <- matrix(rnorm(12), 2, 4)
m
```

```
##            [,1]     [,2]       [,3]       [,4]
## [1,]  1.2629543 1.329799  0.4146414 -0.9285670
## [2,] -0.3262334 1.272429 -1.5399500 -0.2947204
```

Suppose we want to calculate the row means or column means of the matrix `m`.

Try the following code:

```
sapply(m, mean)    # `m` is treated as a vector of 8 values
```

```
## [1]  1.2629543 -0.3262334  1.3297993  1.2724293  0.4146414 -1.5399500 -
0.9285670
## [8] -0.2947204
```

`lapply()` and `sapply()` treat the matrices and arrays as though they were vectors, whereas `apply()` (short for "array apply") evaluates a function *over the margins of an array*.

Usage:

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

- The `MARGIN` argument essentially indicates the dimension(s) that the function will be applied over (the dimension(s) that will be preserved).
- E.g., for a matrix, `1` indicates rows, `2` indicates columns, `c(1, 2)` indicates rows and columns.

```
apply(m, 1, mean)   # row mean

## [1]  0.5197070 -0.2221186

apply(m, 2, mean)   # column mean

## [1]  0.4683605  1.3011143 -0.5626543 -0.6116437
```

- We can pass the optional arguments of `FUN` via `...`:

```
apply(m, 1, quantile, probs = c(0.25, 0.75))  # the 1st and 3rd quartiles

##           [,1]       [,2]
## 25% 0.07883932 -0.6296625
## 75% 1.27966553  0.0970670

apply(m, 1, quantile)     # `probs` defaults to seq(0, 1, 0.25)

##             [,1]       [,2]
## 0%   -0.92856703 -1.5399500
## 25%   0.07883932 -0.6296625
## 50%   0.83879786 -0.3104769
## 75%   1.27966553  0.0970670
## 100%  1.32979926  1.2724293
```

- We can preserve more than one dimension:

```
set.seed(0)
a <- array(rnorm(40), c(2, 2, 10))   # an array with 3 dimensions
apply(a, c(1, 2), mean)              # preserve the 1st and 2nd dimension

##             [,1]       [,2]
## [1,] -0.02307290  0.4249748
## [2,] -0.07941677 -0.1027437
```

Col/Row Sums and Means

There are shorthand functions that perform column/row sums and means, which are much faster and more descriptive.

```
rowSums(m) <=> apply(m, 1, sum)
rowMeans(m) <=> apply(m, 1, mean)
colSums(m) <=> apply(m, 2, sum)
colMeans(m) <=> apply(m, 2, mean)
```

```
apply(m, 2, sum)
```

```
## [1]  0.9367209  2.6022286 -1.1253086 -1.2232875
```

```
colSums(m)
```

```
## [1]  0.9367209  2.6022286 -1.1253086 -1.2232875
```

5.6 Group Apply: `tapply()`

Example 1: Game Scores

```
# create a data frame:
set.seed(1)
game_scores <- data.frame(player = rep(c("Nick", "Charles", "Samuel"), times
= c(4, 3, 5)), score = round(rlnorm(12, 8), -1))
game_scores
```

```
##     player score
## 1     Nick  1590
## 2     Nick  3580
## 3     Nick  1290
## 4     Nick 14700
## 5  Charles  4140
## 6  Charles  1310
## 7  Charles  4850
## 8   Samuel  6240
## 9   Samuel  5300
## 10  Samuel  2200
## 11  Samuel 13520
## 12  Samuel  4400
```

Question: How can we calculate some statistic on a variable that has been split into groups? For example, how to calculate the mean `score` of each `player`?

First, we need to split the data frame into groups using **`split()`**.

Usage:

```
split(x, f, drop = FALSE, ...)
```

- `x` is a vector or data frame containing values to be divided into groups.
- `f` is a factor in the sense that `as.factor(f)` defines the grouping, or a list of such factors in which case their interaction is used for the grouping.

- `split` divides the data in the vector `x` into the groups defined by `f` and returns a *list* of vectors containing the values for each group. The components of the list are named by the levels of `f`.

```
scores_by_player <- split(game_scores$score, game_scores$player)
scores_by_player   # a list of vectors

## $Charles
## [1] 4140 1310 4850
##
## $Nick
## [1]  1590  3580  1290 14700
##
## $Samuel
## [1]  6240  5300  2200 13520  4400
```

Next, apply a function (e.g., `mean()`) to each element of the returned list.

```
means_by_player <- sapply(scores_by_player, mean)
means_by_player

##  Charles     Nick   Samuel
## 3433.333 5290.000 6332.000
```

Alternatively:

```
game_by_player <- split(game_scores, game_scores$player)  # a list of data
frames
sapply(game_by_player, function(x) mean(x$score))

##  Charles     Nick   Samuel
## 3433.333 5290.000 6332.000
```

Example 2: Email Exchange

First, download the data set `small_corpus.csv` from Canvas. Then, read the data:

```
small_corpus <- read.csv("small_corpus.csv", colClasses = c("character",
"factor", "factor", "integer"))
head(small_corpus)

##         time from  to n
## 1 1999-01-04  114  65 2
## 2 1999-01-04  114 169 2
## 3 1999-01-07  114 110 4
## 4 1999-01-07  114 112 4
## 5 1999-01-07  114 169 2
## 6 1999-01-08  114 145 2
```

This data set stores the email records of some firm employees. `from` and `to` are employee IDs, `n` is the number of emails from `from` to `to` on a particular day.

```r
length(levels(small_corpus$from))  # 11
```

```
## [1] 11
```

```r
length(unique(small_corpus$to))    # 21
```

```
## [1] 21
```

Question: How to quantify the intensity of email exchange between each pair of employees?

```r
small_corpus_by_pair <- split(small_corpus$n, list(small_corpus$from,
small_corpus$to))
# `f` is a list of factors whose interaction is used for the grouping
head(small_corpus_by_pair)
```

```
## $`107.107`
## integer(0)
##
## $`112.107`
## [1] 2
##
## $`114.107`
## [1]  2  2  4 12  4
##
## $`145.107`
## integer(0)
##
## $`155.107`
## integer(0)
##
## $`160.107`
## integer(0)
```

The interaction between `from` and `to` results in many levels that are empty.

```r
length(small_corpus_by_pair)   # 231
```

```
## [1] 231
```

231 = 11 * 21, every pair of `from` and `to` is listed.

We can drop empty levels when calling the `split()` function by setting `drop = TRUE`:

```r
small_corpus_by_pair <- split(small_corpus$n, list(small_corpus$from,
small_corpus$to), drop = TRUE)
length(small_corpus_by_pair)    # 41
```

```
## [1] 41
```

```r
head(small_corpus_by_pair)
```

```
## $`112.107`
## [1] 2
```

```
##
## $`114.107`
## [1]  2  2  4 12  4
##
## $`38.11`
## [1] 2 2 4 2 2
##
## $`114.110`
## [1] 4 2 2 2 4 2
##
## $`155.110`
## [1] 4 2 2 4 2 2
##
## $`169.110`
## [1] 2 2 4 2 2 2
```

```
sapply(small_corpus_by_pair, sum)
```

```
## 112.107 114.107   38.11 114.110 155.110 169.110 114.112  38.112 107.114
112.114
##       2      24      12      16      16      14      12       2      14
2
## 155.114 169.114  38.114 114.123 145.128 114.145 155.145 114.155 169.155
114.160
##      16      50       2       2       2       8       2      56       8
2
##  65.160 114.165 169.165  50.167 114.169 155.169  114.22  160.22   65.22
114.29
##       2      10      14      16      60       4       4       5       2
2
##  114.38   65.38  169.46  114.50   50.50   169.6  112.65  114.65   22.65
38.65
##      14       2       2       2       2       2       2      10       2
4
##   38.96
##       2
```

Split-and-Apply Paradigm and `tapply()`

The above two examples show a very common **split-and-apply paradigm** in data analysis. The basic pipeline is as follows:

- Split a data set into pieces (e.g., according to some factor)
- Apply a function to each piece (e.g., mean())

We have an easier way to do this "split-and-apply" paradigm: **`tapply()`**.

Usage:

```
tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

- `tapply()` splits the elements of the vector `X` into groups defined by `INDEX`, and applies the function `FUN` over each group.
- `INDEX` is a list of one or more factors, each of the same length as `X`. The elements are coerced to factors by `as.factor`.
- `tapply()` can be thought of as a combination of `split()` and `sapply()` for vectors only.

```
# the game score example:
tapply(game_scores$score, game_scores$player, mean)

##  Charles     Nick   Samuel
## 3433.333 5290.000 6332.000

# the email exchange example:
tapply(small_corpus$n, list(small_corpus$from, small_corpus$to), sum)

##     107 11 110 112 114 123 128 145 155 160 165 167 169 22 29 38 46 50  6
65 96
## 107  NA NA  NA  NA  14  NA  NA  NA  NA  NA  NA  NA  NA NA NA NA NA NA NA
NA NA
## 112   2 NA  NA  NA   2  NA  NA  NA  NA  NA  NA  NA  NA NA NA NA NA NA NA
2 NA
## 114  24 NA  16  12  NA   2  NA   8  56   2  10  NA  60  4  2 14 NA  2 NA
10 NA
## 145  NA NA  NA  NA  NA  NA   2  NA  NA  NA  NA  NA  NA NA NA NA NA NA NA
NA NA
## 155  NA NA  16  NA  16  NA  NA   2  NA  NA  NA  NA   4 NA NA NA NA NA NA
NA NA
## 160  NA NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  5 NA NA NA NA NA
NA NA
## 169  NA NA  14  NA  50  NA  NA  NA   8  NA  14  NA  NA NA NA NA  2 NA  2
NA NA
## 22   NA NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA NA NA NA NA NA NA
2 NA
## 38   NA 12  NA   2   2  NA  NA  NA  NA  NA  NA  NA  NA NA NA NA NA NA NA
4  2
## 50   NA NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  16  NA NA NA NA NA  2 NA
NA NA
## 65   NA NA  NA  NA  NA  NA  NA  NA  NA   2  NA  NA  NA  2 NA  2 NA NA NA
NA NA
```

[Task 1: The Split-and-Apply Pattern and `tapply()`]

`strikes.csv` contains the data on the political economy of strikes. Read the data into your R session and call the resulting data frame `strikes.df`.

```
strikes.df <- read.csv("strikes.csv")
head(strikes.df)
```

```
##     country year strike.volume unemployment inflation left.parliament
## 1 Australia 1951           296          1.3      19.8            43.0
## 2 Australia 1952           397          2.2      17.2            43.0
## 3 Australia 1953           360          2.5       4.3            43.0
## 4 Australia 1954             3          1.7       0.7            47.0
## 5 Australia 1955           326          1.4       2.0            38.5
## 6 Australia 1956           352          1.8       6.3            38.5
##   centralization density
## 1      0.3748588      NA
## 2      0.3751829      NA
## 3      0.3745076      NA
## 4      0.3710170      NA
## 5      0.3752675      NA
## 6      0.3716072      NA
```

It has 625 observations from 18 countries in 1951-1985 (35 years) and 8 columns: country, year, days on strike per 1000 workers, unemployment, inflation, left-wing share of governmentt, centralization of unions, and union density.

Note that since $18 \times 35 = 630 > 625$, some years are missing from some countries.

(a) Split `strikes.df` by country

Use the `split()` function. Call the resulting list `strikes.by.country`, and show the names of elements of the list, as well as the first 5 rows of the data frame for Canada.

(b) Compute country-level averages for unemployment rate

Use `strikes.by.country` and `sapply()`, compute the average unemployment rate for each country. The expected output is the following:

```
  Australia     Austria     Belgium      Canada     Denmark     Finland
France     Germany
  3.5057143   2.5400000   3.6466667   6.0428571   5.7114286   2.5714286
3.1828571   3.1171429
    Ireland       Italy       Japan Netherlands New.Zealand      Norway
Sweden Switzerland
  7.7714286   6.7257143   1.6028571   3.6914286   1.0028571   1.4285714
2.1371429   0.3285714
         UK         USA
  3.4514286   5.5428571
```

(c) Compute country-level averages for union density

Use `strikes.by.country` and `sapply()`, compute the average union density for each country. The expected output is the following:

```
  Australia     Austria     Belgium      Canada     Denmark     Finland
France     Germany
```

```
   47.78462    60.67692    45.51429    31.83846    66.93077    54.63077
19.61538    33.81538
    Ireland       Italy       Japan Netherlands New.Zealand      Norway
Sweden Switzerland
   58.64231    38.53462    33.21923    36.13846    62.91111    56.92308
72.96154    31.35769
         UK         USA
   45.19231    24.78077
```

Tips: enable `sapply()` to take the named argument `na.rm` for `mean` so as to handle missing values.

(d) Compute multiple groupwise avarages

Using `strikes.by.country` and just one call to `sapply()`, compute the average unemployment rate, inflation rate, and strike volume for each country. The output should be a *matrix* of dimension 3 x 18. Also, with just the one call to `sapply()`, figure out how to make the output matrix have appropriate row names. The expected output is the following:

```
               Australia   Austria    Belgium     Canada    Denmark
Finland     France
unemployment    3.505714  2.540000   3.646667   6.042857   5.711429
2.571429   3.182857
inflation       6.594286  5.102857   4.150000   4.797143   6.582857
7.317143   6.948571
strike.volume 378.600000 25.600000 244.000000 749.542857 194.828571
448.542857 185.400000
               Germany    Ireland      Italy      Japan Netherlands
New.Zealand    Norway
unemployment   3.117143   7.771429   6.725714   1.602857    3.691429
1.002857  1.428571
inflation      3.294286   8.151429   8.005714   5.820000    4.814286
7.691429  6.320000
strike.volume 43.828571 547.428571 997.685714 165.828571   26.114286
259.257143 75.114286
                 Sweden Switzerland         UK        USA
unemployment   2.137143   0.3285714   3.451429   5.542857
inflation      6.434286   3.4171429   7.105714   4.428571
strike.volume 73.485714   3.6571429 322.714286 448.228571
```

Tips: define a (possibly anonymous) function that computes these three averages and return them in a named vector. Recall that for `sapply()`, if the result is a list where every element is a vector of the same length (> 1), a *matrix* is returned.

(e) Let's say we want to compute multiple groupwise averages in a more compact way. Edit the following code to provide the passed-in function, which calls `tapply()` to compute group-wise means.

```
sapply(strikes.df[c("unemployment", "inflation", "strike.volume")], # put
your function definition here
       )
```

The expected output is the following:

```
             unemployment inflation strike.volume
Australia       3.5057143  6.594286    378.600000
Austria         2.5400000  5.102857     25.600000
Belgium         3.6466667  4.150000    244.000000
Canada          6.0428571  4.797143    749.542857
Denmark         5.7114286  6.582857    194.828571
Finland         2.5714286  7.317143    448.542857
France          3.1828571  6.948571    185.400000
Germany         3.1171429  3.294286     43.828571
Ireland         7.7714286  8.151429    547.428571
Italy           6.7257143  8.005714    997.685714
Japan           1.6028571  5.820000    165.828571
Netherlands     3.6914286  4.814286     26.114286
New.Zealand     1.0028571  7.691429    259.257143
Norway          1.4285714  6.320000     75.114286
Sweden          2.1371429  6.434286     73.485714
Switzerland     0.3285714  3.417143      3.657143
UK              3.4514286  7.105714    322.714286
USA             5.5428571  4.428571    448.228571
```

[End of Task 1]

5.7 Summary of `apply` Functions

The `apply` Functions

- `lapply()` applies a function over a list or vector.
- `sapply()` produces a result of the simplest type possible.
- `mapply()` applies a function over a set of arguments.
- `apply()` evaluates a function over the margins of an array.
- `tapply()` groups the elements of a vector and applies a function over the resulted subsets.

Pros and Cons of `apply` Functions Compared to `for` Loops

Pros:

- The code is cleaner (once we're familiar with the concept), easier to code and read, and less error prone because we don't have to deal with subsetting and saving the results.
- `apply` functions can be faster than `for` loops, sometimes dramatically.

Cons:

- Inconsistent syntax.
  - E.g., with `tapply()` and `sapply()`, the simplify parameter is called `simplify`. With `mapply()`, it's called `SIMPLIFY`. With `apply()`, the argument is absent.
- Cover only a partial set of all possible combinations of input and output types.

[Task 2: *apply() Functions]

Task 2 is revised and included in Assignment 1 No need to do it in in-class exercise

We're going to examine data from the 2016 Summer Olympics in Rio de Janeiro. This dataset includes the official statistics on the 11,538 athletes (6,333 men and 5,205 women) and 306 events at the 2016 Rio Olympic Games. You can read more about this data set at https://github.com/flother/rio2016.

Below we read in the data and store it as `rio`.

```
rio <- read.csv("rio.csv")
head(rio)
```

```
##          id           name nationality    sex date_of_birth height weight
## 1 736041664 A Jesus Garcia         ESP   male    1969-10-17   1.72     64
## 2 532037425     A Lam Shin         KOR female    1986-09-23   1.68     56
## 3 435962603    Aaron Brown         CAN   male    1992-05-27   1.98     79
## 4 521041435     Aaron Cook         MDA   male    1991-01-02   1.83     80
## 5  33922579     Aaron Gate         NZL   male    1990-11-26   1.81     71
## 6 173071782    Aaron Royle         AUS   male    1990-01-26   1.80     67
##       sport gold silver bronze info
## 1 athletics    0      0      0
## 2   fencing    0      0      0
## 3 athletics    0      0      1
## 4 taekwondo    0      0      0
## 5   cycling    0      0      0
## 6 triathlon    0      0      0
```

Use `rio` to answer the following 2 questions.

(a) Use `sapply` and `tapply` to count each type of medals (gold, silver and bronze) for each country. Name your result `medal_chart`. The expected output is the following:

```
    gold silver bronze
AFG    0      0      0
ALB    0      0      0
ALG    0      2      0
AND    0      0      0
ANG    0      0      0
ANT    0      0      0
ARG   21      1      0
```

```
ARM    1       3       0
ARU    0       0       0
ASA    0       0       0
AUS   23      34      25
AUT    0       0       2
AZE    1       7      10
BAH    1       0       5
BAN    0       0       0
BAR    0       0       0
BDI    0       1       0
......
```

Tips: similar to your answer to 1(e)

Once `medal_chart` is created, use `order()` function to sort countries in the medal chart first by gold, second by silver, and last by bronze.

(b) Among the countries that had zero medals, which had the most athletes? and how many athletes was this? Wrtie code to answer the question.

[End of Task 2]

5.8 The `plyr` Package

The `plyr` package allows us to smoothly apply the "split-and-apply" strategy.

- It builds on the built-in `*apply` functions and can be regarded as the generalization of `tapply()`.
- It provides a set of consistently named functions with consistently named arguments and gives us control over their input and output formats.
  - has a common syntax
  - requires less code since it takes care of the input and output format
  - can be run in parallel
- The basic format of `plyr` functions is two letters followed by `ply`, with the 1st letter referring to the format in and the 2nd to the format out.
- The letters include:
  - d = data frame
  - a = array (includes matrices)
  - l = list

E.g., `ddply` means: split a *data frame*, apply function, and return results in a *data frame*.

ddply()

Usage:

```
ddply(.data, .variables, .fun = NULL, ..., .progress = "none",
      .inform = FALSE, .drop = TRUE, .parallel = FALSE, .paropts = NULL)
```

- `ddply` splits the dataframe `.data`, applies the function `.fun` on ecah subset, then combines results into a data frame.
- `.data`: the data frame to be processed
- `.variables`: combination of variables to split by; support various specification syntax
  - Character: `c("from", "to")`
  - Numeric: `1:3`
  - Formula: `~ from + to`
- `.fun`: the function to call on each piece
- `...`: extra arguments passed to `.fun`

```
# install.packages("plyr")
library(plyr)
```

Now let's revisit the email exchange example.

Group-Wise Summaries

```
summary_corpus <- ddply(small_corpus, c("from", "to"), summarise, total =
sum(n), mean = mean(n))
nrow(summary_corpus)   # 41 rows, empty subsets are dropped

## [1] 41

head(summary_corpus)

##   from  to total     mean
## 1  107 114    14 3.500000
## 2  112 107     2 2.000000
## 3  112 114     2 2.000000
## 4  112  65     2 2.000000
## 5  114 107    24 4.800000
## 6  114 110    16 2.666667
```

The function `summarise()` creates a new data frame to summarise an existing data frame.

Try `summarise(small_corpus, total = sum(n), mean = mean(n))` to get a rough idea on `summarise()`.

Group-Wise Transformations:

```
normalize_corpus <- ddply(small_corpus, c("from", "to"), mutate, mean =
mean(n), sd = sd(n), normalized = (n - mean)/sd)
head(normalize_corpus)
```

```
##         time from  to n mean       sd normalized
## 1 1999-05-24  107 114 2  3.5 1.914854 -0.7833495
## 2 1999-05-25  107 114 4  3.5 1.914854  0.2611165
## 3 1999-06-09  107 114 6  3.5 1.914854  1.3055824
## 4 1999-06-11  107 114 2  3.5 1.914854 -0.7833495
## 5 1999-06-25  112 107 2  2.0       NA         NA
## 6 1999-05-11  112 114 2  2.0       NA         NA
```

Unlike `summarise()` that creates a new data frame, `mutate()` modifies a data frame by adding new or replacing existing columns.

The transformations are executated iteratively so that later transformations can use the columns created by earlier ones (e.g., the column `normalized` uses the columns `mean` and `sd`).

[Task 3: `ddply()` with the Games Scores Data]

```
set.seed(1)
game_scores <- data.frame(player = rep(c("Nick", "Charles", "Samuel"), times
= c(4, 3, 5)), score = round(rlnorm(12, 8), -1))
game_scores
```

```
##     player score
## 1     Nick  1590
## 2     Nick  3580
## 3     Nick  1290
## 4     Nick 14700
## 5  Charles  4140
## 6  Charles  1310
## 7  Charles  4850
## 8   Samuel  6240
## 9   Samuel  5300
## 10  Samuel  2200
## 11  Samuel 13520
## 12  Samuel  4400
```

(a) Use the function `ddply()` to generate a new data frame that summarises how many scores each player has, and the lowest, highest and mean score of each player. The expected result is as follows.

```
   player count  min   max     mean
1 Charles     3 1310  4850 3433.333
2    Nick     4 1290 14700 5290.000
3  Samuel     5 2200 13520 6332.000
```

(b) Use the function `ddply()` to add two columns to the data frame `game_scores`. Column `total` is the total score each player got, and the column `percent` is the percentage of each score in the total score the player got. The expected result is as follows.

```
    player score total percent
1  Charles  4140 10300   40.19
2  Charles  1310 10300   12.72
3  Charles  4850 10300   47.09
4     Nick  1590 21160    7.51
5     Nick  3580 21160   16.92
6     Nick  1290 21160    6.10
7     Nick 14700 21160   69.47
8   Samuel  6240 31660   19.71
9   Samuel  5300 31660   16.74
10  Samuel  2200 31660    6.95
11  Samuel 13520 31660   42.70
12  Samuel  4400 31660   13.90
```

[End of Task 3]