*apply() functions

Yufeng Huang

Associate Professor of Marketing, Simon Business School

August 16, 2022

Motivation: convert names into "lastname, firstname" format

```
# names of the marketing faculty
names <- c("Kristina Brecko", "Hana Choi", "Paul Ellickson", "Ron Goettler",
        "Avery Haviv", "Yufeng Huang", "Mitch Lovett", "Takeaki Sunada", "Paul Nelson")

# split them to turn into a list
names.splt <- strsplit(names, " ")

# check part of the result
names.splt[1:2]

## [[1]]
## [1] "Kristina" "Brecko"
##
## [[2]]
## [1] "Hana" "Choi"
```

For loop way

```
# initialize, as vector of same length
names.ror <- rep("", length(names))

# for each name, reorder it
for (i in 1:length(names)) {
        # paste0 combines characters without gap, so e.g.
        #    "Choi" combined with ", " and then with "Hana"
        names.ror[i] <- paste(names.splt[[i]][2], names.splt[[i]][1], sep = ", ")
}

# check result
names.ror

## [1] "Brecko, Kristina" "Choi, Hana"       "Ellickson, Paul"  "Goettler, Ron"
## [5] "Haviv, Avery"     "Huang, Yufeng"    "Lovett, Mitch"    "Sunada, Takeaki"
## [9] "Nelson, Paul"

# any problem with this?
```

The sapply() approach

```
# No particular problem with the previous for loop
# but code can be more elegant...

# lapply approach:
# 1) define a function on each vector of "splitted names"
#    e.g. c("Hana", "Choi") becomes scalar "Choi, Hana"
rev.names <- function(x) paste(x[2], x[1], sep = ", ")

# 2) sapply() "applies" the function to every element of the list names.splt
names.ror <- sapply(names.splt, rev.names)

# check result
names.ror

## [1] "Brecko, Kristina" "Choi, Hana"       "Ellickson, Paul"  "Goettler, Ron"
## [5] "Haviv, Avery"     "Huang, Yufeng"    "Lovett, Mitch"    "Sunada, Takeaki"
## [9] "Nelson, Paul"
```

*apply() functions

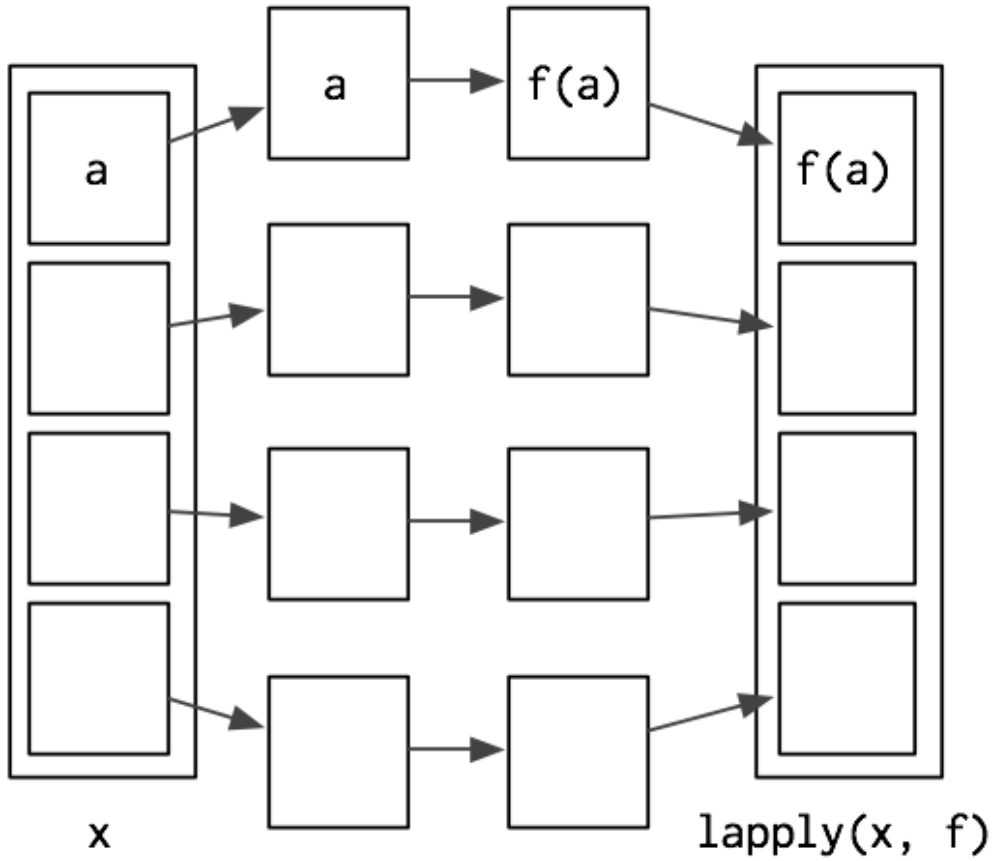
- There is a set of "vectorize" functions
  - lapply()
  - sapply()
  - apply()
  - ...
- They take a function (could be user defined)
  - and apply it to many different objects at the same time, avoiding loops
  - just like aggregate() does (but aggregate is a more specific function)

lapply() and sapply()

lapply()

- lapply() might be the simplest of all apply() functions
  - "l" stands for "list" apply
  - sapply() is lapply() with output simplification
  - "s" stands for "simplified" apply
- Takes a list or vector
  - can work on data frames because they are lists
- Applies a function to **every element** of the list (or vector)
- lapply() returns the output as a list
  - sapply() will simplify the output whenever it can

lapply()



source: Hadley Wickham

Simplest examples

```
# define a list
a <- list(1, 2, 3)      # same if a is a vector c(1, 2, 3)

# recall: cannot vectorize the calculation
log(a)

                ## Error in log(a):  non-numeric argument to mathematical function

# lapply approach
#    "X" and "FUN" are argument names
lapply(X = a, FUN = log)

## [[1]]
## [1] 0
##
## [[2]]
## [1] 0.6931472
##
## [[3]]
## [1] 1.098612
```

Also works with vectors

```
# what if a is a vector?
a <- c(1, 2, 3)

# now can vectorize
#    i.e. element-wise calculations
log(a)

## [1] 0.0000000 0.6931472 1.0986123

# lapply approach
lapply(a, log)

## [[1]]
## [1] 0
##
## [[2]]
## [1] 0.6931472
##
## [[3]]
## [1] 1.098612

# NOTE: can work with vector and result is still a list

# sapply will simplify
sapply(a, log)

## [1] 0.0000000 0.6931472 1.0986123
```

Simplest examples: break-down

```
# b as a list of expanding vectors
b <- list(1, 1:2, 1:3, 1:4, 1:5)

b

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

# for each element of b, calculate mean
res <- rep(NA, length(b))       # initialize
for (i in 1:length(b)) {
        res[i] <- mean(b[[i]])
}
res

## [1] 1.0 1.5 2.0 2.5 3.0

# compare with sapply(),
#    which is basically lapply but also simplifies the result
sapply(b, mean)

## [1] 1.0 1.5 2.0 2.5 3.0
```

Going one step further

```
# use a for-loop to generate the list b
max_len <- 4
b <- list()     # initialize
for (i in 1:max_len) {
        b[[i]] <- 1:i
}
# what is this?

# now we can use lapply() to generate the same thing
lapply(1:max_len, function(i) 1:i)

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

How about data frame?

- So far: lapply() breaks a list into elements, applies the function TO EACH ELEMENT, and combines the result into a list
- How does it work with data frames?
  - notice that data frames are just lists
- So lapply() or sapply() on data frames will do things column by column
  - how sapply() will simplify the result will depend on how it looks like
  - (not just brute-force unlist)

Example 1: summary statistics on the mtcars data

```
# recall built-in mtcars data
head(mtcars, n = 3)
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

```
# how do we take averages for the first 4 variables?
res <- rep(NA, 4)
for (i in 1:4) {
        res[i] <- mean(mtcars[, i])
}

res
```

```
## [1]  20.09062   6.18750 230.72188 146.68750
```

mtcars: the lapply() way

```
# notice that we effectively are still dealing with lists
is.list(mtcars)

## [1] TRUE

# so lapply works with mtcars as it does with lists
lapply(mtcars[1:4], mean)

## $mpg
## [1] 20.09062
##
## $cyl
## [1] 6.1875
##
## $disp
## [1] 230.7219
##
## $hp
## [1] 146.6875
```

sapply() will simplify it

```
# average of first 4 variables
sapply(mtcars[, 1:4], mean)
```

```
##       mpg       cyl      disp        hp
##  20.09062   6.18750 230.72188 146.68750
```

```
# standard deviation of first 4 variables
sapply(mtcars[, 1:4], sd)
```

```
##        mpg        cyl       disp         hp
##   6.026948   1.785922 123.938694  68.562868
```

```
# multiple functions work? only when custom-define into one function
sapply(mtcars[, 1:4], function(x) c(mean(x), sd(x)))
```

```
##             mpg      cyl     disp        hp
## [1,] 20.090625 6.187500 230.7219 146.68750
## [2,]  6.026948 1.785922 123.9387  68.56287
```

```
# also note that sapply() organizes the results beautifully
```

Side: why not aggregate?

```
# aggregate() will work in this case
aggregate(cbind(mpg, cyl, disp, hp) ~ 1, mtcars,
        function(x) c(mean = mean(x), sd = sd(x)))

##    mpg.mean    mpg.sd cyl.mean   cyl.sd disp.mean  disp.sd   hp.mean     hp.sd
## 1 20.090625  6.026948 6.187500 1.785922  230.7219 123.9387 146.68750  68.56287

#    Note: ~ 1 means that I will aggregate by constant 1, which means I get one row
#    also note that result is organized in a different way than sapply()

# however, I cannot use aggregate to generate cumulative results
#    it works but this format not easy to work with...
res1 <- aggregate(cbind(mpg, cyl, disp, hp) ~ 1, mtcars, cumsum)
res1[, 1]       # each "column" looks like a "row", weird

##      [,1] [,2] [,3] [,4]  [,5] [,6]  [,7]  [,8]  [,9] [,10] [,11] [,12] [,13]
## [1,]   21   42 64.8 86.2 104.9  123 137.3 161.7 184.5 203.7 221.5 237.9 255.2
##      [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24] [,25]
## [1,] 270.4 280.8 291.2 305.9 338.3 368.7 402.6 424.1 439.6 454.8 468.1 487.3
##      [,26] [,27] [,28] [,29] [,30] [,31] [,32]
## [1,] 514.6 540.6   571 586.8 606.5 621.5 642.9

# but I could do sapply
sapply(mtcars[, 1:4], cumsum)

##         mpg cyl   disp   hp
##  [1,]  21.0   6  160.0  110
##  [2,]  42.0  12  320.0  220
##  [3,]  64.8  16  428.0  313
##  [4,]  86.2  22  686.0  423
##  [5,] 104.9  30 1046.0  598
##  [6,] 123.0  36 1271.0  703
##  [7,] 137.3  44 1631.0  948
##  [8,] 161.7  48 1777.7 1010
##  [9,] 184.5  52 1918.5 1105
## [10,] 203.7  58 2086.1 1228
## [11,] 221.5  64 2253.7 1351
```

More sophisticated summary statistics structures

```
# Say I want mean, sd, quantiles and IQR (inter-quartile range)
lapply(mtcars[, 1:4], FUN = function(x) {
        list(mean = mean(x), sd = sd(x), quantile(x), IQR = IQR(x))
})

## $mpg
## $mpg$mean
## [1] 20.09062
##
## $mpg$sd
## [1] 6.026948
##
## $mpg[[3]]
##     0%    25%    50%    75%   100%
## 10.400 15.425 19.200 22.800 33.900
##
## $mpg$IQR
## [1] 7.375
##
##
## $cyl
## $cyl$mean
## [1] 6.1875
##
## $cyl$sd
## [1] 1.785922
##
## $cyl[[3]]
##   0%  25%  50%  75% 100%
##    4    4    6    8    8
##
## $cyl$IQR
## [1] 4
##
##
## $disp
```

How did sapply do in this case?

```
# the answer is: not so well...
#    result is a list, although organized like a table
sapply(mtcars[, 1:4], FUN = function(x) {
        list(mean = mean(x), sd = sd(x), mean = quantile(x), mean = IQR(x))
})
```

```
##      mpg       cyl       disp      hp
## mean 20.09062  6.1875    230.7219  146.6875
## sd   6.026948  1.785922  123.9387  68.56287
## mean numeric,5 numeric,5 numeric,5 numeric,5
## mean 7.375     4         205.175   83.5
```

```
# one way is to make the quantile() output "parallel" with the rest by transforming the result
#    to a data frame, effectively forcing each column into a vector
#    but keep in mind that we did not force vector within sapply(); we did it after sapply()
sapply(mtcars[, 1:4], FUN = function(x) {
        c(mean = mean(x), sd = sd(x), quantile(x), IQR = IQR(x))        # note: c()
})
```

```
##            mpg      cyl     disp        hp
## mean 20.090625 6.187500 230.7219 146.68750
## sd    6.026948 1.785922 123.9387  68.56287
## 0%   10.400000 4.000000  71.1000  52.00000
## 25%  15.425000 4.000000 120.8250  96.50000
## 50%  19.200000 6.000000 196.3000 123.00000
## 75%  22.800000 8.000000 326.0000 180.00000
## 100% 33.900000 8.000000 472.0000 335.00000
## IQR   7.375000 4.000000 205.1750  83.50000
```

```r
# second approach: notice that lapply gives a list of lists
#    first layer: variable
#    second layer: function values (including vector-valued quantile(x))
# so thoughts: use another lapply()-like function to
#    take apart the second layer through the first layer

lapply(mtcars[1:4], FUN = function(x)
       list(mean = mean(x), sd = sd(x), quantile(x), IQR = IQR(x)))
```

```
## $mpg
## $mpg$mean
## [1] 20.09062
##
## $mpg$sd
## [1] 6.026948
##
## $mpg[[3]]
##     0%    25%    50%    75%   100%
## 10.400 15.425 19.200 22.800 33.900
##
## $mpg$IQR
## [1] 7.375
##
##
## $cyl
## $cyl$mean
## [1] 6.1875
##
## $cyl$sd
## [1] 1.785922
##
## $cyl[[3]]
##   0%  25%  50%  75% 100%
##    4    4    6    8    8
##
## $cyl$IQR
## [1] 4
##
##
```

Side: how to improve this?

```r
# so another way is to use sapply to go through the original lapply results
res <- lapply(mtcars[1:4], FUN = function(x) {
                list(mean = mean(x), sd = sd(x), quantile(x), IQR = IQR(x))
        })

# sapply with unlist
#    effectively unlisting each element of the list res (note: not unlisting the entire res)
#    personally, I like this approach because the logic is clear
sapply(res, FUN = unlist)
```

```
##            mpg      cyl     disp        hp
## mean 20.090625 6.187500 230.7219 146.68750
## sd    6.026948 1.785922 123.9387  68.56287
## 0%   10.400000 4.000000  71.1000  52.00000
## 25%  15.425000 4.000000 120.8250  96.50000
## 50%  19.200000 6.000000 196.3000 123.00000
## 75%  22.800000 8.000000 326.0000 180.00000
## 100% 33.900000 8.000000 472.0000 335.00000
## IQR   7.375000 4.000000 205.1750  83.50000
```

Example 2: Splitting names in the Baltimore payroll data

- In the Baltimore data, names look like:
  - "Ellickson,Paul B"
- What if we want
  - first = "Paul"
  - last = "Ellickson"
  - and ignore the "B"?
- We can
  - split the name by comma: "Ellickson" and "Paul B"
  - and take first of the first part, so "Ellickson"
  - and first of the second part, so "Paul"

Example 2: for loop way

```
# loaded Baltimore data df.avg
split.name <- strsplit(df.avg$name, split = ',')
# e.g. "Ellickson,Paul B." becomes "Ellickson" and "Paul B."
df.avg$lastname <- character(nrow(df.avg))      # e.g. "Ellickson"
df.avg$firstname <- character(nrow(df.avg))  # e.g. "Paul B." becomes "Paul"

# proc.time() logs system time
t0 <- proc.time()

for (i in 1:dim(df.avg)[1]) {
        split.name.last <- unlist(strsplit(split.name[[i]][1], split = ' '))
        split.name.first <- unlist(strsplit(split.name[[i]][2], split = ' '))
        df.avg$lastname[i] <- split.name.last[1]
        df.avg$firstname[i] <- split.name.first[1]
}
t1 <- proc.time()

# time
t1 - t0

##    user  system elapsed
##    1.16    0.83    1.98

# examine results
head(df.avg$lastname, n = 5)

## [1] "Aaron"   "Aaron"   "Aaron"   "Abaineh" "Abbey"
```

Example 2: sapply() way

```
split.name <- strsplit(df.avg$name, split = ',')
# e.g. "Ellickson,Paul B." becoems "Ellickson" and "Paul B."

split.and.extract <- function(x, n) unlist(strsplit(x[n], split = ' '))[1]      # n takes 1 and 2

t0 <- proc.time()
df.avg$lastname <- sapply(split.name, function(x) split.and.extract(x, 1))
df.avg$firstname <- sapply(split.name, function(x) split.and.extract(x, 2))
t1 <- proc.time()

# much faster...
t1 - t0

##    user  system elapsed
##    0.24    0.00    0.24

# same results
head(df.avg$lastname, n = 5)

## [1] "Aaron"   "Aaron"   "Aaron"   "Abaineh" "Abbey"
```

Example 3: Variations in the sample mean

- In statistics (as you will learn in fall), it is important to also consider variations in your statistical estimates
  - such as the sample mean
  - i.e., the mean of one sample can (and often will) be different across different random samples
- To show this, we can randomly draw sub-samples from mtcars and examine the average mpg[1]
- Some of you might perform these exercises quite often in your job[2]

[1] Formally, this is called "sub-sampling" and is similar to "bootstrapping"

[2] I guess even more so if you work in finance

Example 3: mean of one sample can be different than another

```
r <- nrow(mtcars)

# sample without replacement (default is replace = FALSE)
rand.number <- sample(x = 1:r, size = 10)
mpg.1 <- mtcars[rand.number, "mpg"]
mean(mpg.1)
```

```
## [1] 23.99
```

```
# do this again
#    sample again will give you a different sample
#    unless you set seed, which is much beyond our scope
rand.number <- sample(x = 1:r, size = 10, replace = FALSE)
mpg.2 <- mtcars[rand.number, "mpg"]
mean(mpg.2)
```

```
## [1] 22.09
```

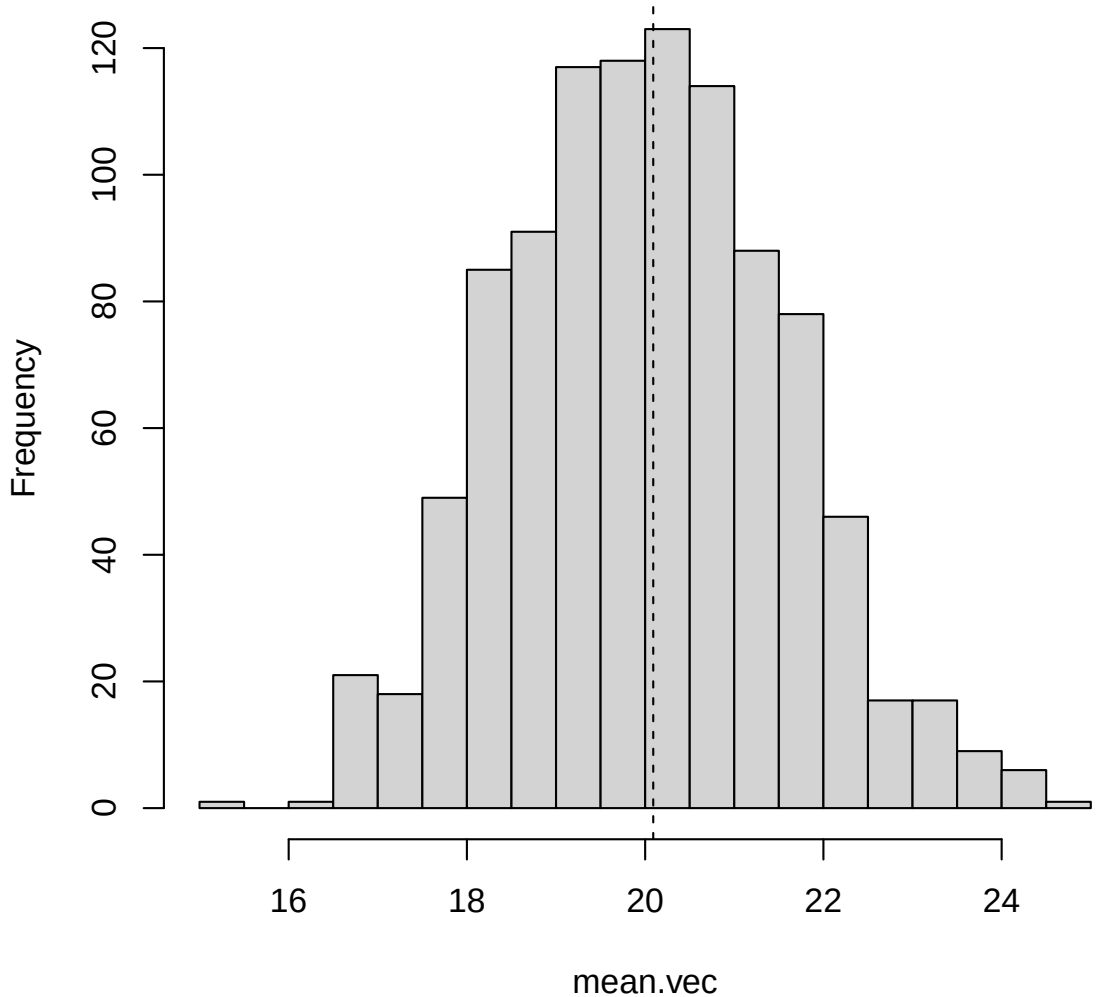
Example 3: sample means, 100 sub-samples of size 10

```
set.seed(121269)        # set seed
m <- 1000       # repetitions
n <- 10         # sample size
r <- nrow(mtcars)       # full sample size

# record time
t0 <- proc.time()

# for-loop way: initialize
mean.vec <- rep(NA, n)

# loop
for (i in 1:m) {
        rand.number <- sample(1:r, n)
        mean.vec[i] <- mean(mtcars[rand.number, "mpg"])
}
t1 <- proc.time()

# examine raw results
head(mean.vec)

## [1] 19.91 21.87 20.53 21.21 18.39 18.16

# average of those sample average, compared with average of full sample
c(mean(mean.vec), mean(mtcars$mpg))

## [1] 20.02279 20.09062

# time difference
t1 - t0

##    user  system elapsed
##    0.02    0.00    0.02
```

Example 3: sample means, 100 sub-samples of size 10

```r
# interlude
#    can we write down the full procedure for each sub-sample, satisfying
#    1) each iteration can be done independently
#    2) procedure can be written in a function?

# 1) is easy in this case because each iteration does not even depend on i
# 2) being independent on i, also easy to be written as a function
#    function(i) mean(mtcars[sample(1:r, n), "mpg"])
#    note: i should be here because we need to tell R the nr. of iterations

set.seed(121269)        # set seed

# record time
t0 <- proc.time()

# sapply way?
mean.vec <- sapply(1:m, function(i) mean(mtcars[sample(1:r, n), "mpg"]))
t1 <- proc.time()

# similar results (now that we have fixed the seed, draws are the same)
head(mean.vec)

## [1] 19.91 21.87 20.53 21.21 18.39 18.16

# time difference (this is a case where time is so short, so either can be quicker)
t1 - t0

##    user  system elapsed
##    0.03    0.00    0.03
```

```r
hist(mean.vec, nclass = 25, main = "sub-sample means (vertical: full-sample mea
abline(v = mean(mtcars$mpg), lty = 2)
```



Example 3: are sub-sample mean variations equal to the theoretical standard error?

```
# THEORETICAL standard error of the mean
#    i.e. measures the precision of the mean estimate
sd(mtcars$mpg) / sqrt(nrow(mtcars))
```

```
## [1] 1.065424
```

```
# EMPIRICAL standard error from simulations
#    i.e. standard deviations of the subsample means
sd(mean.vec)
```

```
## [1] 1.531077
```

```
# they are not the same: smaller sub-samples generate larger variances
#    if you wonder how to properly examine standard errors
#    in simulation, Google "bootstrap"
```

Example 4: forward-simulation of random walk processes

- Let's say the log price of bitcoin follows a random walk process

$$p_t = p_{t-1} + u_t$$

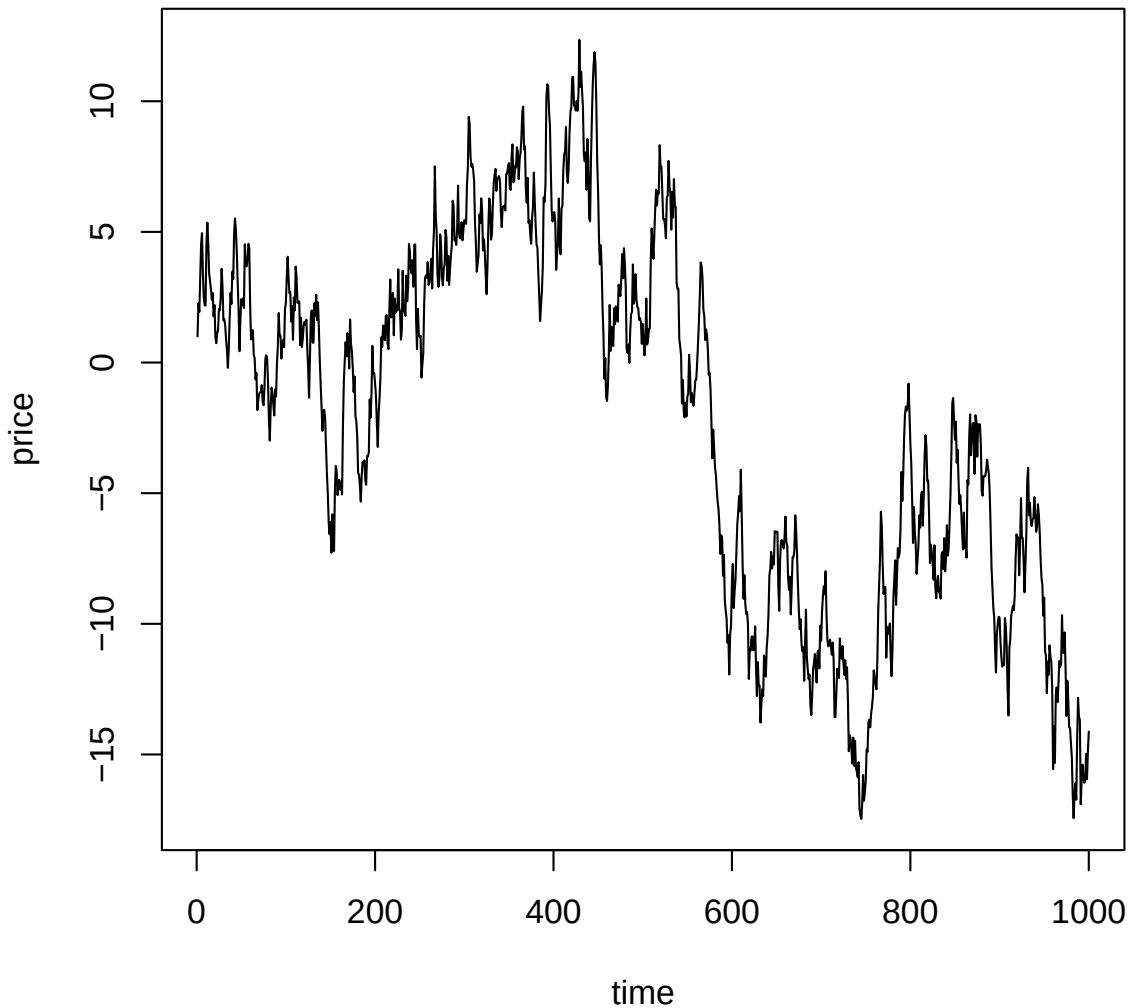
  where u_t follows the standard normal distribution

- The price process is non-stationary; what should we do if we want to characterize its distribution and changes of its distribution over time?
  - we can simulate many such processes and examine the distribution
  - will run one such example to illustrate the usage of loop/apply

Example 4: one random walk process

```r
# set seed (necessary to reproduce any simulation exercises)
set.seed(0)

# simulation of one random walk
Tmax <- 1000    # 1000 days
p_vec <- numeric(Tmax)  # pre-assign results
p_vec[1] <- 1

for (t in 2:Tmax) {
    p_vec[t] <- p_vec[t-1] + rnorm(1)   # rnorm(1): standard normal random vari
}

# plot
plot(1:Tmax, p_vec, type = 'l', xlab = "time", ylab = "price")
```

Example 4: one random walk process



Example 4: How do we get 1,000 such processes?

```
# record time
t0 <- proc.time()

# assign results into a matrix
NSim <- 1000
sim_res <- array(NA, dim = c(NSim, Tmax))

# simulation using a loop
for (s in 1:NSim) {

        # initial value = 1
        sim_res[s, 1] <- 1

        # subsequent values
        for (t in 2:Tmax) {
            sim_res[s, t] <- sim_res[s, t-1] + rnorm(1)
        }

}

# terminal time
t1 <- proc.time()
t1 - t0

##    user  system elapsed
##    1.66    0.04    1.71
```

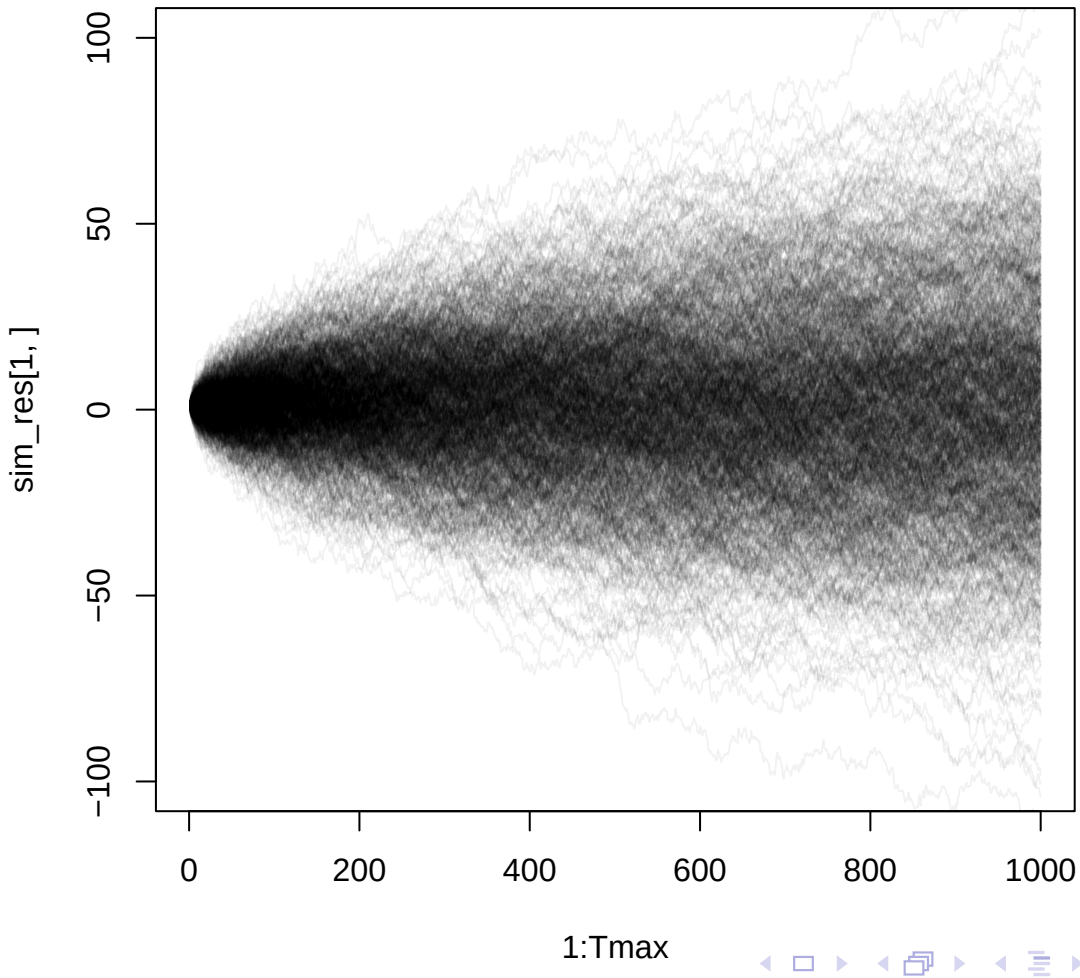
Example 4: sapply() approach?

```
# observe that the forward-sim procedure can be summarized into a function
sim_res <- array(NA, dim = c(NSim, Tmax))

# one_simulation is a function characterizing simulation s
one_simulation <- function(s) { # argument s is useless
    # pre-assign
    p_vec <- numeric(Tmax)
    p_vec[1] <- 1
    # forward-simulation
    for (t in 2:Tmax) {
        p_vec[t] <- p_vec[t-1] + rnorm(1)
    }
    return(p_vec)
}
```

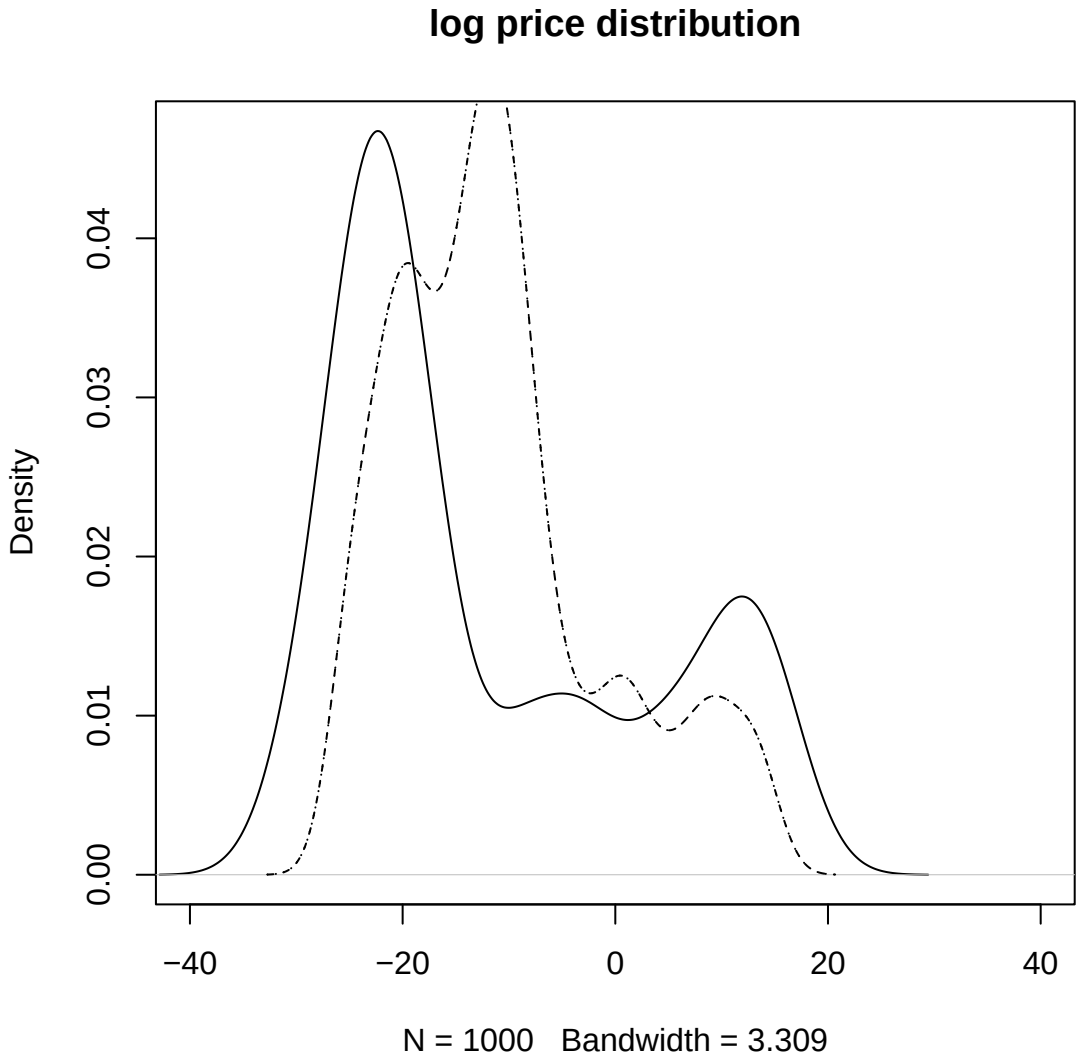
Example 4: sapply() approach?

```
# record time
t0 <- proc.time()

# then the whole process will be
N_sim <- 1000
sim_res <- t(sapply(1:NSim, one_simulation))
#    transpose because sapply will gather each simulation path in columns

# terminal time
#    interestingly, did not save that much time
t1 <- proc.time()
t1 - t0

##    user  system elapsed
##    1.61    0.00    1.61
```

```r
# plot
plot(1:Tmax, sim_res[1, ], type = 'l', col = rgb(0, 0, 0, 0.05),
        ylim = c(-100, 100))    # rgb(r,g,b,alpha)  where alpha is transparency
for (s in 2:NSim) {
    points(1:Tmax, sim_res[s, ], type = 'l', col = rgb(0, 0, 0, 0.05))
}
```



Bitcoin prices at t = 10, 100, 500



Code in the previous slide

```
# estimate density
den1 <- density(sim_res[10, ])
den2 <- density(sim_res[100, ])
den3 <- density(sim_res[500, ])

# plot the three together
plot(den1, type = 'l', lty = 1, xlim = c(-40, 40), main = "log price distributi
points(den2, type = 'l', lty = 2)
points(den2, type = 'l', lty = 3)
```

apply()

apply()

- apply() takes AN ARRAY and applies a function to given index
  - in contrast with lapply/sapply, which work with lists
- Arguments:
  - the array (or matrix)
  - which index is "looped over"
  - which function to use
- Can be used on data frames because they "look like" matrices
  - as we will see, some problem might arise because data frames are not matrices

We did max.col() in Samsung v. HTC

```
# Samsung wants to maximize profit for each column strategy by HTC
#    want to find WHERE the maximum is for each row

# A is a matrix of some numbers
A <- array(c(1, 2, 4, 3, 7, 6, 5, 2, 1, 8, 5, 2), dim = c(3, 4))
A
```

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

```
# could loop over rows of A
res.1 <- numeric(nrow(A))
for (i in 1:nrow(A)) {
        res.1[i] <- which.max(A[i, ])
}
res.1
```

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

```
# code is too cumbersome... could do max.col()
max.col(A)
```

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

What if we want a max.row()?

```
# I want to find WHERE the maximum is for each COLUMN
#    sort of like max.col() but for the rows

# same array
A
```

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

```
# could loop over columns of A
res.2 <- numeric(ncol(A))
for (j in 1:ncol(A)) {
        res.2[j] <- which.max(A[, j])
}
res.2
```

```
## [1] 3 2 1 1
```

```
# also too cumbersome...
max.row(A)
```

```
        ## Error in max.row(A): could not find function "max.row"
```

```
#    but now there's no max.row()
```

Use apply() is simpler

```
# better is
res.3 <- apply(A, MARGIN = 2, FUN = which.max)

res.3
```

```
## [1] 3 2 1 1
```

```
# and the same results
all(res.2 == res.3)
```

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

Different dimensions and notation

```
# take average *given* columns (i.e. along rows)
apply(A, MARGIN = 2, FUN = which.max)
```

```
## [1] 3 2 1 1
```

```
# take standard deviation *given* rows (i.e. along column)
#    also we suppressed argument names
#    now we get back to the max.col() result
apply(A, 1, which.max)
```

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

```
# can fix both dimensions
#    in this case trivial, but good with high-dim arrays
apply(A, MARGIN = c(1, 2), FUN = length)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    1    1    1    1
## [3,]    1    1    1    1
```

```
#    but can't leave MARGIN empty
```

Using apply() on data frames

```r
# data frames are tabular, so apply works in the same logic
B <- data.frame(id = c(1, 2, 3, 4, 5, 6),
        salary = c(2000, 3000, 2000, NA, 2500, NA),
        tenure = c(0, 2, 0, 5, NA, 1)
)
B
```

```
##   id salary tenure
## 1  1   2000      0
## 2  2   3000      2
## 3  3   2000      0
## 4  4     NA      5
## 5  5   2500     NA
## 6  6     NA      1
```

```r
# can do something similar as before
apply(B, 2, function(x) c(mean(x, na.rm = T), sd(x, na.rm = T)))
```

```
##            id    salary   tenure
## [1,] 3.500000 2375.0000 1.600000
## [2,] 1.870829  478.7136 2.073644
```

```r
# equivalent: apply(B, 2, function(x, ...) c(mean(x, ...), sd(x, ...)), na.rm = T)

# can also apply functions to rows
#    e.g. we count how many columns are non-missing
apply(B, 1, function(x) sum(!is.na(x)))
```

```
## [1] 3 3 3 2 2 2
```

apply() internally still works with arrays

```
# how about data frames with different data types?
C <- data.frame(names = c("mike", "pete", "liz", "chloe", "dan", "alice"),
        salary = c(2000, 3000, 2000, NA, 2500, NA),
        tenure = c(0, 2, 0, 5, NA, 1)
)

# does a very bad job because C is internally first coerced to character
#    sd still works due to the way it handles characters
apply(C, 2, function(x) c(mean(x, na.rm = T), sd(x, na.rm = T)))

##      names   salary   tenure
## [1,]    NA       NA       NA
## [2,]    NA 478.7136 2.073644

# that's why you want to use lapply(X, FUN) rather than apply(X, 2, FUN)
```

What just happened?

```
# notice that mean of character will give all NAs
mean(c("2", "3", NA, "5"), na.rm = T)

## [1] NA

# while sd of all characters will coerce character into numeric
sd(c("2", "3", NA, "5"), na.rm = T)

## [1] 1.527525

# different people wrote those code, I guess...

# now, if we see what the following is, we will be able to understand the code:
as.matrix(C)

##      names   salary tenure
## [1,] "mike"  "2000" " 0"
## [2,] "pete"  "3000" " 2"
## [3,] "liz"   "2000" " 0"
## [4,] "chloe" NA     " 5"
## [5,] "dan"   "2500" NA
## [6,] "alice" NA     " 1"
```

Let's reproduce the mtcars example

```r
# let's re-produce the mtcars example
summary.mtcars <- as.data.frame(
        apply(mtcars[1:4], MARGIN = 2, FUN = function(x) {
                list(c(mean = mean(x), sd = sd(x), quantile(x), IQR = IQR(x)))
        })
)
# because apply treats data frame as a matrix, the column names are messy
names(summary.mtcars) <- names(mtcars[1:4])
summary.mtcars
```

```
##             mpg      cyl     disp        hp
## mean 20.090625 6.187500 230.7219 146.68750
## sd    6.026948 1.785922 123.9387  68.56287
## 0%   10.400000 4.000000  71.1000  52.00000
## 25%  15.425000 4.000000 120.8250  96.50000
## 50%  19.200000 6.000000 196.3000 123.00000
## 75%  22.800000 8.000000 326.0000 180.00000
## 100% 33.900000 8.000000 472.0000 335.00000
## IQR   7.375000 4.000000 205.1750  83.50000
```

tapply()

tapply()

- tapply() is essentially a different form of aggregate(): i.e. it takes summary statistics by groups
- Arguments:
  - a data frame that can be broken into groups
  - which variable to work with for each group
  - which function to use
- Will contrast tapply(), aggregate(), cast() and DT[i, j, by]

Simulate a medical test sample

```r
# let's randomly generate a medical test sample
#    i.e. we have treatment and control groups
#    we need to examine: 1) are treatment/control balanced,
#    2) is there an effect of the drug?

# age and treatment
medical <- data.frame(patient = 1:100,
                      age = round(rnorm(100, mean = 60, sd = 12)),
                      treatment = rep(c(TRUE, FALSE), each = 50))

# outcome
medical$outcome <- rnorm(100, mean = 1 + 0.1 * medical$treatment, sd = 5)
```

Goal: average age and outcome given treatment status?

```
# average age given treatment (balance test)
tapply(X = medical$age,
        INDEX = list(treatment = medical$treatment),
        FUN = mean)

## treatment
## FALSE  TRUE
## 57.82 57.88

# average outcome given treatment (treatment effect)?
tapply(X = medical$outcome,
        INDEX = list(treatment = medical$treatment),
        FUN = mean)

## treatment
##    FALSE     TRUE
## 0.904623 1.804361

# you can see that tapply() is very much like an aggregate()
```

aggregate version

```
# average age given treatment (balance test)
aggregate(x = medical$age,
          by = list(treatment = medical$treatment),
          FUN = mean)

##   treatment     x
## 1     FALSE 57.82
## 2      TRUE 57.88

# average BOTH age and outcome given treatment (treatment effect)?
aggregate(formula = cbind(outcome, age) ~ treatment,
          data = medical,
          FUN = mean)

## Error in aggregate.formula(formula = cbind(outcome, age) ~ treatment,
##     :  argument 'x' is  missing - it has been renamed from 'formula'
```

cast version

```
library('reshape')
cast(data = medical,
        formula = . ~ treatment,
        fun.aggregate = mean,
        value = "age")

##   value FALSE  TRUE
## 1 (all) 57.82 57.88

# or do this in the format of aggregate
cast(data = medical,
        formula = treatment ~ .,
        fun.aggregate = mean,
        value = "outcome")

##   treatment    (all)
## 1     FALSE 0.904623
## 2      TRUE 1.804361
```

data.table version

```
library('data.table')
medical_dt <- data.table(medical)

# still my personal favorite
medical_dt[,
        .(age = mean(age), outcome = mean(outcome)),
        treatment]

##    treatment   age  outcome
## 1:      TRUE 57.88 1.804361
## 2:     FALSE 57.82 0.904623
```

Conclusions

- *apply() functions can vectorize your data manipulation efforts
  - shorter time and cleaner code
  - when apply() and when for (i in range)?
- Sometimes your calculation is "sequential", i.e. one step depends on the other
  - in these occasions, use for/while/repeat
- Sometimes your calculation is parallel, i.e. steps are independent to each other, but it is difficult to vectorize your code
  - in these occasions, consider using *apply()
- tapply() is another aggregate-like function
  - comparison between aggregate, tapply, cast and data.table