Lecture 3 R Basics: "apply" functions

Zhe Zhao

Stevens Institute of Technology

zzhao6@stevens.edu

September 14, 2015

Agenda

1. Data Types (cont'd)
   - Data Frame
   - Removing NA
2. "Apply" Functions
   - Vectorized Operation
   - lapply
   - sapply
3. Generating Random Numbers

Import CSV

We use *read.table* or *read.csv* to read tabular data. These two functions are almost identical except their default separators.

Example

```
> read.csv("goog.csv")
> GOOG <- read.csv("goog.csv")
# default value of header is T
> GOOG <- read.csv(file = "goog.csv", header = T)

> head(GOOG) # first several rows of GOOG

# GOOG is a list
> mode(GOOG)
> names(GOOG)
> GOOG$Open
> GOOG$Adj.Close
```

Data Frame

Data frames are used to store tabular data.

- A special type of list.
- Each element of the list (which is a vector) can be thought of as a column.
- Unlike matrices, data frames can store different classes of objects in each column (just like lists); matrices must have every element be the same class.
- Data frames are usually created by calling *read.table()* or *read.csv()*.
- Can be converted to a matrix by calling *data.matrix()*

Data Frame

We can create data frames using the build-in function **data.frame()**.

Example (Creating data frame)

```
> # two vectors
> kids <- c("Joe", "Jill")
> ages <- c(11, 12)
> typeof(kids)
[1] "character"
> typeof(ages)
[1] "double"
>
> # create a dataframe
> df <- data.frame(kids, ages)
> df
  kids ages
1  Joe   11
2 Jill   12
```

Data Frame

Data frames are essentially lists.

Example (names in data frame)

```
> # create a dataframe
> df <- data.frame(kids, ages)
> # create a list
> df2 <- list(kids, ages)
>
> # both are list
> typeof(df)
[1] "list"
> typeof(df2)
[1] "list"
```

Data Frame

Argument *stringsAsFactors*.

Factors in R represent categorical variables, that is for statistical analysis. In finance, this data structure are not frequently used.

Example

```
> # strings as factors
> df <- data.frame(kids, ages)
> df$kids                                # factors
[1] Joe  Jill
Levels: Jill Joe
>
> df <- data.frame(kids, ages, stringsAsFactors = F)
> df$kids                                # strings
[1] "Joe"  "Jill"
```

Data Frame

Create a data frame by reading a csv file.

Example

```
> aapl <- read.csv("AAPL.csv", stringsAsFactors = F)
> head(aapl)
        Date   Open   High    Low  Close   Volume Adj.Close
1 2015-09-11 111.79 114.21 111.76 114.21 49441800    114.21
2 2015-09-10 110.27 113.28 109.90 112.57 62675200    112.57
3 2015-09-09 113.76 114.02 109.77 110.15 84344400    110.15
4 2015-09-08 111.75 112.56 110.32 112.31 54114200    112.31
5 2015-09-04 108.97 110.45 108.51 109.27 49963900    109.27
6 2015-09-03 112.49 112.78 110.04 110.37 52906400    110.37
> typeof(aapl)
[1] "list"
```

Missing Values

Missing values are denoted by **NA** or **NaN** for undefined mathematical operations.

- *is.na()* is used to test objects if they are NA.
- *is.nan()* is used to test for NaN.
- **NA** values have a class also, so there are integer NA, character NA, etc.
- A **NaN** value is also **NA** but the converse is not true.

Removing NA

A common task is to remove missing values from your data.

Example

```
> x <- c(1, 2, 3, NA, NA, 6, NA, 8)
> xna <- is.na(x)     # vectorized operation
> xna
[1] FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE
> x[!xna]
[1] 1 2 3 6 8
```

Vectorized Operation

Suppose we have a function *f()* that we wisht to apply to all elements of a vector x. Instead of looping all elements in x and calling *f()* in each iteration, we can simply call *f()* on x itself.

This can really **simplify our code** and, moreover, give us **a dramatic performance increase** of hunderdsfold or more.

Vectorized Operation

Example

```
> x <- 1:4
> y <- 6:9
> # want to do x + y

> # using for loop
> result <- c()
> for (i in x)
+ {
+     result[i] = x[i] + y[i]
+ }

> result
[1]  7  9 11 13
```

Vectorized Operation

We can directly apply "+" to x and y.

Example

```
> result <- x + y
> result
[1]  7  9 11 13
> rbind(x, y, result)
       [,1] [,2] [,3] [,4]
x         1    2    3    4
y         6    7    8    9
result    7    9   11   13
```

Vectorized Operation

Another example

Example

```
> x <- 1:4
> x > 2
[1] FALSE FALSE  TRUE  TRUE
```

Vectorized Operation

Apply vectorized operation to functions defined by user

Example

```
> # user defined functions
> mySquare <- function(x)
+ {
+     return (x^2)
+ }
```

Vectorized Operation

Example

```
> x <- 1:4
> y <- 6:9
> mySquare(x)
[1]  1  4  9 16
> mySquare(y)
[1] 36 49 64 81
> table1 <- cbind(x, mySquare(x), y, mySquare(y))
> table1
     x    y
[1,] 1  1 6 36
[2,] 2  4 7 49
[3,] 3  9 8 64
[4,] 4 16 9 81
```

Vectorized Operation

Revisiting the coin flipping example.

Example

```
No.heads <- 0
result.Vec <- NULL
for (flips in 1:1000)
{
    # x = c(1, 0), 1 means head, 0 means tail
    tmp <- sample(x=c(1, 0), size=1,
                  replace=T, prob=c(0.5, 0.5))
    No.heads <- No.heads + tmp
    result.Vec <- c(result.Vec,No.heads/flips)
}
plot(1:1000, result.Vec, type="l")     # produce a figure
```

Vectorized Operation

Rewrite the same example in vectors

Example

```
> ?cumsum # please check by yourself
>
> sims <- sample(c(1, 0), size = 1000,
                 replace = T, prob = c(0.5, 0.5))
> cum_sims <- cumsum(sims)             # cumulative Sums
> result.Vec <- cum_sims / (1:1000)    # vectorized operation
>
> plot.ts(result.Vec)
```

"Apply" Functions

lapply

- *lapply* takes two arguments: a list x, a function or a name of function.
- If x is not a list, it will be coerced to a list using as.list().
- *lapply* always returns a list, regardless of the class of the input.

Example

```
> x <- 1:4
> lapply(x, mySquare)
[[1]]
[1] 1

[[2]]
[1] 4

[[3]]
[1] 9

[[4]]
[1] 16
```

sapply

sapply will try to simplify the result of *lapply* if possible.

- If the result is a list where every element is length 1, then a vector is returned.
- If the result is a list where every element is a vector of the same length, a matrix is returned.
- Otherwise a list is returned.

sapply

Example

```
> x <- 1:4
> sapply(x, mySquare)
[1]  1  4  9 16
```

Apply Functions

Another example

Example

```
> x <- list(rnorm(10000), runif(10000, min = 0, max = 1))
> lapply(x, mean)
[[1]]
[1] -0.008742473

[[2]]
[1] 0.5009495

> sapply(x, mean)
[1] -0.008742473  0.500949453
```

Apply Functions

More example

Example

```
> x <- 1:4
> lapply(x, runif)
[[1]]
[1] 0.8414947

[[2]]
[1] 0.6263586 0.8831707

[[3]]
[1] 0.6218396 0.1194392 0.3133257

[[4]]
[1] 0.8740655 0.4518131 0.1776370 0.5959832
```

About Performance

Use "apply" functions will dramaticaly increase the performance of your code

Example

```
> x <- 1:10^5
> system.time(
+     for (i in x)
+     {
+         mySquare(x)
+     }
+ )
   user  system elapsed
 44.170   1.718  45.880
```

About Performance

Here comes the magic.

Example

```
> system.time(
+     sapply(x, mySquare)
+ )
   user  system elapsed
  0.175   0.002   0.176
```

Other Apply Functions

- lapply Apply a Function over a List or Vector
- sapply Simplify the result of lapply

- apply Apply Functions Over Array Margins
- eapply Apply a Function Over Values in an Environment
- mapply Apply a Function to Multiple List or Vector Arguments
- rapply Recursively Apply a Function to a List
- tapply Apply a Function Over a Ragged Array

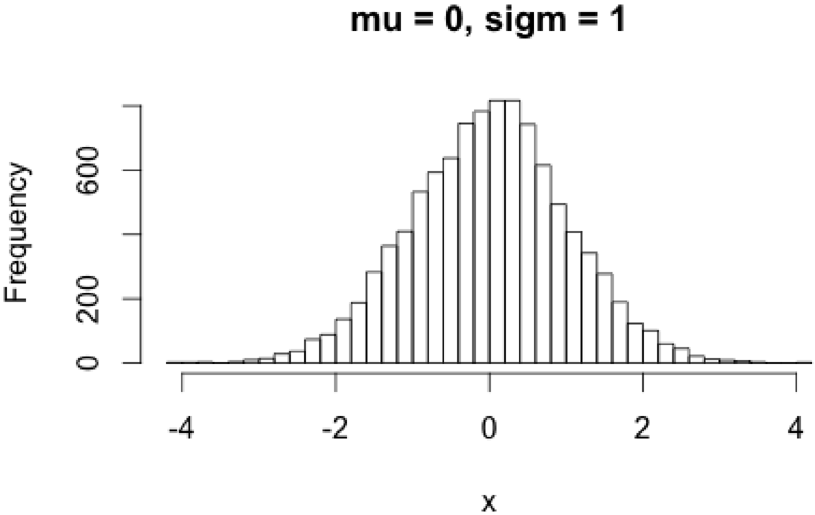
Normal Distribution

Now let's move on how to generate random variables.

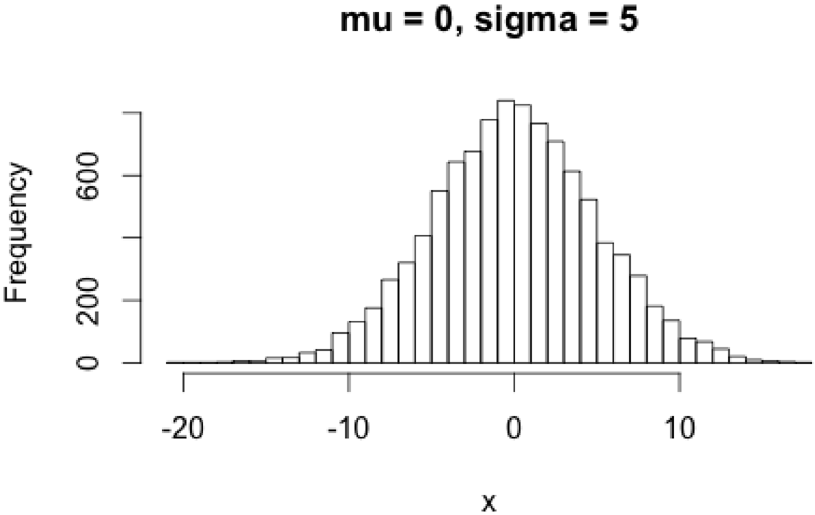
Example

```
> # rnorm(n = , mean = , sd = )

> x <- rnorm(n = 10000, mean = 0, sd = 1)
> hist(x)
> hist(x, nclass = 40)

> # another sigma
> x <- rnorm(n = 10000, mean = 0, sd = 5)
```

Normal Distribution



Normal Distribution



Normal Distribution

set.seed()

Anywho the random numbers R gives you aren't really random. They're pseudo-random. Basically there's a function that outputs numbers that look random. To do this it needs some inputs. The first input it gets will be the 'seed'.

Example

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
> rnorm(5)
[1] -0.8204684  0.4874291  0.7383247  0.5757814 -0.3053884
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

Normal Distribution

- rnorm: generate random Normal variates with a given mean and standard deviation.
- dnorm: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)
- pnorm: evaluate the cumulative distribution function for a Normal distribution
- qnorm: gives the quantile function

Normal Distribution

Example

```
> # Density function
> dnorm(x = 0)     # mean = 0, sd = 1
[1] 0.3989423
> dnorm(x = 1)     # check table if you want
[1] 0.2419707

> # cumulative distribution function
> pnorm(q = 0)
[1] 0.5
> pnorm(q = 5)
[1] 0.9999997
```

Other Distributions

- rt() – t distribution
- rpois() – poisson distribution
- runif() – uniform distribution
- rexp() – exponential distribution

Other Distributions

Example

```
> x <- rpois(1000, lambda = 2)
> hist(x, nclass = 40)
>
> x <- rexp(1000)
> hist(x, nclass = 40)
>
> x <- rt(1000, df = 10)
> hist(x, nclass = 40)
```