Lecture 2: R Basics(2)

Cheng Lu

Overview

- Import CSV
- Data Frame
- Missing values
- Vectorized Operation
- User defined functions
- "Apply" Functions
- Homework

Import CSV

Before importing files, the files have to be in the current working directory:

- Use **getwd()** get current working directory.
- Use **setwd()** set the current working directory to the file location.

Example

```
> getwd()
[1] "C:/Users/demonew/Documents"
> # use "/" not "\" when you set working directory
> setwd("C:/Users/demonew/Documents/Stevens/Graduate Courses/FE515/22s")
> getwd()
[1] "C:/Users/demonew/Documents/Stevens/Graduate Courses/FE515/22s"
```

If you use Rmarkdown, set working directory for each chunk.

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, first row of file is treated as header
GOOG <- read.csv(file = "goog.csv", header = T)
head(GOOG) # first several rows of GOOG
mode(GOOG) # GOOG is a list
class(GOOG) # GOOG is also a data frame
names(GOOG)
GOOG$Open
GOOG$Adj.Close
```

Data Frame

Data frames are used to store tabular data.

- A special type of list.
- Each member of a list (which is a vector) can be thought of as a column with same size.
- Unlike matrices, data frames can store different type of objects in each column (just like lists); matrices must have every element be the same type.
- 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

```
> # 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

```
> # 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

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

Data Frame

Create a data frame by reading a .csv file.

Example

```
> aapl <- read.csv("AAPL.csv")
> 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.

Example

```
> c(-1,0,1)/0
[1] -Inf  NaN  Inf
> is.na(c(1,NA,NaN))
[1] FALSE  TRUE  TRUE
> is.nan(c(1,NA,NaN))
[1] FALSE FALSE  TRUE
```

Missing Values

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
> y <- na.omit(x)
> as.numeric(y)
[1] 1 2 3 6 8
```

Vectorized Operation

Suppose we have a function f() that we wish 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, sometime 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 1:4) {
+ 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
```

User defined functions

Functions are created using **function()** directly.

Example

```
foo <- function(# parameters){
# body
}
foo <- function(# parameters){
# body
return ()
}
```

Here **foo** is a function and we can call the function with **foo(#parameters)**.
This is different from C++ or other programming language, since we define a function using a function: **function()**, and the new function are stored as R object in your environment like anything else. Also the type of the parameters and output are not specified.

User defined functions

Example

```
> print("Hello World!")
[1] "Hello World!"
> PrintHW <- function(){
+   print("Hello World!")
+ }
> PrintHW()
[1] "Hello World!"
> PrintSomething <- function(sth){ #function with parameter
+ print(sth)
+ }
> PrintSomething("HW!")
[1] "HW!"
> PrintSomething(1)
[1] 1
```

User defined functions

Functions can use global variables, but changing variables inside function doesn't affect global environment.

Example

```
> sth <- "Hello World!"
> printsth <- function(){
+   print(sth)
+   sth <- "Hello R World!" # how about "<<-" rather than "<-" ?
+   print(sth)
+ }
> printsth()
[1] "Hello World!"
[1] "Hello R World!"
> sth
[1] "Hello World!"
```

User defined functions

Example

```
> # calculating 1 + 2
> a <- 1 # set value for variable a
> b <- 2 # set value for variable b
> c <- a + b # statements
> c
[1] 3
> # names of the variables become function parameters
> add <- function(a, b){ # add() becomes an object in environment
+ # the statements become function body
+ c <- a + b
+ return (c) # No output if no return, but how about c not return(c)
+ }
> add(1, 2) # call the function with the values of the variables
[1] 3
> result <- add(1, 2) # assign the output to a variable "result"
> result
[1] 3
```

User defined functions

How to write a function?

- Set values for variables and the write down the corresponding statements
- Create function using **function()**:
  - Names of the variables become function parameters
  - The statements become function body
  - Don't forget **return()** if you need outputs
- Call the function with the values of the variables

The function will end after you **return()** something.

User defined functions

There are also a way to update vector when the length of the vector is unknown, suppose we want to construct sequence from 3 to 8.

Example

```
> a <- 3 # set value for variable a
> b <- 8 # set value for variable b
> # statements
> seque <- NULL # equivalent to seque <- c()
> i <- a
> while(i <= b){
+     seque <- c(seque, i) # update sequence
+     i <- i + 1
+ }
> seque
[1] 3 4 5 6 7 8
```

User defined functions

Suppose we want to *write a function* to construct sequence from a to b.

Example

```
> # names of the variables become function parameters
> constrSeq <- function(a, b){
+   # the statements become function body
+   seque <- NULL # equivalent to seque <- c()
+   i <- a
+   while(i <= b){
+     seque <- c(seque, i) # update sequence
+     i <- i + 1
+   }
+   return(seque) # don't forget return
+ }
> constrSeq(3, 8) # call the function with the values of the variables
[1] 3 4 5 6 7 8
```

"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.

"Apply" Functions

Example

```
> x <- 1:4
> lapply(x, mySquare)
[[1]]
[1] 1
[[2]]
[1] 4
[[3]]
[1] 9
[[4]]
[1] 16
```

"Apply Functions"

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.

"Apply" Functions

Example

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

"Apply" Functions

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

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
```

"Apply" Functions

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
- replicate: Repeat evaluation of an expression
- rapply: Recursively Apply a Function to a List
- tapply: Apply a Function Over a Ragged Array

"Apply" Functions

Sometimes "apply" functions will dramatically increase the performance of your code

Example

```
> x <- NULL
> system.time(
+ for (i in 1:10000) {# bind 10000 columns of random variables
+   x <- cbind(x, rnorm(252)) # size of x changes
+ })
   user  system elapsed
  20.99    6.04   27.12
> system.time(x <- replicate(10000, rnorm(252)))
   user  system elapsed
   0.22    0.00    0.21
```

The 'user time' is the CPU time charged for the execution of user instructions of the calling process.
The 'system time' is the CPU time charged for execution by the system on behalf of the calling process.
The third entry is the 'real' elapsed time since the process was started. (see **?proc.time**)

"Apply" Functions

Sometimes using "apply" functions doesn't increase the performance

Example

```
> x <- 1:10^5
> system.time({
+   for(i in 1:length(x)){
+     mySquare(x[i])
+   }
+ })
   user  system elapsed
   0.05    0.00    0.05
> system.time(
+   sapply(x, mySquare)
+ )
   user  system elapsed
   0.08    0.00    0.09
```

"Apply" Functions

Example

```
> x <- 1:4
> y <- x
> mapply(add, x, y)
[1] 2 4 6 8
> outer(x, y, add) # outer is not an apply function
     [,1] [,2] [,3] [,4]
[1,]    2    3    4    5
[2,]    3    4    5    6
[3,]    4    5    6    7
[4,]    5    6    7    8
```

"Apply" Functions

Example

```
> A <- cbind(x, y)
> A
     x y
[1,] 1 1
[2,] 2 2
[3,] 3 3
[4,] 4 4
> apply(A, 1, mean) # calculate means of A for each row, 1 for rows
[1] 1 2 3 4
> apply(A, 2, mean) # calculate means of A for each column, 2 for columns
  x   y
2.5 2.5
```