Lecture 2 R Basics Functions

Zhe Zhao

Stevens Institute of Technology

zzhao6@stevens.edu

September 8, 2015

Overview

1. Data Types (cont'd)
   - Data Frame
   - Removing NA
2. Functions
3. R Coding Style

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)

```
> kids <- c("Joe", "Jill")
> ages <- c(11, 12)
# the two vector are in different types
> typeof(kids)
[1] "character"
> typeof(ages)
[1] "double"
> d <- data.frame(kids, ages)
> d
  kids ages
1  Joe   11
2 Jill   12
```

Data Frame

R objects can also have names, recall that we can use name to access list members. So does data frames.

Example (names in data frame)

```
> names(d)
[1] "kids" "ages"
> d$kids
[1] Joe  Jill
Levels: Jill Joe
> d$ages
[1] 11 12
> # another example:
> # creat a data frame with read.csv()
> dow <- read.csv("DOW.csv")  # symbols in Dow Jones
> names(dow)
[1] "Ticker"  "Company"
```

Data Frame

We can access data frame elements like a list.

Example

```
> d
  kids ages
1  Joe   11
2 Jill   12
> d[[1]]        # a data frame is actually a list
[1] Joe  Jill
Levels: Jill Joe
```

Or you can use matrix-like style.

Example

```
> d[1, 1]
[1] Joe
Levels: Jill Joe
```

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
```

Appetizer example: coin flips

Before we go to function let's do a simulation using what we have learned until now.

Imagin you are flipping a fair coin, say 1000 times. Simulate the probability of heads after each flip, then make a 2-D graph for that probability. On your graph, x should be number of flips and y should be the probability of heads. We are expecting the curve converges to 1/2 since the coin is fair.

Coin flips

Analysis:

- We could keep generating a logical variable, using 1 for heads and 0 for tails.
- We also need a vector, which has 1000 elements, recording how many heads we got after each iteration.
- Another vector can be used for recording 1000 probabilities.
- Functions will be used: *sample(), plot()*.

Example

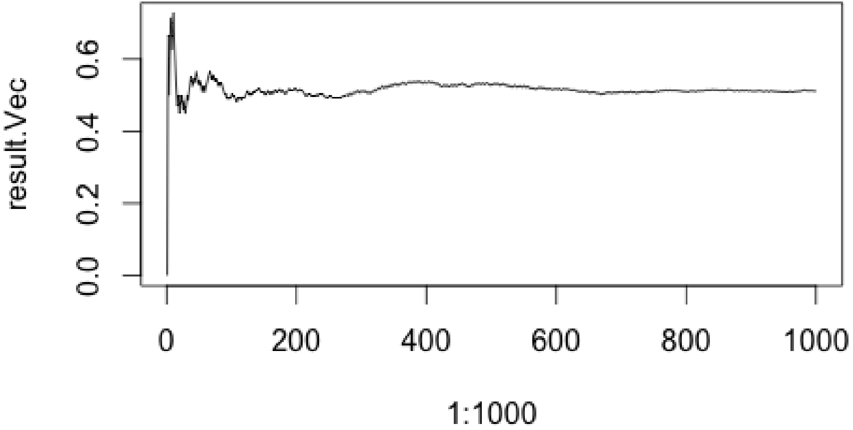
```
# To flip once:
> sample(x = c(1, 0),
         size = 1,
         replace = T,
         prob = c(0.5, 0.5))
[1] 1
```

Coin flips

Example

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

Coin flips



User Defined Functions

Functions are created using *function()* directive and are stored as R objects like anything else. There is actually a function class in R.

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

foo <- function(# parameters)
{
    # a function returns something
    return()
}
```

Function

Example

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

Function

The return value can be assigned to another variable / object

Example

```
> add <- function(a, b)
+ {
+     c <- a + b
+     return (c)
+ }
> add(1, 2)
[1] 3

> result <- add(1, 2)
> result
[1] 3
```

Function

- Functions can be passed as arguments to other functions.
- Functions can be nested, so that you can define a function inside of another function.
- The return value of a function is the last expression in the function body to be evaluated.

Function

Back to our coin flipping example. The following code are taking the head probability as the only argument.

Example

```
coinFlip <- function(headProb) {
    No.heads <- 0
    result.Vec <- NULL
    for (flips in 1:1000)
    {
        tmp <- sample(x=c(1, 0), size=1, replace=T,
                      prob=c(headProb, 1-headProb))

        No.heads <- No.heads + tmp
        result.Vec <- c(result.Vec,No.heads/flips)
    }
    plot(1:1000, result.Vec, type="l")
}
```

Function

Now we have defined a function, to call it in the correct way you need to pass parameters with right type. In this case it has to be a real number between 0 and 1.

Example

```
> coinFlip(0.5)
> coinFlip(0.7)
> coinFlip(0.9)
> coinFlip(1)
```

R Coding Style

- R Interal: R Coding Standards
- Google: R Style Guide
- 4D Pie Charts: R Code Style

R Coding Style

I summarized them and listed some entries here:

- Use meaningful names on variables, functions as well as files.
- The maximum line length is 80 characters.
- At least 4 spaces for indentation.
- Place space arount all binary operators, such as '+', '-', '<-'
- Always place a space after a comma.
- Sufficient comments.