STATS 406 F15: Lab 01

1 Preliminaries

- Thur: 11:30a.m. – 1:00p.m. (Jun Guo)
- Office hrs: (@1720 Science Learning Center, Chemistry Building. Attend any)
  - Tue: 10:00a.m. – 11:30a.m. (Yuan Zhang)
  - Tue: 1:00p.m. – 2:30p.m. (Dao Nguyen)
  - Tue: 3:00p.m. – 4:30p.m. (Jun Guo)

2 Start R coding in Rstudio

- Where to write your code: (Ctrl + Shift + N) or (Windows) File − > New − > R Script
- Saving the file: (Ctrl + S) or File − > Save. Save it with an extension '.R'.
- How to run the code:
  - Running a single line or a selection of part of the code: Ctrl + R (Windows), Command + Enter (Mac)
  - Running the whole file:
    - Type in the Console: source('filename')
      Make sure you are in the right directory.
    - Ctrl + Shift + S (Windows).
    - Set proper working directory.
  - Set proper working directory.

```
## check current working directory
getwd()

## set a new working directory
setwd('/User/guojun/Desktop')    # an example
```

3 Variables and Functions

Variables are objects in ***R*** that store values; we usually initialize variables in ***R*** using the assignment operators = or < −, which are equivalent. We can explicitly assign value to a variable, also, variables can be implicitly declared using the value of another variables or as the result of some function:

```
x <- 6
y <- x
print(y)
[1] 6
```

```
x <- 8
y <- x^2 - 2*x + 1
print(y)
[1] 49
```

Functions in ***R*** is a piece of code which takes input called arguments, perform a task and possibly return a value. The following is a simple function that takes an input variable x and computes x^n:

```
y = function(n, x)
{
    return (x^n)
}
print(y(2, 7))
[1] 49
```

4 Vectors and Matrices

vector is a collection of objects, say variables of the same data type(integer, double, logical, character and complex). Scalar valued variables are essentially vectors of length 1. Longer vectors can be created by *concatenating* a list of scalars by the function **c()**, or by explicit declaration, for example by the function **seq()**.

Exercise: Create the following vectors in specified ways:

1. vector $[1.0, 1.5, 2.0, 2.5, 3.0]$ with **seq(from=, to=, length=)** function;
2. the above vector by concatenating the variable x1 = 1.0 and the vector x2 = $[1.5, 2.0, 2.5, 3.0]$ where x2 is generated by **seq(from=, to=, by=)** function;

Once a vector is created it can accept the same mathematical operations as a scalar variable, and the result is a vector where that operation has been carried out on each element: (the same rule applies to matrix). As an example we revisit the simple function we defined above.

```
n <- seq(0, 5, by=1) ## this is the same as n <- c(0:5) or n <- 0:5
print(n)
[1] 0 1 2 3 4 5
x <- seq(1, 10, length = 10)
print(x)
[1]  1  2  3  4  5  6  7  8  9 10
## compute the values of 2 raised to the powers from 0 to 5;
y1 = y(n, 2)
## compute the squares of 1 to 10;
y2 = y(2, x)
## quote the function exp() in R that computes e^x for x from 1 to 10;
y3 = exp(x)
print(y1)
print(y2)
print(y3)
```

Individual elements of v can be accessed by indexing v. Indices may also be themselves a vector:

```
v <- c(4, 5, 10, 19, 23, 4, 16)
## print the 6th element of v
print(v[6])
## print the 1st, 3rd, and 4th elements of v
index <- c(1, 3, 4)
print( v[index] )
[1] 4 10 19
```

Remember that values in index cannot exceed length(v). This would be like requesting the 8th element of a vector that is only of length 7. Above we have used indexing to select cases from a vector, but we can also use it to delete cases:

```
## select everything except 2nd, 5th, 6th and 7th element from v
index <- c(2,5,6,7)
v[-index]
[1] 4 10 19
```

Other useful basic commands on vectors include

- sort(x): which returns the values in x sorted from lowest to highest
- max(x) and min(x) which return the max/min values in x
- mean(x), median(x), sd(x)

Exercise: Write a line of code that will generate a vector of the form $m^2, (m-1)^2, ..., 1, 0, 1, ..., (m-1)^2, m^2$ for an arbitrary input value, m. Test this code for $m = 5$ to see that

```
[1] 25 16 9 4 1 0 1 4 9 16 25
```

is returned.

Solution: This sequence is the square of the sequence $-m, -m+1, -m+2, \ldots, -1, 0, 1, \ldots, m-2, m-1, m$ so **seq(-m, m, by=1)∧ 2** will suffice (remember to assign a value to m first). |

Much like vectors are lists of scalars, matrices are lists of vectors. They can be created explicitly or by concatenating vectors. Suppose we want to create a variable containing the matrix

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

We could create this explicitly by specifying the values for each entry, or by binding a number of vectors:

```
## Assign all the loading values as a vector to a matrix, specify dimensions with nrow and ncol, note
## that R locate the values column by column.
A <- matrix(c(1:9), nrow=3, ncol=3)
print(A)
[,1] [,2] [,3]
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
## Create A by concatenating vectors
v1 <- seq(1, 3, by=1) ## same as c(1:3)
v2 <- seq(4, 6, by=1) ## same as c(4:6)
v3 <- seq(7, 9, by=1) ## same as c(7:9)
A <- cbind( v1, v2, v3 )
colnames(A) = NULL ## dont worry about this
```

The command cbind creates a matrix where each of the vectors passed to it are the columns of the matrix. Notice that matrices are indexed in the same way as vectors, with the first and second index referring to the row and column numbers, respectively. If one of the two indices are left blank then that entire row/column is returned:

```
# get the second column of A
A[,2]
[1] 4 5 6
# get the third row of A
A[3,]
[1] 3 6 9
# get the entry in third row and 2nd column of A
A[3,2]
[1] 6
# get the entries in first and third rows, and 2nd and third column of A
A[c(1,3),c(2,3)]
     [,1] [,2]
[1,]    4    7
[2,]    6    9
```

Suppose we are given a matrix B

```
B = matrix(rnorm(25),5,5)
```

we need to extract the row with the largest first entry.

```
u = B[,1] ## First column of B
indx = which.max(u) ## OR indx = which(u == max(u))
v = B[indx,]
```

Suppose you want to count the proportion of entries in v which are positive.

```
prop = mean(v>0) ## v>0 is a vector of logicals.
```

Similarly to vectors, usual mathematical operations on a matrix are carried out elementwise. One final basic matrix command that is useful is the transpose function, which returns a matrix with the rows and columns reversed from the input matrix:

```
t(A)
[,1] [,2] [,3]
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
```

Multiply two matrices by matrix multiplication

```
AA = t(A) %*% A
print(AA)
```

Calculate determinant of a square matrix A in

```
print(det(A))
```

5 Data frames

A data frame is used for storing data tables. It is a list of vectors of equal length, each vector usually records a certain attribute of the objects the data tries to describe, we can also view the data frame as a matrix like object with rows and columns to refer to. For example, the following variable dt0 is a data frame containing three vectors N, C, L.

```
N <- c(1, 2, 3)
C <- c("a", "b", "c")
L <- c(TRUE, FALSE, TRUE)
dt0 <- data.frame(N, C, L)       # dt0 is a data frame
class(dt0)  # type of the object ‘dt0’
names(dt0)  # attribute (column vector) names of ‘dt0’
print(dt0)
  N C     L
1 1 a  TRUE
2 2 b FALSE
3 3 c  TRUE

dt0mat = data.matrix(dt0)  # convert a data frame to a matrix object
print(dt0mat)  # notice the change
     N C L
[1,] 1 1 1
[2,] 2 2 0
[3,] 3 3 1
```

Now let’s work on a real data frame ‘airqualities.txt’. First download and save the data to your working directory

```
# import the data from your current working directory
airq = read.table('airquality.txt', header = T, stringsAsFactors = F)

# check the type of the object imported
class(airq)

# check the attribute (each column vector) names
names(airq)

# have a look of the data frame
head(airq)

# suppose we are only interested in complete cases without ‘NA’ records
cpl.cases = complete.cases(airq)
aircompl = airq[cpl.cases, ]    # create a sub data frame accordingly

# we only look at data from the month 7 here, assume it’s July
July.id = which(aircompl[,'Month'] == 7)
July.air = aircompl[July.id, ]    # create a further sub data frame accordingly

# find the day in July with the highest Ozone recording
ozmax.id = which.max(July.air[,'Ozone'])
day.ozmax = July.air[ozmax.id, 'Day']
print(day.ozmax)
[1] 1
```

6 Random number generation

At the heart of many programming tasks in this course is simulating random variables. R has a number of functions that can be used to generate from various distributions:

```
# u contains n random variables uniformly distributed on (a,b)
n <- 10
a <- 1
b <- 5
u <- runif(n, min=a, max=b)
# u contains n normally distributed variables with mean mu and variance sigmasq
n <- 10
mu <- 0
sigmasq <- 2
u <- rnorm(n, mean=mu, sd=sqrt(sigmasq))
# u contains n exponentially distributed variables with parameter lambda
n <- 10
lambda <- 5
u <- rexp(n, rate=lambda)
# u contains n binomial random variables with parameters k, p
n <- 10
k <- 20
p <- .5
u <- rbinom(n, size=k, prob=p)
```

For each of the random number generating functions above (which begin with r), there is a corresponding probability density function, CDF function, and quantile function (inverse CDF), which begin with d, p and q, respectively. For example the normal density, CDF, and quantile functions are dnorm, pnorm, and qnorm. We can use simulation to estimate quantities that may or may not be easy to calculate by hand. For example, recall that the Binomial(n, p) distribution has probability mass function

$$p(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

Using this we can explicitly calculate, for example, $P(X > 3)$ when n and p are given. Suppose that $n = 10$ and $p = .35$ can also use R to calculate this by simulation:

```
n <- 10
p <- .35
#### exact probability
# using the built-in binomial CDF function
1-pbinom(3, size=n, prob=p)
[1] 0.486173
# calculating it ourselves
prob <- 0
k = 4:10
prob <- choose(n, k)*(p^k)*((1-p)^(n-k))
sum(prob)
[1] 0.486173
#### calculate by sampling
# nreps: number of simulated i.i.d. variables-- nreps larger --> more accurate
nreps = 10000
X = rbinom(nreps, size=n, prob=p)
mean(X > 3)
[1] 0.4864
```