Probability and Statistics (IT302) Class No. 25
13^{th} October 2020 Tuesday 10:30AM - 11:00AM

Basic Arithmetic and Objects

R has a command line interface, and will accept simple commands to it. This is marked by a **> symbol**, called the prompt. If you type a command and press return, R will evaluate it and print the result for you.

```
> 6 + 9

[1] 15

> x <- 15
> x - 1

[1] 14
```

The expression x <- 15 creates a variable called x and gives it the value 15. This is called assignment; the variable on the left is assigned to the value on the right. The left hand side must contain only contain a single variable.

```
> x + 4 <- 15     # doesn't work
```

Assignment can also be done with = (or ->).

```
> x = 5
> 5*x -> x
> x

[1] 25
```

The **operators = and <- are identical**, but **many people prefer <-** because it is not used in any other context, but = is, so there is less room for confusion.

Source : R Programming by Robin Evans

Vectors

The key feature which makes R very useful for statistics is that it is vectorized. This means that many operations can be performed point-wise on a vector. The function c() is used to create vectors:

```
> x <- c(1, -1, 3.5, 2)
> x
[1] 1.0  -1.0  3.5  2.0
```

Then if we want to add 2 to everything in this vector, or to square each entry:

```
> x + 2
[1] 3.0  1.0  5.5  4.0

> x^2
[1] 1.00  1.00  12.25  4.00
```

Source : R Programming by Robin Evans

Vectors Contd.

This is very useful in statistics:

```
> sum((x - mean(x))^2)
[1] 10.69
```

Colon operator

```
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10

> -3:4
[1] -3 -2 -1 0 1 2 3 4

> 9:5
[1] 9 8 7 6 5
```

Source : R Programming by Robin Evans

function seq()

More generally, the **function seq()** can generate any arithmetic progression.

```
> seq(from=2, to=6, by=0.4)

[1] 2.0  2.4  2.8  3.2  3.6  4.0  4.4  4.8  5.2  5.6  6.0

> seq(from=-1, to=1, length=6)

[1] -1.0  -0.6  -0.2  0.2  0.6  1.0
```

Source : R Programming by Robin Evans

rep()

Sometimes it's necessary to have repeated values, for which we use rep()

```
> rep(5,3)
[1]  5  5   5
```

```
> rep(2:5,each=3)
[1] 2   2   2    3  3   3   4   4   4   5   5   5
```

```
> rep(-1:3, length.out=10)
[1] -1  0   1   2   3   -1  0   1   2   3
```

We can also use R's vectorization to create more interesting sequences:

```
> 2^(0:10)
[1] 1   2   4   8   16  32  64  128     256     512     1024
```

Source : R Programming by Robin Evans

rep() contd.

```
> 1:3 + rep(seq(from=0,by=10,to=30), each=3)
[1] 1  2  3  11 12 13 21 22 23 31 32 33
```

The last example demonstrates recycling, which is also an important part of vectorization. If we perform a binary operation (such as +) on two vectors of different lengths, the shorter one is used over and over again until the operation has been applied to every entry in the longer one. If the longer length is not a multiple of the shorter length, a warning is given.

```
> 1:10 * c(-1,1)
[1] -1  2  -3  4  -5  6  -7  8  -9  10
```

```
> 1:7 * 1:2
Warning: longer object length is not a multiple of shorter object length
[1] 1  4  3  8  5  12 7
```

Source : R Programming by Robin Evans

Subsetting

It's frequently necessary to extract some of the elements of a larger vector. In R you can use square brackets to select an individual element or group of elements:

```
> x <- c(5,9,2,14,-4)
> x[3]
[1]  2

> # note indexing starts from 1
> x[c(2,3,5)]
[1] 9   2   -4

> x[1:3]
[1] 5   9   2

> x[3:length(x)]
[1] 2   14  -4
```

Source : R Programming by Robin Evans

Subsetting Contd.

There are two other methods for getting subvectors. The First is using a logical vector (i.e. containing TRUE and FALSE) of the same length:

```
> x > 4
[1] TRUE  TRUE  FALSE     TRUE  FALSE
```

```
> x[x > 4]
[1] 5   9   14
```

or using negative indices to specify which elements should not be selected:

```
> x[-1]
[1] 9   2   14  -4
```

```
> x[-c(1,4)]
[1] 9   2   -4
```

Source : R Programming by Robin Evans

Logical Operators

The **comparison operator** > returns a logical vector indicating whether or not the left hand side is greater than the right hand side. Here we demonstrate the other comparison operators:

```
> x <= 2 # less than or equal to
[1] FALSE  FALSE  TRUE FALSE  TRUE
```

```
> x == 2 # equal to
[1] FALSE  FALSE  TRUE FALSE  FALSE
```

```
> x != 2 # not equal to
[1] TRUE TRUE FALSE  TRUE TRUE
```

Note the **double equals sign ==, to distinguish between assignment and comparison**.

Source : R Programming by Robin Evans

Logical Operators Contd.

If we want the elements of x within a range, we can use the following:

```
> (x > 0) & (x < 10) # and
[1] TRUE  TRUE  TRUE  FALSE  FALSE
```

The & operator does a pointwise `and' comparison between the two sides**. Similarly, the **vertical bar | does pointwise `or', and the unary ! Operator performs negation.

```
> (x == 5) | (x > 10)
[1]  TRUE  FALSE  FALSE  TRUE  FALSE

> !(x > 5)
[1] TRUE  FALSE  TRUE  FALSE  TRUE
```

Source : R Programming by Robin Evans

Character Vectors

As you might have noticed in the exercise above, vectors don't have to contain numbers. We can equally create a character vector, in which each entry is a string of text. Strings in R are contained within double quotes ":

```
> x <- c("Hello", "how do you do", "lovely to meet you", 42)
> x
[1] "Hello“       "how do you do"  "lovely to meet you"
[4] "42"
```

Notice that you cannot mix numbers with strings: if you try to do so the number will be converted into a string. Otherwise character vectors are much like their numerical counterparts.

```
> x[2:3]
[1]       "how do you do"         "lovely to meet you“
> x[-4]
[1]       "Hello"           "how do you do"         "lovely to meet you"
```

Source : R Programming by Robin Evans

Matrices

Matrices are much used in statistics, and so play an important role in R. To create a matrix use the function matrix(), specifying elements by column first:

```
> matrix(1:12, nrow=3, ncol=4)

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

This is called column-major order. Of course, we need only give one of the dimensions:

```
> matrix(1:12, nrow=3)
```

Source : R Programming by Robin Evans

Matrices Contd.

unless we want vector recycling to help us:

```
> matrix(1:3, nrow=3, ncol=4)
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    2    2    2    2
[3,]    3    3    3    3
```

Sometimes it's useful to specify the elements by row first

```
> matrix(1:12, nrow=3, byrow=TRUE)
```

Source : R Programming by Robin Evans

Matrices Contd.

```
> diag(3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

```
> diag(1:3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

```
> 1:5 %o% 1:5
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

Source : R Programming by Robin Evans

Matrices Contd.

The last operator performs an outer product, so it creates a matrix with (i; j)-th entry $x_i y_j$. The function outer() generalizes this to any function f on two arguments, to create a matrix with entries $f(x_i; y_j)$.

```
> outer(1:3, 1:4, "+")
     [,1] [,2] [,3] [,4]
[1,]    2    3    4    5
[2,]    3    4    5    6
[3,]    4    5    6    7
```

Source : R Programming by Robin Evans

Matrices Contd.

Matrix multiplication is performed using the operator %*%, which is quite distinct from scalar multiplication *.

```
> A <- matrix(c(1:8,10), 3, 3)
> x <- c(1,2,3)
> A %*% x # matrix multiplication
     [,1]
[1,]   30
[2,]   36
[3,]   45

> A*x # NOT matrix multiplication
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    4   10   16
[3,]    9   18   30
```

Source : R Programming by Robin Evans

Matrices Contd.

Standard functions exist for common mathematical operations on matrices.

```
> t(A)        # transpose
          [,1]        [,2]        [,3]
[1,]      1           2           3
[2,]      4           5           6
[3,]      7           8           10

> det(A)          # determinant
[1]       -3

> diag(A) # diagonal
[1]       1         5         10
```

Source : R Programming by Robin Evans

rbind() and cbind() functions

You can stitch matrices together using the rbind() and cbind() functions.These employ vector recycling:

```
> cbind(A, t(A))

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    4    7    1    2    3
[2,]    2    5    8    4    5    6
[3,]    3    6   10    7    8   10

> rbind(A, 1, 0)

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6   10
[4,]    1    1    1
[5,]    0    0    0
```

Source : R Programming by Robin Evans

Matrices

A matrix is a two-dimensional rectangular data set. It can be created using a vector input to the matrix function.

```
# Create a matrix.
M = matrix( c('a','a','b','c','b','a'), nrow=2,ncol=3,byrow = TRUE)
print(M)
```

When we execute the above code, it produces the following result:

```
     [,1] [,2] [,3]
```

```
[1,] "a"  "a"  "b"
[2,] "c"  "b"  "a"
```

Matrices Contd.

- Matrix is a two dimensional data structure in R programming. Matrix is similar to vectors but additionally contains the dimension attribute.
- All attributes of an object can be checked with the attributes() function (dimension can be checked directly with the dim() function).
- We can check if a variable is a matrix or not with the class() function.

```
> a
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> class(a)
[1] "matrix"

> attributes(a)
$dim
[1] 3 3

> dim(a)
[1] 3 3
```

Source : STATISTICS WITH R PROGRAMMING Lecture Notes

matrix() function

Matrix can be created using the matrix() function. Dimension of the matrix can be defined by passing appropriate value for arguments **nrow** and **ncol**.

Providing value for both dimension is not necessary. If one of the dimension is provided, the other is inferred from length of the data.

We can see that the **matrix is filled column-wise.** This can be reversed to row-wise filling by passing TRUE to the argument byrow.

```
> matrix(1:9, nrow = 3, ncol = 3)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9


> # same result is obtained by providing only one dimension
> matrix(1:9, nrow = 3)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Source : STATISTICS WITH R PROGRAMMING Lecture Notes

matrix() function Contd.

```
> matrix(1:9, nrow=3, byrow=TRUE)    # fill matrix row-wise
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

It is possible to name the rows and columns of matrix during creation by passing a 2 element list to the argument **dimnames**.

```
> x <- matrix(1:9, nrow = 3, dimnames = list(c("X","Y","Z"),
c("A","B","C")))
> x
  A B C
X 1 4 7
Y 2 5 8
Z 3 6 9
```

Source : STATISTICS WITH R PROGRAMMING Lecture Notes

colnames() and rownames() functions

These names can be accessed or changed with two helpful functions **colnames()** and **rownames().**

```
> colnames(x)
[1] "A" "B" "C"
> rownames(x)
[1] "X" "Y" "Z"


> # It is also possible to change names
> colnames(x) <- c("C1","C2","C3")
> rownames(x) <- c("R1","R2","R3")


> x
   C1 C2 C3
R1  1  4  7
R2  2  5  8
R3  3  6  9
```

Source : STATISTICS WITH R PROGRAMMING Lecture Notes

Cbind() and rbind() functions

Another way of creating a matrix is by using functions cbind() and rbind() as in column bind and row bind.

```
> cbind(c(1,2,3),c(4,5,6))
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6


> rbind(c(1,2,3),c(4,5,6))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Source : STATISTICS WITH R PROGRAMMING Lecture Notes

Dim() function

you can also create a matrix from a vector by setting its dimension using dim().

```
> x <- c(1,2,3,4,5,6)
> x
[1] 1 2 3 4 5 6
> class(x)
[1] "numeric"
```

```
> dim(x) <- c(2,3)
> x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> class(x)
[1] "matrix"
```

Source : STATISTICS WITH R PROGRAMMING Lecture Notes

How to access Elements of a matrix?

We can access elements of a matrix using the square bracket [indexing method. Elements can be accessed as var[row, column]. Here rows and columns are vectors.

Using integer vector as index

We specify the row numbers and column numbers as vectors and use it for indexing. If any field inside the bracket is left blank, it selects all. We can use negative integers to specify rows or columns to be excluded.

```
> x
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Source : STATISTICS WITH R PROGRAMMING Lecture Notes

How to access Elements of a matrix? Contd

```
> x[c(1,2),c(2,3)]    # select rows 1 & 2 and columns 2 & 3
     [,1] [,2]
[1,]    4    7
[2,]    5    8


> x[c(3,2),]    # leaving column field blank will select entire
columns
     [,1] [,2] [,3]
[1,]    3    6    9
[2,]    2    5    8
```

Source : STATISTICS WITH R PROGRAMMING Lecture Notes

How to access Elements of a matrix? Contd.

```
> x[,]    # leaving row as well as column field blank will select
entire matrix
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9


> x[-1,]    # select all rows except first
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    3    6    9
```

One thing to notice here is that, if the matrix returned after **indexing is a row matrix or column matrix, the result is given as a vector.**

Source : STATISTICS WITH R PROGRAMMING Lecture Notes

How to access Elements of a matrix? Contd.

```
> x[1,]
[1] 1 4 7
> class(x[1,])
[1] "integer"
```

This behavior can be avoided by using the argument drop = FALSE while indexing.

```
> x[1,,drop=FALSE]  # now the result is a 1X3 matrix rather than a
vector
     [,1] [,2] [,3]
[1,]    1    4    7
> class(x[1,,drop=FALSE])
[1] "matrix"
```

Source : STATISTICS WITH R PROGRAMMING Lecture Notes

How to access Elements of a matrix? Contd.

It is possible to index a matrix with a single vector.

While indexing in such a way, it acts like a vector formed by stacking columns of the matrix one after another. The result is returned as a vector.

```
> x
     [,1] [,2] [,3]
[1,]    4    8    3
[2,]    6    0    7
[3,]    1    2    9

> x[1:4]
[1] 4 6 1 8

> x[c(3,5,7)]
[1] 1 0 3
```

Source : STATISTICS WITH R PROGRAMMING Lecture Notes