A Brief introduction to R

Arnab Maity

NCSU Statistics ~ 5240 SAS Hall ~ amaity[at]ncsu.edu

Contents

R and RStudio

The software R is free to download from the *The Comprehensive R Archive Network* (`https://cran.r-project.org/`). I also encourage you to download RStudio, an integrated development environment, for efficient coding. RStudio is available from `https://www.rstudio.com/` for free.

Basic data types in R

R has the following six basic data types.

- **Character**: such as names like `"Arnab"`,
- **Numeric**: integer and double, e.g., `1L`, `2` or `-3.9`,[1]
- **Logical**: Boolean data, `TRUE`, `FALSE` and `NA`,
- **Complex**: numbers such as `1 + 2i`,
- **Raw**: holds raw bytes.

[1] The specification `1L` explicitly tells R that it is an integer. But `2` is in fact stored as a double.

The **Raw** data type does not come up in most scenarios we encounter in standard programming, and will not be discussed further.

Variable names and assignment operator

Often we need to store our data with a name so that we can use them later. We use assignment operator `<-` to do so. For example, the command `x <- 2` assigns[2] the the value `2` to the name `x` .

[2] The command `x = 2` also works. However, the = sign is also used to specify function arguments.

```
# numeric data
number <- 2
print(number)
```

```
## [1] 2
```

```
# character
name <- "Arnab"
print(name)
```

```
## [1] "Arnab"
```

```
# Logical
bool <- TRUE
print(bool)
```

```
## [1] TRUE
```

Notice that the assignment `bool <- TRUE`, we did not put quotation marks around `TRUE`. This is because `TRUE` is a Boolean constant, not a character string.[3]

You can see the type of a variable by using the command `typeof`, as follows.

```
typeof(number)
```

```
## [1] "double"
```

```
typeof(bool)
```

```
## [1] "logical"
```

[3] What would be the result of the assignment `bool <- "TRUE"`? What type of variable would `bool` be then?

Asking for help

You can view the help page/documentation for any object in R, if such a page is available, by using the `?` or `help()` command. It is particularly useful for complicated functions. Always view the docs of any functions that are new to you.

We have seen the use of `print()` and `typeof()` in the previous sections. Try issuing the commands `?print` and `?typeof` to see what happens. Try `?help`.

Basic operations

At the least, you can you R as a calculator. It performs basic arithmetic operations for numeric data:

- `+`, `-`, $*$ and \ for addition, subtraction, multiplication and division, respectively.
- `^` for exponentiation, `%%` for remainder from division and `%/%` for integer division.[4]

[4] While `%%` and `%/%` can be used for non-integer values, the results may vary in different platforms since they are susceptible to representation error.

Other mathematical functions such as exponential `exp()`, natural logarithm `log()`, trigonometric function `sin()`, `cos()` etc. are also available.

```
x <- 2
y <- 3
x + y
```

```
## [1] 5
```

```
x^2
```

```
## [1] 4
```

```
cos(pi*x/2)
```

```
## [1] -1
```

Notice that, in the last command, we used `pi`. Constants such as `pi` are pre-defined in R.

Relational operations

R has the usual relational operations available:

- <, <=, > and >= for less than, less than or equal to, greater than, greater that or equal to, respectively.
- == and `!=` for equal to and not equal to, respectively.

Each of these operations returns `TRUE/FALSE` value.

```
# numeric data
x <- 2
y <- 3
x == y
```

```
## [1] FALSE
```

```
x != y
```

```
## [1] TRUE
```

```
x <= y
```

```
## [1] TRUE
```

```
# Character data
"Arnab" == "Maity"
```

```
## [1] FALSE
```

Vectors

Vectors are one of the most imortant data structures in R. It is very important that we understand how to create, manipulate, and perform computations using vectors to be effective R users. There are two types of vectors in R: **atomic vector** and **list**.

Atomic vectors

Atomic vectors is a collection of elements of the *same type*. You can create such vectors using the `c()` function in R. For example, shown below is a atomic vector containing `double` data type.

```
dbl_vec <- c(1.2, 3, -5.9)
dbl_vec
```

```
## [1]  1.2  3.0 -5.9
```

Thus atomic vectors can be of any of the basic data types (integer, double, character etc.) discussed above. Even though the vector is printed in a row, by default, **a vector behaves as a column vector.**

If we attempt to put multiple data types in the same vector, they will be coerced to the most flexible type automatically.

```
multi_vec <- c(23.5, "Arnab", TRUE)
multi_vec
```

```
## [1] "23.5"  "Arnab" "TRUE"
```

Note that the double and the logical elements of `multi_vec` were converted to characters.

Vector (numeric) operations with R

In this course, we will mainly deal with numeric vectors when performing data analysis. A vector is an **array of numbers**. Specifically, we will write

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{pmatrix}$$

and call it a *column vector*. We often write $\boldsymbol{x} \in \mathbb{R}^p$. Similarly, a *row vector* is written as

$$\boldsymbol{x}^T = (x_1, x_2, \ldots, x_p).$$

Note that the notation $\boldsymbol{x}^T$ denotes "transpose'' of $\boldsymbol{x}$.[5]

The usual vector operations in linear algebra can be done on these vectors. For two vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^p$, the sum is defined as[6]

$$\boldsymbol{a} + \boldsymbol{b} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \vdots \\ a_p + b_p \end{pmatrix},$$

[5] **Note:** In this course, **we will always take a vector as a column vector by convention, and will always use the transpose to denote a row vector.** Thus the statement "$\boldsymbol{a}$ is a vector" will imply that "$\boldsymbol{a}$ is a *column* vector."

[6] Similarly, the difference is defined as

$$\boldsymbol{a} - \boldsymbol{b} = \begin{pmatrix} a_1 - b_1 \\ a_2 - b_2 \\ \vdots \\ a_p - b_p \end{pmatrix}$$

that is, a vector of same dimension as of $\boldsymbol{a}$ and $\boldsymbol{b}$, where each element is the sum of corresponding elements of $\boldsymbol{a}$ and $\boldsymbol{b}$.

Consider the two vectors as follows.[7]

```
a = c(5.1,  4.9, 4.7, 4.6, 5.0)
b = c(3.5,  3.0, 3.2, 3.1, 3.6)
```

[7] Note that to add (or subtract) $\boldsymbol{a}$ and $\boldsymbol{b}$, the two vectors have to have the same number of elements.

Their sum is:

```
a + b
```

```
## [1] 8.6 7.9 7.9 7.7 8.6
```

Their difference is:

```
a - b
```

```
## [1] 1.6 1.9 1.5 1.5 1.4
```

A vector $\boldsymbol{a}$ can be multiplied by a scalar k by simply multiplying each element of $\boldsymbol{a}$ by k:

$$k\boldsymbol{a} = k \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_p \end{pmatrix} = \begin{pmatrix} ka_1 \\ ka_2 \\ \vdots \\ ka_p \end{pmatrix}$$

In R, we can use the $*$ operator:[8]

[8] We can similarly divide a vector by a scalar by using the / operator.

```
a
```

```
## [1] 5.1 4.9 4.7 4.6 5.0
```

```
2*a
```

```
## [1] 10.2  9.8  9.4  9.2 10.0
```

Multiplication between two vectors is a little more involved. Here we need to define the *inner product* of two vectors. For two vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^p$, the inner product is defined as:

$$\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \boldsymbol{a}^T \boldsymbol{b} = \begin{pmatrix} a_1 & a_2 & \ldots & a_p \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix} = a_1 b_1 + a_2 b_2 + \ldots + a_p b_p = \sum_{j=1}^{p} a_j b_j.$$

Note that *the result is a scalar*. As an example, suppose $\boldsymbol{a}^T = (1, 0, 2, 5)$ and $\boldsymbol{b} = \begin{pmatrix} 2 \\ 3 \\ 1 \\ 6 \end{pmatrix}$. Then we have

$$\boldsymbol{a}^T\boldsymbol{b} = \begin{pmatrix} 1 & 0 & 2 & 5 \end{pmatrix} \times \begin{pmatrix} 2 \\ 3 \\ 1 \\ 6 \end{pmatrix} = (1 \times 2) + (0 \times 3) + (2 \times 1) + (5 \times 6) = 34$$

In R, we can use the `%*%` operator to compute the inner product (or matrix multiplication in general). In this example[9]

```
a <- c(1, 0, 2, 5)
b <- c(2, 3, 1, 6)

t(a) %*% b
```

```
##      [,1]
## [1,]   34
```

Other operations such as exponentiation by a scalar, `log()`, etc are done element wise on a vector.

```
vec_one <- c(1,2,3)
vec_two <- c(4,5,6)
# log transform
log(vec_one)
```

```
## [1] 0.0000000 0.6931472 1.0986123
```

We can access element of a vector by using the `[` operator. For example, to access the first element of `vec_one` we will use `vec_one[1]`.

```
vec_one[1]
```

```
## [1] 1
```

We can assign specific values to elements using `[` and `<-` together.

```
vec_one[2] <- 31
vec_one
```

```
## [1]  1 31  3
```

It is possible, and often desirable to create named vectors, that is, a vector which has names for each element.

[9] **Note:** Be careful to use `%*%`. Be sure to put the `%` signs properly. Just using $*$ without the `%` signs would give you elementwise product:

$$\boldsymbol{a} * \boldsymbol{b} = \begin{pmatrix} a_1 b_1 \\ a_2 b_2 \\ \vdots \\ a_p b_p \end{pmatrix}.$$

In matrix algebra this is referred to as *Hadamard product*.

```
vec_named <- c(math = 91, engligh = 85, history = 99)
vec_named
```

```
##    math engligh history
##      91      85      99
```

For such a vector, we can refer/assign to its elements by both index and name.

```
vec_named[1]
```

```
## math
##   91
```

```
vec_named["math"]
```

```
## math
##   91
```

See also the `names()` function.

Lists

Lists are vectors that can hold different types of elements, unlike atomic vectors. We can create a list using the `list()` function.

```
my_list <- list(number = 23,
                name = "Arnab",
                is_student = FALSE)
my_list
```

```
## $number
## [1] 23
##
## $name
## [1] "Arnab"
##
## $is_student
## [1] FALSE
```

We can access element of a list either by its name, if they exist (in the example above, the names are `number`, `name` and `is_student`) with `$` operator, or using their index with `[[` operator. Assignment of new values can be done the same way with `<-`.

```
# Access elements
my_list$number
```

```
## [1] 23
```

```
my_list[[2]]
```

```
## [1] "Arnab"
```

```
# value assignment
my_list$number <- 50 # changing existing element
my_list$new_data <- -23 # adding a new element
my_list
```

```
## $number
## [1] 50
##
## $name
## [1] "Arnab"
##
## $is_student
## [1] FALSE
##
## $new_data
## [1] -23
```

Lists are quite verstile, and can hold other data structures as well. For example, a list can hold other vectors, lists, matrices etc as well.

```
big_list <- list(vec = c(1,2,3),
                 lst = list("Arnab", FALSE, 34),
                 num = 45,
                 long_lst = list(a = 1, b = list(4,5))
                )
big_list
```

```
## $vec
## [1] 1 2 3
##
## $lst
## $lst[[1]]
## [1] "Arnab"
##
## $lst[[2]]
## [1] FALSE
```

```
##
## $lst[[3]]
## [1] 34
##
##
## $num
## [1] 45
##
## $long_lst
## $long_lst$a
## [1] 1
##
## $long_lst$b
## $long_lst$b[[1]]
## [1] 4
##
## $long_lst$b[[2]]
## [1] 5
```

Lists are used to build other complicated data structures, such as data frames, which we discuss later.

Matrices

Like atomic vectors, a matrix (2D arrays) can hold **only one type of data**. We can create matrices by using the `matrix()` function, or binding multiple vectors by rows or columns using the `rbind()` or `cbind()` functions, respectively.

```
matrix_one <- matrix(1:6, nrow = 2, ncol = 6)
matrix_one
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    3    5    1    3    5
## [2,]    2    4    6    2    4    6
```

```
cbind(vec_one, vec_two)
```

```
##      vec_one vec_two
## [1,]       1       4
## [2,]      31       5
## [3,]       3       6
```

```
rbind(vec_one, vec_two)
```

```
##         [,1] [,2] [,3]
## vec_one    1   31    3
## vec_two    4    5    6
```

We can access the elements with the `[` operator. For matrices we need two indices (one for row and the other for column). Thus `matrix_one[2, 3]` will refer to the element in 2nd row and 3rd column.

```
matrix_one[2, 3]
```

```
## [1] 6
```

Transposing matrices involves turning the first column into the first row, second column into second row and so on. We write $\mathbf{M}^T$ as the transpose of $\mathbf{M}$.

We can use `t()` to take a transpose in R:

```
Mt = t(matrix_one)
Mt
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
## [4,]    1    2
## [5,]    3    4
## [6,]    5    6
```

Addition and subtraction of matrices can be done if the matrices have the *same size*. The sum of two matrices `A` and `B` (of same size) is another matrix (of the same size) where each element is the sum of the corresponding elements of `A` and `B`.

```
A = cbind(c(0.71,  0.61, 0.72, 0.83, 0.92),
          c(0.63,  0.69, 0.77, 0.80, 1.00))
A
```

```
##      [,1] [,2]
## [1,] 0.71 0.63
## [2,] 0.61 0.69
## [3,] 0.72 0.77
## [4,] 0.83 0.80
## [5,] 0.92 1.00
```

```
B = matrix(c(1,2,3,4,5,6,7,8,9,10),5,2)
B
```

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

```
# Summing two matrices
A + B
```

```
##      [,1]  [,2]
## [1,] 1.71  6.63
## [2,] 2.61  7.69
## [3,] 3.72  8.77
## [4,] 4.83  9.80
## [5,] 5.92 11.00
```

```
# Subtracting
A - B
```

```
##       [,1]  [,2]
## [1,] -0.29 -5.37
## [2,] -1.39 -6.31
## [3,] -2.28 -7.23
## [4,] -3.17 -8.20
## [5,] -4.08 -9.00
```

Matrix addition satisfies the usual commutative and associative laws.

$$\begin{aligned} \text{Commutative law:} \quad & \mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A} \\ \text{Associative law:} \quad & \mathbf{A} + (\mathbf{B} + \mathbf{C}) = (\mathbf{A} + \mathbf{B}) + \mathbf{C} \end{aligned}$$

Multiplication of a matrix by a scalar is done by simply multiplying every element in the matrix by the scalar. So if $k = 0.4$, and

$$\mathbf{A} = \begin{pmatrix} 1 & 5 & 8 \\ 1 & 2 & 3 \end{pmatrix},$$

we can calculate $k\mathbf{A}$ as:

$$k\mathbf{A} = 0.4 \times \begin{pmatrix} 1 & 5 & 8 \\ 1 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 0.4 & 2 & 3.2 \\ 0.4 & 0.8 & 1.6 \end{pmatrix}.$$

Matrix multiplication however follows vector multiplication, and therefore does not follow the same rules as basic multiplication. To multiply two matrices `A` and `B`, one must first check that the *number of columns in `A` is exactly the same as the number of rows in `B`*. Otherwise, we can not multiply these two matrices. More generally,

$$\boldsymbol{A}_{m\times n} \times \boldsymbol{B}_{n\times p} = \boldsymbol{C}_{m\times p}.$$

Let $\boldsymbol{A}$ be of size $m \times n$; represent $\boldsymbol{A}$ using its row vectors $\boldsymbol{a}_1^T, \boldsymbol{a}_2^T, \ldots, \boldsymbol{a}_m^T$. Let $\boldsymbol{B}$ be of size $n \times p$; represent $\boldsymbol{B}$ using its columns vectors $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_p$. The multiplication operation for matrices is defined as:

$$\mathbf{AB} = \begin{pmatrix} a_1^T \\ a_2^T \\ \ldots \\ a_m^T \end{pmatrix} \begin{pmatrix} b_1 & b_2 & \ldots & b_p \end{pmatrix} = \begin{pmatrix} a_1^T b_1 & a_1^T b_2 & \ldots & a_1^T b_p \\ a_2^T b_1 & a_2^T b_2 & \ldots & a_2^T b_p \\ \vdots & \vdots & & \vdots \\ a_m^T b_1 & a_m^T b_2 & \ldots & a_m^T b_p \end{pmatrix}$$

Thus, (i, j)-th element of $\mathbf{AB}$ is the inner product of i-th row of $\boldsymbol{A}$ and j-th column of $\boldsymbol{B}$.

Consider the following example.

```
A = cbind(c(0.71,  0.61, 0.72, 0.83, 0.92),
          c(0.63,  0.69, 0.77, 0.80, 1.00))
A
```

```
##      [,1] [,2]
## [1,] 0.71 0.63
## [2,] 0.61 0.69
## [3,] 0.72 0.77
## [4,] 0.83 0.80
## [5,] 0.92 1.00
```

```
B = matrix(c(1,2,3,4,5,6,7,8,9,10),2,5)
B
```

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

Here `A` has 2 columns and `B` has two rows, and hence we can multiply `A` with `B`. In R, we only need to use the `%*%` operator to ensure we are getting matrix multiplication:

```
C = A %*% B
C
```

```
##      [,1] [,2]  [,3]  [,4]  [,5]
## [1,] 1.97 4.65  7.33 10.01 12.69
## [2,] 1.99 4.59  7.19  9.79 12.39
## [3,] 2.26 5.24  8.22 11.20 14.18
## [4,] 2.43 5.69  8.95 12.21 15.47
## [5,] 2.92 6.76 10.60 14.44 18.28
```

Just to check, look at $\boldsymbol{C}_{23}$, the $(2,3)$-th element of $\boldsymbol{C}$.

$$\boldsymbol{C}_{23} = 7.19 = (0.61, 0.69)\begin{pmatrix} 5 \\ 6 \end{pmatrix} = (5 \times 0.61) + (6 \times 0.69) = 7.19.$$

You will get an error message if you multiply non-conformable matrices.[10]

```
B %*% t(A)
```

```
## Error in B %*% t(A): non-conformable arguments
```

[10] Dimesion of `B` is 2×5 but dimension of `t(A)` is 2×5. Thus number of columns in `B` is not the same as number of columns in `t(A)`.

Unlike addition, matrix multiplication is not commutative:

$$\begin{aligned} \text{(non-commutative)} \quad & \mathbf{AB} \neq \mathbf{BA} \\ \text{Associative law} \quad & \mathbf{A}(\mathbf{BC}) = (\mathbf{AB})\mathbf{C} \end{aligned}$$

The distributive laws of multiplication over addition still apply.

$$\begin{aligned} \mathbf{A}(\mathbf{B}+\mathbf{C}) &= \mathbf{AB}+\mathbf{AC} \\ (\mathbf{A}+\mathbf{B})\mathbf{C} &= \mathbf{AC}+\mathbf{BC} \end{aligned}$$

We have the following rules for transposes.

$$\begin{aligned} (\mathbf{A}+\mathbf{B})^T &= \mathbf{A}^T + \mathbf{B}^T \\ (\mathbf{AB})^T &= \mathbf{B}^T\mathbf{A}^T \end{aligned}$$

Other functions such as inverse, determinant, eigen decomposition, SVD etc are also available, when appropriate, using the functions `solve()`, `det()`, `eigen()`,`svd()`, respectively. There are many more functions related to matrices in R. We leave the reader to explore as needed.

Data frames

When we work with real data, atomic vectors and matrices may not be enough to store them if the data set contains different data types, such as numbers, characters, factors etc. R has a useful data structure, which is built upon list, called a `data frame`. A data frame can have different columns holding different data types.

We can create a data frame using the `data.frame()` function.

```
df <- data.frame(name = c("Arnab", "Ana"),
                 grade = c(80, 93),
                 is_graduate = c(FALSE, TRUE)
                 )
df
```

```
##    name grade is_graduate
## 1 Arnab    80       FALSE
## 2   Ana    93        TRUE
```

Notice that, in a data frame, **each column must have the same number of elements**.

A data frame has name for each row (accessed os set using `rownames()`), and names for each column (accessed/set by `colnames()`.

```
rownames(df)
```

```
## [1] "1" "2"
```

```
colnames(df)
```

```
## [1] "name"        "grade"       "is_graduate"
```

We can check the size of a data frame by using the `dim()` function. The functions `nrow()` and `ncol()` give us number of rows and columns, respectively.

```
dim(df)
```

```
## [1] 2 3
```

```
ncol(df)
```

```
## [1] 3
```

We can access columns of a data frame by either index (`df[,1]`) or by name (`df["name"]` or `df$name`). Row can be accessed by index (`df[1,]`). We can put mulpliple rows and columns as well.

```
df[c(1,2), c("name", "grade")]
```

```
##    name grade
## 1 Arnab    80
## 2   Ana    93
```

Control flow

Often we encounter situations where we need to perform a task if a condition is satisfied (e.g., if numeric grade is greater than 70, set letter grade to "S", otherwise set letter grade to "U"). Such operations can be done using the `if / else` statement. The basic form of `if/else` statement is:

if(condition) task_one else task_two

Here `condition` is a Boolean variable. If condition is `TRUE`, then `task_one` executes. Otherwise, `task_two` executes.

```
numeric_grade <- 85
letter_grade <- NA
if(numeric_grade > 70){
  letter_grade <- "S"
} else {
  letter_grade <- "U"
}
letter_grade
```

```
## [1] "S"
```

The condition in the if statement has to be a scalar. If we supply a vector valued condition, only the first element would be used.

Functions

Often we want to repeat a specific algorithm/set of steps multiple times. Rather than copying and pasting the same piece of code multiple times, it is recommended to write a function. Just like mathematics, a function will have a set of input arguments and a set of output. We have already seen the `print()`, `typeof()` and `t()` functions.

As an example, suppose we want to write a function that takes a vector x and returns `$sin(1 / xˆ2)$`. The following function does the job.

```
my_fun <- function(x){
  res <- sin(1/xˆ2)
  return(res)
}
```

Let us analyze the code above. The `my_fun` piece is simply what we named our function. The `function` keyword defines the function with the arguments provided in the parentheses (i.e, x). The code within `{` and `}` is the body of the function that does the the actual computation. It creates a new variable `res` that stores $sin(1/x^2)$, and then returns the value. [11]

Let's call the function for a specific value of x.

```
y <- 2
my_fun(y)
```

```
## [1] 0.247404
```

```
x <- c(1,2,3)
my_fun(x)
```

```
## [1] 0.8414710 0.2474040 0.1108826
```

Notice that the function works with a single number as its argument as well as a vector argument. This is because the code `sin(1/x^2)` works when `x` is a vector with element-wise operations.

More complicated functions are also possible and often needed. Try practicing by writing a function that takes a matrix X and a vector y, and outputs the vector $(X^T X)^{-1} X^T y$.

[11] We could have just used `return(sin(1/x^2))`.