R Basics Part II (induction week)

Dr Lina Song

Fri, Sep 30 2022

Table of contents

1 Class Objective

- Learn R data types and common functions for each data type
  - matrix, data frame, list
- Learn R programming basics

  - variables, conditional statement, loops, and user-defined functions

2 Matrices

2.1 Matrices: creating matrices

Creating matrices: matrix()

- A matrix can be created using the command `matrix()`
  - the first argument is the vector to be converted into matrix
  - the second argument is the number of rows
  - the last argument is the number of cols (optional)

```
matrix(1:9, nrow = 3, ncol = 3)
```

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

> ! Important
>
> R by default inserts elements **vertically** by **columns**

- R will fill in the matrix by order and discard the remaining elements once fully filled

```
matrix(1:9, nrow = 3, ncol = 2)
```

```
Warning in matrix(1:9, nrow = 3, ncol = 2): data length [9] is not a sub-
multiple or multiple of the number of columns [2]
```

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

- R will fill in the matrix by order and recycle to fill in the remaining elements

```
matrix(1:9, nrow = 3, ncol = 4)
```

```
Warning in matrix(1:9, nrow = 3, ncol = 4): data length [9] is not a sub-
multiple or multiple of the number of columns [4]
```

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

Creating matrices: inserting by row

However, we can ask R to insert by rows by setting the `byrow` argument.

```
matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
```

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

Creating matrices: concatenation of matrices cbind() and rbind()

We can use `cbind()` and `rbind()` to concatenate vectors and matrices into new matrices.

- `cbind()` does the column binding

```
x <- cbind(1:3, 4:6) # column bind
x
```

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

- `cbind()` can also operate on matrices.

```
cbind(x,x)
```

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

- `rbind()` does the row binding

```
rbind(7:9, 10:12) # row bind
```

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

2.2 Matrices: indexing and subsetting

Matrices have two dimensions: rows and columns. Therefore, to extract elements from a matrix, we just need to specify which row(s) and which column(s) we want.

```r
x
```

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

- Extract an element
  - 1 is specified for row index, so we will extract elements from the first row
  - 1 is specified for column index, so we will extract elements from the the second column
  - Altogether, we extract the single element in row 1, column 2.

```r
x[1,2] # the element in the 1st row, 2nd column
```

```
[1] 4
```

- If we leave blank for a dimension, we extract all elements of that dimension.
  - 1 is specified for row index, so we will extract elements from the first row
  - Nothing is specified for column index, so we will extract all elements from all columns
  - Altogether, we extract all elements in the first row

```r
x[1,] # all elements in the first row
```

```
[1] 1 4
```

Exercise

1. Extract all elements in the second column
2. Extract all elements in the first and third rows

2.3 Matrices: operations

Let's use 3 matrices `x`, `y`, and `z`:

```r
x <- matrix(1:6, nrow = 3)
y <- matrix(1:6, byrow = T, nrow = 2)
```

- Functions will be vectorized over all elements in a matrix

```r
x
```

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

```r
z<- x^2
z
```

```
     [,1] [,2]
[1,]    1   16
[2,]    4   25
[3,]    9   36
```

Matrices' operations: matrix addition and multiplication

- If the two matrices are of the same dimensions, they can do element-wise operations, including the `*`

```r
x + z   # elementwise addition
```

```
     [,1] [,2]
[1,]    2   20
[2,]    6   30
[3,]   12   42
```

```r
x * z
```

```
     [,1] [,2]
[1,]    1   64
[2,]    8  125
[3,]   27  216
```

- We can also use `%*%` to indicate matrix multiplication

```r
x%*%y # matrix multiplication
```

```
     [,1] [,2] [,3]
[1,]   17   22   27
[2,]   22   29   36
[3,]   27   36   45
```

Matrices' operations: inverse and transpose

- We use `t()` to do matrix transpose

```r
t(x) # transpose
```

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

- We use `solve()` to get the inverse of an matrix

```r
solve(t(x)%*%x) # inverse; must be on a square matrix
```

```
           [,1]       [,2]
[1,]  1.4259259 -0.5925926
[2,] -0.5925926  0.2592593
```

3 Data Frames

3.1 Data Frames: creating dataframe

Data Frames: create dataframe using data.frame()

- Data Frame is the R object that we will deal with most of the time in the MSc program. You can think of `data.frame` as a spreadsheet in excel.

```r
df <-  data.frame(id = 1:4,
  name = c("David", "Yongdong", "Anil", "Wei"),
  wage = rnorm(n=4, mean = 10^5, sd = 10^3),
  male = c(T, T, T, T)
  )
df
```

```
  id     name      wage male
1  1    David 100917.17 TRUE
2  2 Yongdong 100152.49 TRUE
3  3     Anil 100935.73 TRUE
4  4      Wei  99163.52 TRUE
```

- Data frames can also be created from external sources, e.g., from a csv file or database.

3.2 Data Frames: Basics

- Each row stands for an `observation`; each column stands for a `variable`.
- Each column should have a **unique** name.
- Each column must contain the same data type, but the different columns can store different data types.
  - compare with matrix?
- Each column must be of same length, because rows have the same length across variables.

3.3 Data Frames: check dimensions and variable types

- You can verify the size of the `data.frame` using the command `dim()`; or `nrow()` and `ncol()`

```r
dim(df)
```

```
[1] 4 4
```

```r
nrow(df)
```

```
[1] 4
```

```r
ncol(df)
```

```
[1] 4
```

- You can get the data type info using the command `str()`

```r
class(df)
```

```
[1] "data.frame"
```

```r
str(df)
```

```
'data.frame':	4 obs. of  4 variables:
 $ id  : int  1 2 3 4
 $ name: chr  "David" "Yongdong" "Anil" "Wei"
 $ wage: num  100917 100152 100936 99164
 $ male: logi  TRUE TRUE TRUE TRUE
```

- Get the variables names

```r
names(df)
```

```
[1] "id"   "name" "wage" "male"
```

3.4 Data Frames: summary

- Summarize the data frame

```r
summary(df)
```

```
       id           name                wage             male        
 Min.   :1.00   Length:4           Min.   : 99164   Mode:logical  
 1st Qu.:1.75   Class :character   1st Qu.: 99905   TRUE:4        
 Median :2.50   Mode  :character   Median :100535                 
 Mean   :2.50                      Mean   :100292                 
 3rd Qu.:3.25                      3rd Qu.:100922                 
 Max.   :4.00                      Max.   :100936                 
```

3.5 Data Frames: subsetting

Since a dataframe is essentially a matrix, all the subsetting syntax with matrices can be applied here.

```r
df$name # subset a column
```

```
[1] "David"    "Yongdong" "Anil"     "Wei"     
```

```r
df[,c(2,3)] # can also subset like a matrix
```

```
      name      wage
1    David 100917.17
2 Yongdong 100152.49
3     Anil 100935.73
4      Wei  99163.52
```

We are interesting in the cylinders and the weights of inefficient cars (lower than 15 miles per gallon).

```r
poll_cars <- mtcars[mtcars$mpg<15, c("cyl", "wt")] # remember to assign the generated dataframe to
poll_cars
```

```
                    cyl    wt
Duster 360            8 3.570
Cadillac Fleetwood    8 5.250
Lincoln Continental   8 5.424
Chrysler Imperial     8 5.345
Camaro Z28            8 3.840
```

4 Other data structures (Optional)

4.1 Arrays

- We can use `array()` to generate a high-dimensional array
- Just like vectors and matrices, arrays can include only data types of the same kind.
- A 3D array is basically a combination of matrices each laid on top of other

```r
x <- 1:4
x <- array(data = x, dim = c(2,3,2))
x
```

```
, , 1

     [,1] [,2] [,3]
[1,]    1    3    1
[2,]    2    4    2

, , 2

     [,1] [,2] [,3]
[1,]    3    1    3
[2,]    4    2    4
```

4.2 Lists

A list is an R object that can contain anything. List is pretty useful when you need to store objects for latter use.

```r
x <- 1:2
y <- c("a", "b")
L <- list( numbers = x, letters = y)
```

4.3 Lists: indexing and subsetting

There are many ways to extract a certain element from a list.

- by index
- by the name of the element
- by dollar sign $

```
L[[1]] # extract the first element
```

```
[1] 1 2
```

```
L[['numbers']] # based on element name
```

```
[1] 1 2
```

```
L$numbers # extract the element called numbers
```

```
[1] 1 2
```

After extracting the element, we can work on the element further:

```
L$numbers[1:3] > 2
```

```
[1] FALSE FALSE    NA
```

5 Programming Basics

5.1 if/else

Sometimes, you want to run your code based on different conditions. For instance, if the observation is a missing value, then use the population average to impute the missing value. This is where `if/else` kicks in.

```
if (condition == TRUE) {
  action 1
} else if (condition == TRUE ){
  action 2
} else {
  action 3
}
```

Example 1:

```r
a <- 15

if (a > 10) {
larger_than_10 <- TRUE
} else {
  larger_than_10 <- FALSE
}

larger_than_10
```

```
[1] TRUE
```

Example 2:

```r
x <- -5
if(x > 0){
  print("x is a non-negative number")
} else {
  print("x is a negative number")
}
```

```
[1] "x is a negative number"
```

5.2 Loops

As the name suggests, in a loop the program repeats a set of instructions many times, until the stopping criteria is met.

Loop is very useful for repetitive jobs.

```r
for (i in 1:10){ # i is the iterator
  # loop body: gets executed each time
  # the value of i changes with each iteration
}
```

5.3 Nested loops

We can also nest loops into other loops.

```r
x <- cbind(1:3, 4:6) # column bind
x
```

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

```
y <- cbind(7:9, 10:12) # row bind
y
```

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

```
z <- x

for (i in 1:nrow(x)) {
  for (j in 1:ncol(x)){
    z[i,j] <- x[i,j] + y[i,j]
  }
}

z
```

```
     [,1] [,2]
[1,]    8   14
[2,]   10   16
[3,]   12   18
```

5.4 Functions

A function takes the argument as input, run some specified actions, and then return the result to us.

Functions are very useful. When we would like to test different ideas, we can combine functions with loops: We can write a function which takes different parameters as input, and we can use a loop to go through all the possible combinations of parameters.

User-defined function syntax

Here is how to define a function in general:

```
function_name <- function(arg1 ,arg2 = default_value){
  # write the actions to be done with arg1 and arg2
  # you can have any number of arguments, with or without defaults
  return() # the last line is to return some value
}
```

Example:

```r
magic <- function( x, y){
  return(x^2 + y)
}

magic(1,3)
```

```
[1] 4
```

5.5 A comprehensive example

Task: write a function, which takes a vector as input, and returns the max value of the vector

```r
get_max <- function(input){
  max_value <- input[1]
  for (i in 2:length(input) ) {
    if (input[i] > max_value) {
      max <- input[i]
    }
  }

  return(max)
}

get_max(c(-1,3,2))
```

```
[1] 2
```

Exercise

Write your own version of `which.max()` function