Lecture 1: R Basics(1)

Cheng Lu

Overview

- Atomic Class
- Explicit Coercion
- Vector
- Matrix
- Arithmetic Operations
- List
- Subsetting
- "for" loop
- if-else statement
- "while" loop

Atomic Class

R has five basic or "atomic" classes of objects:

- numeric (real number). e.g. 3.1, 4.2
- integer. e.g. 3L, 4L
- character (or string). e.g. 'a', 'b'
- complex. e.g. 3+4i, 2-3i
- logical. e.g. TRUE, F

A vector only support one type of object. For example, 3.1, 4.2 or 'a', 'b', are actually **one-element vectors**.

We use some embedded R functions, such as mode(), typeof(), storage.mode() to inspect which mode the variable belongs to.

Atomic Class

The results of modes and storage modes for the different vector types are listed in the following table.

typeof	**mode**	**storage.mode**
logical	logical	logical
integer	numeric	integer
double	numeric	double
complex	complex	complex
character	character	character

Table: Vector Types

Explicit Coercion

R objects are often coerced to different types during computations. We do explicit coercion using **as.*** functions, if available.

Example (Explicit Coercion)

```
> -3:3 # create a vector [-3,-2,-1,0,1,2,3]
[1] -3 -2 -1  0  1  2  3
> x <- -3:3 # use "<-" to assign the vector to a variable x
> x # now x is in your environment, and x = [-3, -2, -1, 0, 1, 2, 3]
[1] -3 -2 -1 0 1 2 3
> typeof(x)
[1] "integer"
> as.numeric(x)
[1] -3 -2 -1 0 1 2 3
> as.logical(x)
[1] TRUE TRUE TRUE FALSE TRUE TRUE TRUE
> as.character(x)
[1] "-3" "-2" "-1" "0" "1" "2" "3"
```

Vector

Vector: the workhorse in R

- The simplest data structure in R.
- All elements of a vector must have the same mode.
- **length(x)** returns the number of elements in vector x.

Example (Creating Vectors)

```
> x <- c(4, 5, 6) # numeric
> x <- c(TRUE, FALSE) # logical
> x <- c(T, F) # logical
> x <- c('a', 'b', 'c') # character
> x <- 1:100 # integer
> x <- c(1+0i, 3+5i) # complex
```

Vector

There are some simple ways to create vectors: **c()**, **seq()**, **rep()**, and **:** (colon).

Example (Creating Vectors)

```
> x <- seq(10, 20, by = 2)
> x
[1] 10 12 14 16 18 20
> y <- rep(x = c(1, 2, 3), 2) # this "x" is different from the former x
> y
[1] 1 2 3 1 2 3
> z <- c(x, 0, y)
> z
[1] 10 12 14 16 18 20 0 1 2 3 1 2 3
> a <- 1:length(x)+1 # ':' has higher precedence than '+'
> a
[1] 2 3 4 5 6 7
```

Vector

Example (Arithmetic Operations)

```
> 2 * x + 1
[1] 21 25 29 33 37 41
> x / y
[1] 10.000000  6.000000  4.666667 16.000000  9.000000  6.666667
> x ^ y # power
[1]   10  144 2744   16  324 8000
> x %% y # x mod y
[1] 0 0 2 0 0 2
> x %/% y # integer division
[1] 10  6  4 16  9  6
```

Matrix

Technically, a matrix is just a vector with two subscripts: the number of rows and the number of columns.wrong)

Example

```
> x <- 1:9 # x = [1,2,...,9]
> matrix(x, nrow = 3, ncol = 3) # create a matrix using the vector x
[,1] [,2] [,3]
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
> matrix(x, nrow = 3, ncol = 3, byrow=T) # by row
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 4 5 6
[3,] 7 8 9
> A <- matrix(x, nrow = 3, ncol = 3, byrow=T) # assign above matrix to "A"
```

Matrix

We can also bind two vectors to generate matrices

Example (Matrix cont'd)

```
> x <- 1:3
> y <- 4:6
> A1 <- rbind(x, y)
> A1
[,1] [,2] [,3]
x 1 2 3
y 4 5 6
> A2 <- cbind(x, y)
> A2
x y
[1,] 1 4
[2,] 2 5
[3,] 3 6
```

Matrix

Consider a matrix A, **nrow(A)** and **ncol(A)** will return the number of rows and number of columns of A.

Matrix Operations

Command	Meaning
t(A)	transpose of A
det(A)	determinant of A
eigen(A)$values	eigenvalues of A
eigen(A)$vectors	eigenvectors of A
A * B	element-wise multiplication
A %*% B	matrix multiplication
solve(A, b)	solve linear equation $Ax = b$
solve(A)	inverse of A

Table: Matrix Operations

Arithmetic Operations

Other Operations for Vector or Matrix

- sum(A), summation of all elements in A
- prod(A), product of all elements in A
- max(A), maximum element in A
- min(A), minimum element in A
- exp(A), exponential of each element in A
- log(A), logarithm of each element in A
- abs(A), absolute value of each element in A

Also use '?' to access documentations, e.g. if you don't know function **cumsum()**, type **?cumsum** for help.

Arithmetic Operations

Example

```
> x <- 1:5
> sum(x)
[1] 15
> cumsum(x)
[1]  1  3  6 10 15
> cumprod(x)
[1]   1   2   6  24 120
> max(x, pi) # max{1,2,3,4,5,pi}
[1] 5
> pmax(x, pi) # [max{1,pi},max{2,pi},...,max{5,pi}]. How about pmin() ?
[1] 3.141593 3.141593 3.141593 4.000000 5.000000
> mean(x) # mean value or average
[1] 3
```

Arithmetic Operations

Let $S_0 = 100$, $K = 100$, $T_1 = 1, \sigma = 0.2, r = 0.05$, calculate

$$d_1 = \frac{\ln \frac{S_0}{K} + (r + \frac{1}{2}\sigma^2)T_1}{\sigma\sqrt{T_1}}$$

Example

```
> S0 <- 100
> K <- 100
> T1 <- 1
> sigma <- 0.2
> r <- 0.05
> d1 <- (log(S0/K) + (r+0.5*sigma^2)*T1)/(sigma*sqrt(T1))
> d1
[1] 0.35
```

Arithmetic Operations

Calculate

$$\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

when $x = 0$, and $x = [-3, -2, \ldots, 3]$

Example

```
> x <- 0
> 1/sqrt(2*pi)*exp(-x^2/2)
[1] 0.3989423
> x <- -3:3
> 1/sqrt(2*pi)*exp(-x^2/2)
[1] 0.004431848 0.053990967 0.241970725 0.398942280 0.241970725 0.053990967
[7] 0.004431848
```

Logical Operations

Example

```
> a <- pi # a = 3.1415926...
> a > 3
[1] TRUE
> x <- rep(c(1,2,3),2) # repeat [1,2,3] twice
> x
[1] 1 2 3 1 2 3
> x <= 2
[1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE
> y <- rep(3,6) # repeat 3 by 6 times
> y
[1] 3 3 3 3 3 3
> x == y
[1] FALSE FALSE  TRUE FALSE FALSE  TRUE
> x = y # same as x <- y
```

Logical Operations

Example

```
> 3 != 4 && 5 + 4 == 4 + 5 # "!=" means not equal
[1] TRUE
> 3 == 4 || 5 + 4 <= 6 + 3
[1] TRUE
> x <- c(T, F, T, F)
> y <- c(T, T, F, F)
> !x
[1] FALSE  TRUE FALSE  TRUE
> x & y
[1]  TRUE FALSE FALSE FALSE
> x | y
[1]  TRUE  TRUE  TRUE FALSE
```

List

- Lists are special type of vector that can contain elements of different types.
- Similar to a dictionary in Python, or a struct in C.
- Very important, forming the basis for data frames, object-oriented programming.

List

Example (Creating Lists)

```
> l <- list("John", 12345, "Male")
> l
[[1]]
[1] "John"
[[2]]
[1] 12345
[[3]]
[1] "Male"
```

List

Lists can have names, and you access members by '$'.

Example (List with names)

```
> l <- list(name = "John", ID = 12345, gender = "Male")
> l
$name
[1] "John"
$ID
[1] 12345
$gender
[1] "Male"
> l$name # access list member by '$'
[1] "John"
> l$ID
[1] 12345
```

Subsetting

Example

```
> x <- 1:10
> x[3] # the 3rd element of vector x
[1] 3
> x[c(3,7)] # the 3rd and 7th elements of x
[1] 3 7
> x[-2] # all the elements in x except the 2nd one
[1]  1  3  4  5  6  7  8  9 10
> x[-c(2,4,6)]
[1]  1  3  5  7  8  9 10
```

Subsetting

Example

```
> m <- matrix(x, nrow = 2, byrow = T)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
> m[2, 2] # the element in the 2nd row and 2nd column of matrix m
[1] 7
> m[2, c(1, 3)] # the elements in the 2nd row and 1st and 3rd column
[1] 6 8
> m[, -c(2,3,4)]
     [,1] [,2]
[1,]    1    5
[2,]    6   10
```

Subsetting

Example

```
> y <- c('a', 'b', 'c')
> l <- list(numbers = x, chars = y)
> l
$numbers
[1] 1 2 3 4 5 6 7 8 9 10
$chars
[1] "a" "b" "c"
> l[[1]] # subsetting of a list
[1] 1 2 3 4 5 6 7 8 9 10
> l[["numbers"]] # same as l[[1]]
[1] 1 2 3 4 5 6 7 8 9 10
```

Subsetting

Example

```
> l[[1]][3] # nested subsetting
[1] 3
> l[[c(1, 3)]] # same as l[[1]][3]
[1] 3
> l[[1]][-c(2:5)]
[1]  1  6  7  8  9 10
> l$chars[c(1,3,4)]
[1] "a" "c" NA
```

Subsetting

Example (Assign Values)

```
> a <- x[c(2:5)]
> a
[1] 2 3 4 5
> m[,4:5] <- 0
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    0    0
[2,]    6    7    8    0    0
> l$numbers[-c(2,3,4)] <- 0
> l
$numbers
 [1] 0 2 3 4 0 0 0 0 0 0
$chars
[1] "a" "b" "c"
```

for loop

For loops take an iterator variable and assign it successive values from a vector. For loops are most commonly used for iterating over the elements of an object (vector, list, etc).

Example

```
# Suppose we want to print numbers from 1 to 5
# print(1)
# print(2)
# ...
# print(5)
for (i in 1:5){
  print(i) # every iteration increases 1
}
# equivalent to
x <- 1:5
for(i in x){
  print(i)
}
```

for loop

Example

```
> # calculate 1 + 2 + 3 + ... + 100 = 5050
> # sums <- 0
> # sums <- sums + 1
> # sums <- sums + 2
> # ...
> # sums <- sums + 100
> # sums
> sums <- 0
> for (i in 1:100){
+ sums = sums + i
+ }
> sums
[1] 5050
```

Sometimes loops are used for calculating recursive formula, suppose you want to calculate the 100th term of the sequence 1, 3, 5, 7,....
Then you have the formula $S_n = S_{n-1} + 2$ and $S_1 = 1$, and you need to calculate S_{100}:

Example

```
> S <- rep(NA, 100) # initialization of a vector of NA with 100 elements
> S[1] <- 1 # first element
> for (n in 2:100) {
+   S[n] <- S[n-1] + 2 # recursive formula
+ }
> S[100] # the 100th term
[1] 199
```

Question: How about if the sequence start at the 0th term S_0 and you want to find S_{100}? But the 0th term of a vector S[0] does not exists in R. One solution: Let S[1]$\leftarrow S_0$ and find S[101].

for loop

Example

```
> A2 <- matrix(NA, nrow = 3, ncol = 4) # NA matrix
> A1 <- matrix(1:12, nrow = 3, byrow = T)
> for (i in 1:3) # i is row index
+ {
+   for (j in 1:4) # j is col index
+   {
+       A2[i, j] <- A1[i, j] * 10
+   }
+ }
> A2
     [,1] [,2] [,3] [,4]
[1,]   10   20   30   40
[2,]   50   60   70   80
[3,]   90  100  110  120
```

if-else statement

Conditional statements are features of a programming language which perform different computations or actions depending on whether a programmer-specified boolean condition evaluates to true or false.

Example

```
> x <- F
> if (x) # x needs to be a logical value
+ {
+   "x is TRUE"
+ } else {
+   "x is FALSE"
+ }
[1] "x is FALSE"
# or ...
> ifelse(x, "x is TRUE", "x is FALSE")
[1] "x is FALSE"
```

if-else statement

Another if-else example.

Example

```
x <- 1:10
if (length(x) > 15)
{
    print("x is a long array")
} else {
    print("x is short array")
}
```

if-else statement

Example

```
> # print all even numbers between 1 and 10
> # idea: test all numbers between 1 and 10 to see if they are divisible by 2
> # if(1 %% 2 == 0){
> #   print(1)
> # }
> # ...
> # if(10 %% 2 == 0){
> #   print(10)
> # }
> for (i in 1:10) {
+   if(i %% 2 == 0){ # if "i" is divisible by 2
+     print(i)
+   }
+ }
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
```

if-else statement

Example

```
> # print all even numbers between 1 and 10 which are greater than 5
> for (i in 1:10) {
+   if(i %% 2 == 0 && i > 5){ # if "i" is divisible by 2 and greater than 5
+     print(i)
+   }
+ }
[1] 6
[1] 8
[1] 10
```

while loop

Another way of doing iteration: while loop.

Example

```
i <- 1
while(TRUE){ # infinite loop, press 'stop' to stop printing
  print(i)
  i <- i + 1
}

i <- 1
while(i <= 100){ # print number from 1 to 100
  print(i)
  i <- i + 1
}
```

while loop

Example

```
# calculate 1 + 2 + ... + 100 = 5050 with while loop
# sums <- 0
# i <- 1
# sums <- sums + i
# i <- i + 1
# repeat the above 2 sentences when i <= 100
sums <- 0
i <- 1
while (i <= 100){
  sums <- sums + i
  i <- i + 1
}
sums
```