R for Business and Economics

Baris Soybilgen

Contents

This refresher shows fundamentals of R programming language. This refresher is based on Matloff's (2011) "The Art of R Programming".

Data structures in R

vector

`vector` is the basic data structure of the R programming language. We can access values (index) in vectors using `[]`.

```
x <- c(1,2,4)
x
## [1] 1 2 4
x[3]
## [1] 4
x[2:3]
## [1] 2 4
```

We can apply functions over `vector`.

```
length(x)
## [1] 3
mode(x) # Get or set the type or storage mode of an object.
## [1] "numeric"
typeof(x) # Determines the (R internal) type or storage mode of any object
## [1] "double"
class(x) # Displays the class of an object
## [1] "numeric"
```

`vector` can contain a single data type of logical, integer, double, or character.

```
y <- T
str(y)
##  logi TRUE
z <- c(T,12)
str(z)
```

```
## num [1:2] 1 12
v <- c(T,12,"abc")
str(v)
## chr [1:3] "TRUE" "12" "abc"
```

We can manipulate `vector` easily using indexing techniques.

```
x <- c(88,5,12,13)
x <- c(x[1:3],168,x[4])
str(x)
## num [1:5] 88 5 12 168 13
str(x[c(1,3)])
## num [1:2] 88 12
str(x[-1]) # minus is for deleting elements
## num [1:4] 5 12 168 13
str(x[-1:-2])
## num [1:3] 12 168 13
str(x[1:(length(x)-1)])
## num [1:4] 88 5 12 168
str(x[-length(x)])
## num [1:4] 88 5 12 168
```

We can further generate `vector` with `seq()`, `rep()`, and `vector()` functions.

```
str(seq(from = 12,to = 30,by = 3))
## num [1:7] 12 15 18 21 24 27 30
str(seq(from = 1,to = 2,length = 10))
## num [1:10] 1 1.11 1.22 1.33 1.44 ...
str(rep(NA,4))
## logi [1:4] NA NA NA NA
str(rep(c(5,12,13),3))
## num [1:9] 5 12 13 5 12 13 5 12 13
str(vector(length=2))
## logi [1:2] FALSE FALSE
```

One of the important features of `vector` in `R` is recycling. Applying an operation to two vectors requires them to be the same length, `R` automatically recycles, or repeats, the shorter one, until it is long enough to match the longer one. However, using complex recycling operations is generally a bad idea. Furthermore, `R` will show a warning when you are performing an operation on vectors, and the vectors are not of the same length.

```
c(1,2,4) + c(6,0,9,20,22)
## Warning in c(1, 2, 4) + c(6, 0, 9, 20, 22): longer object length is not a
## multiple of shorter object length
## [1]  7  2 13 21 24
```

For advanced indexing, we can use `comparison operators` and `subset()`. We can also use `which()` to get the actual index value.

```
z <- c(5,2,-3,8)
str(z[z * z > 8])
## num [1:3] 5 -3 8
z[z>3] <- 0
str(z)
## num [1:4] 0 2 -3 0
str(subset(z,z > 1))
```

```
##  num 2
which(z * z > 1)
## [1] 2 3
```

To check whether all elements in a `vector` are true according to the given argument, we use `all()`. To check whether at least one of the values in a `vector` is true according to the given argument, we use `any()`.

```
x <- 1:10
str(any(x > 8))
##  logi TRUE
str(all(x > 3))
##  logi FALSE
```

Warning: Above, we use `NA` to create a vector with missing values. The `R` documentation defines `NA` as a logical constant of length one which contains a missing value indicator. In addition to `NA`, `R` has a similar structure called `NULL` which represents the null object and is often returned by expressions and functions whose values are undefined. The important difference between `NA` and `NULL`: `NA` shows missing values and has a logical value, whereas `NULL` values are counted as nonexistent and have no logical value.

```
u <- NULL
length(u)
## [1] 0
NULL > 0
## logical(0)
v <- NA
length(v)
## [1] 1
NA > 0
## [1] NA
```

`matrix`

`matrix` is a two-dimensional data structure in R programming. `matrix` is similar to `vector` but additionally contains the dimension attribute.

```
m <- rbind(c(10,11),c(12,13))
dim(m)
## [1] 2 2
str(m)
##  num [1:2, 1:2] 10 12 11 13
str(m[2,2])
##  num 13
str(m[1,]) # row 1
##  num [1:2] 10 11
str(m[,2]) # column 2
##  num [1:2] 11 13
```

You can see the difference between `mode()` and `class()` in `matrix`.

```
mode(m) # Get or set the type or storage mode of an object.
## [1] "numeric"
typeof(m) # Determines the (R internal) type or storage mode of any object
## [1] "double"
class(m) # Displays the class of an object
## [1] "matrix" "array"
```

We can use arithmetic operations over `matrix` and `vector`. We can also use matrix multiplication.

```r
x <- c(1,2)
str(m*2)
##  num [1:2, 1:2] 20 24 22 26
str(x*2)
##  num [1:2] 2 4
str(m*x)
##  num [1:2, 1:2] 10 24 11 26
str(m*c(2,4))
##  num [1:2, 1:2] 20 48 22 52
str(m*c(2,4,6,8))
##  num [1:2, 1:2] 20 48 66 104
str(m%*%c(1,1)) #matrix multiplication
##  num [1:2, 1] 21 25
```

`matrix` can be created by `rbind()`, `cbind()`, and `matrix()`.

```r
str(matrix(c(1,2,3,4),nrow = 2,ncol = 2))
##  num [1:2, 1:2] 1 2 3 4
str(rbind(c(1,2),c(3,4)))
##  num [1:2, 1:2] 1 3 2 4
str(cbind(c(1,2),c(3,4)))
##  num [1:2, 1:2] 1 2 3 4
```

Filtering in `matrix` also conducted similar to `vector`.

```r
x <- cbind(c(1,2,3),c(2,3,4))
str(x)
##  num [1:3, 1:2] 1 2 3 2 3 4
str(x[x[,2] >= 3,])
##  num [1:2, 1:2] 2 3 3 4
z <- c(5,12,13)
str(x[z %% 2 == 1,])
##  num [1:2, 1:2] 1 3 2 4
str(which(x > 2))
##  int [1:3] 3 5 6
```

Warning: When conducting filtering and indexing on `matrix`, if one dimension is reduced to 1, `R` could convert `matrix` to `vector`. To prevent this, we use `drop=F`.

```r
x <- cbind(c(1,2,3),c(2,3,4))
str(x) # matrix
##  num [1:3, 1:2] 1 2 3 2 3 4
str(x[1,]) # vector
##  num [1:2] 1 2
str(x[1,,drop = F]) # matrix
##  num [1, 1:2] 1 2
```

In `R`, `matrix` with more than 2 dimensions is called `array`. We can create `array` using `array()`.

```r
a <- array(data = c(1,2,3,4,5,6,7,8),dim = c(2,2,2))
str(a)
##  num [1:2, 1:2, 1:2] 1 2 3 4 5 6 7 8
str(attributes(a)) # return dimensions of the array as list
## List of 1
##  $ dim: int [1:3] 2 2 2
```

`list`

`list` is the object which contains elements of different types – like strings, numbers, vectors, and another list inside it. `list` can also contain a matrix or a function as its elements. We can create a `list` using `list()`.

```
j <- list(name = c("Joe", "Mary"), salary = c(55000, 60000), union = T)
str(j)
## List of 3
##  $ name  : chr [1:2] "Joe" "Mary"
##  $ salary: num [1:2] 55000 60000
##  $ union : logi TRUE
```

We can accomplish indexing `list` using both `[]` and `$`.

```
str(j$salary)
##  num [1:2] 55000 60000
str(j[["salary"]])
##  num [1:2] 55000 60000
str(j[[2]])
##  num [1:2] 55000 60000
str(class(j[[2]]))
##  chr "numeric"
```

If we use `[]` instead `[[]]`, we would obtain another list instead of vector.

```
str(j[2])
## List of 1
##  $ salary: num [1:2] 55000 60000
str(class(j[2]))
##  chr "list"
```

As in `matrix` and `vector`, we can manipulate elements in `list` using indexing techniques.

```
l <- list(a = "abc",b = 12)
str(l)
## List of 2
##  $ a: chr "abc"
##  $ b: num 12
l$c <- "sailing"
l[[4]] <- 28
l[5:7] <- c(F,T,T)
str(l)
## List of 7
##  $ a: chr "abc"
##  $ b: num 12
##  $ c: chr "sailing"
##  $  : num 28
##  $  : logi FALSE
##  $  : logi TRUE
##  $  : logi TRUE
str(l[[4]])
##  num 28
l$b <- NULL # delete z$b
str(l)
## List of 6
##  $ a: chr "abc"
##  $ c: chr "sailing"
```

```
##  $  : num 28
##  $  : logi FALSE
##  $  : logi TRUE
##  $  : logi TRUE
str(l[[4]])
##  logi FALSE
```

You can produce a vector from `list` which contains all the atomic components using `unlist()`.

```
j <- list(name = c("Joe", "Mary"),salary = c(55000, 60000),union = T)
str(j)
## List of 3
##  $ name  : chr [1:2] "Joe" "Mary"
##  $ salary: num [1:2] 55000 60000
##  $ union : logi TRUE
v <- unlist(j)
v
##   name1   name2 salary1 salary2   union
##   "Joe"  "Mary" "55000" "60000"  "TRUE"
names(v) <- NULL # delete names in vector
v
## [1] "Joe"   "Mary"  "55000" "60000" "TRUE"
unname(unlist(j)) # both unlist and delete names
## [1] "Joe"   "Mary"  "55000" "60000" "TRUE"
```

We can also create a recursive `list`.

```
b <- list(u = 5,v = 12)
c <- list(w = 13)
a <- list(b,c)
str(a)
## List of 2
##  $ :List of 2
##   ..$ u: num 5
##   ..$ v: num 12
##  $ :List of 1
##   ..$ w: num 13
```

`data frame`

A `data frame` is a `matrix` like structure in which each column contains values of one variable, and each row contains one set of values from each column. We can create `data frame` using `data.frame()`.

```
kids <- c("Jack","Jill","Jane")
ages <- c(12,10,13)
height <- c(130,120,140)
d <- data.frame(kids,ages,height,stringsAsFactors = FALSE)
str(d)
## 'data.frame':    3 obs. of  3 variables:
##  $ kids  : chr  "Jack" "Jill" "Jane"
##  $ ages  : num  12 10 13
##  $ height: num  130 120 140
```

Filtering and indexing in `data frame` is a mixture of `list` and `matrix`.

```
str(d[[1]]) # obtain as vector
##  chr [1:3] "Jack" "Jill" "Jane"
str(d[1]) # obtain as data frame
## 'data.frame':    3 obs. of  1 variable:
##  $ kids: chr  "Jack" "Jill" "Jane"
str(d$kids)
##  chr [1:3] "Jack" "Jill" "Jane"
str(d[1:2,])
## 'data.frame':    2 obs. of  3 variables:
##  $ kids  : chr  "Jack" "Jill"
##  $ ages  : num  12 10
##  $ height: num  130 120
str(d[1:2,2]) # reduced to vector
##  num [1:2] 12 10
str(d[1:2,2,drop = F]) # keep data frame structure
## 'data.frame':    2 obs. of  1 variable:
##  $ ages: num  12 10
```

When working with data frames, you will frequently come across NA values. `na.rm = TRUE` argument and `complete.cases()` function can make our life easier when working with `NA`.

```
d[2,3] <- NA
mean(d$height) # return NA because d[3,3] is NA
## [1] NA
mean(d$height,na.rm = TRUE) # compute mean of non-NA values
## [1] 135
complete.cases(d) # return a logical vector show which rows have no missing values
## [1]  TRUE FALSE  TRUE
str(d[complete.cases(d),])
## 'data.frame':    2 obs. of  3 variables:
##  $ kids  : chr  "Jack" "Jane"
##  $ ages  : num  12 13
##  $ height: num  130 140
```

`factor`

`factor` is the data structure taht takes on a limited number of different values; such variables are often referred to as categorical variables. It is best used to represent categorical variables when conducting data analysis. Both numeric and character variables can be made into `factor`, but its levels will always be character values. `factor` is stored as a vector of integer values with a corresponding set of character values. Use `str()` to further understand this. For more information, see here.

```
x <- c(5,12,13,12)
xf <- factor(x)
xf
## [1] 5  12 13 12
## Levels: 5 12 13
str(xf)
##  Factor w/ 3 levels "5","12","13": 1 2 3 2
unclass(xf)
## [1] 1 2 3 2
## attr(,"levels")
## [1] "5"  "12" "13"
attr(xf,"levels")
```

```
## [1] "5"  "12" "13"
length(xf)
## [1] 4
```

After creating `factor`, it is not possible to add another value that is not shown at levels. You need to first add the new value to levels, then add the new value to `factor`.

```
xf[3] <- 14
## Warning in `[<-.factor`(`*tmp*`, 3, value = 14): invalid factor level, NA
## generated
xf
## [1] 5    12   <NA> 12
## Levels: 5 12 13
xff <- factor(xf,levels = c(5,12,13,14))
xff[3] <- 14
xff
## [1] 5  12 14 12
## Levels: 5 12 13 14
str(xff)
##  Factor w/ 4 levels "5","12","13",..: 1 2 4 2
```

Control Statements

`if-else`

The syntax for `if` looks like this.

```
x <- 15
if(x > 0) {
  print(paste(x, "is a positive number"))
}
## [1] "15 is a positive number"
```

We can also add `else` after `if`.

```
x <- 7
if(x > 0) {
  print(paste(x, "is a positive number"))
} else {
  print(paste(x, "is a negative number or zero"))
}
## [1] "7 is a positive number"
```

If `if-else` includes one statement each, we can also use the compact form.

```
x <- 3
y <- if(x == 0) x else x+1
if(x == 0) y <- x else y <- x+1
print(paste("x = ", x, "and y = ", y))
## [1] "x =  3 and y =  4"
```

We can also use `else if` after `if`. After `else if`, we can also use `else` or `else if` again but these are not necessary.

```
x <- 3
if (x > 0) {
  print(paste(x, "is a positive number"))
```

```r
} else if (x < 0) {
  print(paste(x, "is a negative number"))
} # Option 1
## [1] "3 is a positive number"
x <- 0
if (x > 0) {
  print(paste(x, "is a positive number"))
} else if (x < 0) {
  print(paste(x, "is a negative number"))
} else if (x == 0) {
  print(paste(x, "is zero"))
} # Option 2
## [1] "0 is zero"
if (x > 0) {
  print(paste(x, "is a positive number"))
} else if (x < 0) {
  print(paste(x, "is a negative number"))
} else {
  print(paste(x, "is zero"))
} # Option 3
## [1] "0 is zero"
```

Loops

The most frequently used loop, `for`, looks like this.

```r
for (i in 2015:2020) {
  print(paste("The year is", i))
}
## [1] "The year is 2015"
## [1] "The year is 2016"
## [1] "The year is 2017"
## [1] "The year is 2018"
## [1] "The year is 2019"
## [1] "The year is 2020"
```

We have one statement, we can also use the compact form.

```r
for (i in 2015:2020) print(paste("The year is", i))
## [1] "The year is 2015"
## [1] "The year is 2016"
## [1] "The year is 2017"
## [1] "The year is 2018"
## [1] "The year is 2019"
## [1] "The year is 2020"
```

Another loop style is `while`.

```r
i <- 2015
while (i < 2021) {
  print(i)
  i <- i+1
}
## [1] 2015
## [1] 2016
## [1] 2017
```

```
## [1] 2018
## [1] 2019
## [1] 2020
```

A typical looping sequence can be altered using the `break` or the `next` statement. A `break` statement is used inside a loop to stop the iterations and flow the control outside of the loop. A `next` statement is useful when we want to skip the current iteration of a loop without terminating it.

```
for (i in 1:10) {
  if (!i %% 2){
    next
  }
  print(i)
}
## [1] 1
## [1] 3
## [1] 5
## [1] 7
## [1] 9

for (i in 1:5) {
  if (i > 3 ){
    break
  }
  print(i)
}
## [1] 1
## [1] 2
## [1] 3
```

Functions

Up to this point, we use `R` built-in functions to do some basic operations on data structures. Instead of using `R` built in functions, we can also create our custom functions. In general, a function needs a name, arguments, and a return value. A function will return the value of the last statement executed unless a return statement is explicitly called.

```
is.prime <- function(x) {
  y <- T
  if (x==2) {
    y <- T
  } else if (x<2) {
    warning("input should be equal to or greater than 2 \n")
    y <- "input should be equal to or greater than 2"
  } else {
    for(i in 2:ceiling(x / 2)) {
      if(x %% i == 0) {
        y <- F
        break
      }
      }
    }
return(y)
}
for (i in 1:20) {
```

```r
  if (is.prime(i)==T) print(paste(i, "is a prime number"))
}
## Warning in is.prime(i): input should be equal to or greater than 2
## [1] "2 is a prime number"
## [1] "3 is a prime number"
## [1] "5 is a prime number"
## [1] "7 is a prime number"
## [1] "11 is a prime number"
## [1] "13 is a prime number"
## [1] "17 is a prime number"
## [1] "19 is a prime number"
```