SIOB 296 Introduction to Programming with R

Eric Archer (eric.archer@noaa.gov)

Week 02: January 13, 2019

indexing review, coercion, vectorization, factors, matrices and arrays, lists, data frames

Reading: The Book of R

Chapter 3 Matrices and Arrays
Chapter 5 Lists and Data Frames

Coercion

Many objects can be coerced from one class to another using `as.<class>` functions. If you have a `numeric` vector, it can be coerced to `character` or `logical`:

```
as.character(1:5)
```

```
[1] "1" "2" "3" "4" "5"
```

```
# when going from numeric to logical, 0 = FALSE, all other numbers are TRUE
as.logical(c(-1, -0.5, 0, 1, 3.5, 6))
```

```
[1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
```

Going from `character` to `numeric` or `logical`:

```
as.numeric(c("-5", "0.3", "3.14x", "hello", "a4"))
```

```
Warning: NAs introduced by coercion
```

```
[1] -5.0  0.3   NA   NA   NA
```

```
as.logical(c("hello", "T", "false", "True", "n", "1"))
```

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

Going from `logical` to `character` or `numeric`:

```
as.character(c(T, F, TRUE, FALSE))
```

```
[1] "TRUE"  "FALSE" "TRUE"  "FALSE"
```

```
as.numeric(c(T, F, TRUE, FALSE))
```

```
[1] 1 0 1 0
```

When coercing a `logical` to `numeric` `T = 1` and `F = 0`. This has some useful properties. To count the number of elements that meet a condition, we can use this feature with the `sum` function:

```
x <- sample(1:5, 100, replace = T)
x
```

```
  [1] 4 2 2 4 1 4 1 4 5 2 1 3 4 3 5 4 1 5 2 4 4 5 5 1 3 3 5 1 1 2 1 3 2 2 5 1 1
 [38] 2 3 2 2 5 2 3 4 1 5 1 3 1 4 2 2 3 4 1 5 4 1 4 5 2 2 2 2 3 1 1 2 4 3 2 1 1
 [75] 2 2 2 5 5 5 4 1 4 1 2 3 4 4 4 3 1 1 4 5 3 1 5 2 1 1
```

```
sum(x == 1)
```

```
[1] 26
```

Likewise, to calculate the proportion of things that meet a condition, we use the same trick with `mean`:

```
mean(x <= 2)
```

```
[1] 0.5
```

Vectorization

A central component of R operations is the idea of "vectorization". It is a built-in capability in R that makes doing operations on multiple elements in a vector faster and intuitive. The essence of vectorization is that operations between multiple R vectors will recycle elements in the smaller object to the size of the larger object. This is most easily seen in vector algebra.

```
# Add two vectors of equal length
1:5 + 21:25
```

```
[1] 22 24 26 28 30
```

```
# Add two vectors where one is a multiple of the other
1:10 + 1:2
```

```
 [1]  2  4  4  6  6  8  8 10 10 12
```

```
# Add two vectors where one is not the multiple of the other
1:10 + 1:3
```

```
Warning in 1:10 + 1:3: longer object length is not a multiple of shorter object
length
```

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

Here's an example of vectorization with logical indexing.

```
# Select every other element
x <- 1:10
x[c(T, F)]
```

```
[1] 1 3 5 7 9
```

```
# Select every third element
x[c(T, F, F, F)]
```

```
[1] 1 5 9
```

Whenever possible, try to find ways to take advantage of vectorization. It will be faster than doing explicit loops using `for()` or `*apply()` functions.

Factors

Factors are special vectors where the unique values are stored as numbers and mapped to character levels

```r
x <- factor(c("yellow", "blue", "green", "blue", "Blue", "yellow"))
x
```

```
[1] yellow blue   green  blue   Blue   yellow
Levels: blue Blue green yellow
```

```r
# Notice that the values are numerics
str(x)
```

```
 Factor w/ 4 levels "blue","Blue",..: 4 1 3 1 2 4
```

```r
# ... but the class isn't
is.numeric(x)
```

```
[1] FALSE
```

```r
# ... nor is it character
is.character(x)
```

```
[1] FALSE
```

```r
# Here's the class
class(x)
```

```
[1] "factor"
```

```r
# and the storage mode
mode(x)
```

```
[1] "numeric"
```

The numeric and original character vectors can be obtained by **coercion** using the `as.<class>` set of functions:

```r
as.numeric(x)
```

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

```r
as.character(x)
```

```
[1] "yellow" "blue"   "green"  "blue"   "Blue"   "yellow"
```

A factor has both `levels` and `labels`. The `levels` are the set of values that might have existed in the original vector and the `labels` are the representations of the `levels`.

```r
# The sample function takes a random sample from a vector with or without replacement
x <- sample(x = letters[1:4], size = 10, replace = TRUE)
xf <- factor(x)
xf
```

```
 [1] d b b c d c a c b d
Levels: a b c d
```

```r
# Here are the levels
levels(xf)
```

```
[1] "a" "b" "c" "d"
```

```r
# We can change the order of the levels (note doesn't change order of values in vector)
xf.lvl <- factor(x, levels = c("c", "b", "d", "a"))
xf.lvl
```

```
 [1] d b b c d c a c b d
Levels: c b d a
```

```r
# Adding a level that doesn't exist has no effect on data, but includes level in list of levels
xf.lvl <- factor(x, levels = c("c", "e", "b", "d", "a"))
xf.lvl
```

```
 [1] d b b c d c a c b d
Levels: c e b d a
```

```r
# Omitting a level causes all values with that level to be NA
xf.lvl <- factor(x, levels = c("b", "d", "a"))
xf.lvl
```

```
 [1] d    b    b    <NA> d    <NA> a    <NA> b    d
Levels: b d a
```

```r
# Labels will match order of levels
xf.lbl <- factor(x, labels = c("Z", "Y", "X", "W"))
xf.lbl
```

```
 [1] W Y Y X W X Z X Y W
Levels: Z Y X W
```

```r
# But you must have as many labels as levels
xf.lbl <- factor(x, labels = c("Z", "Y", "X"))
```

```
Error in factor(x, labels = c("Z", "Y", "X")): invalid 'labels'; length 3 should be 1 or 4
```

Matrices

Matrices are always two-dimensional objects having a certain number of rows and columns. They contain only one kind (atomic mode) of data (e.g., numeric, character, logical). They are created by supplying a vector of values to the `matrix()` function and specifying how many rows and/or how many columns to dimension it by.

```r
# Create a matrix
x <- 1:24
mat <- matrix(x, nrow = 4)
mat
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24
```

```r
# How many elements are in the matrix?
length(mat)
```

```
[1] 24
```

```r
#How many rows and columns?
nrow(mat)
```

```
[1] 4
```

```r
ncol(mat)
```

```
[1] 6
```

Cells are selected by `[row, column]`

```
mat[2, 3]
```

```
[1] 10
```

Selecting a single row or single column returns a vector

```
mat[3, ]
```

```
[1]  3  7 11 15 19 23
```

```
mat[, 4]
```

```
[1] 13 14 15 16
```

Use `drop = F` to select a single row or column and return a matrix

```
mat[4, , drop = F]
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    4    8   12   16   20   24
```

```
mat[, 2, drop = F]
```

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

Select several rows or columns

```
mat[c(1, 3, 4), ]
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    3    7   11   15   19   23
[3,]    4    8   12   16   20   24
```

```
mat[, 2:5]
```

```
     [,1] [,2] [,3] [,4]
[1,]    5    9   13   17
[2,]    6   10   14   18
[3,]    7   11   15   19
[4,]    8   12   16   20
```

Select rows, exclude columns

```
mat[1:3, -(2:4)]
```

```
     [,1] [,2] [,3]
[1,]    1   17   21
[2,]    2   18   22
[3,]    3   19   23
```

Change a value in the matrix

```
mat[2, 5] <- NA
```

Change an entire column

```
mat[, 3] <- 100:103
```

Adding a column or row

```
mat.plus.col <- cbind(mat, 100:103)
mat.plus.row <- rbind(300:305, mat)
```

Assign row and column names

```
rownames(mat) <- c("first", "second", "third", "fourth")
colnames(mat) <- letters[1:ncol(mat)]
```

Choose rows and columns by name

```
mat["first", c("e", "c", "d")]
```

```
  e   c   d
 17 100  13
```

Choose columns by logical vectors

```
mat[, c(T, T, F, F, T, F)]
```

```
       a b  e
first  1 5 17
second 2 6 NA
third  3 7 19
fourth 4 8 20
```

Transpose a matrix

```
t(mat)
```

```
  first second third fourth
a     1      2     3      4
b     5      6     7      8
c   100    101   102    103
d    13     14    15     16
e    17     NA    19     20
f    21     22    23     24
```

Add, subtract, multiply, or divide a matrix by a scalar

```
mat * 5
```

```
        a  b   c  d   e   f
first   5 25 500 65  85 105
second 10 30 505 70  NA 110
third  15 35 510 75  95 115
fourth 20 40 515 80 100 120
```

```
mat / 3
```

```
               a        b        c        d        e        f
first  0.3333333 1.666667 33.33333 4.333333 5.666667 7.000000
second 0.6666667 2.000000 33.66667 4.666667       NA 7.333333
third  1.0000000 2.333333 34.00000 5.000000 6.333333 7.666667
fourth 1.3333333 2.666667 34.33333 5.333333 6.666667 8.000000
```

```
mat ^ 2
```

```
        a  b     c   d   e   f
first   1 25 10000 169 289 441
second  4 36 10201 196  NA 484
```

```
third   9 49 10404 225 361 529
fourth 16 64 10609 256 400 576
```

Add a vector to a matrix

```
mat + 1000:1003
```

```
          a    b    c    d    e    f
first  1001 1005 1100 1013 1017 1021
second 1003 1007 1102 1015   NA 1023
third  1005 1009 1104 1017 1021 1025
fourth 1007 1011 1106 1019 1023 1027
```

If you want to recycle over columns, you need to transpose the matrix, then transpose the result:

```
prod1 <- t(mat) * c(5, 10, 50, 100, 500, 1000)
prod1
```

```
  first second third fourth
a     5     10    15     20
b    50     60    70     80
c  5000   5050  5100   5150
d  1300   1400  1500   1600
e  8500     NA  9500  10000
f 21000  22000 23000  24000
```

```
t(prod1)
```

```
        a  b    c    d     e     f
first   5 50 5000 1300  8500 21000
second 10 60 5050 1400    NA 22000
third  15 70 5100 1500  9500 23000
fourth 20 80 5150 1600 10000 24000
```

Vectorization happens when vectors and matrices are multiplied, but remember that this happens by row:

```
mat <- matrix(1:24, nrow = 4)
mat * c(5, 10, 50, 100)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    5   25   45   65   85  105
[2,]   20   60  100  140  180  220
[3,]  150  350  550  750  950 1150
[4,]  400  800 1200 1600 2000 2400
```

Row and column sums or means

```
rowSums(mat)
```

```
[1] 66 72 78 84
```

```
colMeans(mat)
```

```
[1]  2.5  6.5 10.5 14.5 18.5 22.5
```

Arrays

Arrays are multi-dimensional objects that also contain only a single atomic mode of data. They are indexed the same way as matrices, but created by specifying the number of dimensions.

```
# 1 dimensional array (= vector)
arr.vec <- array(x)
arr.vec
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

```
# 2 dimensional array (= matrix)
arr.mat <- array(x, dim = c(3, 8))
arr.mat
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    4    7   10   13   16   19   22
[2,]    2    5    8   11   14   17   20   23
[3,]    3    6    9   12   15   18   21   24
```

```
# 3 dimensional array
arr.3d <- array(x, dim = c(3, 4, 2))
arr.3d
```

```
, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

The number of dimensions of an object can be obtained with `dim()`.

```
dim(arr.mat)
```

```
[1] 3 8
```

```
dim(arr.3d)
```

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

An array or matrix can be redimensioned as well.

```
dim(arr.mat) <- c(2, 4, 3)
arr.mat
```

```
, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]    9   11   13   15
[2,]   10   12   14   16
```

```
, , 3

     [,1] [,2] [,3] [,4]
[1,]   17   19   21   23
[2,]   18   20   22   24
```

Lists

Lists are one-dimensional objects where each element can be any kind of object.

```
x <- list(1, letters[1:5], matrix(100:119, 5))
x
```

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

[[2]]
[1] "a" "b" "c" "d" "e"

[[3]]
     [,1] [,2] [,3] [,4]
[1,]  100  105  110  115
[2,]  101  106  111  116
[3,]  102  107  112  117
[4,]  103  108  113  118
[5,]  104  109  114  119
```

```
str(x)
```

```
List of 3
 $ : num 1
 $ : chr [1:5] "a" "b" "c" "d" ...
 $ : int [1:5, 1:4] 100 101 102 103 104 105 106 107 108 109 ...
```

```
class(x)
```

```
[1] "list"
```

```
mode(x)
```

```
[1] "list"
```

A useful piece of information is that lists are special vectors:

```
is.list(x)
```

```
[1] TRUE
```

```
is.vector(x)
```

```
[1] TRUE
```

If you use a single bracket ([) to index a list, you will get a list back:

```
y <- x[2]
str(y)
```

```
List of 1
 $ : chr [1:5] "a" "b" "c" "d" ...
```

```
length(y)
```

```
[1] 1
```

To get the actual object back, you have to use double brackets (`[[`):

```
z <- x[[2]]
str(z)
```

```
 chr [1:5] "a" "b" "c" "d" "e"
```

```
length(z)
```

```
[1] 5
```

List elements can have names and they can be used for indexing like vectors, but single brackets still return a list and double brackets return the object:

```
x2 <- list(first = 1, lets = letters[1:5], third = matrix(30:53, 4))
x2["first"]
```

```
$first
[1] 1
```

```
x2[["first"]]
```

```
[1] 1
```

The dollar sign (`$`) is a special operator for lists with names that returns the same thing as double brackets:

```
x2$first
```

```
[1] 1
```

List names can be changed with `names`:

```
names(x2) <- c("a.number", "some.letters", "a.matrix")
x2
```

```
$a.number
[1] 1

$some.letters
[1] "a" "b" "c" "d" "e"

$a.matrix
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   30   34   38   42   46   50
[2,]   31   35   39   43   47   51
[3,]   32   36   40   44   48   52
[4,]   33   37   41   45   49   53
```

A list can contain a list, and if you know the names, you can chain the `$`:

```
x2$new.element <- list(numbers = 1:5, matrix = matrix(11:25, 3))
x2
```

```
$a.number
[1] 1
```

```
$some.letters
[1] "a" "b" "c" "d" "e"

$a.matrix
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   30   34   38   42   46   50
[2,]   31   35   39   43   47   51
[3,]   32   36   40   44   48   52
[4,]   33   37   41   45   49   53

$new.element
$new.element$numbers
[1] 1 2 3 4 5

$new.element$matrix
     [,1] [,2] [,3] [,4] [,5]
[1,]   11   14   17   20   23
[2,]   12   15   18   21   24
[3,]   13   16   19   22   25
```

```
x2$new.element$matrix
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]   11   14   17   20   23
[2,]   12   15   18   21   24
[3,]   13   16   19   22   25
```

To remove an element from a list, you assign `NULL` to that element:

```
x2$some.letters <- NULL
x2
```

```
$a.number
[1] 1

$a.matrix
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   30   34   38   42   46   50
[2,]   31   35   39   43   47   51
[3,]   32   36   40   44   48   52
[4,]   33   37   41   45   49   53

$new.element
$new.element$numbers
[1] 1 2 3 4 5

$new.element$matrix
     [,1] [,2] [,3] [,4] [,5]
[1,]   11   14   17   20   23
[2,]   12   15   18   21   24
[3,]   13   16   19   22   25
```

Lists can be grown using the `c` function:

```
x <- list(a = 1, b = 2:6, c = letters)
z <- c(x, g = T)
z
```

```
$a
[1] 1

$b
[1] 2 3 4 5 6

$c
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"

$g
[1] TRUE
```

We use `dimnames` to add names to arrays. They have to be specified as lists:

```
arr <- array(1:24, dim = c(3, 4, 2))
dimnames(arr) <- list(letters[1:3], LETTERS[1:4], c("one", "two"))
arr
```

```
, , one

  A B C  D
a 1 4 7 10
b 2 5 8 11
c 3 6 9 12

, , two

   A  B  C  D
a 13 16 19 22
b 14 17 20 23
c 15 18 21 24
```

Data Frames

Data frames are two-dimensional objects that are normally used to represent data where the rows are observations and the columns are variables.

```
ids <- c(1213, 2435, 5367, 6745, 3592)
loc <- c("north", "north", "north", "west", "south")
len <- c(9.9, 4.5, 7.7, 3.4, 2.0)
wght <- c(270, 130, 235, 90, 88)

df <- data.frame(id = ids, location = loc, len = len, wt = wght)

str(df)
```

```
'data.frame':   5 obs. of  4 variables:
 $ id      : num  1213 2435 5367 6745 3592
 $ location: Factor w/ 3 levels "north","south",..: 1 1 1 3 2
 $ len     : num  9.9 4.5 7.7 3.4 2
 $ wt      : num  270 130 235 90 88
```

```
nrow(df)
```

```
[1] 5
```

```
ncol(df)
```

```
[1] 4
```

Data frames are actually special lists where every column is an element that is the same length:

```
is.data.frame(df)
```

```
[1] TRUE
```

```
is.list(df)
```

```
[1] TRUE
```

```
is.vector(df)
```

```
[1] FALSE
```

```
length(df)
```

```
[1] 4
```

Data frames are indexed the same way as matrices:

```
df[1, ]
```

```
    id location len  wt
1 1213    north 9.9 270
```

```
df[, "len"]
```

```
[1] 9.9 4.5 7.7 3.4 2.0
```

```
df[, c("id", "wt")]
```

```
    id  wt
1 1213 270
2 2435 130
3 5367 235
4 6745  90
5 3592  88
```

Columns can also be returned as a vector using the `$`:

```
df$wt
```

```
[1] 270 130 235  90  88
```

Data frames are often indexed by a column within the data frame itself. For instance, we want to select only the rows where length is less than 5:

```
df[df$len < 5, ]
```

```
    id location len  wt
2 2435    north 4.5 130
4 6745     west 3.4  90
5 3592    south 2.0  88
```

Notice that when we do this, we are placing the condition in the row slot of the indexing brackets. The point of this is that we are creating a logical condition as long as there are rows and using this logical vector to

index.

Here's a more complex example:

```
df[df$wt > 200 & df$len < 8, ]
```

```
    id location len  wt
3 5367    north 7.7 235
```

We can also choose which columns to return at the same time:

```
df[df$location != "north", c("id", "len", "wt")]
```

```
    id len wt
4 6745 3.4 90
5 3592 2.0 88
```

The `subset` function is a convenient way to index a data.frame without using the `$` notation:

```
subset(df, wt > 200, c("id", "location"))
```

```
    id location
1 1213    north
3 5367    north
```