matrices and arrays in R

A matrix is a vector with two additional attributes: the number of rows and the number of columns. Matrices are also vectors; they have modes (classes). However, vectors are not one-rows/column matrices.

Matrices are special cases of R type object: arrays. Unlike matrices, arrays can be multi-dimensional; having rows, columns, and layers. Matric row and column subscripts (indexes) begin with 1. The internal storage of a matrix is column-major order, meaning all of column 1 is stored initially, then all of column 2, and so on.

However, it can be specified as an argument to build a matrix with row-major order, opposed to the default:

```
> y <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)     #build 2x2 matrix
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> y <- matrix(1:4 nrow = 2)                          #only nrow or ncol needed
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> y <- matrix(nrow = 2, ncol = 2)                    #individually specifying elements
> y
> y[1, 1] <- 1
> y[2, 1] <- 2
> y[3, 1] <- 3
> y[1, 2] <- 3
> y[2, 2] <- 4
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> m <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, byrow = TRUE)   #specify byrow argument
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Common Linear Algebra operations can be performed on matrices, such as matrix multiplication, matrix scalar multiplication, matrix addition, matrix subtraction, etc.

```
> y %*% y                #mathematical matrix multiplication
     [,1] [,2]
[1,]    7   15
[2,]   10   22

> 3*y                    #mathematical multiplication of matrix by scalar
     [,1] [,2]
[1,]    3    9
[2,]    6   12

> y+y                    #mathematical matrix addition
     [,1] [,2]
[1,]    2    6
[2,]    4    8
```

matrix subsetting [ ] in R

Matrices can be subset like vectors, with the addition of column arguments.

```
> z <- matrix(c(1,2,3,4,1,1,0,0,1,0,1,0), ncol = 3)     #build 3x4 matrix z
> z
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    1    0
[3,]    3    0    1
[4,]    4    0    0

> z[,2:3]                                              #subset columns 2-3 of z
     [,1] [,2]
[1,]    1    1
[2,]    1    0
[3,]    0    1
[4,]    0    0

> z[1:2,1]                                         #subset rows 1-2 and column 1 of z
[1] 1 2

> z[,-1]                                        #subset the complement of z, column 1
     [,1] [,2]
[1,]    1    1
[2,]    1    0
[3,]    0    1
[4,]    0    0

> z[1,] <- matrix(c(7,7,7), nrow = 3)                  #assign 1st row with vector
> z
     [,1] [,2] [,3]
[1,]    7    7    7
[2,]    2    1    0
[3,]    1    0    1
[4,]    4    0    0

> x <- matrix(nrow = 5, ncol = 4)    #assign matrix z to rows 2-5 and columns 2-4 of x
> x[2:5,2:4] <- y
> x
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA    7    7    7
[3,]   NA    2    1    0
[4,]   NA    1    0    1
[5,]   NA    4    0    0
```

matrix filtering in R

Matrices can be filtered similar to vectors; however, it is important to remain mindful of the applied syntax.

```
> x <- matrix(c(1,2,2,3,3,4), nrow = 3, byrow = TRUE)
> x
     [,1] [,2]
[1,]    1    2
[2,]    2    3
[3,]    3    4

> x[x[,2] >= 3,]
     [,1] [,2]
[1,]    2    3
[2,]    3    4

> j <- x[,2] >= 3
> j
[1] FALSE  TRUE  TRUE

> x[j,]
     [,1] [,2]
[1,]    2    3
[2,]    3    4
```

The assignment of variable **j** to filter matrix **x** is a vectorized operation; all of the below hold true:

… The object x[, 2] is a vector
… The operator >= compares two vectors
… The number 3 was recycled to a vector of 3's

Similarly, a variable can be used outside of the variable that the filtering is applied:

```
> m <- matrix(1:6, nrow = 3)    #build a 3x2 matrix m
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> m[m[,1] >1 & m[,2] > 5,]    #filter matrix m to the row(s) where the first column
[1] 3 6                        values > 1 and the second column values > 5
```

Note that the above operation should have returned a 1x2 matrix, as opposed to a two-element vector. This is caused by the incorrect data type being assigned. The solution to avoiding this outcome is to apply the drop argument in the function; this will tell R to retain the two-dimensional nature of the data.

Additionally, since matrices are vectors, vector operations can be applied appropriately:

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

> which(m > 2)          #returns a 4-element vector of all matrix elements of m > 2
[1] 3 4 5 6
```

applying functions to matrix rows and columns $f(x)$ in R

The apply( ) functions are among the most used functions in R; apply( ), tapply( ), lapply( ). Each instructs R to call user-defined functions against each row or each column in a matrix.

A generalized formation of the apply( ) function:

```
> apply(X, margin, FUN, ...)
```

The arguments in the above apply( ) function form are as follows:

… **X** is the matrix
… **margin** is the dimension; **1** applies the function to **rows**, **2** applies the function to **columns**
… **FUN** is the applied function
… **...** are optional arguments to be supplied to function **f**

Below applies the mean( ) function matrix **z** as an example:

```
> z <- matrix(1:6, nrow = 3)     #build 3x2 matrix z
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> apply(z,2,mean)         #apply the mean function to matrix z by columns
[1] 2 5

> colMeans(z, 2L)         #apply the colMeans function to matrix z by columns
[1] 2 5
```

Note that the colMeans( ) function performs the same operation and could be used instead; however, the above example is for illustrative purposes of the apply( ) function.

```
> f <- function(x) x/c(2,8)     #build function f that divides the elements
> y <- apply(z,1,f)              of x by the two-dimensional vector(2, 8)
> y
     [,1]  [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75

> t(y)                           #transpose matrix y to reflect the
     [,1]  [,2]                   dimensions of the original matrix z
[1,]  0.5 0.500
[2,]  1.0 0.625
[3,]  1.5 0.750
```

adding and deleting matrix rows and columns in R

Matrices are fixed in terms of rows and columns; thus, the latter cannot be technically *deleted*. Instead, matrices can be ***assigned*** and ***reassigned*** resulting in the same net effect.

Revisiting the reassignment of **vectors** to change size:

```
> x <- c(12, 5, 13, 16, 8)                          #build five-element vector
> x
[1] 12  5 13 16  8

> x <- c(x, 20)                                     #append value 20
> x
[1] 12  5 13 16  8 20

> x <- c(x[1:3], 20, x[4:6])                        #insert value 20
> x
[1] 12  5 13 20 16  8 20

> x <- x[-2:-4]                                     #delete elements 2 through 4
> x
[1] 12 16  8 20
```

Note: reassignment occurs often unseen. For example, even the assignment **x[2] <- 12** is a reassignment.

Analogous operations can be used to change the size of a **matrix**:

```
> one                                                 #build four-element vector “one”
[1] 1 1 1 1

> z <- matrix(c(1:4,1,1,0,0,1,0,1,0), nrow = 4)       #build 4x3 matrix z
> z
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    1    0
[3,]    3    0    1
[4,]    4    0    0

> cbind(one,z)                                        #bind vector “one” to matrix z
     one
[1,]   1 1 1 1
[2,]   1 2 1 0
[3,]   1 3 0 1
[4,]   1 4 0 0

> cbind(1,z)                                   #bind values 1 to matrix z by recycling
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    1    2    1    0
[3,]    1    3    0    1
[4,]    1    4    0    0
```

```
> m <- matrix(1:6, nrow = 3)            #build 3x2 matrix m
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> m <- m[c(1,3),]                       #delete row 2 through reassignment
> m
     [,1] [,2]
[1,]    1    4
[2,]    3    6
```

matrix and vector distinction in R

Considering that matrices are also vectors:

```
z <- matrix(1:8, nrow = 4)              #build 4x2 matrix z
> z
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

As a vector, the length of matrix z can be queried:

```
> length(z)
[1] 8
```

However, a matrix is in its own class as a “matrix” object (an S3 class).

```
> class(z)                                #print the class of matrix z
[1] "matrix"

> attributes(z)                           #print the attributes of matrix z
$dim
[1] 4 2

> dim(z)                                  #the dimensions can also be printed
[1] 4 2

> nrow(z)                                 #print the number of matrix z rows
[1] 4

> ncol(z)                                 #print the number of matrix z columns

[1] 2
```

Lastly, the objects themselves can be printed by calling their names:

```
> nrow
function (x)
dim(x)[1L]
<bytecode: 0x00000000095e36c8>
<environment: namespace:base>
```

The benefits of object-oriented programming are shown with functions whose arguments are a matrix, or matrices. The number of rows/columns are available rather than needing to be supplied in arguments.

avoiding dimension reduction in R

There are many cases in statistics where dimension reduction is the goal. However, the latter is only the case when the result of losing dimensions is the *intent*.

```
> z                         #build 4x2 matrix z
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8

> r <- z[2,]         #subset matrix z for row 2; the result is a vector, not a matrix
> r
[1] 2 6
```

Noting that the result of the above operation is a vector, the original matrix class is lost. Concretely, the result is a two-element vector, as opposed to a 1x2 matrix. The latter is proven as follows:

```
> attributes(z)                    #matrix z is a 4x2 matrix
$dim
[1] 4 2

> attributes(r)                    #vector r has no discernable attributes
NULL

> str(z)                           #matrix z has 2 indices for rows and columns
 int [1:4, 1:2] 1 2 3 4 5 6 7 8

> str(r)                           #vector r has a single index of range 1:2
 int [1:2] 2 6
```

The problems caused by unintentional dimension reductions within R code can result in general purpose code performing as expected in normal conditions; but failing in special cases. For example, assume a submatrix is extracted from a given matrix, followed by matrix operations on the submatrix. If the submatrix only has a single row, R coerces to a vector and fails in the following matrix operations.

Unintentional dimensionality reduction can be avoided through the use of the **drop** argument.

```
> r <- z[2,, drop = FALSE]       #maintain matrix integrity withy the drop() argument
> r
     [,1] [,2]
[1,]    2    6

> dim(r)                        #the dimensions of matrix r now consists of 2 columns
[1] 1 2
```

The consideration of **drop** as an argument is due to **“[“** actually serving as a function, like “+” noted prior.

```
> z[3,2]                       #subset matrix z to return row 3 and column 2
[1] 7

> "[" (z,3,2)      #subset matrix with the “[“ function z to return row 3 and column 2
[1] 7
```

Additionally, the **as.matrix( )** function can be used to treat an existing vector as a matrix.

```
> u <- c(1,2,3)            #construct three-element vector u
> u
[1] 1 2 3

> attributes(u)            #vector u contains no discernible attributes, as expected
NULL

> v <- as.matrix(u)        #construct matrix v by treating vector u as a matrix
> v
     [,1]
[1,]    1
[2,]    2
[3,]    3

> attributes(v)            #matrix v contains dimensions of 3 rows and 1 column
$dim
[1] 3 1
```

naming matrix rows and columns in R

Rows and columns in matrices are often referred to by their corresponding numbers. However, matrix rows and numbers can be assigned names with the **rownames( )** and **colnames( )** functions as follows:

```
> z                              #print 2x2 matrix z
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> colnames(z)                    #matrix z has no names assigned to columns
NULL

> colnames(z) <- c("a", "b")     #assign names to columns of matrix z
> z
     a b
[1,] 1 3
[2,] 2 4

> colnames(z)                    #matrix z has 2 columns names “a” and “b”
[1] "a" "b"

> z[,"a"]                        #column names can be used as references
[1] 1 2
```

higher-dimensional arrays in R

A typical matrix in R consists of rows and corresponding observations; a two-dimensional data structure. Assume the hypothetical where the same observations occur at different time frames amongst the sample (students taking tests in multiple periods during a school term). Time becomes the third dimension in R and these datasets are called ***arrays***. The following example illustrates a series of two tests students take during a given period:

```
> firsttest <- matrix(c(46,21,50,30,25,48), ncol = 2)      #construct fist test matrix
> secondtest <- matrix(c(46,41,50,43,35,49), nrow = 3)     #construct fist test matrix

> firsttest
     [,1] [,2]
[1,]   46   30
[2,]   21   25
[3,]   50   48

> secondtest
     [,1] [,2]
[1,]   46   43
[2,]   41   35
[3,]   50   49
```

The **array( )** function is defined above to construct a two-layer array, each consisting of three rows and two columns. The number of layers are assigned with the second **'2'** in the **dim = c(3,2,2)** argument.

```
> attributes(tests)                #the attributes of the tests array are printed
$dim
[1] 3 2 2

> tests[3,2,1]                 #the score on the second portion of test 1 for student 3
[1] 48

>                              #merge firsttest and secondtest matrices into an array
>
> tests <- array(c(firsttest, secondtest), dim = c(3,2,2))
> tests
, , 1

     [,1] [,2]
[1,]   46   30
[2,]   21   25
[3,]   50   48

, , 2

     [,1] [,2]
[1,]   46   43
[2,]   41   35
[3,]   50   49
```

In additional to building a three-dimensional array by combining the two matrices above, four-dimensional matrices can be constructed by combining two, or more, three-dimensional arrays, and so on.

A common use of arrays is applied to calculating tables, discussed later.