Matrices and Arrays

Bok, Jong Soon
javaexpert@nate.com
www.javaexpert.info

Matrices

- Are a vector with two additional attributes : the number of rows and the number of columns.
- Since matrices are vectors, they also have modes, such as numeric and character.

Arrays

- Can be multidimensional.
- For example, a three-dimensional array would consist of rows, columns, and layers, not just rows and columns as in the matrix case.

Creating Matrices

- `matrix(data, nrow, ncol, byrow, dimnames)`
  - `data` is the input vector which becomes the data elements of the matrix.
  - `nrow` is the number of rows to be created.
  - `ncol` is the number of columns to be created.
  - `byrow` is a logical clue. If `TRUE` then the input vector elements are arranged by row.
  - `dimname` is the names assigned to the rows and columns.

Creating Matrices (Cont.)

- Matrix row and column subscripts begin with `1`.
- For example, the upper-left corner of the matrix `a` is denoted `a[1,1]`.
- The internal storage of a matrix is in *column-major order*, meaning that first all of column 1 is stored, then all of column 2, and so on.

Creating Matrices (Cont.)

- One way to create a matrix is by using the **matrix()** function:

Console ~/

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

Creating Matrices (Cont.)

- When print out **y**, R shows its notation for rows and columns.

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

Creating Matrices (Cont.)

- Another way to build **y** is to specify elements individually:

```
Console ~/
>
> y <- matrix(nrow=2, ncol=2)
> y[1,1] <- 1
> y[2,1] <- 2
> y[1,2] <- 3
> y[2,2] <- 4
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4
>
```

Creating Matrices (Cont.)

- Though internal storage of a matrix is in *column-major* order, you can set the **`byrow`** argument in **`matrix()`** to **`true`** to indicate that the data is coming in *row-major* order.

```
Console ~/
> m <- matrix(c(1, 2, 3, 4, 5, 6), nrow=2, byrow=TRUE)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
>
```

General Matrix Operations

- **Performing Linear Algebra Operations on Matrices.**

```
Console ~/
> # Mathematical matrix multiplication
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> y %*% y
     [,1] [,2]
[1,]    7   15
[2,]   10   22
>
> # Mathematical multiplication of matrix by scalar
> 3 * y
     [,1] [,2]
[1,]    3    9
[2,]    6   12
>
> # Mathematical matrix addition
> y + y
     [,1] [,2]
[1,]    2    6
[2,]    4    8
```

Matrix Indexing

```
Console ~/
> z <- matrix(c(1, 2, 3, 4, 1, 1, 0, 0, 1, 0, 1, 0), nrow=4)
> z
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    1    0
[3,]    3    0    1
[4,]    4    0    0
>
> z[, 2:3]
     [,1] [,2]
[1,]    1    1
[2,]    1    0
[3,]    0    1
[4,]    0    0
>
```

Matrix Indexing (Cont.)

Console ~/

```
> y <- matrix(c(11, 12, 21, 22, 31, 32), nrow=3, byrow = T)
> y
     [,1] [,2]
[1,]   11   12
[2,]   21   22
[3,]   31   32
>
> y[2:3, ]
     [,1] [,2]
[1,]   21   22
[2,]   31   32
>
> y[2:3, 2]
[1] 22 32
>
```

Matrix Indexing (Cont.)

```
Console ~/
> y <- matrix(c(1, 2, 3, 4, 5, 6), nrow=3)
> y
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
>
> y[c(1, 3), ] <- matrix(c(1, 1, 8, 12), nrow=2)
> y
     [,1] [,2]
[1,]    1    8
[2,]    2    5
[3,]    1   12
>
```

Matrix Indexing (Cont.)

Console ~/

```
> x <- matrix(nrow=3, ncol=3)
> y <- matrix(c(4, 5, 2, 3), nrow=2)
> y
     [,1] [,2]
[1,]    4    2
[2,]    5    3
>
> x[2:3, 2:3] <- y
> x
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA    4    2
[3,]   NA    5    3
>
```

Matrix Indexing (Cont.)

- Negative subscripts, used with vectors to exclude certain elements, work the same way with matrices:

```
Console ~/

> y <- matrix(c(1, 2, 3, 4, 5, 6), nrow=3)
> y
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
>
> y[-2, ]
     [,1] [,2]
[1,]    1    4
[2,]    3    6
>
```

Filtering on Matrices

- Filtering can be done with matrices, just as with vectors.
- You must be careful with the syntax, though.

Console C:/R Home/

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

Console C:/R Home/

```
>
> j <- x[,2] >= 3
> j
[1] FALSE  TRUE  TRUE
>
> x[j,]
     [,1] [,2]
[1,]    2    3
[2,]    3    4
```

Filtering on Matrices (Cont.)

- **Look!**

```
Console C:/R Home/
> x <- matrix(c(1, 2, 3, 2, 3, 4), nrow=3)
> x
     [,1] [,2]
[1,]    1    2
[2,]    2    3
[3,]    3    4
>
> x[x[, 2] >= 3,]
     [,1] [,2]
[1,]    2    3
[2,]    3    4
```

- The object **`x[,2]`** is a vector.
- The operator **`>=`** compares two vectors.
- The number **3** was recycled to a vector of 3s.

Filtering on Matrices (Cont.)

- **Look!**

```
Console C:/R Home/
> x
     [,1] [,2]
[1,]    1    2
[2,]    2    3
[3,]    3    4
>
> z <- c(5, 12, 13)
> x[z %% 2 == 1, ]
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

- The expression `z %% 2 == 1` tests each element of z for being an odd number, thus yielding (`TRUE`, `FALSE`, `TRUE`).
- As a result, we extracted the first and third rows of `x`.

Filtering on Matrices (Cont.)

- Look!

Console C:/R Home/

```
> m <- matrix(c(1,2,3,4,5,6), nrow=3)
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
>
> m[m[,1] > 1 & m[,2] > 5, ]
[1] 3 6
```

- First, the expression **`m[,1] > 1`** compares each element of the first column of m to 1 and returns (**FALSE**,**TRUE**,**TRUE**).
- The second expression, **`m[,2] > 5`**, similarly returns (**FALSE**,**FALSE**,**TRUE**).
- We then take the logical **AND** of (**FALSE**,**TRUE**,**TRUE**) and (**FALSE**,**FALSE**,**TRUE**), yielding (**FALSE**,**FALSE**,**TRUE**).

Filtering on Matrices (Cont.)

- **Look!**

Console C:/R Home/

```
> m <- matrix(c(5, 2, 9, -1, 10, 11), nrow=3)
> m
     [,1] [,2]
[1,]    5   -1
[2,]    2   10
[3,]    9   11
>
> which(m > 2)
[1] 1 3 5 6
```

Applying Functions to Matrix Rows and Columns

- One of the most famous and most used features of R is the `*apply()` family of functions.
- That is, `apply()`, `tapply()`, and `lapply()`.
- Here, we'll look at `apply()`, which instructs R to call a user-specified function on each of the rows or each of the columns of a matrix.

Using the apply() Function

- **Is the general form of apply for matrices:**

```
apply(m,dimcode,f,fargs)
```

- **The arguments:**
  - `m` is the matrix.
  - `dimcode` is the dimension, equal to 1 if the function applies to rows or 2 for columns.
  - `f` is the function to be applied.
  - `fargs` is an optional set of arguments to be supplied to `f`.

Using the apply() Function (Cont.)

Console C:/R Home/

```
>
> z <- matrix(c(1, 2, 3, 4, 5, 6), nrow=3)
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
>
> apply(z, 2, mean)
[1] 2 5
```

Using the apply() Function (Cont.)

- **Look!**

```
Console C:/R Home/
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
>
> f <- function(x) x/c(2,8)
> y <- apply(z, 1, f)
> y
     [,1]  [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75
```

- `f()` function divides a two-element vector by the vector `(2,8)`.
- Recycling would be used if `x` had a length longer than `2`.
- The call to `apply()` asks R to call `f()` on each of the rows of `z`.
- The first such row is `(1,4)`, so in the call to `f()`, the actual argument corresponding to the formal argument `x` is `(1,4)`.
- Thus, R computes the value of `(1,4)/(2,8)`, which in R's element-wise vector arithmetic is `(0.5,0.5)`.

Using the apply() Function (Cont.)

- If the function to be applied returns a vector of **k** components, then the result of `apply()` will have **k** rows.
- Can use the matrix transpose function `t()` to change it if necessary.

Console C:/R Home/

```
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
>
> f
function(x) x/c(2,8)
<bytecode: 0x00000000037ae178>
>
> t(apply(z, 1, f))
     [,1]  [,2]
[1,]  0.5 0.500
[2,]  1.0 0.625
[3,]  1.5 0.750
```

Using the apply() Function (Cont.)

Console C:/R Home/

```
> copymaj <- function(rw, d){
+     maj <- sum(rw[1:d]) / d
+     return(if(maj > 0.5) 1 else 0)
+ }
>
> x <- matrix(c(1,1,1,0,0,1,0,1,1,1,0,1,1,1,1,1,0,0,1,0), nrow=4)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    1    1    0
[2,]    1    1    1    1    0
[3,]    1    0    0    1    1
[4,]    0    1    1    1    0
>
> apply(x, 1, copymaj, 3)
[1] 1 1 0 1
>
> apply(x, 1, copymaj, 2)
[1] 0 1 0 0
```

Adding and Deleting Matrix Rows and Columns

- Technically, matrices are of fixed length and dimensions, so we cannot add or delete rows or columns.
- However, matrices can be reassigned.
- Can achieve the same effect as if we had directly done additions or deletions.

Changing the Size of a Matrix

- **Look!**

Console C:/R Home/

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

- `x` is originally of length 5, which we extend to 6 via concatenation and then reassignment.
- We didn't literally change the length of `x` but instead created a new vector from `x` and then assigned `x` to that new vector.

Changing the Size of a Matrix (Cont.)

- The **`rbind()`** (*row bind*) and **`cbind()`** (*column bind*) functions let you add rows or columns to a matrix.

Console C:/R Home/

```
> one <- c(1, 1, 1, 1)
> one
[1] 1 1 1 1
>
> z <- matrix(c(1,2,3,4,1,1,0,0,1,0,1,0), nrow=4)
> z
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    1    0
[3,]    3    0    1
[4,]    4    0    0
>
> cbind(one, z)
     one
[1,]   1 1 1 1
[2,]   1 2 1 0
[3,]   1 3 0 1
[4,]   1 4 0 0
```

Changing the Size of a Matrix (Cont.)

- **cbind()** creates a new matrix by combining a column of 1s with the columns of **z**.

Console C:/R Home/

```
> one
[1] 1 1 1 1
>
> z
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    1    0
[3,]    3    0    1
[4,]    4    0    0
>
> cbind(1, z)
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    1    2    1    0
[3,]    1    3    0    1
[4,]    1    4    0    0
```

Changing the Size of a Matrix (Cont.)

- Can also use the **`rbind()`** and **`cbind()`** functions as a quick way to create small matrices.

Console C:/R Home/

```
> q <- cbind(c(1,2), c(3, 4))
> q
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Changing the Size of a Matrix (Cont.)

- Can delete rows or columns by reassignment.

Console C:/R Home/

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

More on the Vector/Matrix Distinction

Console C:/R Home/

```
> z <- matrix(1:8, nrow=4)
> z
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
>
> length(z)
[1] 8
>
> class(z)
[1] "matrix"
>
> attributes(z)
$dim
[1] 4 2
```

- As `z` is still a vector, we can query its `length`.
- But as a matrix, `z` is a bit more than a vector.
- Most of R consists of `S3` classes.
- S3 components are denoted by dollar signs (`$`).
- The matrix class has one attribute, named `dim`, which is a vector containing the numbers of rows and columns in the matrix.

More on the Vector/Matrix Distinction (Cont.)

- You can also obtain dim via the **dim()** function.

```
> dim(z)
[1] 4 2
```

- The numbers of rows and columns are obtainable individually via the **nrow()** and **ncol()** functions.

```
> nrow(z)
[1] 4
>
> ncol(z)
[1] 2
```

Avoiding Unintended Dimension Reduction

- In the world of statistics, dimension reduction is a good thing, with many statistical procedures aimed to do it well.
- If we are working with, say, 10 variables and can reduce that number to 3 that still capture the essence of our data, we're happy.
- However, in R, something else might merit the name *dimension reduction* that we may sometimes wish to avoid.

Avoiding Unintended Dimension Reduction

Console C:/R Home/

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

- `r`, it's a vector format, not a matrix format.
- In other words, `r` is a vector of length 2, rather than a `1-by-2` matrix.

Avoiding Unintended Dimension Reduction (Cont.)

```
> z
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
>
> attributes(z)
$dim
[1] 4 2

> r
[1] 2 6
>
> attributes(r)
NULL
>
> str(z)
 int [1:4, 1:2] 1 2 3 4 5 6 7 8
>
> str(r)
 int [1:2] 2 6
```

- R informs us that `z` has row and column numbers, while `r` does not.
- Similarly, `str()` tells us that `z` has indices ranging in `1:4` and `1:2`, for rows and columns, while `r`'s indices simply range in `1:2`.
- No doubt about it—`r` is a vector, not a matrix.

Avoiding Unintended Dimension Reduction (Cont.)

Console C:/R Home/

```
> z
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
>
> r <- z[2,, drop=FALSE]
> r
     [,1] [,2]
[1,]    2    6
>
> dim(r)
[1] 1 2
```

- Fortunately, R has a way to suppress this dimension reduction.
- The `drop` argument.
- Here's an example, using the matrix `z`.
- Now `r` is a `1-by-2` matrix, not a two-element vector.

Avoiding Unintended Dimension Reduction (Cont.)

Console C:/R Home/

```
>
> z
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
>
> z[3,2]
[1] 7
>
> "["(z, 3, 2)
[1] 7
>
```

- [is actually a function, just as is the case for operators like +.

Avoiding Unintended Dimension Reduction (Cont.)

Console C:/R Home/

```
> u <- c(1,2,3)
>
> u
[1] 1 2 3
>
> v <- as.matrix(u)
>
> attributes(u)
NULL
>
> attributes(v)
$dim
[1] 3 1
```

- If you have a vector that you wish to be treated as a matrix, you can use the `as.matrix()` function.

Naming Matrix Rows and Columns

```
Console C:/R Home/
> z <- matrix(1:4, nrow=2)
> z
     [,1] [,2]
[1,]    1    3
[2,]    2    4
>
> colnames(z)
NULL
>
> colnames(z) <- c("a", "b")
>
> z
     a b
[1,] 1 3
[2,] 2 4
>
> colnames(z)
[1] "a" "b"
>
> z[,"a"]
[1] 1 2
```

- The natural way to refer to rows and columns in a matrix is via the row and column numbers.
- However, you can also give names to these entities.
- These names can then be used to reference specific columns.
- The function `rownames()` works similarly.

Higher-Dimensional Arrays

- The matrix is then a two-dimensional data structure.
- But suppose also have data taken at different times, one data point per person per variable per time.
- Time then becomes the third dimension, in addition to rows and columns.
- In R, such data sets are called *arrays*.

Higher-Dimensional Arrays (Cont.)

- Here's the data for the first test.

Console C:/R Home/

```
> firsttest <- matrix(c(46, 21, 50, 30, 25, 50), nrow=3)
> firsttest
     [,1] [,2]
[1,]   46   30
[2,]   21   25
[3,]   50   50
>
```

- Student 1 had scores of 46 and 30 on the first test, student 2 scored 21 and 25, and so on.

Higher-Dimensional Arrays (Cont.)

- **Here's the data for the second test.**

```
> secondtest <- matrix(c(46, 41, 50, 43, 35, 50), nrow=3)
> secondtest
     [,1] [,2]
[1,]   46   43
[2,]   41   35
[3,]   50   50
>
```

Higher-Dimensional Arrays (Cont.)

- Now let's put both tests into one data structure, which we'll name `tests`.
- We'll arrange it to have two *layers*—one layer per test—with three rows and two columns within each layer.
- We'll store `firsttest` in the first layer and `secondtest` in the second.
- In layer 1, there will be three rows for the three students' scores on the first test, with two columns per row for the two portions of a test.
- We use R's array function to create the data structure:

```
> tests <- array(data=c(firsttest,secondtest),dim=c(3,2,2))
```

Higher-Dimensional Arrays (Cont.)

- In the argument `dim=c(3,2,2)`, we are specifying two layers (this is the second 2), each consisting of three rows and two columns.

Console C:/R Home/

```
> firsttest
     [,1] [,2]
[1,]   46   30
[2,]   21   25
[3,]   50   50
>
> secondtest
     [,1] [,2]
[1,]   46   43
[2,]   41   35
[3,]   50   50
>
> tests <- array(data=c(firsttest,secondtest),dim=c(3,2,2))
>
> attributes(tests)
$dim
[1] 3 2 2
```

Higher-Dimensional Arrays (Cont.)

Console C:/R Home/

```
>
> tests
, , 1

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

, , 2

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

>
> tests[3,2,1]
[1] 50
```

- Each element of **tests** now has three subscripts, rather than two as in the matrix case.
- The first subscript corresponds to the first element in the **$dim** vector, the second subscript corresponds to the second element in the vector, and so on.