MTH 2520 R Notes 5

3 Higher Dimensional Arrays

3.1 Creating and Examining Arrays

- An ***array*** is like a matrix, but it can have more than two dimensions (e.g. it can have rows, columns, and *layers*).
- Here are some useful functions for creating and examining arrays:

```
array()          # Create an array, from a vector, with dimensions
                 # specified by the argument dim
dim()            # Returns the dimensions of an array
is.array()       # Indicates whether or not an object is an array
```

- Here's an array with three dimensions (rows, columns, and layers). It has 3 rows, 3 columns, and 2 layers:

```
> x <- array(1:18, dim = c(3, 3, 2))
> x

, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18

> dim(x)

[1] 3 3 2
```

```
> is.array(x)

[1] TRUE
```

- Note the order in which R fills the array with the elements of the vector passed to `array()`.

Section 3.1 Exercises

Exercise 1 Here's an array `y`:

```
> y


, , 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
```

a) What will be the result of:

```
> dim(y)
```

b) Write a command that creates the array `y` (and use it to check your answer to part *a*).

Exercise 2 *Matrices* are actually *arrays* with only two dimensions. Guess what the outputs of `is.matrix()` and `is.array()` will be, then check your answers:

```
> y <- matrix(1:9, nrow = 3, ncol = 3)
> is.matrix(y)
> is.array(y)
```

3.2 Naming Array Dimensions

- We can view or change the names of the dimensions (rows, columns, layers, etc.) of an array using:

```
dimnames()        # View the names of an array's dimensions or assign
                  # names via a list of character vectors
```

- Here's an example (using the array `x` from Section 3.1):

```
> dimnames(x)

NULL
```

We see that currently `x` doesn't have any dimension names.

We can assign dimension names to `x` using the following:

```
> dimnames(x) <- list(Row = c("R1", "R2", "R3"),
+                     Col = c("C1", "C2", "C3"),
+                     Layer = c("L1", "L2"))
> x

, , Layer = L1

    Col
Row  C1 C2 C3
  R1  1  4  7
  R2  2  5  8
  R3  3  6  9

, , Layer = L2

    Col
Row  C1 C2 C3
  R1 10 13 16
  R2 11 14 17
  R3 12 15 18
```

Note that `dimnames()` requires that the array's dimension names be in the form of a *list* (as seen on the right side of the assignment operator `<-` above).

- As a more realistic example, in the array `Temps` below, the three dimensions correspond to longitude, latitude, and time. It contains temperatures (°K) recorded in the northern hemisphere at four longitudes (represented by rows), three latitudes (columns), and three months (layers):

```
> Temps

, , Time = Jan

     Lat
```

```
Long   75  45  15
  0   273 284 305
  90  239 260 299
  180 251 278 299
  270 236 257 288

, , Time = Apr

     Lat
Long   75  45  15
  0   270 291 314
  90  260 287 302
  180 258 279 299
  270 247 274 292

, , Time = Jul

     Lat
Long   75  45  15
  0   279 297 309
  90  279 301 301
  180 273 284 301
  270 277 288 292
```

```
> dim(Temps)

[1] 4 3 3
```

It was created using the following commands:

```
> Temps.vec <- c(273, 239, 251, 236, 284, 260, 278, 257, 305, 299, 299, 288,
  270, 260, 258, 247, 291, 287, 279, 274, 314, 302, 299, 292, 279, 279, 273,
  277, 297, 301, 284, 288, 309, 301, 301, 292)

> Temps <- array(Temps.vec, dim = c(4, 3, 3))

> dimnames(Temps) <- list(
      Long = c("0", "90" ,"180", "270"),
      Lat = c("75", "45", "15"),
      Time = c("Jan", "Apr", "Jul"))
```

Section 3.2 Exercises

Exercise 3 Add dimension names to the array y that you created in Exercise 1 so that you end up with this:

```
> y
```

```
, , Layer = L1

    Col
Row  C1 C2 C3 C4
  R1  1  4  7 10
  R2  2  5  8 11
  R3  3  6  9 12

, , Layer = L2

    Col
Row  C1 C2 C3 C4
  R1 13 16 19 22
  R2 14 17 20 23
  R3 15 18 21 24
```

3.3 Array Indexing Using [ ]

3.3.1 Accessing Array Elements

- We access array elements (rows, columns, layers, etc.) using square brackets:

```
[ , , ]    # Access array elements via their dimension indices (row,
           # column, layer, etc.) separated by commas
```

- For example, using the `Temps` array from above, to get the value in the 2nd row (90th longitude), 3rd column (15th latitude), and 2nd layer (April), type:

```
> Temps[2, 3, 2]                    # Returns a single value

[1] 302
```

To get all the values (January, April, and July) in the 2nd row (90th longitude) and 3rd column (15th latitude), type:

```
> Temps[2, 3,  ]                    # Returns a vector

Jan Apr Jul
299 302 301
```

and to all get the values in the 2nd layer (April), type:

```
> Temps[ , , 2]                     # Returns a matrix
```

```
     Lat
Long   75  45  15
  0   270 291 314
  90  260 287 302
  180 258 279 299
  270 247 274 292
```

Notice if three indices are specified, an single value is returned. If two are specified, a vector is returned. If only one is specified, a matrix is returned.

3.3.2 Rearranging Array Elements

- We can also use square brackets to rearrange rows, columns, layers, etc. of an array.

For example, notice that in the `Temps` array above, the latitudes (columns) are in decreasing order. To rearrange the columns so that the latitudes are in increasing order, we could type:

```
> Temps[ , c(3, 2, 1), ]

, , Time = Jan

     Lat
Long   15  45  75
  0   305 284 273
  90  299 260 239
  180 299 278 251
  270 288 257 236

, , Time = Apr

     Lat
Long   15  45  75
  0   314 291 270
  90  302 287 260
  180 299 279 258
  270 292 274 247

, , Time = Jul

     Lat
Long   15  45  75
  0   309 297 279
  90  301 301 279
  180 301 284 273
  270 292 288 277
```

Section 3.3 Exercises

Exercise 4 Here's the `Temps` array again (with columns in the original order):

```
> Temps


, , Time = Jan

     Lat
Long   75  45  15
  0   273 284 305
  90  239 260 299
  180 251 278 299
  270 236 257 288

, , Time = Apr

     Lat
Long   75  45  15
  0   270 291 314
  90  260 287 302
  180 258 279 299
  270 247 274 292

, , Time = Jul

     Lat
Long   75  45  15
  0   279 297 309
  90  279 301 301
  180 273 284 301
  270 277 288 292
```

After creating the `Temps` array (using the commands given earlier in this section), guess what the results of each of the following will be, then check your answers.

a) `> Temps[3, 1, 2]`

b) `> Temps[1, , 3]`

c) `> is.vector(Temps[1, , 3])`

d) `> Temps[1, , ]`

e) `> is.matrix(Temps[1, , ])`

Exercise 5 Here's a small array `x`:

```
> x <- array(1:8, dim = c(2, 2, 2))
> dimnames(x) <- list(Row = c("R1", "R2"),
                      Col = c("C1", "C2"),
                      Layer = c("L1", "L2"))
> x


, , Layer = L1

    Col
Row  C1 C2
  R1  1  3
  R2  2  4

, , Layer = L2

    Col
Row  C1 C2
  R1  5  7
  R2  6  8
```

The following commands rearrange the rows, columns, or layers of `x`. Guess what the result of each command will be, then check your answers:

a) `> x[c(2, 1), , ]`

b) `> x[, c(2, 1), ]`

c) `> x[, , c(2, 1)]`

Exercise 6 Write a command that returns the `Temps` array in the order July (layer 1), April (layer 2), January (layer 3), as below:

```
, , Time = Jul

     Lat
Long  75  45  15
  0  279 297 309
  90 279 301 301
  180 273 284 301
  270 277 288 292

, , Time = Apr

     Lat
Long  75  45  15
  0  270 291 314
```

```
  90  260 287 302
  180 258 279 299
  270 247 274 292

, , Time = Jan

     Lat
Long   75  45  15
  0   273 284 305
  90  239 260 299
  180 251 278 299
  270 236 257 288
```

3.4 Transposing an Array

- We've seen that we can transpose a matrix (change its rows to columns) using `t()`.

  Similarly, we can rearrange (permute) an array's dimensions (i.e. transpose the array) using `aperm()`:

```
aperm()    # Transpose an array by rearranging (permuting) its
           # dimensions
```

- `aperm()` has two main arguments, `a`, the array to be transposed, and `perm`, a vector containing the values 1, 2, 3, etc. (for rows, columns, layers, etc.) whose order indicates the desired permutation of the array dimensions.
- For example, using the `Temps` array, if we want latitudes to be represented by rows, and longitudes by columns, we type:

```
> aperm(Temps, c(2, 1, 3))       # Change columns to rows, leave layers intact

, , Time = Jan

    Long
Lat    0  90 180 270
  75 273 239 251 236
  45 284 260 278 257
  15 305 299 299 288

, , Time = Apr

    Long
Lat    0  90 180 270
  75 270 260 258 247
```

```
  45 291 287 279 274
  15 314 302 299 292

, , Time = Jul

    Long
Lat    0  90 180 270
  75 279 279 273 277
  45 297 301 284 288
  15 309 301 301 292
```

3.5 The `apply()` Function

- We can apply a function over one (or more) array dimensions using `apply()`:

```
apply()          # Apply a function over one (or more) array dimensions
                 # (rows, columns, layers, etc.)
```

- `apply()` has three main arguments, `X`, an array, `FUN`, the function to be applied, and `MARGIN`, specifying which dimension(s) the function should be applied over (`MARGIN = 1` for rows, `MARGIN = 2` for columns, `MARGIN = 3` for layers, etc.).
- For example, using the `Temps` array from above, to find the mean temperature, averaged over the spatial region (rows and columns), for each month (layer), type:

```
> apply(Temps, MARGIN = 3, FUN = mean)      # Keeps the 3rd dimension (layers)

     Jan      Apr      Jul
272.4167 281.0833 290.0833
```

Notice that above, we've *kept* the 3rd dimension (layers, which correspond to months), and averaged over the other two dimensions (rows and columns, which correspond to longitudes and latitudes).

In general, the argument `MARGIN` in `apply()` refers to the dimension of the array that we want to *keep*, and the function `FUN` is applied across the other dimensions.

For example, to find the mean temperature, separately for each latitude (column), we type:

```
> apply(Temps, MARGIN = 2, FUN = mean)      # Keeps the 2nd dimension (columns)

      75       45       15
261.8333 281.6667 300.0833
```

Above, we've *kept* the 2nd dimension (columns, which correspond to latitudes), and averaged over the other two dimensions (rows and layers, which correspond to longitudes and months).

To find the mean temperature, averaged over months separately for each location (longitude, latitude combination), we type:

```
> apply(Temps, MARGIN = c(1, 2), FUN = mean)      # Keeps the 1st and 2nd dimen-
                                                  # sions (rows and columns)
```

```
     Lat
Long        75       45       15
  0   274.0000 290.6667 309.3333
  90  259.3333 282.6667 300.6667
  180 260.6667 280.3333 299.6667
  270 253.3333 273.0000 290.6667
```

Above, we've *kept* the 1st and 2nd dimensions (rows and columns, which correspond to longitudes and latitudes), and averaged over the 3rd dimension (layers, which correspond to months).

Section 3.5 Exercises

Exercise 7 Here's a small array `x`:

```
> x <- array(c(0, 2, 2, 8, 2, 0, 8, 2), dim = c(2, 2, 2))
> dimnames(x) <- list(Row = c("R1", "R2"),
                      Col = c("C1", "C2"),
                      Layer = c("L1", "L2"))
> x

, , Layer = L1

    Col
Row  C1 C2
  R1  0  2
  R2  2  8

, , Layer = L2

    Col
Row  C1 C2
  R1  2  8
  R2  0  2
```

Guess what result of each following commands will be, then check your answers:

a) `> apply(x, MARGIN = 1, FUN = mean)`

b) `> apply(x, MARGIN = 3, FUN = mean)`

c) `> apply(x, MARGIN = c(1, 2), FUN = mean)`

3.6 Arrays are Vectors with a Dimensions Attribute

- Technically, an array is just a vector with a dimensions *attribute*. Recall that in R, an *attribute* is a piece of *metadata*m that is, a piece of data about a data set.

  For example, consider this small array `x`:

```
> x <- array(1:8, dim = c(2, 2, 2))
> x


, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

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

  `x` has one attribute, its dimensions:

```
> attributes(x)


$dim
[1] 2 2 2
```

- If we pass an array to a function that expects a vector, like `sum()`, R treats the array as if it was a vector:

```
> sum(x)


[1] 36
```

- We can explicitly coerce an array to a vector using:

```
as.vector()                    # Coerce an object to a vector
```

- For example (using the array `x` from above):

```
> as.vector(x)

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

Section 3.6 Exercises

Exercise 8 Here's a small array `x`:

```
> x <- array(8:1, dim = c(2, 2, 2))
> x


, , 1

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

, , 2

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

Guess what the following commands will return, then check your answers:

a) `> sum(x)`

b) `> as.vector(x)`