MTH 3220 R Notes 3

3 Matrices

3.1 Creating and Examining Matrices

- *Matrices* are one way of storing data in a two-dimensional layout (i.e. in rows and columns). They're easily created and examined in R using the functions:

```
matrix()            # Create a matrix, from a vector, with nrow rows
                    # and ncol columns
dim()               # Returns the dimensions (number of rows and
                    # columns) of a matrix
nrow(); ncol()      # Number of rows, number of columns of a matrix
is.matrix()         # Indicates whether or not an object is a matrix
```

- Here's an example showing how to create a matrix from a vector using `matrix()`:

```
x <- matrix(1:9, nrow = 3, ncol = 3)
x

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

is.matrix(x)

## [1] TRUE

dim(x)

## [1] 3 3
```

- By default, the matrix is created by filling in its *columns*, left to right. An optional argument, `byrow`, can be set to `TRUE` if we want to create it by filling its *rows*:

```
x <- matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
x

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

- If there are too few elements in the vector passed to `matrix()` to fill the matrix, the vector elements are *recycled* until the matrix has been filled.

If the length of the vector isn't a multiple (or sub-multiple) of the number of rows (or columns if `byrow = TRUE`), a warning message is printed:

```
x <- matrix(c(1, 2, 3, 4), nrow = 3, ncol = 3)

## Warning in matrix(c(1, 2, 3, 4), nrow = 3, ncol = 3):  data length [4] is not a sub-multiple
or multiple of the number of rows [3]

x

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

- We can also create matrices using:

```
cbind()            # Create a matrix by "binding" vectors together
                   # in columns.
rbind()            # Create a matrix by "binding" vectors together
                   # in rows.
```

- `cbind()` "binds" two or more vectors of the same length into columns of a matrix, and `rbind()` "binds" them into rows. For example:

```
x <- cbind(c(1,2,3), c(4,5,6), c(7,8,9))
x

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

Section 3.1 Exercises

Exercise 1 Here's a matrix `x`:

```
x <- matrix(c(8, 8, 8, 4, 4, 4), nrow = 2, byrow = T)
x

##      [,1] [,2] [,3]
## [1,]    8    8    8
## [2,]    4    4    4
```

Guess what the result of each of the following will be, then check your answers.

a) `is.matrix(x)`

b) `dim(x)`

c) `nrow(x)`

d) `ncol(x)`

Exercise 2 If there are too few elements in the vector passed to `matrix()` to fill the matrix, the vector elements are *recycled* until the matrix has been filled.

If the length of the vector isn't a multiple (or sub-multiple) of the number of rows (or columns if `byrow = TRUE`), a warning message is printed.

Guess what the result of each of the following will be, then check your answers.

a) `matrix(c(1, 2, 3), nrow = 3, ncol = 2)`

b) `matrix(c(1, 2, 3), nrow = 4, ncol = 2)`

Exercise 3 Here's a matrix `x`:

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    2
```

a) Write a command that creates the matrix using `matrix()`.

b) Write a command that creates it using `cbind()`.

c) Write a command that creates it using `rbind()`

3.2 General Matrix Operations

3.2.1 Matrix Arithmetic and Recycling

Matrix Arithmetic

- The operators '`+`', '`-`', '`*`', '`/`', and '`^`' operate on matrices *elementwise*. For example, consider the matrices `x` and `y`:

```
x

##      [,1] [,2] [,3]
## [1,]    5    5    5
## [2,]    4    4    4
## [3,]    3    3    3

y

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    1    2    3
## [3,]    1    2    3
```

To add 2 to each element of `x`, we type:

```
x + 2

##      [,1] [,2] [,3]
## [1,]    7    7    7
## [2,]    6    6    6
## [3,]    5    5    5
```

and to add each element of `y` to the corresponding element of `x`, type:

```
x + y

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

- The built-in R functions that are *vectorized*, such as `sqrt()`, `log()`, `cos()`, etc. also operate elementwise on matrices.

Recycling

- We can also carry out arithmetic operations ('`+`', '`-`', '`*`', '`/`', and '`^`') using a *vector* and a matrix. For example:

```
x.mat

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

y.vec

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

x.mat - y.vec

##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
```

Notice that elements of the vector are matched with the columns of the matrix from left to right.

- If the vector is too short, its elements are recycled. If the length of the vector isn't a sub-multiple of the total number of elements in the matrix, a warning message is printed.

3.2.2 Matrix Indexing Using `[ ]`

Accessing Matrix Elements, Rows, or Columns

- We access matrix elements, rows, or columns using square brackets:

```
[ , ]        # Access matrix elements via their row and column indices
             # (separated by a comma)
```

- To extract a specific element, specify its row and column indices in square brackets `[ ]`, separated by a comma. For example, here's the matrix `x` from above:

```
x

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

To extract the element in the 2nd row and 3rd column, we type:

```
x[2, 3]

## [1] 8
```

- An entire row is accessed by leaving the column index blank. For example the 2nd row of `x` is obtained via:

```
x[2, ]

## [1] 2 5 8
```

Likewise, an entire column is accessed by leaving the row index blank:

```
x[, 3]

## [1] 7 8 9
```

Replacing Matrix Elements, Rows, or Columns

- The assignment operator `<-` can be used to assign a new value to a matrix element:

```
x[2, 3] <- 0
x

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    0
## [3,]    3    6    9
```

- To replace an entire column, we leave the row index blank:

```
x[, 3] <- c(0, 0, 0)
x

##      [,1] [,2] [,3]
## [1,]    1    4    0
## [2,]    2    5    0
## [3,]    3    6    0
```

Replacing an entire row is done in a similar manner, but we leave the column index blank.

- Negative indices in square brackets (e.g. `x[-1, ]`) return all but that row or column of a matrix.

Adding and Deleting Matrix Rows or Columns

- We can add a row or column to an existing matrix using `rbind()` and `cbind()`:

```
x

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

cbind(x, c(10, 10, 10))

##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   10
## [3,]    3    6    9   10
```

Rearranging Matrix Rows or Columns

- Square brackets can also be used to rearrange (permute) the rows or columns of a matrix. To do so, we specify the desired permutation before or after the comma. For example, here's the matrix `x` again:

```
x

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

To obtain `x` with its first and third columns swapped, we type:

```
x[ , c(3, 2, 1)]

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

Transposing a Matrix

- Sometimes we need to ***transpose*** a matrix, i.e. turn its rows into columns and its columns into rows. We use the function:

```
t()                    # Transpose a matrix
```

- Here's an example:

```
x
```

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

```
t(x)
```

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

Filtering on Matrices

- We can extract (or *filter* out) certain rows from a matrix by providing the appropriate `"logical"` vector in square brackets (before the comma). For example:

```
x
```

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

```
x[c(TRUE, FALSE, TRUE), ]
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    3    6    9
```

  Above, the rows corresponding to `TRUE` in the `"logical"` vector are extracted from `x`.

- Now consider the following data set corresponding to employees at a manufacturing plant:

Age	Years Experience	Paid Sick Days
43	17	3
22	4	7
37	6	2
35	6	0
31	3	0
27	4	1
55	16	0
45	19	2
35	7	3
39	8	1
40	14	0
44	11	1
60	21	1

  Here are the data stored as a matrix in R:

```
employees
```

```
##       Age Years Experience Paid Sick Days
##  [1,]  43               17              3
##  [2,]  22                4              7
##  [3,]  37                6              2
##  [4,]  35                6              0
##  [5,]  31                3              0
##  [6,]  27                4              1
##  [7,]  55               16              0
##  [8,]  45               19              2
##  [9,]  35                7              3
## [10,]  39                8              1
## [11,]  40               14              0
## [12,]  44               11              1
## [13,]  60               21              1
```

(We'll see later how the column headings were added to the matrix).

- To determine which employees are older than 40, we type:

```
employees[ , 1] > 40

##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
## [12]  TRUE  TRUE
```

Note that the result is a `"logical"` vector. Now, to extract those employees' rows from the `employees` matrix, we provide this `"logical"` vector in square brackets before the comma:

```
employees[employees[ , 1] > 40, ]

##      Age Years Experience Paid Sick Days
## [1,]  43               17              3
## [2,]  55               16              0
## [3,]  45               19              2
## [4,]  44               11              1
## [5,]  60               21              1
```

Section 3.2 Exercises

Exercise 4 Arithmetic operators ('+', '−', '∗', etc.) and many of R's built-in functions (e.g. `sqrt()` and `log()`) operate *elementwise* on matrices.

Consider the matrices `x` and `y`:

```
x <- matrix(c(1, 4, 9, 16), nrow = 2, ncol = 2)
x

##      [,1] [,2]
## [1,]    1    9
## [2,]    4   16
```

```
y <- matrix(c(2, 1, 2, 1), nrow = 2, ncol = 2)
y
```

```
##      [,1] [,2]
## [1,]    2    2
## [2,]    1    1
```

Guess what the result of each of the following will be, then check your answers.

a) `x + 1`

b) `sqrt(x)`

c) `x * y`

Exercise 5 When arithmetic operations are performed with a matrix and a vector, the elements of the vector are matched with the columns of the matrix.

Consider the following matrix and vector:

```
x.mat <- matrix(1:6, nrow = 3, ncol = 2)
x.mat
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
y.vec <- 1:6
y.vec
```

```
## [1] 1 2 3 4 5 6
```

Guess what the result of the following will be, then check your answer.

```
x.mat + y.vec
```

Exercise 6 If an arithmetic operation is performed with a matrix and a vector, and the vector is too short, its elements are recycled. If the length of the vector isn't a sub-multiple of the total number of elements in the matrix, a warning message is printed.

Consider the following matrix and vectors:

```
x.mat <- matrix(1:6, nrow = 2, ncol = 3)
x.mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
y.vec <- 1:3
y.vec
```

```
## [1] 1 2 3
```

```
z.vec <- 1:5
z.vec
```

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

In each case below, the guess whether R will print a warning message, then check your answers.

a) `x.mat - y.vec`

b) `x.mat - z.vec`

Exercise 7 To access individual elements or entire rows or columns of a matrix, we specify their indices in square brackets `[ ]`, separated by a comma. Consider the following matrix:

```
x <- matrix(1:9, nrow = 3, ncol = 3)
x
```

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

Guess what the result of each of the following will be, then check your answers.

a) `x[1, 3]`

b) `x[1, ]`

c) `x[, 3]`

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

e) `x[c(1, 2), 3]`

f) `x[-1, ]`

g) `x[-1, 3]`

Exercise 8 To rearrange the rows or columns of a matrix, we specify the rearrangement in square brackets either before or after a comma. Consider the following matrix:

```
x <- matrix(1:6, nrow = 2, ncol = 3)
x

##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Guess what the result of each of the following will be, then check your answers.

a) `x[, c(3, 2, 1)]`

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

Exercise 9 We can extract (or *filter* out) certain rows of a matrix by specifying an appropriate `"logical"` vector before a comma in square brackets. Consider the following matrix:

```
x <- matrix(1:6, nrow = 3, ncol = 2)
x

##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Guess what the result of the following will be, then check your answer.

```
x[c(TRUE, TRUE, FALSE), ]
```

Exercise 10 Consider the following data on six kids and their pets:

Kid	Age	Number of Siblings	Number of Pets	Age of Oldest Pet
1	11	1	2	5
2	8	2	1	3
3	12	1	2	2
4	9	0	0	NA
5	12	3	0	NA
6	10	4	1	6

These data can be stored as a matrix using the following command:

```
kidsPets <- cbind(1:6, c(11, 8, 12, 9, 12, 10), c(1, 2, 1, 0, 3, 4),
                  c(2, 1, 2, 0, 0, 1), c(5, 3, 2, NA, NA, 6))
```

The `colnames()` function can be used to add column headings to the matrix:

```
colnames(kidsPets) <- c("Kid", "Age", "Number of Siblings",
                        "Number of Pets", "Age of Oldest Pet")
```

a) Here's a `"logical"` vector indicating whether a kid is over the age of 10:

```
kidsPets[, 2] > 10

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

In words, what will the following command do? Check your answer.

```
kidsPets[kidsPets[, 2] > 10, ]
```

b) In words, what will the following command do? Check your answer.

```
kidsPets[kidsPets[, 4] > 0, ]
```

3.3 The `apply()` Function

- Sometimes we need to apply a function separately to each row (or each column) of a matrix. To do so, we use:

```
apply()         # Apply a function separately to each row (or each
                # column) of a matrix
```

- `apply()` has three main arguments, `x`, a matrix, `MARGIN`, for indicating rows or columns (`MARGIN = 1` for rows and `MARGIN = 2` for columns), and `FUN`, the function to be applied to the rows (or columns) of `x`.
- For example, below, `apply()` is used to compute the sum of each row:

```
x

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

apply(X = x, MARGIN = 1, FUN = sum)

## [1] 12 15 18
```

Note that `apply()` returns a vector, each element of which is the sum of the corresponding row in `x`.

- If the function specified by `FUN` requires additional arguments, they're passed through `apply()` after `FUN`. For example, here are the arguments for the function `sample()`:

```
args(sample)

## function (x, size, replace = FALSE, prob = NULL)
## NULL
```

Its two main arguments are `x`, a vector, and `size`, a sample size. It generates a random sample of the specified `size` from the elements of the vector.

To use `apply()` to randomly select one value from each row of the matrix `x` from above, we type:

```
apply(X = x, MARGIN = 1, FUN = sample, size = 1)

## [1] 1 2 9
```

Note that the sample `size` was specified after `FUN` in the call to `apply()`.

Section 3.3 Exercises

Exercise 11 Consider the following matrix `x`:

```
x <- matrix(c(8, 6, 3, 6, 5, 7, 2, 1, 9, 4, 7, 6, 8, 3, 4, 3),
              nrow = 4, ncol = 4)
x
```

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

Guess what the result of each of the following will be, then check your answers.

a) `apply(x, MARGIN = 1, FUN = min)`

b) `apply(x, MARGIN = 2, FUN = min)`

c) `apply(x, MARGIN = 1, FUN = which.min)`

Exercise 12 R has several built-in datasets, one of which is `USPersonalExpenditure`, a matrix with five rows and five columns consisting of U.S. personal expenditures (in billions of dollars) in the categories: Food and Tobacco, Household Operation, Medical and Health, Personal Care, and Private Education for the years 1940, 1945, 1950, 1955 and 1960.

Type:

```
USPersonalExpenditure
```

to view the data set.

a) Write a command that uses `apply()`, with `FUN = mean`, to find the mean expenditure for each of the five expenditure categories (rows).

b) Write a command that uses `apply()` to find the mean expenditure for each of the five years (columns).

3.4 Naming Matrix Rows and Columns

- We can assign names to the rows or columns of a matrix, or get the names if they already exist, using:

```
rownames()        # Get or assign the row names of a matrix
colnames()        # Get or assign the column names of a matrix
```

- Here's an example:

```
x

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

colnames(x) <- c("ColA", "ColB",  "ColC")          # Assign column names
x

##      ColA ColB ColC
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

colnames(x)                                        # Get the column names

## [1] "ColA" "ColB" "ColC"
```

- To remove the row or column names, we assign `NULL` to them:

```
colnames(x) <- NULL
```

Section 3.4 Exercises

Exercise 13 Consider the following matrix `x`:

```
x <- matrix(1:9, nrow = 3, ncol = 3)
```

a) After creating the matrix, write a command involving `colnames()` that assigns the names `"A", "B", "C"` to its columns.

b) Now write a command that removes the column names from `x`.

3.5 Matrices are Vectors with Two Dimensions

- Internally, R stores the elements of a matrix as a vector (but keeps track of the matrix's dimensions). Therefore, a matrix will often behave like a vector. For example:

```
x                                   # x is a matrix

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

length(x)                           # x behaves like a vector here

## [1] 9

mean(x)                             # x behaves like a vector here too

## [1] 5
```

- We can explicitly coerce a matrix to a vector using:

```
as.vector()          # Coerce a matrix (or other R object) to a vector
```

- For example, to coerce the matrix `x` from above to a vector, type:

```
as.vector(x)

## [1] 1 2 3 4 5 6 7 8 9
```

Notice that R links the columns of the matrix `x` together end to end to form the vector.

Section 3.5 Exercises

Exercise 14 Technically, a matrix is just a vector with two dimensions. So when you pass a matrix to a function that's expecting a vector, like `sum()`, `mean()`, or `max()`, R behaves as if the matrix is a vector.

Consider again the following matrix `x`:

```
x <- matrix(c(1, 1, 4, 2, 3, 1), nrow = 3, ncol = 2)
x

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

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

a) `sum(x)`

b) `mean(x)`

c) `max(x)`

Exercise 15 Consider again the matrix `x`:

```
x <- matrix(c(1, 1, 4, 2, 3, 1), nrow = 3, ncol = 2)
x
```

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

Guess what the result of the following will be, then check your answer:

```
as.vector(x)
```