Base R Data Structures: Matrices

Common Data Structures

A data scientist needs to deal with data! We need to have a firm foundation in the ways that we can store our data in R. This section goes through the most commonly used 'built-in' R objects that we'll use.

- There are five major data structures used in R

  1. Atomic Vector (1d)
  2. Matrix (2d)
  3. Array (nd)
  4. Data Frame (2d)
  5. List (1d)

Dimension	Homogeneous (elements all the same)	Heterogeneous (elements may differ)
1d	Atomic Vector	List
2d	Matrix	Data Frame
nd	Array	

Matrix

Vectors are useful to know about but generally not great for dealing with a dataset. When we think of a dataset, we often think about a spreadsheet having rows corresponding to **observations** and columns corresponding to **variables**.

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

Here the observations correspond to measurements on flowers. The variables that were measured on each flower are the columns.

We can see that a vector is not really useful for handling this kind of 'standard' data since it is 1D. However, we can think of vectors as the building blocks for more complicated types of data.

As matrices are homogeneous (all elements must be the same type), we can think of a matrix as a collection of vectors of the same **type and length** (although this is not formally how it works in R). Think of each vector as a column of the matrix.

Creating a Matrix

- We can create a matrix using the `matrix()` function. Looking at the help for `matrix` we see the function definition as:

```
matrix(data = NA,
       nrow = 1,
       ncol = 1,
       byrow = FALSE,
       dimnames = NULL)
```

where `data` is

> an optional data vector (including a list or expression vector). Non-atomic classed R objects are coerced by as.vector and all attributes discarded.

Ok, so really we need to supply a **vector** of data. Then the `nrow` and `ncol` arguments define how many rows and columns. The `byrow` argument is a Boolean telling R whether or not to fill in the matrix by row or by column (the default). `dimnames` is an optional **attribute**. We'll look at that shortly.

A basic call to `matrix()` might look like this:

```
my_mat <- matrix(c(1, 3, 4, -1, 5, 6),
                 nrow = 3,
                 ncol = 2)
my_mat
```

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

Notice how the values fill in the first column then the second column. This is the default behavior but can be changed via the `byrow` argument.

```
my_mat <- matrix(c(1, 3, 4, -1, 5, 6),
                 nrow = 3,
                 ncol = 2,
                 byrow = TRUE)
my_mat
```

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

Let's think of a matrix as a collection of vectors of the same **type and length** (how you might think of a dataset). We could construct the matrix by giving appropriate vectors of the same type and length.

```
#populate vectors
x <- rep(0.2, times = 6)
y <- c(1, 3, 4, -1, 5, 6)
```

- Check they are the same `type` of element. In this case, as long as they are both `numeric`, we can put them together and not lose any precision (`integers` are coerced to `doubles` if needed).

```
str(x)
```

```
 num [1:6] 0.2 0.2 0.2 0.2 0.2 0.2
```

```
is.numeric(x)
```

```
[1] TRUE
```

```
str(y)
```

```
 num [1:6] 1 3 4 -1 5 6
```

```
is.numeric(y)
```

```
[1] TRUE
```

- A `length()` function exists. We can use that to see the vectors have the same number of elements.

```
#check 'length'
length(x)
```

```
[1] 6
```

```
length(y)
```

```
[1] 6
```

- Now construct the matrix by *combining* the vectors together with `c()`.

```
my_mat2 <- matrix(c(x, y), ncol = 2)
my_mat2
```

```
     [,1] [,2]
[1,]  0.2    1
[2,]  0.2    3
[3,]  0.2    4
[4,]  0.2   -1
[5,]  0.2    5
```

```
[6,]  0.2    6
```

As `byrow = FALSE` is the default, the vectors become the columns!

That being said, the way to really think about the matrix is as a long vector we are giving dimensions to. In fact, if we coerce our matrix to a vector explicitly (via the `as.vector()` function), we see the original data vector we passed in.

```
as.vector(my_mat2)
```

```
 [1]  0.2  0.2  0.2  0.2  0.2  0.2  1.0  3.0  4.0 -1.0  5.0  6.0
```

- Matrices don't have to be made up of numeric data. As long as the elements are all the same type, we are good to go!

```
x <- c("Hi", "There", "!")
y <- c("a", "b", "c")
z <- c("One", "Two", "Three")
matrix(c(x, y, z), nrow = 3)
```

```
     [,1]    [,2] [,3]  
[1,] "Hi"    "a"  "One" 
[2,] "There" "b"  "Two" 
[3,] "!"     "c"  "Three"
```

- Notice that we don't need to specify the number of columns. R figures that out!

- We do need to be careful about how R **recycles** things:

```
matrix(c(x, y, z), ncol = 2)
```

```
Warning in matrix(c(x, y, z), ncol = 2): data length [9] is not a sub-
multiple
or multiple of the number of rows [5]
```

```
     [,1]    [,2]   
[1,] "Hi"    "c"    
[2,] "There" "One"  
[3,] "!"     "Two"  
[4,] "a"     "Three"
[5,] "b"     "Hi"   
```

- We were 1 element short of being able to fill a matrix with 2 columns and 5 rows (5 chosen to make sure all the data was included). R recycles the first element we passed to fill in the last value. This is a common thing that R does. It can be useful for shorthanding things but otherwise is just something we need to be aware of as a common error to be fixed!

```
matrix(0, nrow = 2, ncol = 2)
```

```
     [,1] [,2]
[1,]    0    0
```

```
[1,]    0    0
[2,]    0    0
```

Matrix Attributes

Similar to a vector, a matrix can have attributes (this is true of any R object!). The common attributes associated with a matrix are the `dim` or dimensions and the `dimnames` or dimension names.

```
my_iris <- as.matrix(iris[, 1:4])
head(my_iris)
```

```
     Sepal.Length Sepal.Width Petal.Length Petal.Width
[1,]          5.1         3.5          1.4         0.2
[2,]          4.9         3.0          1.4         0.2
[3,]          4.7         3.2          1.3         0.2
[4,]          4.6         3.1          1.5         0.2
[5,]          5.0         3.6          1.4         0.2
[6,]          5.4         3.9          1.7         0.4
```

```
str(my_iris)
```

```
 num [1:150, 1:4] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 - attr(*, "dimnames")=List of 2
  ..$ : NULL
  ..$ : chr [1:4] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width"
```

```
attributes(my_iris)
```

```
$dim
[1] 150   4

$dimnames
$dimnames[[1]]
NULL

$dimnames[[2]]
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
```

We can access these attributes through functions designed for them:

```
dim(my_iris)
```

```
[1] 150   4
```

```
dimnames(my_iris)
```

```
[[1]]
NULL

[[2]]
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
```

Here we can name the rows and the columns of a matrix by assigning a `list()` to the `dimnames` attribute. (Lists are covered shortly.)

```
dimnames(my_iris) <- list(
  1:150, #first list element is a vector for the row names
  c("Sepal Length", "Sepal Width", "Petal Length", "Petal Width")
)
head(my_iris)
```

```
  Sepal Length Sepal Width Petal Length Petal Width
1          5.1         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
4          4.6         3.1          1.5         0.2
5          5.0         3.6          1.4         0.2
6          5.4         3.9          1.7         0.4
```

We can assign these when we create a matrix!

```
my_mat3 <- matrix(c(runif(10),
                    rnorm(10),
                    rgamma(10, shape = 1, scale = 1)),
                  ncol = 3,
                  dimnames = list(1:10, c("Uniform", "Normal", "Ga
my_mat3
```

```
     Uniform      Normal     Gamma
1  0.4231678  1.90951023 0.1895390
2  0.2578190  1.00936720 0.1174303
3  0.5439424  0.16105580 0.2143738
4  0.7515014 -1.08958063 0.6573588
5  0.2452878 -0.70252775 0.2337714
6  0.9510920 -0.04129549 2.8102538
7  0.8407916  0.38951588 1.9387043
8  0.3350867  0.35077086 0.2072771
9  0.3458813  1.75740781 1.1187380
10 0.8857804 -0.77953385 0.4975693
```

Accessing Elements of a Matrix

We saw that `[]` allowed us to access the elements of a vector. We'll use the same syntax to get at elements of a matrix. However, we now have two dimensions! That means we need to specify which row elements and which column elements. This is done by using square brackets with a comma in between our indices.

- Notice the default row names and column names! This gives us hints on this syntax!

```
mat <- matrix(c(1:4, 20:17), ncol = 2)
mat
```

```
     [,1] [,2]
[1,]    1   20
```

```
[1,]    1   20
[2,]    2   19
[3,]    3   18
[4,]    4   17
```

- If we leave an index blank, we get the entirety of that index back.

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

```
[1] 19
```

```
mat[ , 1]
```

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

```
mat[2, ]
```

```
[1]  2 19
```

```
mat[2:4, 1]
```

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

```
mat[c(2, 4), ]
```

```
     [,1] [,2]
[1,]    2   19
[2,]    4   17
```

Notice that R simplifies the result where possible. That is, returns an atomic vector if you have only 1 dimension and a matrix if two.

Also, if you only give a single value in the `[]` then R uses the count of the value in the matrix (essentially treating the matrix elements as a long vector). These counts go down columns first.

- If you do have `dimnames` associated, then you can access elements with those.

```
mat <- matrix(c(1:4, 20:17), ncol = 2,
          dimnames = list(NULL,
                c("First", "Second"))
        )
mat
```

```
     First Second
[1,]     1     20
[2,]     2     19
[3,]     3     18
[4,]     4     17
```

```
mat[, "First"]
```

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

Arrays

Arrays are the n-dimensional extension of matrices. Like matrices, they must have all elements of the same type.

```
my_array <- array(1:24, dim = c(4, 2, 3))
my_array
```

```
, , 1

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

, , 2

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

, , 3

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

Accessing elements is similar to vectors and matrices!

```
my_array[1, 1, 1]
```

```
[1] 1
```

```
my_array[4, 2, 1]
```

```
[1] 8
```

Arrays are often used in deep learning.

Recap!

- A 2D object where all elements are of the same type
- Access elements via `[ , ]`

- Not ideal for data since all elements must be the same type (see data frames!)