2 - Mathematical Objects

Mathematics and mathematical objects provide a foundation on which most of data science is built. These mathematical objects include scalar values, 1-dimensional vectors, 2-dimensional matrices and even higher dimensional data structures. This section will explore these objects and how one can initialize, create, and manipulate them using R or Python. This can be handled in R with built-in functions and array creation methods but in Python we must introduce the NumPy module.

2.1 - Python NumPy Module

As per the NumPy documentation page [8] Array Programming with NumPy, "NumPy is a fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more."

The NumPy documentation page [8] includes a section with instructions on downloading and installing the NumPy module. For example if you are using `pip` NumPy can be installed with the call `pip install numpy` . Once installed it can be loaded/imported with the `import` call as done below. It should be noted that version 1.19.1 of NumPy is being used.

```
import numpy as np # NumPy
print(np.__version__)
```

```
1.23.2
```

GOOD TIP: Import the NumPy module with a shorter name such as `np` (this was done above). Each time a NumPy function is used its name must be called first. For example to use the NumPy `random()` function one must make the call `np.random()` . If NumPy had been imported under a different name such as `numpy_module` then the call would have been `numpy_module.random()` . Using shorter and clear module names can save you some time coding.

2.2 - Vectors (1D Arrays)

An important distinction between numeric arrays is those which use floating point values (real numbers) and those which use integers values. There is usually a preference in using one over the other for most applications; for example, integer arrays can be used for indexing.

Ex. One dimensional numeric arrays of the form:

$$x_i \in \mathbb{R},\ i = 1, \dots, n,\ \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n$$

2.2.1 - Numpy Vectors

According to the NumPy documentation [8], there are six general mechanisms for creating arrays:

1. Conversion from other Python structures (i.e. lists and tuples)
2. Intrinsic NumPy array creation functions (e.g. arange, ones, zeros, etc.)
3. Replicating, joining, or mutating existing arrays
4. Reading arrays from disk, either from standard or custom formats
5. Creating arrays from raw bytes through the use of strings or buffers
6. Use of special library functions (e.g., random)

```
# arange(n) gives an array of size n with elements from 0 to n-1
X = np.arange(3)
X
```

```
array([0, 1, 2])
```

Note that the NumPy `arange()` function returns an integer array where as the `ones()` and `zeros()` functions return floating point arrays by default.

```
# ones(n) gives array of size n with all elements equal to 1
X = np.ones((1,3)) # 1 row 3 columns
X
```

```
array([[1., 1., 1.]])
```

```
# zeros(n) gives array of size n with all elements equal to 0
X = np.zeros((3,1)) # 3 rows 1 column
X
```

```
array([[0.],
       [0.],
       [0.]])
```

When manually creating a NumPy array the function `np.array()` is used. If any one of the values is set to floating point then the entire array will be a floating point array (all values change).

```
# array() allows for manual array creation
X = np.array([0,0,0.]) # "0." enforces floating point
X
```

```
array([0., 0., 0.])
```

2.2.2 - Indexing NumPy Vectors

NumPy arrays are 0-indexed and indexing makes use of square brackets. The indexing of [0] will give the first element in the array. Placing an array inside the indexing bracket allows one to index any number of specific elements.

```
X = np.array([1,2,3,4])
X
```

```
array([1, 2, 3, 4])
```

```
X[0] # first element
```

```
1
```

```
X[3] # last element
```

```
4
```

```
X[0:3] # use colon sytax n:m to get elements from postion n to m-1
```

```
array([1, 2, 3])
```

```
X[[0,1,2,3,0,0,1,1]] # insert array to get any elements
```

```
array([1, 2, 3, 4, 1, 1, 2, 2])
```

2.2.3 - R Vectors

A vector is the most common and basic data structure in R. Technically, a vector can be one of two types: atomic vectors and lists. It is a collection of elements that are most commonly composed of characters, logical, integers or numeric values. Vectors can be initialized in R using various functions or manual options such as colon syntax `:` , `rep()` function, `numeric()` function and manual `c()` function.

```
# colon sytax "n:m" gives an array of size (m-n) with elements from n to m
X <- 0:2
print(X)
```

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

```
# rep(x, n) gives an array of the element x repeated n times
X <- rep(1,3)
print(X)
```

```
[1] 1 1 1
```

```
# numeric(n) can intialize array of size n with all elements equal to 0
X <- numeric(3)
print(X)
```

```
[1] 0 0 0
```

```
# c() allows for manual array creation
X <- c(1,2,3)
print(X)
```

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

2.2.4 - Indexing R Vectors

Vectors in R are not 0-indexed and indexing makes use of square brackets. The indexing of [1] will give the first element in the vector. As with Python, placing a vector inside the indexing bracket allows one to index any number of specific elements.

```
X <- c(1,2,3,4)
print(X)
```

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

```
X[1] # first element
```

1

```
X[4] # last element
```

4

```
print(X[1:3]) # use colon sytax n:m to get elements from postion n to m
```

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

```
print(X[c(1,2,3,4,1,1,2,2)]) # insert array to get any elements
```

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

2.3 - Matrices (2D Arrays)

All numeric matrices in the following examples use only floating point values:

$$x_{ij} \in \mathbb{R},\ i = 1, \dots, n,\ j = 1, \dots, m,\ X = \begin{bmatrix} x_{11} & \dots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \dots & x_{nm} \end{bmatrix} \in \mathbb{R}^{n \times m}$$

2.3.1 - NumPy Matrices

In Python, a matrix is a two-dimensional data structure arranged into rows and columns. Using NumPy, a matrix can be initialized the exact same way as the vectors but by using nested lists. Note that NumPy includes a Matrix data object (`np.matrix`) but using the array object (`np.array()`) is all that is needed. The same six general mechanisms for NumPy creating arrays (outlined above) can be applied to creating matrices or 2D arrays.

Note that NumPy includes an identity matrix function `np.eye(n)` which can create an $n \times n$ idenity matrix.

```
M = np.eye(2) # identity matrix
M
```

```
array([[1., 0.],
       [0., 1.]])
```

```
M = np.array([[1.,2],[3,4]]) # manual matrix creation
M
```

```
array([[1., 2.],
       [3., 4.]])
```

Matrices can be created with NumPy using the array operation or by changing the dimensional structure of a NumPy array with the `reshape()` function. Python fills the matrix in order of rows first, which is opposite of R.

```
m = np.array([1.,2,3,4]) # matrix from a vector
M = m.reshape(2,2)
M
```

```
array([[1., 2.],
       [3., 4.]])
```

2.3.2 - Indexing NumPy Matrices

NumPy matrix arrays are indexed with the same tools as those for vector arrays but one must now consider the additional dimension (across rows and columns).

```
M = np.array([[1,2],[3,4]])
M
```

```
array([[1, 2],
       [3, 4]])
```

```
M[0,0] # first row, first column (top left)
```

```
1
```

```
M[1,1] # last row, last columnn (bottom right)
```

```
4
```

```
M[:,0] # first column
```

```
array([1, 3])
```

```
M[1,:] # last row
```

```
array([3, 4])
```

```
M[[0,1,1], [1,1,1]] # specific elements
```

```
array([2, 4, 4])
```

```
M[[[0],[1],[1]], [1,1,1]] # specific rows and elements
```

```
array([[2, 2, 2],
       [4, 4, 4],
       [4, 4, 4]])
```

2.3.3 - R Matrices

In R, matrices are an extension of the numeric or character vectors. They are simply an atomic vector with dimensions: the number of rows and columns. The elements of a matrix in R must be of the same data type. Matrices can be created in R using the `matrix()` operation or by changing the dimensional structure of a vector with the `dim()` function.

```
M <- matrix(nrow = 2, ncol = 2) # matrix with elements not set
print(M)
```

```
     [,1] [,2]
[1,]   NA   NA
[2,]   NA   NA
```

```
M <- matrix(c(1,2,3,4), ncol = 2, nrow = 2) # manual matrix creation
print(M)
```

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

```
m <- c(1,2,3,4) # matrix from vector
dim(m) <- c(2,2)
print(m)
```

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

It should be noted how R fills the contents of a matrix, going along each column from top to bottom row. To change this filling to rows first, one can use `byrow = TRUE` inside the matrix call.

```
M <- matrix(c(1,2,3,4), nrow = 2, byrow = TRUE) # filled by row first
print(M)
```

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

2.3.4 - Indexing R Matrices

Matrices in R are indexed with the same tools as those for vectors in R but one must now consider the additional dimension (across rows and columns).

```
M <- matrix(c(1,2,3,4), nrow = 2, byrow = TRUE)
print(M)
```

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

```
M[1,1] # first row, first column (top left)
```

1

```
M[2,2] # last row, last columnn (bottom right)
```

4

```
print(M[,1]) # first column
```

```
[1] 1 3
```

```
print(M[2,]) # last row
```

```
[1] 3 4
```

```
print(M[c(1,2,2), 2]) # specific elements
```

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

```
print(M[c(1,2,2), c(2,2,2)]) # specific rows and elements
```

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

---

[8] Harris, C.R., Millman, K.J., van der Walt, S.J. et al, 2020. Array Programming with NumPy, [link]