

## 4.1: Array Representations of Vectors, Matrices, and Tensors

Thus far, we have discussed simple "one-dimensional" (1D) arrays, which are linear sequences of numbers. In linear algebra terms, 1D arrays represent **vectors**. The array length corresponds to the "vector dimension" (e.g., a 1D array of length 3 corresponds to a 3-vector). In accordance with Scipy terminology, we will use the word "dimension" to refer to the dimensionality of the array (called the **rank** in linear algebra), and not the vector dimension.

You are probably familiar with the fact that vectors can be represented using **index notation**, which is pretty similar to Python's notation for addressing 1D arrays. Consider a length- $d$  vector

$$\vec{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{d-1} \end{bmatrix}. \quad (4.1.1)$$

The  $j$ th element can be written as

$$x_j \leftrightarrow \mathbf{x}[j], \quad (4.1.2)$$

where  $j = 0, 1, \dots, d-1$ . The notation on the left is mathematical index notation, and the notation on the right is Python's array notation. Note that we are using 0-based indexing, so that the first element has index 0 and the last element has index  $d-1$ .

A **matrix** is a collection of numbers organized using two indices, rather than a single index like a vector. Under 0-based indexing, the elements of an  $m \times n$  matrix are:

$$\mathbf{M} = \begin{bmatrix} M_{00} & M_{01} & \cdots & M_{0,n-1} \\ M_{10} & M_{11} & \cdots & M_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ M_{m-1,0} & M_{m-1,1} & \cdots & M_{m-1,n-1} \end{bmatrix}. \quad (4.1.3)$$

More generally, numbers that are organized using multiple indices are collectively referred to as **tensors**. Tensors can have more than two indices. For example, vector cross products are computed using the **Levi-Civita tensor**  $\varepsilon$ , which has three indices:

$$(\vec{A} \times \vec{B})_i = \sum_{jk} \varepsilon_{ijk} A_j B_k. \quad (4.1.4)$$

### 4.1.1 Multi-Dimensional Arrays

In Python, tensors are represented by **multi-dimensional arrays**, which are similar to 1D arrays except that they are addressed using more than one index. For example, matrices are represented by 2D arrays, and the  $(i, j)$ th component of an  $m \times n$  matrix is written in Python notation as follows:

$$M_{ij} \leftrightarrow \mathbf{M}[i, j] \quad \text{for } i = 0, \dots, m-1, \quad j = 0, \dots, n-1. \quad (4.1.5)$$

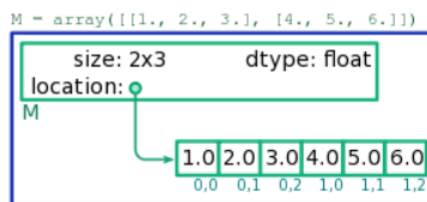


Figure 4.1.1: Memory model of a 2D array.

The way multi-dimensional arrays are laid out in memory is very similar to the memory layout of 1D arrays. There is a book-keeping block, which is associated with the array name, and which stores information about the array size (including the number of indices and the size of each index), as well as the memory location of the array contents. The elements lie in a sequence of storage blocks, in a specific order (depending on the array size). This arrangement is shown schematically in Fig. 4.1.1.

When Python needs to access any element of a multi-dimensional array, it knows exactly which memory location the element is stored in. The location can be worked out from the size of the multi-dimensional array, and the memory location of the first element. In Fig. 4.1.1, for example, `M` is a  $2 \times 3$  array, containing 6 storage blocks laid out in a specific sequence. If we need to access `M[1,1]`, Python knows that it needs to jump to the storage block four blocks down from the  $(0,0)$  block. Hence, reading/writing the elements of a multi-dimensional array is an  $O(1)$  operation, just like for 1D arrays.

In the following subsections, we will describe how multi-dimensional arrays can be created and manipulated in Python code.

#### Note

There is also a special Scipy class called `matrix` which can be used to represent matrices. *Don't use this.* It's a layer on top of Scipy's multi-dimensional array facilities, mostly intended as a crutch for programmers transitioning from Matlab. Arrays are better to use, and more consistent with the rest of Scipy.

### 4.1.2 Creating Multi-Dimensional Arrays

You can create a 2D array with specific elements using the `array` command, with an input consisting of a list of lists:

```
>>> x = array([[1., 2., 3.], [4., 5., 6.]])
>>> x
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

The above code creates a 2D array (i.e. a matrix) named `x`. It is a  $2 \times 3$  array, containing the elements  $x_{00} = 1$ ,  $x_{01} = 2$ ,  $x_{02} = 3$ , etc. Similarly, you can create a 3D array by supplying an input consisting of a list of lists of lists; and so forth.

It is more common, however, to create multi-dimensional arrays using `ones` or `zeros`. These functions return arrays whose elements are all initialized to 0.0 and 1.0, respectively. (You can then assign values to the elements as desired.) To do this, instead of specifying a number as the input (which would create a 1D array of that size), you should specify a tuple as the input. For example,

```
>>> x = zeros((2,3))
>>> x
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>>
>>> y = ones((3,2))
>>> y
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

There are many more ways to create multi-dimensional arrays, which we'll discuss when needed.

### 4.1.3 Basic Array Operations

To check on the dimension of an array, consult its `ndim` slot:

```
>>> x = zeros((5,4))
>>> x.ndim
2
```

To determine the exact shape of the array, use the `shape` slot (the shape is stored in the form of a tuple):

```
>>> x.shape
(3, 5)
```

To access the elements of a multi-dimensional array, use square-bracket notation:  $M[2, 3]$ ,  $T[0, 4, 2]$ , etc. Just remember that each component is zero-indexed.

Multi-dimensional arrays can be sliced, similar to 1D arrays. There is one important feature of multi-dimensional slicing: if you specify a single value as one of the indices, the slice results in an array of smaller dimensionality. For example:

```
>>> x = array([[1., 2., 3.], [4., 5., 6.]])
>>> x[:, 0]
array([ 1.,  4.] )
```

In the above code,  $x$  is a 2D array of size  $2 \times 3$ . The slice  $x[:, 0]$  specifies the value 0 for index 1, so the result is a 1D array containing the elements  $[x_{00}, x_{10}]$ .

If you don't specify all the indices of a multi-dimensional array, the omitted indices implicitly included, and run over their entire range. For example, for the above  $x$  array,

```
>>> x[1]
array([ 4.,  5.,  6.] )
```

This is also equivalent to  $x[1, :]$ .

#### 4.1.4 Arithmetic Operations

The basic arithmetic operations can all be performed on multi-dimensional arrays, and act on the arrays element-by-element. For example,

```
>>> x = ones((2,3))
>>> y = ones((2,3))
>>> z = x + y
>>> z
array([[ 2.,  2.,  2.],
       [ 2.,  2.,  2.]])
```

You can think of this in terms of index notation:

$$z_{ij} = x_{ij} + y_{ij}. \quad (4.1.6)$$

What is the runtime for performing such arithmetic operations on multi-dimensional arrays? With a bit of thinking, we can convince ourselves that the runtime scales linearly with the number of elements in the multi-dimensional array, because the arithmetic operation is performed on each individual index. For example, the runtime for adding a pair of  $M \times N$  matrices scales as  $O(MN)$ .

##### Note

The multiplication operator  $*$  also acts element-by-element. It does *not* refer to matrix multiplication!

For example,

```
>>> x = ones((2,3))
>>> y = ones((2,3))
>>> z = x * y
```

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

In index notation, we can think of the  $*$  operator as doing this:

$$z_{ij} = x_{ij} y_{ij}. \quad (4.1.7)$$

By contrast, matrix multiplication is  $z_{ij} = \sum_k x_{ik} y_{kj}$ . We'll see how this is accomplished in the next subsection.

### 4.1.5 The Dot Operation

The most commonly-used function for array multiplication is the `dot` function, which takes two array inputs  $x$  and  $y$  and returns their "dot product". It constructs a product by summing over the *last* index of array  $x$ , and over the *next-to-last* index of array  $y$  (or over its last index, if  $y$  is a 1D array). This may sound like a complicated rule, but you should be able to convince yourself that it corresponds to the appropriate type of multiplication operation for the most common cases encountered in linear algebra:

- If  $x$  and  $y$  are both 1D arrays (vectors), then `dot` corresponds to the usual dot product between two vectors:

$$z = \text{dot}(x, y) \leftrightarrow z = \sum_k x_k y_k \quad (4.1.8)$$

- If  $x$  is a 2D array and  $y$  is a 1D array, then `dot` corresponds to right-multiplying a matrix by a vector:

$$z = \text{dot}(x, y) \leftrightarrow z_i = \sum_k x_{ik} y_k \quad (4.1.9)$$

- If  $x$  is a 1D array and  $y$  is a 2D array, then `dot` corresponds to left-multiplication:

$$z = \text{dot}(x, y) \leftrightarrow z_i = \sum_k x_k y_{ki} \quad (4.1.10)$$

- If  $x$  and  $y$  are both 2D arrays, `dot` corresponds to matrix multiplication:

$$z = \text{dot}(x, y) \leftrightarrow z_{ij} = \sum_k x_{ik} y_{kj} \quad (4.1.11)$$

- The rule applies to higher-dimensional arrays as well. For example, two rank-3 tensors are multiplied together in this way:

$$z = \text{dot}(x, y) \leftrightarrow z_{ijpq} = \sum_k x_{ijk} y_{pkq} \quad (4.1.12)$$

Should you need to perform more general products than what the `dot` function provides, you can use the `tensordot` function. This takes two array inputs,  $x$  and  $y$ , and a tuple of two integers specifying which components of  $x$  and  $y$  to sum over. For example, if  $x$  and  $y$  are 2D arrays,

$$z = \text{tensordot}(x, y, (0, 1)) \leftrightarrow z_{ij} = \sum_k x_{ki} y_{jk} \quad (4.1.13)$$

What is the runtime for `dot` and `tensordot`? Consider a simple case: matrix multiplication of an  $M \times N$  matrix with an  $N \times P$  matrix. In index notation, this has the form

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj}, \quad \text{for } i \in \{0, \dots, M-1\}, \quad j \in \{0, \dots, P-1\}. \quad (4.1.14)$$

The resulting matrix has a total of  $(M \times P)$  indices to be computed. Each of these calculations requires a sum involving  $O(N)$  arithmetic operations. Hence, the total runtime scales as  $O(MNP)$ . By similar reasoning, we can figure out the runtime scaling for any tensor product between two tensors: it is the product of the sizes of the unsummed indices, times the size of the summed index. For example, for a `tensordot` product between an  $M \times N \times P$  tensor and a  $Q \times S \times P$  tensor, summing over the last index of each tensor, the runtime would scale as  $O(MNPQS)$ .

This page titled [4.1: Array Representations of Vectors, Matrices, and Tensors](#) is shared under a [CC BY-SA 4.0](#) license and was authored, remixed, and/or curated by [Y. D. Chong](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.