**10.3 Functions and Operations on Arrays**

NumPy has functions that take arrays as arguments, and operations that can be performed on entire arrays.

Functions

NumPy has ***universal functions*** (***ufuncs***) to which an entire array can be passed. The function is evaluated on every element, returning an array with the same dimensions as the original. These include trig functions such as **sin**, **cos**, **tan**, etc. For example:

from math import pi

x = np.linspace(0, 2\*pi, 10)

y = np.sin(x)

print(x)

print(y)

[0. 0.6981317 1.3962634 2.0943951 2.7925268 3.4906585

4.1887902 4.88692191 5.58505361 6.28318531]

[ 0.00000000e+00 6.42787610e-01 9.84807753e-01 8.66025404e-01

3.42020143e-01 -3.42020143e-01 -8.66025404e-01 -9.84807753e-01

-6.42787610e-01 -2.44929360e-16]

Other ufuncs include the square root function **sqrt**, the absolute value function **abs**, and exponential functions. For example, we could get the absolute value of every element in a 2D array:

rng = np.random.default\_rng()

nums = rng.integers(-10,10, (3,5))

print(nums)

print()

print(np.abs(nums))

[[ 1 -1 7 -10 3]

[ 6 2 6 -5 -3]

[ -1 1 -10 -6 9]]

[[ 1 1 7 10 3]

[ 6 2 6 5 3]

[ 1 1 10 6 9]]

There are ***aggregation functions*** that perform tasks such as summing elements, multiplying elements, and finding minimum and maximum values. By default, the aggregation is performed on the entire array, but to aggregate on just rows or columns, the axis can be specified.

For example, let’s first create a 2D array:

>>> rng = np.random.default\_rng()

>>> nums = rng.integers(60, 100, (4,2))

>>> print(nums)

[[96 91]

[68 64]

[60 97]

[77 77]]

To find the overall minimum, the **min** function is used:

>>> np.min(nums)

60

To find the minimum for each column, specify axis 0:

>>> np.min(nums, axis = 0)

array([60, 64])

To find the minimum for each row, specify axis 1:

>>> smin = np.min(nums, axis = 1)

>>> smin

array([91, 64, 60, 77])

Other unfuncs that work the same way (overall, or specify the axis) include **max**, **mean**, **median**, **std** (standard deviation), and **var** (variance).

The aggregation functions also include **sum** and **prod**. For example, to get an overall sum:

>>> rng = np.random.default\_rng()

>>> nums = rng.integers(0,10, (3,4))

>>> print(nums)

[[3 9 7 3]

[1 0 2 8]

[2 0 0 1]]

>>> np.sum(nums)

36

To get a sum for every column:

>>> np.sum(nums, axis = 0)

array([ 6, 9, 9, 12])

Operations

Numerical operations can be performed on entire arrays. ***Scalar operations*** involve an array and one scalar. For example, to add 5 to every element in an array:

>>> rowvec = array([-3, 28, 6])

>>> rowvec + 5

array([ 2, 33, 11])

Operations such as addition, subtraction, multiplication, division, and exponentiation can be performed on arrays of any dimension. For example, to divide every element in a 2D array by 2:

rng = np.random.default\_rng()

nums = rng.integers(10,20, (2,5))

print(nums)

print()

print(nums/2)

[[18 17 13 11 15]

[14 12 13 12 16]]

[[9. 8.5 6.5 5.5 7.5]

[7. 6. 6.5 6. 8. ]]

***Array operations*** are operations that are performed element by element on two arrays that have the same dimensions.

For example, the following creates two 1 x 5 arrays, or vectors, and adds the corresponding elements to create another 1 x 5 array. This is array addition.

>>> vec1 = array([6, 2, 7, 5, 9])

>>> vec2 = array([3, 1, 4, 2, 1])

>>> vec1 + vec2

array([ 9, 3, 11, 7, 10])

Similarly, array multiplication is an element-by-element operation:

rng = np.random.default\_rng()

mat1 = rng.integers(0,10, (2,3))

print(mat1)

print()

mat2 = rng.integers(0,10, (2,3))

print(mat2)

print()

print(mat1\*mat2)

[[3 9 3]

[4 0 0]]

[[4 2 6]

[8 1 3]]

[[12 18 18]

[32 0 0]]

Note: this is array multiplication, not matrix multiplication. Matrix multiplication is not performed element-by-element, and will be described in the next section.

Matrix Operations and Matrix Properties

***Matrix multiplication*** has a very specific meaning. By “matrix” here we are referring to 2D arrays. First of all, to multiply a matrix A by a matrix B to result in a matrix C, the number of columns of A must be the same as the number of rows of B. If the matrix A has dimensions *m x n*, that means that matrix B must have dimensions *n x* *something*; called *p*.

The terminology is that the ***inner dimensions*** (the *n*s) must be the same. The resulting matrix C has the same number of rows as A and the same number of columns as B (i.e., the ***outer dimensions*** *m x p*). In mathematical notation,

[A]*m x n* [B]*n x p* = [C]*m x p*

This only defines the size of C, not how to find the elements of C.

The elements of the matrix C are defined as the sum of products of corresponding elements in the rows of A and columns of B, or in other words,

cij = .

In the following example, A is *2 x 3* and B is *3 x 4*; the inner dimensions are both 3, so performing the matrix multiplication A\*B is possible (note that B\*A would not be possible). C will have as its size the outer dimensions *2 x 4*. The elements in C are obtained using the summation just described. The first row of C is obtained using the first row of A and in succession the columns of B. For example, C(1,1) is 3\*1+8\*4+0\*0 or 35. C(1,2) is 3\*2+8\*5+0\*2 or 46.

A B C

 \* 

In NumPy, the matrix multiplication operator is @.

>>> A = array([[3, 8, 0],[1, 2, 5]])

>>> B = array([[1, 2, 3, 1], [4, 5, 1, 2], [0, 2, 3, 0]])

>>> C = A @ B

>>> print(C)

[[35 46 17 19]

[ 9 22 20 5]]

Properties of Square Matrices

A ***square matrix*** is a matrix in which the number of rows is the same as the number of columns. The definitions that follow in this section only apply to square matrices.

The ***diagonal*** of a square matrix is the set of terms aii for which the row and column indices are the same, so from the upper left element to the lower right. For example, for the following matrix the diagonal consists of 1, 6, 11, and 16.



A square matrix is a ***diagonal matrix*** if all values that are not on the diagonal are 0. The numbers on the diagonal, however, do not have to be all nonzero, although frequently they are. Mathematically, this is written as aij = 0 for i ~= j. The following is an example of a diagonal matrix:



The NumPy **diag** function can be used to create a diagonal matrix, by passing the numbers to be on the diagonal (either as a 1D array or a list).

>>> diagnums = [4, 5, 2, 7]

>>> np.diag(diagnums)

array([[4, 0, 0, 0],

[0, 5, 0, 0],

[0, 0, 2, 0],

[0, 0, 0, 7]])

If instead a diagonal matrix is passed to the **diag** function, it returns the diagonal as a 1D array.

>>> myd = np.diag([33, 2, 11])

>>> np.diag(myd)

array([33, 2, 11])

So, the **diag** function can be used two ways: (i) pass a matrix and it returns a vector, or (ii) pass a vector and it returns a matrix!

A square matrix is an ***identity*** matrix called I if *aij* = 1 for *i* == *j* and *aij*= 0 for *i* ~= *j*. In other words, all of the numbers on the diagonal are 1 and all others are 0. The following is a *3 x 3* identity matrix:



Note that any identity matrix is a special case of a diagonal matrix.

Identity matrices are very important and useful. NumPy has a built-in function **eye** that will create an *n x n* identity matrix, given the value of *n*:

>>> np.eye(4)

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

[0., 1., 0., 0.],

[0., 0., 1., 0.],

[0., 0., 0., 1.]])

A square matrix is ***symmetric*** if *aij = aji* for all *i*, *j*. In other words, all of the values opposite the diagonal from each other must be equal to each other. In this example, there are three pairs of values opposite the diagonals, all of which are equal (the 2s, the 9s, and the 4s).

**10.4 Assigning and Copying Array Variables**

NumPy arrays contain the actual data, and also ***metadata*** such as the data type.

Since array variables are mutable, assigning one array variable to another works like lists. No new object is created.

>>> vec1 = array([6, 2, 7, 5, 9])

>>> vec2 = vec1

>>> vec2 is vec1

True

So, changing one variable changes the other since they are both referring to the same array.

>>> vec1[2] = 11

>>> print(vec1)

>>> print(vec2)

[ 6 2 11 5 9]

[ 6 2 11 5 9]

Assigning a new array to either of the variables creates a completely new location.

>>> vec1 = np.arange(2,6)

>>> print(vec1)

>>> print(vec2)

[2 3 4 5]

[ 6 2 11 5 9]

If it is desired to have two array variables store the same values, but not to refer to the same lo-cation, the **copy** function can be used instead of the assignment operator.

This creates what is called a ***deep copy***, and copies all of the metadata.

>>> vec1 = array([6, 2, 7, 5, 9])

>>> vec2 = np.copy(vec1)

>>> vec2 is vec1

False