# Numpy Basics

Jasmine Jinwal edited this page 14 days ago · [2 revisions](https://github.com/JasmineJinwal/Project_MTech/wiki/Numpy-Basics/_history)

* ndarray.ndim the number of axes (dimensions) of the array.
* ndarray.shape the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the number of axes, ndim.
* ndarray.size the total number of elements of the array. This is equal to the product of the elements of shape.
* ndarray.dtype an object describing the type of the elements in the array. One can create or specify dtype’s using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.
* ndarray.itemsize the size in bytes of each element of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to ndarray.dtype.itemsize.
* ndarray.data the buffer containing the actual elements of the array. Normally, we won’t need to use this attribute because we will access the elements in an array using indexing facilities.

For example, you can create an array from a regular Python list or tuple using the array function. The type of the resulting array is deduced from the type of the elements in the sequences.

import numpy as np a = np.array([2,3,4]) -- write array in this format always a array([2, 3, 4]) a.dtype dtype('int64') b = np.array([1.2, 3.5, 5.1]) b.dtype dtype('float64')

np.zeros( (3,4) ) array([[ 0., 0., 0., 0.], [ 0., 0., 0., 0.], [ 0., 0., 0., 0.]]) np.ones( (2,3,4), dtype=np.int16 ) # dtype can also be specified array([[[ 1, 1, 1, 1], [ 1, 1, 1, 1], [ 1, 1, 1, 1]], [[ 1, 1, 1, 1], [ 1, 1, 1, 1], [ 1, 1, 1, 1]]], dtype=int16)

**Basic Operations**

Arithmetic operators on arrays apply elementwise. A new array is created and filled with the result.

a = np.array( [20,30,40,50] ) b = np.arange( 4 ) b array([0, 1, 2, 3]) c = a-b c array([20, 29, 38, 47]) b\*\*2 array([0, 1, 4, 9]) 10\*np.sin(a) array([ 9.12945251, -9.88031624, 7.4511316 , -2.62374854]) a<35 array([ True, True, False, False])

some operations, such as \***(+= and** =), act in place to modify an existing array rather than create a new one. When operating with arrays of different types, the type of the resulting array corresponds to the more general or precise one (a behavior known as upcasting).

Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the ndarray class.

a = np.random.random((2,3)) a array([[ 0.18626021, 0.34556073, 0.39676747], [ 0.53881673, 0.41919451, 0.6852195 ]]) a.sum() 2.5718191614547998 a.min() 0.1862602113776709 a.max() 0.6852195003967595

NumPy provides familiar mathematical functions such as sin, cos, and exp. In NumPy, these are called “universal functions”(ufunc). Within NumPy, these functions operate elementwise on an array, producing an array as output.

B = np.arange(3) B array([0, 1, 2]) np.exp(B) array([ 1. , 2.71828183, 7.3890561 ]) np.sqrt(B) array([ 0. , 1. , 1.41421356]) C = np.array([2., -1., 4.]) np.add(B, C) array([ 2., 0., 6.])

# What is Numpy ?

Jasmine Jinwal edited this page 11 days ago · [2 revisions](https://github.com/JasmineJinwal/Project_MTech/wiki/What-is-Numpy-?/_history)

NumPy is the 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.

At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

* NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
* The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python’s built-in sequences.

* A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today’s scientific/mathematical Python-based software, just knowing how to use Python’s built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

The points about sequence size and speed are particularly important in scientific computing. As a simple example, consider the case of each element in a 1-D sequence with the corresponding element in another sequence of the same length. If the data are stored in two Python lists, a and b, we could iterate over each element:

c = [] for i in range(len(a)): c.append(a[i]\*b[i])

This produces the correct answer, but if a and b each contain millions of numbers, we will pay the price for the inefficiencies of looping in Python. We could accomplish the same task much more quickly in C by writing (for clarity we neglect variable declarations and initializations, memory allocation, etc.)

for (i = 0; i < rows; i++): { c[i] = a[i]\*b[i]; }

This saves all the overhead involved in interpreting the Python code and manipulating Python objects, but at the expense of the benefits gained from coding in Python. Furthermore, the coding work required increases with the dimensionality of our data. In the case of a 2-D array, for example, the C code (abridged as before) expands to

for (i = 0; i < rows; i++): { for (j = 0; j < columns; j++): { c[i][j] = a[i][j]\*b[i][j]; } }

NumPy gives us the best of both worlds: element-by-element operations are the “default mode” when an ndarray is involved, but the element-by-element operation is speedily executed by pre-compiled C code. In NumPy

c = a \* b

does what the earlier examples do, at near-C speeds, but with the code simplicity we expect from something based on Python. Two of NumPy’s features which are the basis of much of its power: vectorization and broadcasting. Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just “behind the scenes” in optimized, pre-compiled C code. Vectorized code has many advantages, among which are:

* vectorized code is more concise and easier to read

fewer lines of code generally means fewer bugs

the code more closely resembles standard mathematical notation (making it easier, typically, to correctly code mathematical constructs)

vectorization results in more “Pythonic” code. Without vectorization, our code would be littered with inefficient and difficult to read for loops.

Broadcasting is the term used to describe the implicit element-by-element behavior of operations; generally speaking, in NumPy all operations, not just arithmetic operations, but logical, bit-wise, functional, etc., behave in this implicit element-by-element fashion, i.e., they broadcast. Moreover, in the example above, a and b could be multidimensional arrays of the same shape, or a scalar and an array, or even two arrays of with different shapes, provided that the smaller array is “expandable” to the shape of the larger in such a way that the resulting broadcast is unambiguous.