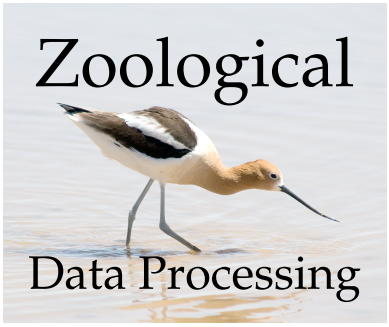
NumPy quick reference



John W. Shipman

2016-05-30 12:28

Abstract

A guide to the more common functions of *NumPy,* a numerical computation module for the Python programming language.

This publication is available in Web form[1] and also as a PDF document[2]. Please forward any comments to <john@nmt.edu>.

Table of Contents

[1] http://www.nmt.edu/~shipman/soft/numpy/
[2] http://www.nmt.edu/~shipman/soft/numpy/numpy.pdf
[3] http://creativecommons.org/licenses/by-nc/3.0/

1. Introduction

The purpose of Python's *NumPy* module is to bring Python's power and elegance to bear on mathematical and scientific problems.

Major parts of this product include:

- *NumPy* provides basic numerical functions, especially for multi-dimensional arrays and mathematical computation.
- *SciPy* builds on *NumPy* to provide features for scientific applications.
- *matplotlib* provides plotting and related graphic display capabilities. Its function set is intended to be quickly usable by people who are familiar with *Matlab*, although it is built entirely on top of Python and thus can be integrated with any other Python-based application.
- *ipython* is an enhanced conversational interface that allows you to interact with Python in parallel with a running application.

2. Online resources

- *Numpy and Scipy Documentation*[4]: This is the full official documentation set.
- *Guide to NumPy*[5] is a general introduction to *NumPy*.

3. Importing *NumPy*

The *NumPy* module provides hundreds of useful mathematical functions, as well as constants like `pi` (π) and `e` (the base of natural logarithms).

There are two ways to make these functions and constants available to your program.

- Section 3.1, "Adding *NumPy* to your namespace" (p. 2): for small programs and experimenting.
- Section 3.2, "Practicing safe namespace hygiene" (p. 3): this is how the professionals do it.

3.1. Adding *NumPy* to your namespace

If your program is primarily doing computation, and you don't use a lot of other modules, place this statement with the other `import` statements at the top of your script, and it will add all the names from the *NumPy* module to your namespace.

```
from numpy import *
```

You can then refer directly to the constants like `pi` and functions like `array()` and `arange()`. Example:

```
>>> from numpy import *
>>> print pi, e
3.14159265359 2.71828182846
>>> print arange(0.0, 1.0, 0.1)
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9]
```

[4] http://docs.scipy.org/doc/
[5] http://web.mit.edu/dvp/Public/numpybook.pdf

3.2. Practicing safe namespace hygiene

When you start developing larger applications, and especially when you start using multiple modules like *SciPy* and the `matplotlib` plotting system, it can become difficult to remember which functions came from which module.

The cure for this problem is to import the entire module and then refer to things inside the module using the syntax "`M.thing`" where `M` is the name of the module and `thing` is the name of the item within the module.

Place a line like this with your other `import` statements:

```
import numpy as np
```

Then you use `np.array()` to create an array, or use `np.arange()` to create an arithmetic progression array, and so forth.

In the rest of this document, we will assume that you use the second form of import. For example:

```
>>> import numpy as np
>>> print np.pi, np.e
3.14159265359 2.71828182846
>>> print np.arange(0.0, 1.0, 0.1)
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9]
```

The term *namespace hygiene* refers to the commendable practice of keeping items from different namespaces separate, so you can tell what comes from where.

4. Basic types

NumPy has a much richer set of data types than standard Python. The type object for a *NumPy* type is called a `dtype`.

Each type has a type name. This table shows the names and describes their functions. Seven of these types are functionally identical to standard Python types, so their names have an underscore (_) appended to differentiate them. The exact meanings may depend on the underlying hardware platform.

`bool_`	Boolean, `True` or `False`.
`byte`	One-byte signed integer.
`short`	C-language `short` integer.
`intc`	C-language `int` integer.
`int_`	Python `int`.
`int8`, `int16`, `int32`	Signed integers with 8, 16, and 32 bits respectively.
`longlong`	Signed arbitrary-length integer.
`intp`	An integer sized to hold a memory address.
`ubyte`	Unsigned 8-bit integer.
`ushort`	Unsigned C `short`.
`uintc`	Unsigned C `int`.
`uint`	Unsigned integer.
`uint8`, `uint16`, `uint32`	Unsigned integers of 8, 16, and 32 bits respectively.
`ulonglong`	Unsigned arbitrary-length integer.

`uintp`	Unsigned integer big enough to hold an address.
`single`	Single-precision float.
`float_`	Python `float`.
`longfloat`	Double-precision float.
`float32`, `float64`	Floats of 32 and 64 bits, respectively.
`csingle`	Single-precision complex.
`complex_`	Python `complex`.
`clongfloat`	Double-precision complex.
`object_`	For arrays of arbitrary Python objects.
`str_`	Eight-bit character string.
`unicode_`	Thirty-two-bit character string.

5. `ndarray`: The N-dimensional array

Use the `np.array()` constructor to create an array with any number of dimensions.

```
np.array(object, dtype=None)
```

`object`
Values to be used in creating the array. This can be a sequence (to create a 1-d array), a sequence of sequences (for a 2-d array), a sequence of sequences of sequences, and so on. Each sequence may be either a list or a tuple.

The `object` argument may also be an existing array. The new array will be a copy, and you can use the `dtype` argument to force the copy to be a different type.

`dtype`
To force the new array to be a given type, use one of the type words as the second argument. For example, `array([1,2,3], dtype=np.float_)` will give you an array with those three values converted to floating-point type.

5.1. One-dimensional arrays

To create a vector of values, use `np.array(S)`, where `S` is a sequence (list or tuple). Unlike Python lists, when you print an array, it doesn't show commas between the elements.

```
>>> import numpy as np
>>> d1=np.array([2.4, -1.5, 3.0, 8.8])
>>> print d1
[ 2.4 -1.5  3.   8.8]
```

To retrieve one value from a vector, use normal Python indexing: position 0 is the first element, position 2 is the third element, and so on.

```
>>> print d1[0]
2.4
>>> print d1[2]
3.0
```

You can use normal Python slicing on arrays as well.

```
>>> print d1[1:3]
[-1.5  3. ]
```

If you want to force the array to use a specific type, use the `dtype=D` keyword argument to `np.array()`, where `D` is one of the `dtype` type objects described in Section 4, "Basic types" (p. 3). In the example below, the first array will have `int` (integer) type, and the second one will have type `float`.

```
>>> print np.array([0, 1, 2, 3])
[0 1 2 3]
>>> print np.array([0, 1, 2, 3], dtype=np.float_)
[ 0.  1.  2.  3.]
```

If you don't have all the values together at once that you need to build an array, you can instead create an array of zeroes and fill the values in later. The argument to the `np.zeros()` function is a sequence containing the dimensions. In this example, we use `[6]` as the argument; this gives us a one-dimensional array with six zeros in it.

```
>>> z = np.zeros([6])
>>> print z
[ 0.  0.  0.  0.  0.  0.]
>>> z[3] = 46.4
>>> z[5] = 82.2
>>> print z
[  0.    0.    0.   46.4   0.   82.2]
```

5.2. The `arange()` function: Arithmetic progression

Use the `np.arange()` function to build a vector containing an arithmetic progression such as `[0.0 0.1 0.2 0.3]`.

```
>>> print np.arange(0.0, 0.4, 0.1)
[ 0.   0.1  0.2  0.3]
>>> print np.arange(5.0, -0.5, -0.5)
[ 5.   4.5  4.   3.5  3.   2.5  2.   1.5  1.   0.5  0. ]
```

Here is the general form:

```
np.arange(start, stop=None, step=1, dtype=None)
```

`start`
: The first value in the sequence.

`stop`
: The limiting value: the last element of the sequence will never be greater than *or equal to* this value (assuming that the `step` value is positive; for negative `step` values, the last element of the sequence will always be greater than the `stop` value).

```
>>> print np.arange(1.0, 4.0)
[ 1.  2.  3.]
```

If you omit the `stop` value, you will get a sequence starting at zero and using `start` as the limiting value.

```
>>> print np.arange(4)
[0 1 2 3]
```

`step`
: The common difference between successive values of the array. The default value is one.

dtype
Use this argument to force representation using a specific type.

```
>>> print np.arange(10)
[0 1 2 3 4 5 6 7 8 9]
>>> print np.arange(10, dtype=np.float_)
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
```

5.3. Two-dimensional arrays

To create a two-dimensional array (matrix), use `np.array()` as demonstrated above, but use a sequence of sequences to provide the values.

```
>>> d2 = np.array([(0, 1, 2, 3), (4, 5, 6, 7), (8, 9, 10, 11)])
>>> print d2
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

To retrieve a value from a matrix *M*, use an expression of the form *M*`[`*row*`, `*col*`]` where *row* is the row position and `col` is the column position.

```
>>> print d2[0,2]
2
>>> print d2[2, 3]
11
```

You can use slicing to get one row or column. A slice operation has this general form:

```
M[rows, cols]
```

In this form, *rows* and *cols* may be either regular Python slice operations (such as `2:5` to select the third through fifth items), or they may be just ":" to select all the elements in that dimension.

In this example, we extract a 2×3 submatrix, containing rows 0 and 1, and columns 0, 1, and 2.

```
>>> print d2[0:2, 0:3]
[[0 1 2]
 [4 5 6]]
```

This example extracts all the rows, but only the first three columns.

```
>>> print d2[:,0:3]
[[ 0  1  2]
 [ 4  5  6]
 [ 8  9 10]]
```

In this example we select all the columns, but only the first two rows.

```
>>> print d2[0:2,:]
[[0 1 2 3]
 [4 5 6 7]]
```

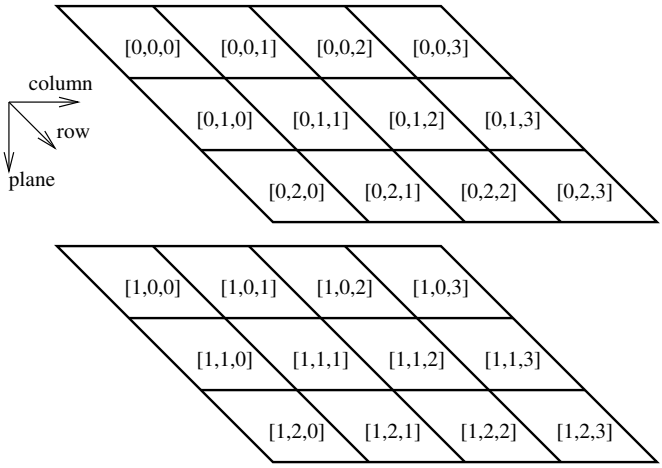
You can use the `np.zeros()` function to create an empty matrix. The argument is a sequence (list or tuple) of the dimensions; we'll use a tuple this time.

```
>>> z2 = np.zeros((2,7))
>>> print z2
```

```
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]]
```

5.4. Three or more dimensions

The figure below shows the numbering of indices of a 3-d array. The column index is always last. The row index precedes the column index for arrays of two or more dimensions. For a 3-d array, the "plane" index is the first index.



Here is a conversational example of the creation of an array shaped like the above illustration.

```
>>> import numpy as np
>>> d3 = np.array(
...  [[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]],
...   [[12,13,14,15], [16,17,18,19], [20,21,22,23]]])
>>> print d3
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

To extract one element from a three-dimensional array, use an expression of this form:

```
A[plane, row, col]
```

For example:

```
>>> d3[0,0,0]
0
>>> d3[0,0,1]
1
>>> d3[0,1,0]
4
>>> d3[1,0,0]
12
>>> d3[1,2,3]
23
```

Slicing generalizes to any number of dimensions. For example:

```
>>> print d3
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
>>> d3[:,:,1:3]
array([[[ 1,  2],
        [ 5,  6],
        [ 9, 10]],

       [[13, 14],
        [17, 18],
        [21, 22]]])
```

5.5. Array attributes

These attributes are available from any `ndarray`.

`.ndim`
The number of dimensions of this array.

`.shape`
A tuple of the array's dimensions.

`.dtype`
The array's type as a `dtype` instance.

```
>>> print d1
[ 2.4 -1.5  3.   8.8]
>>> print d1.ndim, d1.shape, d1.dtype
1 (4,) float64
>>> print d2
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> print d2.ndim, d2.shape, d2.dtype
2 (3, 4) int32
>>> print d3
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
>>> print d3.ndim, d3.shape, d3.dtype
3 (2, 3, 4) int32
```

5.6. Array methods

You may find these methods on `ndarray` instances to be useful.

`A.astype(T)`
Creates a new array with the elements from `A`, but as type `T`, where `T` is one of the `dtype` values discussed in Section 4, "Basic types" (p. 3).

```
>>> s1 = np.arange(5, 10)
>>> print s1
[5 6 7 8 9]
>>> s2 = s1.astype(np.float_)
>>> print s2
[ 5.  6.  7.  8.  9.]
```

`A.copy()`
Creates a new `ndarray` as an exact copy of `A`.

```
>>> s3 = s2.copy()
>>> s2[2] = 73.88
>>> print s2
[  5.     6.    73.88   8.     9.  ]
>>> print s3
[ 5.  6.  7.  8.  9.]
```

`A.reshape(dims)`
Returns a new array that is a copy of the values `A` but having a shape given by `dims`.

```
>>> a1 = np.arange(0.0, 12.0, 1.0)
>>> print a1
[  0.   1.   2.   3.   4.   5.   6.   7.   8.   9.  10.  11.]
>>> a2 = a1.reshape( (2,6) )
>>> print a2
[[  0.   1.   2.   3.   4.   5.]
 [  6.   7.   8.   9.  10.  11.]]
```

`A.resize(dims)`
Changes the shape of array `A`, but does so *in place*.

```
print a1
[  0.   1.   2.   3.   4.   5.   6.   7.   8.   9.  10.  11.]
>>> a1.resize([2,6])
>>> print a1
[[  0.   1.   2.   3.   4.   5.]
 [  6.   7.   8.   9.  10.  11.]]
```

`A.mean()`
Returns the mean of the values in `A`.

`A.var()`
Returns the variance of the values in `A`.

```
>>> print a1.mean(), a1.var()
5.5 11.9166666667
```

There are many other array methods; these are just some of the more common ones.

6. Universal functions (*ufuncs*)

The usual mathematical operators (`+ - * /`) generalize to *NumPy* arrays, as well as a number of *ufuncs* (universal functions) defined by *NumPy*.

For example, to add two vectors `v1` and `v2` of the same length, the "+" operator gives you an element-by-element sum.

```
>>> v1 = np.arange(0.6, 1.6, 0.1)
>>> print v1
[ 0.6  0.7  0.8  0.9  1.   1.1  1.2  1.3  1.4  1.5]
>>> v2 = np.arange(40.0, 50.0, 1.0)
>>> print v2
[ 40.  41.  42.  43.  44.  45.  46.  47.  48.  49.]
>>> print v1+v2
[ 40.6  41.7  42.8  43.9  45.   46.1  47.2  48.3  49.4  50.5]
```

The other common operators generalize in the same way.

```
>>> print v1*v2
[ 24.   28.7  33.6  38.7  44.   49.5  55.2  61.1  67.2  73.5]
>>> print v1-v2
[-39.4 -40.3 -41.2 -42.1 -43.  -43.9 -44.8 -45.7 -46.6 -47.5]
>>> print v1**2
[ 0.36  0.49  0.64  0.81  1.    1.21  1.44  1.69  1.96  2.25]
```

You can also use the "+" operator to add a constant value to every element of an array. This is called *broadcasting*.

```
>>> print v1
[ 0.6  0.7  0.8  0.9  1.   1.1  1.2  1.3  1.4  1.5]
>>> print v1 + 0.4
[ 1.   1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9]
```

All the usual Python mathematical operators will broadcast across arrays.

```
>>> print v1 * 10
[  6.   7.   8.   9.  10.  11.  12.  13.  14.  15.]
>>> print v1*10+100
[ 106.  107.  108.  109.  110.  111.  112.  113.  114.  115.]
```

In addition, these *NumPy* functions can be used on arrays, either to operate element-by-element or to broadcast values.

`np.abs(`*a*`)`	Absolute value.
`np.arccos(`*a*`)`	Inverse cosine.
`np.arcsin(`*a*`)`	Inverse sine.
`np.arctan(`*a*`)`	Inverse tangent.
`np.arctan2(`*y*`, `*x*`)`	Computes the arctangent of the slope whose Δy is y and whose Δx is x.
`np.cos(`*a*`)`	Cosine.
`np.exp(`*a*`)`	Exponential, e^a.
`np.log(`*a*`)`	Natural log.
`np.log10(`*a*`)`	Common log (base 10).
`np.sin(`*a*`)`	Sine.
`np.sqrt(`*a*`)`	Square root.
`np.tan(`*a*`)`	Tangent.

Examples:

```
>>> print np.abs(np.arange(-4, 5))
[4 3 2 1 0 1 2 3 4]
>>> angles = np.arange(0.0, np.pi*9.0/4.0, np.pi/4.0)
>>> print angles/np.pi
[ 0.    0.25  0.5   0.75  1.    1.25  1.5   1.75  2.  ]
>>> print np.sin(angles)
[  0.00000000e+00   7.07106781e-01   1.00000000e+00   7.07106781e-01
   1.22460635e-16  -7.07106781e-01  -1.00000000e+00  -7.07106781e-01
  -2.44921271e-16]
>>> print np.cos(angles)
[  1.00000000e+00   7.07106781e-01   6.12303177e-17  -7.07106781e-01
  -1.00000000e+00  -7.07106781e-01  -1.83690953e-16   7.07106781e-01
   1.00000000e+00]
>>> print np.tan(angles)
[  0.00000000e+00   1.00000000e+00   1.63317787e+16  -1.00000000e+00
  -1.22460635e-16   1.00000000e+00   5.44392624e+15  -1.00000000e+00
  -2.44921271e-16]
>>> deltaYs = np.array((0, 1, 0, -1))
>>> deltaXs = np.array((1, 0, -1, 0))
>>> quadrants = np.arctan2(deltaYs, deltaXs)
>>> print quadrants
[ 0.          1.57079633  3.14159265 -1.57079633]
>>> print quadrants/np.pi
[ 0.   0.5  1.  -0.5]
```

7. Dot and cross products

To find the matrix (dot) product of two arrays *a1* and *a2*, use the function `np.dot(a1, a2)`.

```
>>> a1 = np.array([[1, 2, -4], [3, -1, 5]])
>>> a2 = np.array([[6, -3], [1, -2], [2, 4]])
>>> print a1
[[ 1  2 -4]
 [ 3 -1  5]]
>>> print a2
[[ 6 -3]
 [ 1 -2]
 [ 2  4]]
>>> np.dot(a1, a2)
array([[  0, -23],
       [ 27,  13]])
```

Similarly, `np.cross(x, y)` returns the cross-product of vectors *x* and *y*.

```
>>> x = (1, 2, 0)
>>> y = (4, 5, 6)
>>> print np.cross(x, y)
[12 -6 -3]
```

8. Linear algebra functions

A number of linear algebra functions are available as sub-module `linalg` of `numpy`.

`np.linalg.det(a)`
Returns the determinant of a 2-d array *a*.

```
>>> m = np.array(((2,3), (-1, -2)))
>>> print m
[[ 2  3]
 [-1 -2]]
>>> print np.linalg.det(m)
-1.0
```

np.linalg.inv(*a*)
Returns the matrix inverse of a non-singular 2-d array *a*.

```
>>> good = np.array(((2.1, 3.2), (4.3, 5.4)))
>>> print np.linalg.inv(good)
[[-2.23140496  1.32231405]
 [ 1.7768595  -0.8677686 ]]
```

np.linalg.norm(*a*)
Returns the Frobenius norm of array *a*.

```
>>> print good
[[ 2.1  3.2]
 [ 4.3  5.4]]
>>> print np.linalg.norm(good)
7.89303490427
```

np.linalg.solve(*A*, *b*)
Solves systems of simultaneous linear equations. Given an N×N array of coefficients *A*, and a length-N vector of constants *b*, returns a length-N vector containing the solved values of the variables.

Here's an example system:

```
    2x +  y = 19
     x - 2y = 2
```

Here's a conversational example showing the solution (x=8 and y=3):

```
>>> coeffs = np.array([[2,1], [1,-2]])
>>> print coeffs
[[ 2  1]
 [ 1 -2]]
>>> consts = np.array((19, 2))
>>> print consts
[19  2]
>>> print np.linalg.solve(coeffs, consts)
[ 8.  3.]
```