Search the docs ...

What is NumPy?
Installation
NumPy quickstart
NumPy: the absolute basics for beginners
NumPy fundamentals
Array creation
Indexing on `ndarrays`
I/O with NumPy
Data types
Broadcasting
Byte-swapping
Structured arrays
Writing custom array containers
Subclassing ndarray
Universal functions (`ufunc`) basics
Copies and views
Interoperability with NumPy
Miscellaneous
NumPy for MATLAB users
Building from source
Using NumPy C-API
NumPy Tutorials
NumPy How Tos
For downstream package authors
F2PY user guide and reference manual
Glossary
Under-the-hood Documentation for developers
Reporting bugs
Release notes
NumPy license

Array creation

See also

Array creation routines

Introduction

There are 6 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)

You can use these methods to create ndarrays or Structured arrays. This document will cover general methods for ndarray creation.

1) Converting Python sequences to NumPy Arrays

NumPy arrays can be defined using Python sequences such as lists and tuples. Lists and tuples are defined using `[...]` and `(...)`, respectively. Lists and tuples can define ndarray creation:

- a list of numbers will create a 1D array,
- a list of lists will create a 2D array,
- further nested lists will create higher-dimensional arrays. In general, any array object is called an **ndarray** in NumPy.

```
>>> a1D = np.array([1, 2, 3, 4])
>>> a2D = np.array([[1, 2], [3, 4]])
>>> a3D = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

When you use **`numpy.array`** to define a new array, you should consider the dtype of the elements in the array, which can be specified explicitly. This feature gives you more control over the underlying data structures and how the elements are handled in C/C++ functions. If you are not careful with `dtype` assignments, you can get unwanted overflow, as such

```
>>> a = np.array([127, 128, 129], dtype=np.int8)
>>> a
array([ 127, -128, -127], dtype=int8)
```

An 8-bit signed integer represents integers from -128 to 127. Assigning the `int8` array to integers outside of this range results in overflow. This feature can often be misunderstood. If you perform calculations with mismatching `dtypes`, you can get unwanted results, for example:

```
>>> a = np.array([2, 3, 4], dtype=np.uint32)
>>> b = np.array([5, 6, 7], dtype=np.uint32)
>>> c_unsigned32 = a - b
>>> print('unsigned c:', c_unsigned32, c_unsigned32.dtype)
unsigned c: [4294967293 4294967293 4294967293] uint32
>>> c_signed32 = a - b.astype(np.int32)
>>> print('signed c:', c_signed32, c_signed32.dtype)
signed c: [-3 -3 -3] int64
```

Notice when you perform operations with two arrays of the same dtype: uint32, the resulting array is the same type. When you perform operations with different dtype, NumPy will assign a new type that satisfies all of the array elements involved in the computation, here uint32 and int32 can both be represented in as int64.

The default NumPy behavior is to create arrays in either 32 or 64-bit signed integers (platform dependent and matches C int size) or double precision floating point numbers, int32/int64 and float, respectively. If you expect your integer arrays to be a specific type, then you need to specify the dtype while you create the array.

2) Intrinsic NumPy array creation functions

NumPy has over 40 built-in functions for creating arrays as laid out in the Array creation routines. These functions can be split into roughly three categories, based on the dimension of the array they create:

1. 1D arrays
2. 2D arrays
3. ndarrays

1 - 1D array creation functions

The 1D array creation functions e.g. **numpy.linspace** and **numpy.arange** generally need at least two inputs, start and stop.

numpy.arange creates arrays with regularly incrementing values. Check the documentation for complete information and examples. A few examples are shown:

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(2, 10, dtype=float)
array([2., 3., 4., 5., 6., 7., 8., 9.])
>>> np.arange(2, 3, 0.1)
array([2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])
```

Note: best practice for **numpy.arange** is to use integer start, end, and step values. There are some subtleties regarding dtype. In the second example, the dtype is defined. In the third example, the array is dtype=float to accommodate the step size of 0.1. Due to roundoff error, the stop value is sometimes included.

numpy.linspace will create arrays with a specified number of elements, and spaced equally between the specified beginning and end values. For example:

```
>>> np.linspace(1., 4., 6)
array([1. ,  1.6,  2.2,  2.8,  3.4,  4. ])
```

The advantage of this creation function is that you guarantee the number of elements and the starting and end point. The previous arange(start, stop, step) will not include the value stop.

2 - 2D array creation functions

The 2D array creation functions e.g. **numpy.eye**, **numpy.diag**, and **numpy.vander** define properties of special matrices represented as 2D arrays.

np.eye(n, m) defines a 2D identity matrix. The elements where i=j (row index and column index are equal) are 1 and the rest are 0, as such:

```
>>> np.eye(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> np.eye(3, 5)
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.]])
```

numpy.diag can define either a square 2D array with given values along the diagonal *or* if given a 2D array returns a 1D array that is only the diagonal elements. The two array creation functions can be helpful while doing linear algebra, as such:

```
>>> np.diag([1, 2, 3])
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
>>> np.diag([1, 2, 3], 1)
array([[0, 1, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 3],
       [0, 0, 0, 0]])
>>> a = np.array([[1, 2], [3, 4]])
>>> np.diag(a)
array([1, 4])
```

`vander(x, n)` defines a Vandermonde matrix as a 2D NumPy array. Each column of the Vandermonde matrix is a decreasing power of the input 1D array or list or tuple, `x` where the highest polynomial order is `n-1`. This array creation routine is helpful in generating linear least squares models, as such:

```
>>> np.vander(np.linspace(0, 2, 5), 2)
array([[0. , 1. ],
      [0.5, 1. ],
      [1. , 1. ],
      [1.5, 1. ],
      [2. , 1. ]])
>>> np.vander([1, 2, 3, 4], 2)
array([[1, 1],
       [2, 1],
       [3, 1],
       [4, 1]])
>>> np.vander((1, 2, 3, 4), 4)
array([[ 1,  1,  1,  1],
       [ 8,  4,  2,  1],
       [27,  9,  3,  1],
       [64, 16,  4,  1]])
```

3 - general ndarray creation functions

The ndarray creation functions e.g. **numpy.ones**, **numpy.zeros**, and **random** define arrays based upon the desired shape. The ndarray creation functions can create arrays with any dimension by specifying how many dimensions and length along that dimension in a tuple or list.

numpy.zeros will create an array filled with 0 values with the specified shape. The default dtype is `float64`:

```
>>> np.zeros((2, 3))
array([[0., 0., 0.],
       [0., 0., 0.]])
>>> np.zeros((2, 3, 2))
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],

       [[0., 0.],
        [0., 0.],
        [0., 0.]]])
```

numpy.ones will create an array filled with 1 values. It is identical to `zeros` in all other respects as such:

```
>>> np.ones((2, 3))
array([[1., 1., 1.],
       [1., 1., 1.]])
>>> np.ones((2, 3, 2))
array([[[1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.]]])
```

The **random** method of the result of `default_rng` will create an array filled with random values between 0 and 1. It is included with the **numpy.random** library. Below, two arrays are created with shapes (2,3) and (2,3,2), respectively. The seed is set to 42 so you can reproduce these pseudorandom numbers:

```
>>> from numpy.random import default_rng
>>> default_rng(42).random((2,3))
array([[0.77395605, 0.43887844, 0.85859792],
       [0.69736803, 0.09417735, 0.97562235]])
>>> default_rng(42).random((2,3,2))
array([[[0.77395605, 0.43887844],
        [0.85859792, 0.69736803],
        [0.09417735, 0.97562235]],
       [[0.7611397 , 0.78606431],
        [0.12811363, 0.45038594],
        [0.37079802, 0.92676499]]])
```

numpy.indices will create a set of arrays (stacked as a one-higher dimensioned array), one per dimension with each representing variation in that dimension:

```
>>> np.indices((3,3))
array([[[0, 0, 0],
        [1, 1, 1],
        [2, 2, 2]],
       [[0, 1, 2],
        [0, 1, 2],
        [0, 1, 2]]])
```

This is particularly useful for evaluating functions of multiple dimensions on a regular grid.

3) Replicating, joining, or mutating existing arrays

Once you have created arrays, you can replicate, join, or mutate those existing arrays to create new arrays. When you assign an array or its elements to a new variable, you have to explicitly **numpy.copy** the array, otherwise the variable is a view into the original array. Consider the following example:

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
>>> b = a[:2]
>>> b += 1
>>> print('a =', a, '; b =', b)
a = [2 3 3 4 5 6] ; b = [2 3]
```

In this example, you did not create a new array. You created a variable, `b` that viewed the first 2 elements of `a`. When you added 1 to `b` you would get the same result by adding 1 to `a[:2]`. If you want to create a *new* array, use the **numpy.copy** array creation routine as such:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = a[:2].copy()
>>> b += 1
>>> print('a = ', a, 'b = ', b)
a =  [1 2 3 4] b =  [2 3]
```

For more information and examples look at Copies and Views.

There are a number of routines to join existing arrays e.g. **numpy.vstack**, **numpy.hstack**, and **numpy.block**. Here is an example of joining four 2-by-2 arrays into a 4-by-4 array using block:

```
>>> A = np.ones((2, 2))
>>> B = np.eye(2, 2)
>>> C = np.zeros((2, 2))
>>> D = np.diag((-3, -4))
>>> np.block([[A, B], [C, D]])
array([[ 1.,  1.,  1.,  0.],
       [ 1.,  1.,  0.,  1.],
       [ 0.,  0., -3.,  0.],
       [ 0.,  0.,  0., -4.]])
```

Other routines use similar syntax to join ndarrays. Check the routine's documentation for further examples and syntax.

4) Reading arrays from disk, either from standard or custom formats

This is the most common case of large array creation. The details depend greatly on the format of data on disk. This section gives general pointers on how to handle various formats. For more detailed examples of IO look at How to Read and Write files.

Standard Binary Formats

Various fields have standard formats for array data. The following lists the ones with known Python libraries to read them and return NumPy arrays (there may be others for which it is possible to read and convert to NumPy arrays so check the last section as well)

```
HDF5: h5py
FITS: Astropy
```

Examples of formats that cannot be read directly but for which it is not hard to convert are those formats supported by libraries like PIL (able to read and write many image formats such as jpg, png, etc).

Common ASCII Formats

Delimited files such as comma separated value (csv) and tab separated value (tsv) files are used for programs like Excel and LabView. Python functions can read and parse these files line-by-line. NumPy has two standard routines for importing a file with delimited data **numpy.loadtxt** and **numpy.genfromtxt**. These functions have more involved use cases in Reading and writing files. A simple example given a simple.csv:

```
$ cat simple.csv
x, y
0, 0
1, 1
2, 4
3, 9
```

Importing simple.csv is accomplished using loadtxt:

```
>>> np.loadtxt('simple.csv', delimiter = ',', skiprows = 1)
array([[0., 0.],
       [1., 1.],
       [2., 4.],
       [3., 9.]])
```

More generic ASCII files can be read using **scipy.io** and Pandas.

5) Creating arrays from raw bytes through the use of strings or buffers

There are a variety of approaches one can use. If the file has a relatively simple format then one can write a simple I/O library and use the NumPy `fromfile()` function and `.tofile()` method to read and write NumPy arrays directly (mind your byteorder though!) If a good C or C++ library exists that read the data, one can wrap that library with a variety of techniques though that certainly is much more work and requires significantly more advanced knowledge to interface with C or C++.

6) Use of special library functions (e.g., SciPy, Pandas, and OpenCV)

NumPy is the fundamental library for array containers in the Python Scientific Computing stack. Many Python libraries, including SciPy, Pandas, and OpenCV, use NumPy ndarrays as the common format for data exchange, These libraries can create, operate on, and work with NumPy arrays.

Previous
NumPy fundamentals

Next
Indexing on `ndarrays`