NumPy User Guide

Release 1.17.0rc1

Written by the NumPy community

July 12, 2019

CONTENTS

This guide is intended as an introductory overview of NumPy and explains how to install and make use of the most important features of NumPy. For detailed reference documentation of the functions and classes contained in the package, see the reference.

CHAPTER

ONE

SETTING UP

1.1 What is NumPy?

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 multiplying 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. Indeed, the NumPy idiom is even simpler! This last example illustrates two of NumPy's features which are the basis of much of its power: vectorization and broadcasting.

1.1.1 Why is NumPy Fast?

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. For detailed "rules" of broadcasting see *`numpy.doc.broadcasting`*.

1.1.2 Who Else Uses NumPy?

NumPy fully supports an object-oriented approach, starting, once again, with *ndarray*. For example, *ndarray* is a class, possessing numerous methods and attributes. Many of its methods are mirrored by functions in the outer-most NumPy namespace, allowing the programmer to code in whichever paradigm they prefer. This flexibility has allowed the NumPy array dialect and NumPy *ndarray* class to become the *de-facto* language of multi-dimensional data interchange used in Python.

1.2 Installing NumPy

In most use cases the best way to install NumPy on your system is by using a pre-built package for your operating system. Please see https://scipy.org/install.html for links to available options.

For instructions on building for source package, see *Building from source*. This information is useful mainly for advanced users.

CHAPTER

TWO

QUICKSTART TUTORIAL

2.1 Prerequisites

Before reading this tutorial you should know a bit of Python. If you would like to refresh your memory, take a look at the Python tutorial.

If you wish to work the examples in this tutorial, you must also have some software installed on your computer. Please see https://scipy.org/install.html for instructions.

2.2 The Basics

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In NumPy dimensions are called *axes*.

For example, the coordinates of a point in 3D space `[1, 2, 1]` has one axis. That axis has 3 elements in it, so we say it has a length of 3. In the example pictured below, the array has 2 axes. The first axis has a length of 2, the second axis has a length of 3.

```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

NumPy's array class is called `ndarray`. It is also known by the alias `array`. Note that `numpy.array` is not the same as the Standard Python Library class `array.array`, which only handles one-dimensional arrays and offers less functionality. The more important attributes of an `ndarray` object are:

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.

2.2.1 An example

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<type 'numpy.ndarray'>
```

2.2.2 Array Creation

There are several ways to create arrays.

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])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

A frequent error consists in calling `array` with multiple numeric arguments, rather than providing a single list of numbers as an argument.

```
>>> a = np.array(1,2,3,4)    # WRONG
>>> a = np.array([1,2,3,4])  # RIGHT
```

`array` transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

The type of the array can also be explicitly specified at creation time:

```
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

The function `zeros` creates an array full of zeros, the function `ones` creates an array full of ones, and the function `empty` creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array is `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)
>>> np.empty( (2,3) )                                 # uninitialized, output may vary
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```

To create sequences of numbers, NumPy provides a function analogous to `range` that returns arrays instead of lists.

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> np.arange( 0, 2, 0.3 )                 # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

When `arange` is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function `linspace` that receives as an argument the number of elements that we want, instead of the step:

```
>>> from numpy import pi
>>> np.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
>>> x = np.linspace( 0, 2*pi, 100 )        # useful to evaluate function at lots of␣
↪points
>>> f = np.sin(x)
```

See also:

`array`, `zeros`, `zeros_like`, `ones`, `ones_like`, `empty`, `empty_like`, `arange`, `linspace`, `numpy.random.rand`, `numpy.random.randn`, `fromfunction`, `fromfile`

2.2.3 Printing Arrays

When you print an array, NumPy displays it in a similar way to nested lists, but with the following layout:

- the last axis is printed from left to right,

- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

One-dimensional arrays are then printed as rows, bidimensionals as matrices and tridimensionals as lists of matrices.

```
>>> a = np.arange(6)                         # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>>
>>> b = np.arange(12).reshape(4,3)           # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
>>> c = np.arange(24).reshape(2,3,4)         # 3d array
>>> print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]
 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

See *below* to get more details on `reshape`.

If an array is too large to be printed, NumPy automatically skips the central part of the array and only prints the corners:

```
>>> print(np.arange(10000))
[   0    1    2 ..., 9997 9998 9999]
>>>
>>> print(np.arange(10000).reshape(100,100))
[[   0    1    2 ...,   97   98   99]
 [ 100  101  102 ...,  197  198  199]
 [ 200  201  202 ...,  297  298  299]
 ...,
 [9700 9701 9702 ..., 9797 9798 9799]
 [9800 9801 9802 ..., 9897 9898 9899]
 [9900 9901 9902 ..., 9997 9998 9999]]
```

To disable this behaviour and force NumPy to print the entire array, you can change the printing options using `set_printoptions`.

```
>>> np.set_printoptions(threshold=np.nan)
```

2.2.4 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
```

(continues on next page)

(continued from previous page)

```
>>> 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])
```

Unlike in many matrix languages, the product operator * operates elementwise in NumPy arrays. The matrix product can be performed using the @ operator (in python >=3.5) or the `dot` function or method:

```
>>> A = np.array( [[1,1],
...               [0,1]] )
>>> B = np.array( [[2,0],
...               [3,4]] )
>>> A * B                       # elementwise product
array([[2, 0],
       [0, 4]])
>>> A @ B                       # matrix product
array([[5, 4],
       [3, 4]])
>>> A.dot(B)                    # another matrix product
array([[5, 4],
       [3, 4]])
```

Some operations, such as += and *=, act in place to modify an existing array rather than create a new one.

```
>>> a = np.ones((2,3), dtype=int)
>>> b = np.random.random((2,3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[ 3.417022  ,  3.72032449,  3.00011437],
       [ 3.30233257,  3.14675589,  3.09233859]])
>>> a += b                  # b is not automatically converted to integer type
Traceback (most recent call last):
  ...
TypeError: Cannot cast ufunc add output from dtype('float64') to dtype('int64') with␣
↪casting rule 'same_kind'
```

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).

```
>>> a = np.ones(3, dtype=np.int32)
>>> b = np.linspace(0,pi,3)
>>> b.dtype.name
'float64'
>>> c = a+b
>>> c
array([ 1.        ,  2.57079633,  4.14159265])
>>> c.dtype.name
'float64'
```

(continues on next page)

(continued from previous page)

```
>>> d = np.exp(c*1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```

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
```

By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the `axis` parameter you can apply an operation along the specified axis of an array:

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)                            # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1)                            # min of each row
array([0, 4, 8])
>>>
>>> b.cumsum(axis=1)                         # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

2.2.5 Universal Functions

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.])
```

(continues on next page)

(continued from previous page)

```
>>> np.add(B, C)
array([ 2.,  0.,  6.])
```

See also:

all, any, apply_along_axis, argmax, argmin, argsort, average, bincount, ceil, clip, conj, corrcoef, cov, cross, cumprod, cumsum, diff, dot, floor, inner, *inv*, lexsort, max, maximum, mean, median, min, minimum, nonzero, outer, prod, re, round, sort, std, sum, trace, transpose, var, vdot, vectorize, where

2.2.6 Indexing, Slicing and Iterating

One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
>>> a[:6:2] = -1000    # equivalent to a[0:6:2] = -1000; from start to position 6,␣
↪exclusive, set every 2nd element to -1000
>>> a
array([-1000,     1, -1000,    27, -1000,   125,   216,   343,   512,   729])
>>> a[ : :-1]                                 # reversed a
array([  729,   512,   343,   216,   125, -1000,    27, -1000,     1, -1000])
>>> for i in a:
...     print(i**(1/3.))
...
nan
1.0
nan
3.0
nan
5.0
6.0
7.0
8.0
9.0
```

Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas:

```
>>> def f(x,y):
...     return 10*x+y
...
>>> b = np.fromfunction(f,(5,4),dtype=int)
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
```

(continues on next page)

(continued from previous page)

```
>>> b[0:5, 1]                       # each row in the second column of b
array([ 1, 11, 21, 31, 41])
>>> b[ : ,1]                        # equivalent to the previous example
array([ 1, 11, 21, 31, 41])
>>> b[1:3, : ]                      # each column in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

When fewer indices are provided than the number of axes, the missing indices are considered complete slices:

```
>>> b[-1]                                  # the last row. Equivalent to b[-1,:]
array([40, 41, 42, 43])
```

The expression within brackets in `b[i]` is treated as an `i` followed by as many instances of `:` as needed to represent the remaining axes. NumPy also allows you to write this using dots as `b[i,...]`.

The **dots** (`...`) represent as many colons as needed to produce a complete indexing tuple. For example, if `x` is an array with 5 axes, then

- `x[1,2,...]` is equivalent to `x[1,2,:,:,:]`,
- `x[...,3]` to `x[:,:,:,:,3]` and
- `x[4,...,5,:]` to `x[4,:,:,5,:]`.

```
>>> c = np.array( [[[  0,  1,  2],               # a 3D array (two stacked 2D arrays)
...                 [ 10, 12, 13]],
...                [[100,101,102],
...                 [110,112,113]]])
>>> c.shape
(2, 2, 3)
>>> c[1,...]                                   # same as c[1,:,:] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[...,2]                                   # same as c[:,:,2]
array([[  2,  13],
       [102, 113]])
```

Iterating over multidimensional arrays is done with respect to the first axis:

```
>>> for row in b:
...     print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

However, if one wants to perform an operation on each element in the array, one can use the `flat` attribute which is an iterator over all the elements of the array:

```
>>> for element in b.flat:
...     print(element)
...
0
1
2
```

(continues on next page)

(continued from previous page)

```
3
10
11
12
13
20
21
22
23
30
31
32
33
40
41
42
43
```

See also:

Indexing, arrays.indexing (reference), `newaxis`, `ndenumerate`, `indices`

2.3 Shape Manipulation

2.3.1 Changing the shape of an array

An array has a shape given by the number of elements along each axis:

```
>>> a = np.floor(10*np.random.random((3,4)))
>>> a
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
>>> a.shape
(3, 4)
```

The shape of an array can be changed with various commands. Note that the following three commands all return a modified array, but do not change the original array:

```
>>> a.ravel()  # returns the array, flattened
array([ 2.,  8.,  0.,  6.,  4.,  5.,  1.,  1.,  8.,  9.,  3.,  6.])
>>> a.reshape(6,2)  # returns the array with a modified shape
array([[ 2.,  8.],
       [ 0.,  6.],
       [ 4.,  5.],
       [ 1.,  1.],
       [ 8.,  9.],
       [ 3.,  6.]])
>>> a.T  # returns the array, transposed
array([[ 2.,  4.,  8.],
       [ 8.,  5.,  9.],
       [ 0.,  1.,  3.],
       [ 6.,  1.,  6.]])
>>> a.T.shape
```

(continues on next page)

(continued from previous page)

```
(4, 3)
>>> a.shape
(3, 4)
```

The order of the elements in the array resulting from ravel() is normally “C-style”, that is, the rightmost index “changes the fastest”, so the element after a[0,0] is a[0,1]. If the array is reshaped to some other shape, again the array is treated as “C-style”. NumPy normally creates arrays stored in this order, so ravel() will usually not need to copy its argument, but if the array was made by taking slices of another array or created with unusual options, it may need to be copied. The functions ravel() and reshape() can also be instructed, using an optional argument, to use FORTRAN-style arrays, in which the leftmost index changes the fastest.

The `reshape` function returns its argument with a modified shape, whereas the `ndarray.resize` method modifies the array itself:

```
>>> a
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
>>> a.resize((2,6))
>>> a
array([[ 2.,  8.,  0.,  6.,  4.,  5.],
       [ 1.,  1.,  8.,  9.,  3.,  6.]])
```

If a dimension is given as -1 in a reshaping operation, the other dimensions are automatically calculated:

```
>>> a.reshape(3,-1)
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
```

See also:

`ndarray.shape`, `reshape`, `resize`, `ravel`

2.3.2 Stacking together different arrays

Several arrays can be stacked together along different axes:

```
>>> a = np.floor(10*np.random.random((2,2)))
>>> a
array([[ 8.,  8.],
       [ 0.,  0.]])
>>> b = np.floor(10*np.random.random((2,2)))
>>> b
array([[ 1.,  8.],
       [ 0.,  4.]])
>>> np.vstack((a,b))
array([[ 8.,  8.],
       [ 0.,  0.],
       [ 1.,  8.],
       [ 0.,  4.]])
>>> np.hstack((a,b))
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
```

The function `column_stack` stacks 1D arrays as columns into a 2D array. It is equivalent to `hstack` only for 2D arrays:

```
>>> from numpy import newaxis
>>> np.column_stack((a,b))     # with 2D arrays
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
>>> a = np.array([4.,2.])
>>> b = np.array([3.,8.])
>>> np.column_stack((a,b))     # returns a 2D array
array([[ 4., 3.],
       [ 2., 8.]])
>>> np.hstack((a,b))           # the result is different
array([ 4., 2., 3., 8.])
>>> a[:,newaxis]               # this allows to have a 2D columns vector
array([[ 4.],
       [ 2.]])
>>> np.column_stack((a[:,newaxis],b[:,newaxis]))
array([[ 4.,  3.],
       [ 2.,  8.]])
>>> np.hstack((a[:,newaxis],b[:,newaxis]))   # the result is the same
array([[ 4.,  3.],
       [ 2.,  8.]])
```

On the other hand, the function `row_stack` is equivalent to `vstack` for any input arrays. In general, for arrays of with more than two dimensions, `hstack` stacks along their second axes, `vstack` stacks along their first axes, and `concatenate` allows for an optional arguments giving the number of the axis along which the concatenation should happen.

Note

In complex cases, `r_` and `c_` are useful for creating arrays by stacking numbers along one axis. They allow the use of range literals (":")

```
>>> np.r_[1:4,0,4]
array([1, 2, 3, 0, 4])
```

When used with arrays as arguments, `r_` and `c_` are similar to `vstack` and `hstack` in their default behavior, but allow for an optional argument giving the number of the axis along which to concatenate.

See also:

`hstack`, `vstack`, `column_stack`, `concatenate`, `c_`, `r_`

2.3.3 Splitting one array into several smaller ones

Using `hsplit`, you can split an array along its horizontal axis, either by specifying the number of equally shaped arrays to return, or by specifying the columns after which the division should occur:

```
>>> a = np.floor(10*np.random.random((2,12)))
>>> a
array([[ 9.,  5.,  6.,  3.,  6.,  8.,  0.,  7.,  9.,  7.,  2.,  7.],
       [ 1.,  4.,  9.,  2.,  2.,  1.,  0.,  6.,  2.,  2.,  4.,  0.]])
>>> np.hsplit(a,3)   # Split a into 3
[array([[ 9.,  5.,  6.,  3.],
       [ 1.,  4.,  9.,  2.]]), array([[ 6.,  8.,  0.,  7.],
       [ 2.,  1.,  0.,  6.]]), array([[ 9.,  7.,  2.,  7.],
```

(continues on next page)

(continued from previous page)

```
       [ 2.,  2.,  4.,  0.]])]
>>> np.hsplit(a,(3,4))   # Split a after the third and the fourth column
[array([[ 9.,  5.,  6.],
       [ 1.,  4.,  9.]]), array([[ 3.],
       [ 2.]]), array([[ 6.,  8.,  0.,  7.,  9.,  7.,  2.,  7.],
       [ 2.,  1.,  0.,  6.,  2.,  2.,  4.,  0.]])]
```

`vsplit` splits along the vertical axis, and `array_split` allows one to specify along which axis to split.

2.4 Copies and Views

When operating and manipulating arrays, their data is sometimes copied into a new array and sometimes not. This is often a source of confusion for beginners. There are three cases:

2.4.1 No Copy at All

Simple assignments make no copy of array objects or of their data.

```
>>> a = np.arange(12)
>>> b = a            # no new object is created
>>> b is a           # a and b are two names for the same ndarray object
True
>>> b.shape = 3,4    # changes the shape of a
>>> a.shape
(3, 4)
```

Python passes mutable objects as references, so function calls make no copy.

```
>>> def f(x):
...     print(id(x))
...
>>> id(a)                           # id is a unique identifier of an object
148293216
>>> f(a)
148293216
```

2.4.2 View or Shallow Copy

Different array objects can share the same data. The `view` method creates a new array object that looks at the same data.

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a                        # c is a view of the data owned by a
True
>>> c.flags.owndata
False
>>>
>>> c.shape = 2,6                      # a's shape doesn't change
>>> a.shape
```

(continues on next page)

(continued from previous page)

```
(3, 4)
>>> c[0,4] = 1234                      # a's data changes
>>> a
array([[   0,    1,    2,    3],
       [1234,    5,    6,    7],
       [   8,    9,   10,   11]])
```

Slicing an array returns a view of it:

```
>>> s = a[ : , 1:3]     # spaces added for clarity; could also be written "s = a[:,
↪1:3]"
>>> s[:] = 10           # s[:] is a view of s. Note the difference between s=10 and␣
↪s[:]=10
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

2.4.3 Deep Copy

The copy method makes a complete copy of the array and its data.

```
>>> d = a.copy()                          # a new array object with new data is␣
↪created
>>> d is a
False
>>> d.base is a                           # d doesn't share anything with a
False
>>> d[0,0] = 9999
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

Sometimes copy should be called after slicing if the original array is not required anymore. For example, suppose a is a huge intermediate result and the final result b only contains a small fraction of a, a deep copy should be made when constructing b with slicing:

```
>>> a = np.arange(int(1e8))
>>> b = a[:100].copy()
>>> del a  # the memory of ``a`` can be released.
```

If `b = a[:100]` is used instead, a is referenced by b and will persist in memory even if `del a` is executed.

2.4.4 Functions and Methods Overview

Here is a list of some useful NumPy functions and methods names ordered in categories. See routines for the full list.

Array Creation arange, array, copy, empty, empty_like, eye, fromfile, fromfunction, identity, linspace, logspace, mgrid, ogrid, ones, ones_like, *r*, zeros, zeros_like

Conversions ndarray.astype, atleast_1d, atleast_2d, atleast_3d, mat

Manipulations `array_split`, `column_stack`, `concatenate`, `diagonal`, `dsplit`, `dstack`, `hsplit`, `hstack`, `ndarray.item`, `newaxis`, `ravel`, `repeat`, `reshape`, `resize`, `squeeze`, `swapaxes`, `take`, `transpose`, `vsplit`, `vstack`

Questions `all`, `any`, `nonzero`, `where`

Ordering `argmax`, `argmin`, `argsort`, `max`, `min`, `ptp`, `searchsorted`, `sort`

Operations `choose`, `compress`, `cumprod`, `cumsum`, `inner`, `ndarray.fill`, `imag`, `prod`, `put`, `putmask`, `real`, `sum`

Basic Statistics `cov`, `mean`, `std`, `var`

Basic Linear Algebra `cross`, `dot`, `outer`, `linalg.svd`, `vdot`

2.5 Less Basic

2.5.1 Broadcasting rules

Broadcasting allows universal functions to deal in a meaningful way with inputs that do not have exactly the same shape.

The first rule of broadcasting is that if all input arrays do not have the same number of dimensions, a "1" will be repeatedly prepended to the shapes of the smaller arrays until all the arrays have the same number of dimensions.

The second rule of broadcasting ensures that arrays with a size of 1 along a particular dimension act as if they had the size of the array with the largest shape along that dimension. The value of the array element is assumed to be the same along that dimension for the "broadcast" array.

After application of the broadcasting rules, the sizes of all arrays must match. More details can be found in *Broadcasting*.

2.6 Fancy indexing and index tricks

NumPy offers more indexing facilities than regular Python sequences. In addition to indexing by integers and slices, as we saw before, arrays can be indexed by arrays of integers and arrays of booleans.

2.6.1 Indexing with Arrays of Indices

```
>>> a = np.arange(12)**2                       # the first 12 square numbers
>>> i = np.array( [ 1,1,3,8,5 ] )              # an array of indices
>>> a[i]                                       # the elements of a at the positions i
array([ 1,  1,  9, 64, 25])
>>>
>>> j = np.array( [ [ 3, 4], [ 9, 7 ] ] )      # a bidimensional array of indices
>>> a[j]                                       # the same shape as j
array([[ 9, 16],
       [81, 49]])
```

When the indexed array `a` is multidimensional, a single array of indices refers to the first dimension of `a`. The following example shows this behavior by converting an image of labels into a color image using a palette.

```
>>> palette = np.array( [ [0,0,0],                # black
...                       [255,0,0],              # red
...                       [0,255,0],              # green
...                       [0,0,255],              # blue
...                       [255,255,255] ] )       # white
>>> image = np.array( [ [ 0, 1, 2, 0 ],           # each value corresponds to a color␣
↪in the palette
...                     [ 0, 3, 4, 0 ]  ] )
>>> palette[image]                            # the (2,4,3) color image
array([[[  0,   0,   0],
        [255,   0,   0],
        [  0, 255,   0],
        [  0,   0,   0]],
       [[  0,   0,   0],
        [  0,   0, 255],
        [255, 255, 255],
        [  0,   0,   0]]])
```

We can also give indexes for more than one dimension. The arrays of indices for each dimension must have the same shape.

```
>>> a = np.arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> i = np.array( [ [0,1],                        # indices for the first dim of a
...                 [1,2] ] )
>>> j = np.array( [ [2,1],                        # indices for the second dim
...                 [3,3] ] )
>>>
>>> a[i,j]                                    # i and j must have equal shape
array([[ 2,  5],
       [ 7, 11]])
>>>
>>> a[i,2]
array([[ 2,  6],
       [ 6, 10]])
>>>
>>> a[:,j]                                    # i.e., a[ : , j]
array([[[ 2,  1],
        [ 3,  3]],
       [[ 6,  5],
        [ 7,  7]],
       [[10,  9],
        [11, 11]]])
```

Naturally, we can put `i` and `j` in a sequence (say a list) and then do the indexing with the list.

```
>>> l = [i,j]
>>> a[l]                                      # equivalent to a[i,j]
array([[ 2,  5],
       [ 7, 11]])
```

However, we can not do this by putting `i` and `j` into an array, because this array will be interpreted as indexing the first dimension of a.

```
>>> s = np.array( [i,j] )
>>> a[s]                                     # not what we want
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: index (3) out of range (0<=index<=2) in dimension 0
>>>
>>> a[tuple(s)]                              # same as a[i,j]
array([[ 2,  5],
       [ 7, 11]])
```

Another common use of indexing with arrays is the search of the maximum value of time-dependent series:

```
>>> time = np.linspace(20, 145, 5)                 # time scale
>>> data = np.sin(np.arange(20)).reshape(5,4)      # 4 time-dependent series
>>> time
array([  20.  ,   51.25,   82.5 ,  113.75,  145.  ])
>>> data
array([[ 0.        ,  0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ,  0.6569866 ],
       [ 0.98935825,  0.41211849, -0.54402111, -0.99999021],
       [-0.53657292,  0.42016704,  0.99060736,  0.65028784],
       [-0.28790332, -0.96139749, -0.75098725,  0.14987721]])
>>>
>>> ind = data.argmax(axis=0)                  # index of the maxima for each series
>>> ind
array([2, 0, 3, 1])
>>>
>>> time_max = time[ind]                       # times corresponding to the maxima
>>>
>>> data_max = data[ind, range(data.shape[1])] # => data[ind[0],0], data[ind[1],1]...
>>>
>>> time_max
array([  82.5 ,   20.  ,  113.75,   51.25])
>>> data_max
array([ 0.98935825,  0.84147098,  0.99060736,  0.6569866 ])
>>>
>>> np.all(data_max == data.max(axis=0))
True
```

You can also use indexing with arrays as a target to assign to:

```
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a[[1,3,4]] = 0
>>> a
array([0, 0, 2, 0, 0])
```

However, when the list of indices contains repetitions, the assignment is done several times, leaving behind the last value:

```
>>> a = np.arange(5)
>>> a[[0,0,2]]=[1,2,3]
>>> a
array([2, 1, 3, 3, 4])
```

This is reasonable enough, but watch out if you want to use Python's += construct, as it may not do what you expect:

```
>>> a = np.arange(5)
>>> a[[0,0,2]]+=1
>>> a
array([1, 1, 3, 3, 4])
```

Even though 0 occurs twice in the list of indices, the 0th element is only incremented once. This is because Python requires "a+=1" to be equivalent to "a = a + 1".

2.6.2 Indexing with Boolean Arrays

When we index arrays with arrays of (integer) indices we are providing the list of indices to pick. With boolean indices the approach is different; we explicitly choose which items in the array we want and which ones we don't.

The most natural way one can think of for boolean indexing is to use boolean arrays that have *the same shape* as the original array:

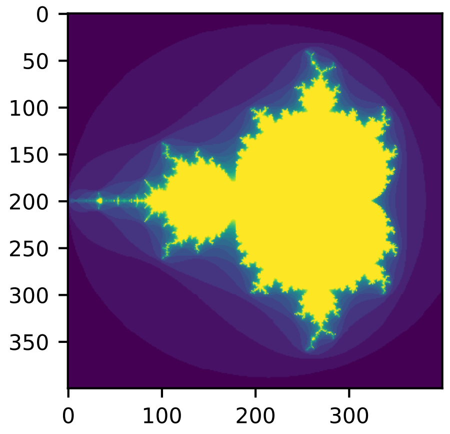
```
>>> a = np.arange(12).reshape(3,4)
>>> b = a > 4
>>> b                                          # b is a boolean with a's shape
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]])
>>> a[b]                                       # 1d array with the selected elements
array([ 5,  6,  7,  8,  9, 10, 11])
```

This property can be very useful in assignments:

```
>>> a[b] = 0                                   # All elements of 'a' higher than 4␣
↪become 0
>>> a
array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]])
```

You can look at the following example to see how to use boolean indexing to generate an image of the Mandelbrot set:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> def mandelbrot( h,w, maxit=20 ):
...     """Returns an image of the Mandelbrot fractal of size (h,w)."""
...     y,x = np.ogrid[ -1.4:1.4:h*1j, -2:0.8:w*1j ]
...     c = x+y*1j
...     z = c
...     divtime = maxit + np.zeros(z.shape, dtype=int)
...
...     for i in range(maxit):
...         z = z**2 + c
...         diverge = z*np.conj(z) > 2**2            # who is diverging
...         div_now = diverge & (divtime==maxit)  # who is diverging now
...         divtime[div_now] = i                  # note when
...         z[diverge] = 2                        # avoid diverging too much
...
...     return divtime
>>> plt.imshow(mandelbrot(400,400))
>>> plt.show()
```



The second way of indexing with booleans is more similar to integer indexing; for each dimension of the array we give a 1D boolean array selecting the slices we want:

```
>>> a = np.arange(12).reshape(3,4)
>>> b1 = np.array([False,True,True])             # first dim selection
>>> b2 = np.array([True,False,True,False])       # second dim selection
>>>
>>> a[b1,:]                                   # selecting rows
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[b1]                                     # same thing
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[:,b2]                                   # selecting columns
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
>>>
>>> a[b1,b2]                                  # a weird thing to do
array([ 4, 10])
```

Note that the length of the 1D boolean array must coincide with the length of the dimension (or axis) you want to slice. In the previous example, `b1` has length 3 (the number of *rows* in `a`), and `b2` (of length 4) is suitable to index the 2nd axis (columns) of `a`.

2.6.3 The ix_() function

The `ix_` function can be used to combine different vectors so as to obtain the result for each n-uplet. For example, if you want to compute all the a+b*c for all the triplets taken from each of the vectors a, b and c:

```
>>> a = np.array([2,3,4,5])
>>> b = np.array([8,5,4])
>>> c = np.array([5,4,6,8,3])
```

(continues on next page)

(continued from previous page)

```
>>> ax,bx,cx = np.ix_(a,b,c)
>>> ax
array([[[2]],
       [[3]],
       [[4]],
       [[5]]])
>>> bx
array([[[8],
        [5],
        [4]]])
>>> cx
array([[[5, 4, 6, 8, 3]]])
>>> ax.shape, bx.shape, cx.shape
((4, 1, 1), (1, 3, 1), (1, 1, 5))
>>> result = ax+bx*cx
>>> result
array([[[42, 34, 50, 66, 26],
        [27, 22, 32, 42, 17],
        [22, 18, 26, 34, 14]],
       [[43, 35, 51, 67, 27],
        [28, 23, 33, 43, 18],
        [23, 19, 27, 35, 15]],
       [[44, 36, 52, 68, 28],
        [29, 24, 34, 44, 19],
        [24, 20, 28, 36, 16]],
       [[45, 37, 53, 69, 29],
        [30, 25, 35, 45, 20],
        [25, 21, 29, 37, 17]]])
>>> result[3,2,4]
17
>>> a[3]+b[2]*c[4]
17
```

You could also implement the reduce as follows:

```
>>> def ufunc_reduce(ufct, *vectors):
...    vs = np.ix_(*vectors)
...    r = ufct.identity
...    for v in vs:
...        r = ufct(r,v)
...    return r
```

and then use it as:

```
>>> ufunc_reduce(np.add,a,b,c)
array([[[15, 14, 16, 18, 13],
        [12, 11, 13, 15, 10],
        [11, 10, 12, 14,  9]],
       [[16, 15, 17, 19, 14],
        [13, 12, 14, 16, 11],
        [12, 11, 13, 15, 10]],
       [[17, 16, 18, 20, 15],
        [14, 13, 15, 17, 12],
        [13, 12, 14, 16, 11]],
       [[18, 17, 19, 21, 16],
        [15, 14, 16, 18, 13],
        [14, 13, 15, 17, 12]]])
```

The advantage of this version of reduce compared to the normal ufunc.reduce is that it makes use of the Broadcasting Rules in order to avoid creating an argument array the size of the output times the number of vectors.

2.6.4 Indexing with strings

See *Structured arrays*.

2.7 Linear Algebra

Work in progress. Basic linear algebra to be included here.

2.7.1 Simple Array Operations

See linalg.py in numpy folder for more.

```
>>> import numpy as np
>>> a = np.array([[1.0, 2.0], [3.0, 4.0]])
>>> print(a)
[[ 1.  2.]
 [ 3.  4.]]

>>> a.transpose()
array([[ 1.,  3.],
       [ 2.,  4.]])

>>> np.linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

>>> u = np.eye(2) # unit 2x2 matrix; "eye" represents "I"
>>> u
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> j = np.array([[0.0, -1.0], [1.0, 0.0]])

>>> j @ j        # matrix product
array([[-1.,  0.],
       [ 0., -1.]])

>>> np.trace(u)  # trace
2.0

>>> y = np.array([[5.], [7.]])
>>> np.linalg.solve(a, y)
array([[-3.],
       [ 4.]])

>>> np.linalg.eig(j)
(array([ 0.+1.j,  0.-1.j]), array([[ 0.70711678+0.j        ,  0.70711678-0.j        ],
       [ 0.00000000-0.70711678j,  0.00000000+0.70711678j]]))
```

```
Parameters:
    square matrix
Returns
    The eigenvalues, each repeated according to its multiplicity.
    The normalized (unit "length") eigenvectors, such that the
    column ``v[:,i]`` is the eigenvector corresponding to the
    eigenvalue ``w[i]`` .
```

2.8 Tricks and Tips

Here we give a list of short and useful tips.

2.8.1 "Automatic" Reshaping

To change the dimensions of an array, you can omit one of the sizes which will then be deduced automatically:

```
>>> a = np.arange(30)
>>> a.shape = 2,-1,3  # -1 means "whatever is needed"
>>> a.shape
(2, 5, 3)
>>> a
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],
        [24, 25, 26],
        [27, 28, 29]]])
```

2.8.2 Vector Stacking

How do we construct a 2D array from a list of equally-sized row vectors? In MATLAB this is quite easy: if `x` and `y` are two vectors of the same length you only need do `m=[x;y]`. In NumPy this works via the functions `column_stack`, `dstack`, `hstack` and `vstack`, depending on the dimension in which the stacking is to be done. For example:

```
x = np.arange(0,10,2)                     # x=([0,2,4,6,8])
y = np.arange(5)                          # y=([0,1,2,3,4])
m = np.vstack([x,y])                      # m=([[0,2,4,6,8],
                                          #      [0,1,2,3,4]])
xy = np.hstack([x,y])                     # xy =([0,2,4,6,8,0,1,2,3,4])
```

The logic behind those functions in more than two dimensions can be strange.

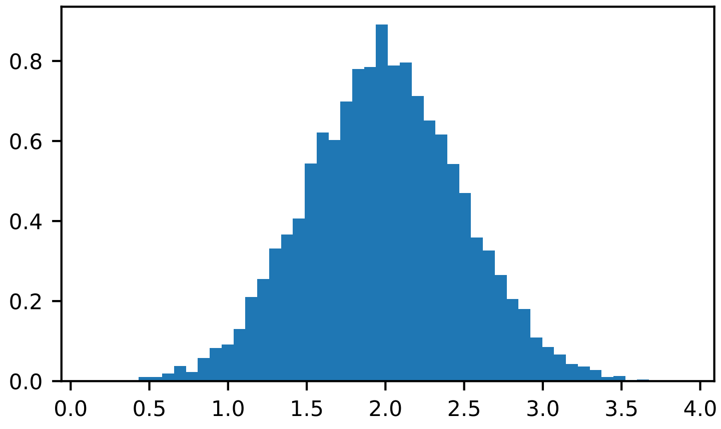
See also:

NumPy for Matlab users

2.8.3 Histograms

The NumPy `histogram` function applied to an array returns a pair of vectors: the histogram of the array and the vector of bins. Beware: `matplotlib` also has a function to build histograms (called `hist`, as in Matlab) that differs from the one in NumPy. The main difference is that `pylab.hist` plots the histogram automatically, while `numpy.histogram` only generates the data.

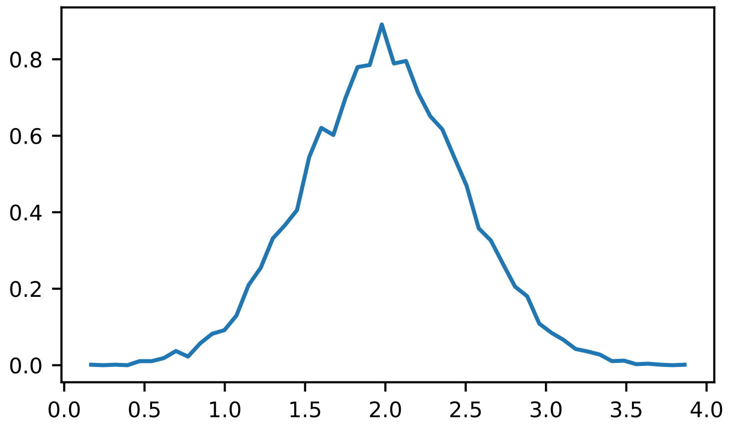
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> # Build a vector of 10000 normal deviates with variance 0.5^2 and mean 2
>>> mu, sigma = 2, 0.5
>>> v = np.random.normal(mu,sigma,10000)
>>> # Plot a normalized histogram with 50 bins
>>> plt.hist(v, bins=50, density=1)       # matplotlib version (plot)
>>> plt.show()
```



```
>>> # Compute the histogram with numpy and then plot it
>>> (n, bins) = np.histogram(v, bins=50, density=True)  # NumPy version (no plot)
>>> plt.plot(.5*(bins[1:]+bins[:-1]), n)
>>> plt.show()
```

2.9 Further reading

- The Python tutorial
- reference
- SciPy Tutorial
- SciPy Lecture Notes
- A matlab, R, IDL, NumPy/SciPy dictionary


0.0
0.2
0.4
0.6
0.8
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

CHAPTER

THREE

NUMPY BASICS

3.1 Data types

See also:

Data type objects

3.1.1 Array types and conversions between types

NumPy supports a much greater variety of numerical types than Python does. This section shows which are available, and how to modify an array's data-type.

The primitive types supported are tied closely to those in C:

Numpy type	C type	Description
np.bool	`bool`	Boolean (True or False) stored as a byte
np.byte	`signed char`	Platform-defined
np.ubyte	`unsigned char`	Platform-defined
np.short	`short`	Platform-defined
np.ushort	`unsigned short`	Platform-defined
np.intc	`int`	Platform-defined
np.uintc	`unsigned int`	Platform-defined
*np.int_*	`long`	Platform-defined
np.uint	`unsigned long`	Platform-defined
np.longlong	`long long`	Platform-defined
np.ulonglong	`unsigned long long`	Platform-defined
np.half / *np.float16*		Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
np.single	`float`	Platform-defined single precision float: typically sign bit, 8 bits exponent, 23 bits mantissa
np.double	`double`	Platform-defined double precision float: typically sign bit, 11 bits exponent, 52 bits mantissa.
np.longdouble	`long double`	Platform-defined extended-precision float
np.csingle	`float complex`	Complex number, represented by two single-precision floats (real and imaginary components)
np.cdouble	`double complex`	Complex number, represented by two double-precision floats (real and imaginary components).
np.clongdouble	`long double complex`	Complex number, represented by two extended-precision floats (real and imaginary components).

Since many of these have platform-dependent definitions, a set of fixed-size aliases are provided:

Numpy type	C type	Description
np.int8	`int8_t`	Byte (-128 to 127)
np.int16	`int16_t`	Integer (-32768 to 32767)
np.int32	`int32_t`	Integer (-2147483648 to 2147483647)
np.int64	`int64_t`	Integer (-9223372036854775808 to 9223372036854775807)
np.uint8	`uint8_t`	Unsigned integer (0 to 255)
np.uint16	`uint16_t`	Unsigned integer (0 to 65535)
np.uint32	`uint32_t`	Unsigned integer (0 to 4294967295)
np.uint64	`uint64_t`	Unsigned integer (0 to 18446744073709551615)
np.intp	`intptr_t`	Integer used for indexing, typically the same as `ssize_t`
np.uintp	`uintptr_t`	Integer large enough to hold a pointer
np.float32	`float`	
np.float64 / *np.float_*	`double`	Note that this matches the precision of the builtin python *float*.
np.complex64	`float complex`	Complex number, represented by two 32-bit floats (real and imaginary components)
np.complex128 / *np.complex_*	`double complex`	Note that this matches the precision of the builtin python *complex*.

NumPy numerical types are instances of `dtype` (data-type) objects, each having unique characteristics. Once you have imported NumPy using

```
>>> import numpy as np
```

the dtypes are available as `np.bool_`, `np.float32`, etc.

Advanced types, not listed in the table above, are explored in section *Structured arrays*.

There are 5 basic numerical types representing booleans (bool), integers (int), unsigned integers (uint) floating point (float) and complex. Those with numbers in their name indicate the bitsize of the type (i.e. how many bits are needed to represent a single value in memory). Some types, such as `int` and `intp`, have differing bitsizes, dependent on the platforms (e.g. 32-bit vs. 64-bit machines). This should be taken into account when interfacing with low-level code (such as C or Fortran) where the raw memory is addressed.

Data-types can be used as functions to convert python numbers to array scalars (see the array scalar section for an explanation), python sequences of numbers to arrays of that type, or as arguments to the dtype keyword that many numpy functions or methods accept. Some examples:

```
>>> import numpy as np
>>> x = np.float32(1.0)
>>> x
1.0
>>> y = np.int_([1,2,4])
>>> y
array([1, 2, 4])
>>> z = np.arange(3, dtype=np.uint8)
>>> z
array([0, 1, 2], dtype=uint8)
```

Array types can also be referred to by character codes, mostly to retain backward compatibility with older packages such as Numeric. Some documentation may still refer to these, for example:

```
>>> np.array([1, 2, 3], dtype='f')
array([ 1.,  2.,  3.], dtype=float32)
```

We recommend using dtype objects instead.

To convert the type of an array, use the .astype() method (preferred) or the type itself as a function. For example:

```
>>> z.astype(float)
array([  0.,   1.,   2.])
>>> np.int8(z)
array([0, 1, 2], dtype=int8)
```

Note that, above, we use the *Python* float object as a dtype. NumPy knows that `int` refers to `np.int_`, `bool` means `np.bool_`, that `float` is `np.float_` and `complex` is `np.complex_`. The other data-types do not have Python equivalents.

To determine the type of an array, look at the dtype attribute:

```
>>> z.dtype
dtype('uint8')
```

dtype objects also contain information about the type, such as its bit-width and its byte-order. The data type can also be used indirectly to query properties of the type, such as whether it is an integer:

```
>>> d = np.dtype(int)
>>> d
dtype('int32')

>>> np.issubdtype(d, np.integer)
True

>>> np.issubdtype(d, np.floating)
False
```

3.1.2 Array Scalars

NumPy generally returns elements of arrays as array scalars (a scalar with an associated dtype). Array scalars differ from Python scalars, but for the most part they can be used interchangeably (the primary exception is for versions of Python older than v2.x, where integer array scalars cannot act as indices for lists and tuples). There are some exceptions, such as when code requires very specific attributes of a scalar or when it checks specifically whether a value is a Python scalar. Generally, problems are easily fixed by explicitly converting array scalars to Python scalars, using the corresponding Python type function (e.g., `int`, `float`, `complex`, `str`, `unicode`).

The primary advantage of using array scalars is that they preserve the array type (Python may not have a matching scalar type available, e.g. `int16`). Therefore, the use of array scalars ensures identical behaviour between arrays and scalars, irrespective of whether the value is inside an array or not. NumPy scalars also have many of the same methods arrays do.

3.1.3 Overflow Errors

The fixed size of NumPy numeric types may cause overflow errors when a value requires more memory than available in the data type. For example, `numpy.power` evaluates `100 * 10 ** 8` correctly for 64-bit integers, but gives 1874919424 (incorrect) for a 32-bit integer.

```
>>> np.power(100, 8, dtype=np.int64)
10000000000000000
>>> np.power(100, 8, dtype=np.int32)
1874919424
```

The behaviour of NumPy and Python integer types differs significantly for integer overflows and may confuse users expecting NumPy integers to behave similar to Python's `int`. Unlike NumPy, the size of Python's `int` is flexible. This means Python integers may expand to accommodate any integer and will not overflow.

NumPy provides `numpy.iinfo` and `numpy.finfo` to verify the minimum or maximum values of NumPy integer and floating point values respectively

```
>>> np.iinfo(np.int) # Bounds of the default integer on this system.
iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
>>> np.iinfo(np.int32) # Bounds of a 32-bit integer
iinfo(min=-2147483648, max=2147483647, dtype=int32)
>>> np.iinfo(np.int64) # Bounds of a 64-bit integer
iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
```

If 64-bit integers are still too small the result may be cast to a floating point number. Floating point numbers offer a larger, but inexact, range of possible values.

```
>>> np.power(100, 100, dtype=np.int64) # Incorrect even with 64-bit int
0
>>> np.power(100, 100, dtype=np.float64)
1e+200
```

3.1.4 Extended Precision

Python's floating-point numbers are usually 64-bit floating-point numbers, nearly equivalent to `np.float64`. In some unusual situations it may be useful to use floating-point numbers with more precision. Whether this is possible in numpy depends on the hardware and on the development environment: specifically, x86 machines provide hardware floating-point with 80-bit precision, and while most C compilers provide this as their `long double` type, MSVC (standard for Windows builds) makes `long double` identical to `double` (64 bits). NumPy makes the compiler's `long double` available as `np.longdouble` (and `np.clongdouble` for the complex numbers). You can find out what your numpy provides with `np.finfo(np.longdouble)`.

NumPy does not provide a dtype with more precision than C `long doubles`; in particular, the 128-bit IEEE quad precision data type (FORTRAN's `REAL*16`) is not available.

For efficient memory alignment, `np.longdouble` is usually stored padded with zero bits, either to 96 or 128 bits. Which is more efficient depends on hardware and development environment; typically on 32-bit systems they are padded to 96 bits, while on 64-bit systems they are typically padded to 128 bits. `np.longdouble` is padded to the system default; `np.float96` and `np.float128` are provided for users who want specific padding. In spite of the names, `np.float96` and `np.float128` provide only as much precision as `np.longdouble`, that is, 80 bits on most x86 machines and 64 bits in standard Windows builds.

Be warned that even if `np.longdouble` offers more precision than python `float`, it is easy to lose that extra precision, since python often forces values to pass through `float`. For example, the `%` formatting operator requires its arguments to be converted to standard python types, and it is therefore impossible to preserve extended precision even if many decimal places are requested. It can be useful to test your code with the value `1 + np.finfo(np.longdouble).eps`.

3.2 Array creation

See also:

Array creation routines

3.2.1 Introduction

There are 5 general mechanisms for creating arrays:

1) Conversion from other Python structures (e.g., lists, tuples)
2) Intrinsic numpy array creation objects (e.g., arange, ones, zeros, etc.)
3) Reading arrays from disk, either from standard or custom formats
4) Creating arrays from raw bytes through the use of strings or buffers
5) Use of special library functions (e.g., random)

This section will not cover means of replicating, joining, or otherwise expanding or mutating existing arrays. Nor will it cover creating object arrays or structured arrays. Both of those are covered in their own sections.

3.2.2 Converting Python array_like Objects to NumPy Arrays

In general, numerical data arranged in an array-like structure in Python can be converted to arrays through the use of the array() function. The most obvious examples are lists and tuples. See the documentation for array() for details for its use. Some objects may support the array-protocol and allow conversion to arrays this way. A simple way to find out if the object can be converted to a numpy array using array() is simply to try it interactively and see if it works! (The Python Way).

Examples:

```
>>> x = np.array([2,3,1,0])
>>> x = np.array([2, 3, 1, 0])
>>> x = np.array([[1,2.0],[0,0],(1+1j,3.)]) # note mix of tuple and lists,
    and types
>>> x = np.array([[ 1.+0.j, 2.+0.j], [ 0.+0.j, 0.+0.j], [ 1.+1.j, 3.+0.j]])
```

3.2.3 Intrinsic NumPy Array Creation

NumPy has built-in functions for creating arrays from scratch:

zeros(shape) 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.]])
```

ones(shape) will create an array filled with 1 values. It is identical to zeros in all other respects.

arange() will create arrays with regularly incrementing values. Check the docstring for complete information on the various ways it can be used. A few examples will be given here:

```
>>> 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 that there are some subtleties regarding the last usage that the user should be aware of that are described in the arange docstring.

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 one can guarantee the number of elements and the starting and end point, which arange() generally will not do for arbitrary start, stop, and step values.

indices() will create a set of arrays (stacked as a one-higher dimensioned array), one per dimension with each representing variation in that dimension. An example illustrates much better than a verbal description:

```
>>> 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.

3.2.4 Reading Arrays From Disk

This is presumably the most common case of large array creation. The details, of course, depend greatly on the format of data on disk and so this section can only give general pointers on how to handle various formats.

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

Comma Separated Value files (CSV) are widely used (and an export and import option for programs like Excel). There are a number of ways of reading these files in Python. There are CSV functions in Python and functions in pylab (part of matplotlib).

More generic ascii files can be read using the io package in scipy.

Custom Binary Formats

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++.

Use of Special Libraries

There are libraries that can be used to generate arrays for special purposes and it isn't possible to enumerate all of them. The most common uses are use of the many array generation functions in random that can generate arrays of random values, and some utility functions to generate special matrices (e.g. diagonal).

3.3 I/O with NumPy

3.3.1 Importing data with `genfromtxt`

NumPy provides several functions to create arrays from tabular data. We focus here on the `genfromtxt` function.

In a nutshell, `genfromtxt` runs two main loops. The first loop converts each line of the file in a sequence of strings. The second loop converts each string to the appropriate data type. This mechanism is slower than a single loop, but gives more flexibility. In particular, `genfromtxt` is able to take missing data into account, when other faster and simpler functions like `loadtxt` cannot.

Note: When giving examples, we will use the following conventions:

```
>>> import numpy as np
>>> from io import StringIO
```

Defining the input

The only mandatory argument of `genfromtxt` is the source of the data. It can be a string, a list of strings, or a generator. If a single string is provided, it is assumed to be the name of a local or remote file, or an open file-like object with a `read` method, for example, a file or `io.StringIO` object. If a list of strings or a generator returning strings is provided, each string is treated as one line in a file. When the URL of a remote file is passed, the file is automatically downloaded to the current directory and opened.

Recognized file types are text files and archives. Currently, the function recognizes `gzip` and `bz2` (*bzip2*) archives. The type of the archive is determined from the extension of the file: if the filename ends with `'.gz'`, a `gzip` archive is expected; if it ends with `'bz2'`, a `bzip2` archive is assumed.

Splitting the lines into columns

The `delimiter` argument

Once the file is defined and open for reading, `genfromtxt` splits each non-empty line into a sequence of strings. Empty or commented lines are just skipped. The `delimiter` keyword is used to define how the splitting should take place.

Quite often, a single character marks the separation between columns. For example, comma-separated files (CSV) use a comma (`,`) or a semicolon (`;`) as delimiter:

```
>>> data = u"1, 2, 3\n4, 5, 6"
>>> np.genfromtxt(StringIO(data), delimiter=",")
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

Another common separator is `"\t"`, the tabulation character. However, we are not limited to a single character, any string will do. By default, `genfromtxt` assumes `delimiter=None`, meaning that the line is split along white spaces (including tabs) and that consecutive white spaces are considered as a single white space.

Alternatively, we may be dealing with a fixed-width file, where columns are defined as a given number of characters. In that case, we need to set `delimiter` to a single integer (if all the columns have the same size) or to a sequence of integers (if columns can have different sizes):

```
>>> data = u"  1  2  3\n  4  5 67\n890123  4"
>>> np.genfromtxt(StringIO(data), delimiter=3)
array([[   1.,    2.,    3.],
       [   4.,    5.,   67.],
       [ 890.,  123.,    4.]])
>>> data = u"123456789\n   4  7 9\n   4567 9"
>>> np.genfromtxt(StringIO(data), delimiter=(4, 3, 2))
array([[ 1234.,   567.,    89.],
       [    4.,     7.,     9.],
       [    4.,   567.,     9.]])
```

The `autostrip` argument

By default, when a line is decomposed into a series of strings, the individual entries are not stripped of leading nor trailing white spaces. This behavior can be overwritten by setting the optional argument `autostrip` to a value of `True`:

```
>>> data = u"1, abc , 2\n 3, xxx, 4"
>>> # Without autostrip
>>> np.genfromtxt(StringIO(data), delimiter=",", dtype="|U5")
array([['1', ' abc ', ' 2'],
       ['3', ' xxx', ' 4']],
      dtype='|U5')
>>> # With autostrip
>>> np.genfromtxt(StringIO(data), delimiter=",", dtype="|U5", autostrip=True)
array([['1', 'abc', '2'],
       ['3', 'xxx', '4']],
      dtype='|U5')
```

The `comments` argument

The optional argument `comments` is used to define a character string that marks the beginning of a comment. By default, `genfromtxt` assumes `comments='#'`. The comment marker may occur anywhere on the line. Any character present after the comment marker(s) is simply ignored:

```
>>> data = u"""#
... # Skip me !
... # Skip me too !
... 1, 2
... 3, 4
... 5, 6 #This is the third line of the data
... 7, 8
... # And here comes the last line
... 9, 0
... """
>>> np.genfromtxt(StringIO(data), comments="#", delimiter=",")
[[ 1.  2.]
 [ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]
 [ 9.  0.]]
```

New in version 1.7.0: When `comments` is set to `None`, no lines are treated as comments.

Note: There is one notable exception to this behavior: if the optional argument `names=True`, the first commented line will be examined for names.

Skipping lines and choosing columns

The `skip_header` and `skip_footer` arguments

The presence of a header in the file can hinder data processing. In that case, we need to use the `skip_header` optional argument. The values of this argument must be an integer which corresponds to the number of lines to skip at the beginning of the file, before any other action is performed. Similarly, we can skip the last `n` lines of the file by using the `skip_footer` attribute and giving it a value of `n`:

```
>>> data = u"\n".join(str(i) for i in range(10))
>>> np.genfromtxt(StringIO(data),)
array([ 0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.])
>>> np.genfromtxt(StringIO(data),
...               skip_header=3, skip_footer=5)
array([ 3.,   4.])
```

By default, `skip_header=0` and `skip_footer=0`, meaning that no lines are skipped.

The `usecols` argument

In some cases, we are not interested in all the columns of the data but only a few of them. We can select which columns to import with the `usecols` argument. This argument accepts a single integer or a sequence of integers corresponding to the indices of the columns to import. Remember that by convention, the first column has an index of 0. Negative integers behave the same as regular Python negative indexes.

For example, if we want to import only the first and the last columns, we can use `usecols=(0, -1)`:

```
>>> data = u"1 2 3\n4 5 6"
>>> np.genfromtxt(StringIO(data), usecols=(0, -1))
array([[ 1.,  3.],
       [ 4.,  6.]])
```

If the columns have names, we can also select which columns to import by giving their name to the `usecols` argument, either as a sequence of strings or a comma-separated string:

```
>>> data = u"1 2 3\n4 5 6"
>>> np.genfromtxt(StringIO(data),
...               names="a, b, c", usecols=("a", "c"))
array([(1.0, 3.0), (4.0, 6.0)],
      dtype=[('a', '<f8'), ('c', '<f8')])
>>> np.genfromtxt(StringIO(data),
...               names="a, b, c", usecols=("a, c"))
    array([(1.0, 3.0), (4.0, 6.0)],
          dtype=[('a', '<f8'), ('c', '<f8')])
```

Choosing the data type

The main way to control how the sequences of strings we have read from the file are converted to other types is to set the `dtype` argument. Acceptable values for this argument are:

- a single type, such as `dtype=float`. The output will be 2D with the given dtype, unless a name has been associated with each column with the use of the `names` argument (see below). Note that `dtype=float` is the default for `genfromtxt`.
- a sequence of types, such as `dtype=(int, float, float)`.
- a comma-separated string, such as `dtype="i4,f8,|U3"`.
- a dictionary with two keys `'names'` and `'formats'`.
- a sequence of tuples `(name, type)`, such as `dtype=[('A', int), ('B', float)]`.
- an existing `numpy.dtype` object.
- the special value `None`. In that case, the type of the columns will be determined from the data itself (see below).

In all the cases but the first one, the output will be a 1D array with a structured dtype. This dtype has as many fields as items in the sequence. The field names are defined with the `names` keyword.

When `dtype=None`, the type of each column is determined iteratively from its data. We start by checking whether a string can be converted to a boolean (that is, if the string matches `true` or `false` in lower cases); then whether it can be converted to an integer, then to a float, then to a complex and eventually to a string. This behavior may be changed by modifying the default mapper of the `StringConverter` class.

The option `dtype=None` is provided for convenience. However, it is significantly slower than setting the dtype explicitly.

Setting the names

The `names` argument

A natural approach when dealing with tabular data is to allocate a name to each column. A first possibility is to use an explicit structured dtype, as mentioned previously:

```
>>> data = StringIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, dtype=[(_, int) for _ in "abc"])
array([(1, 2, 3), (4, 5, 6)],
      dtype=[('a', '<i8'), ('b', '<i8'), ('c', '<i8')])
```

Another simpler possibility is to use the `names` keyword with a sequence of strings or a comma-separated string:

```
>>> data = StringIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, names="A, B, C")
array([(1.0, 2.0, 3.0), (4.0, 5.0, 6.0)],
      dtype=[('A', '<f8'), ('B', '<f8'), ('C', '<f8')])
```

In the example above, we used the fact that by default, `dtype=float`. By giving a sequence of names, we are forcing the output to a structured dtype.

We may sometimes need to define the column names from the data itself. In that case, we must use the `names` keyword with a value of `True`. The names will then be read from the first line (after the `skip_header` ones), even if the line is commented out:

```
>>> data = StringIO("So it goes\n#a b c\n1 2 3\n 4 5 6")
>>> np.genfromtxt(data, skip_header=1, names=True)
array([(1.0, 2.0, 3.0), (4.0, 5.0, 6.0)],
      dtype=[('a', '<f8'), ('b', '<f8'), ('c', '<f8')])
```

The default value of `names` is `None`. If we give any other value to the keyword, the new names will overwrite the field names we may have defined with the dtype:

```
>>> data = StringIO("1 2 3\n 4 5 6")
>>> ndtype=[('a',int), ('b', float), ('c', int)]
>>> names = ["A", "B", "C"]
>>> np.genfromtxt(data, names=names, dtype=ndtype)
array([(1, 2.0, 3), (4, 5.0, 6)],
      dtype=[('A', '<i8'), ('B', '<f8'), ('C', '<i8')])
```

The `defaultfmt` argument

If `names=None` but a structured dtype is expected, names are defined with the standard NumPy default of `"f%i"`, yielding names like `f0`, `f1` and so forth:

```
>>> data = StringIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, dtype=(int, float, int))
array([(1, 2.0, 3), (4, 5.0, 6)],
      dtype=[('f0', '<i8'), ('f1', '<f8'), ('f2', '<i8')])
```

In the same way, if we don't give enough names to match the length of the dtype, the missing names will be defined with this default template:

```
>>> data = StringIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, dtype=(int, float, int), names="a")
array([(1, 2.0, 3), (4, 5.0, 6)],
      dtype=[('a', '<i8'), ('f0', '<f8'), ('f1', '<i8')])
```

We can overwrite this default with the `defaultfmt` argument, that takes any format string:

```
>>> data = StringIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, dtype=(int, float, int), defaultfmt="var_%02i")
array([(1, 2.0, 3), (4, 5.0, 6)],
      dtype=[('var_00', '<i8'), ('var_01', '<f8'), ('var_02', '<i8')])
```

Note: We need to keep in mind that `defaultfmt` is used only if some names are expected but not defined.

Validating names

NumPy arrays with a structured dtype can also be viewed as `recarray`, where a field can be accessed as if it were an attribute. For that reason, we may need to make sure that the field name doesn't contain any space or invalid character, or that it does not correspond to the name of a standard attribute (like `size` or `shape`), which would confuse the interpreter. `genfromtxt` accepts three optional arguments that provide a finer control on the names:

`deletechars` Gives a string combining all the characters that must be deleted from the name. By default, invalid characters are `~!@#$%^&*()-=+~\|]}[{';: /?.>,<`.

`excludelist` Gives a list of the names to exclude, such as `return`, `file`, `print`… If one of the input name is part of this list, an underscore character (`'_'`) will be appended to it.

`case_sensitive` Whether the names should be case-sensitive (`case_sensitive=True`), converted to upper case (`case_sensitive=False` or `case_sensitive='upper'`) or to lower case (`case_sensitive='lower'`).

Tweaking the conversion

The `converters` argument

Usually, defining a dtype is sufficient to define how the sequence of strings must be converted. However, some additional control may sometimes be required. For example, we may want to make sure that a date in a format `YYYY/MM/DD` is converted to a `datetime` object, or that a string like `xx%` is properly converted to a float between 0 and 1. In such cases, we should define conversion functions with the `converters` arguments.

The value of this argument is typically a dictionary with column indices or column names as keys and a conversion functions as values. These conversion functions can either be actual functions or lambda functions. In any case, they should accept only a string as input and output only a single element of the wanted type.

In the following example, the second column is converted from as string representing a percentage to a float between 0 and 1:

```
>>> convertfunc = lambda x: float(x.strip("%"))/100.
>>> data = u"1, 2.3%, 45.\n6, 78.9%, 0"
>>> names = ("i", "p", "n")
>>> # General case .....
>>> np.genfromtxt(StringIO(data), delimiter=",", names=names)
array([(1.0, nan, 45.0), (6.0, nan, 0.0)],
      dtype=[('i', '<f8'), ('p', '<f8'), ('n', '<f8')])
```

We need to keep in mind that by default, `dtype=float`. A float is therefore expected for the second column. However, the strings `' 2.3%'` and `' 78.9%'` cannot be converted to float and we end up having `np.nan` instead. Let's now use a converter:

```
>>> # Converted case ...
>>> np.genfromtxt(StringIO(data), delimiter=",", names=names,
...               converters={1: convertfunc})
array([(1.0, 0.023, 45.0), (6.0, 0.78900000000000003, 0.0)],
      dtype=[('i', '<f8'), ('p', '<f8'), ('n', '<f8')])
```

The same results can be obtained by using the name of the second column (`"p"`) as key instead of its index (1):

```
>>> # Using a name for the converter ...
>>> np.genfromtxt(StringIO(data), delimiter=",", names=names,
...               converters={"p": convertfunc})
array([(1.0, 0.023, 45.0), (6.0, 0.78900000000000003, 0.0)],
      dtype=[('i', '<f8'), ('p', '<f8'), ('n', '<f8')])
```

Converters can also be used to provide a default for missing entries. In the following example, the converter `convert` transforms a stripped string into the corresponding float or into -999 if the string is empty. We need to explicitly strip the string from white spaces as it is not done by default:

```
>>> data = u"1, , 3\n 4, 5, 6"
>>> convert = lambda x: float(x.strip() or -999)
>>> np.genfromtxt(StringIO(data), delimiter=",",
...               converters={1: convert})
array([[   1., -999.,    3.],
       [   4.,    5.,    6.]])
```

Using missing and filling values

Some entries may be missing in the dataset we are trying to import. In a previous example, we used a converter to transform an empty string into a float. However, user-defined converters may rapidly become cumbersome to manage.

The `genfromtxt` function provides two other complementary mechanisms: the `missing_values` argument is used to recognize missing data and a second argument, `filling_values`, is used to process these missing data.

`missing_values`

By default, any empty string is marked as missing. We can also consider more complex strings, such as `"N/A"` or `"???"` to represent missing or invalid data. The `missing_values` argument accepts three kind of values:

a string or a comma-separated string This string will be used as the marker for missing data for all the columns

a sequence of strings In that case, each item is associated to a column, in order.

a dictionary Values of the dictionary are strings or sequence of strings. The corresponding keys can be column indices (integers) or column names (strings). In addition, the special key `None` can be used to define a default applicable to all columns.

`filling_values`

We know how to recognize missing data, but we still need to provide a value for these missing entries. By default, this value is determined from the expected dtype according to this table:

Expected type	Default
`bool`	`False`
`int`	`-1`
`float`	`np.nan`
`complex`	`np.nan+0j`
`string`	`'???'`

We can get a finer control on the conversion of missing values with the `filling_values` optional argument. Like `missing_values`, this argument accepts different kind of values:

a single value This will be the default for all columns

a sequence of values Each entry will be the default for the corresponding column

a dictionary Each key can be a column index or a column name, and the corresponding value should be a single object. We can use the special key `None` to define a default for all columns.

In the following example, we suppose that the missing values are flagged with `"N/A"` in the first column and by `"???"` in the third column. We wish to transform these missing values to 0 if they occur in the first and second column, and to -999 if they occur in the last column:

```
>>> data = u"N/A, 2, 3\n4, ,???"
>>> kwargs = dict(delimiter=",",
...               dtype=int,
...               names="a,b,c",
...               missing_values={0:"N/A", 'b':" ", 2:"???"},
...               filling_values={0:0, 'b':0, 2:-999})
>>> np.genfromtxt(StringIO(data), **kwargs)
```

(continues on next page)

(continued from previous page)

```
array([(0, 2, 3), (4, 0, -999)],
      dtype=[('a', '<i8'), ('b', '<i8'), ('c', '<i8')])
```

usemask

We may also want to keep track of the occurrence of missing data by constructing a boolean mask, with `True` entries where data was missing and `False` otherwise. To do that, we just have to set the optional argument `usemask` to `True` (the default is `False`). The output array will then be a `MaskedArray`.

Shortcut functions

In addition to `genfromtxt`, the `numpy.lib.io` module provides several convenience functions derived from `genfromtxt`. These functions work the same way as the original, but they have different default values.

`recfromtxt` Returns a standard `numpy.recarray` (if `usemask=False`) or a `MaskedRecords` array (if `usemaske=True`). The default dtype is `dtype=None`, meaning that the types of each column will be automatically determined.

`recfromcsv` Like `recfromtxt`, but with a default `delimiter=","`.

3.4 Indexing

See also:

Indexing routines

Array indexing refers to any use of the square brackets ([]) to index array values. There are many options to indexing, which give numpy indexing great power, but with power comes some complexity and the potential for confusion. This section is just an overview of the various options and issues related to indexing. Aside from single element indexing, the details on most of these options are to be found in related sections.

3.4.1 Assignment vs referencing

Most of the following examples show the use of indexing when referencing data in an array. The examples work just as well when assigning to an array. See the section at the end for specific examples and explanations on how assignments work.

3.4.2 Single element indexing

Single element indexing for a 1-D array is what one expects. It work exactly like that for other standard Python sequences. It is 0-based, and accepts negative indices for indexing from the end of the array.

```
>>> x = np.arange(10)
>>> x[2]
2
>>> x[-2]
8
```

Unlike lists and tuples, numpy arrays support multidimensional indexing for multidimensional arrays. That means that it is not necessary to separate each dimension's index into its own set of square brackets.

```
>>> x.shape = (2,5) # now x is 2-dimensional
>>> x[1,3]
8
>>> x[1,-1]
9
```

Note that if one indexes a multidimensional array with fewer indices than dimensions, one gets a subdimensional array. For example:

```
>>> x[0]
array([0, 1, 2, 3, 4])
```

That is, each index specified selects the array corresponding to the rest of the dimensions selected. In the above example, choosing 0 means that the remaining dimension of length 5 is being left unspecified, and that what is returned is an array of that dimensionality and size. It must be noted that the returned array is not a copy of the original, but points to the same values in memory as does the original array. In this case, the 1-D array at the first position (0) is returned. So using a single index on the returned array, results in a single element being returned. That is:

```
>>> x[0][2]
2
```

So note that `x[0,2] = x[0][2]` though the second case is more inefficient as a new temporary array is created after the first index that is subsequently indexed by 2.

Note to those used to IDL or Fortran memory order as it relates to indexing. NumPy uses C-order indexing. That means that the last index usually represents the most rapidly changing memory location, unlike Fortran or IDL, where the first index represents the most rapidly changing location in memory. This difference represents a great potential for confusion.

3.4.3 Other indexing options

It is possible to slice and stride arrays to extract arrays of the same number of dimensions, but of different sizes than the original. The slicing and striding works exactly the same way it does for lists and tuples except that they can be applied to multiple dimensions as well. A few examples illustrates best:

```
>>> x = np.arange(10)
>>> x[2:5]
array([2, 3, 4])
>>> x[:-7]
array([0, 1, 2])
>>> x[1:7:2]
array([1, 3, 5])
>>> y = np.arange(35).reshape(5,7)
>>> y[1:5:2,::3]
array([[ 7, 10, 13],
       [21, 24, 27]])
```

Note that slices of arrays do not copy the internal array data but only produce new views of the original data. This is different from list or tuple slicing and an explicit `copy()` is recommended if the original data is not required anymore.

It is possible to index arrays with other arrays for the purposes of selecting lists of values out of arrays into new arrays. There are two different ways of accomplishing this. One uses one or more arrays of index values. The other involves giving a boolean array of the proper shape to indicate the values to be selected. Index arrays are a very powerful tool that allow one to avoid looping over individual elements in arrays and thus greatly improve performance.

It is possible to use special features to effectively increase the number of dimensions in an array through indexing so the resulting array acquires the shape needed for use in an expression or with a specific function.

3.4.4 Index arrays

NumPy arrays may be indexed with other arrays (or any other sequence- like object that can be converted to an array, such as lists, with the exception of tuples; see the end of this document for why this is). The use of index arrays ranges from simple, straightforward cases to complex, hard-to-understand cases. For all cases of index arrays, what is returned is a copy of the original data, not a view as one gets for slices.

Index arrays must be of integer type. Each value in the array indicates which value in the array to use in place of the index. To illustrate:

```
>>> x = np.arange(10,1,-1)
>>> x
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[np.array([3, 3, 1, 8])]
array([7, 7, 9, 2])
```

The index array consisting of the values 3, 3, 1 and 8 correspondingly create an array of length 4 (same as the index array) where each index is replaced by the value the index array has in the array being indexed.

Negative values are permitted and work as they do with single indices or slices:

```
>>> x[np.array([3,3,-3,8])]
array([7, 7, 4, 2])
```

It is an error to have index values out of bounds:

```
>>> x[np.array([3, 3, 20, 8])]
<type 'exceptions.IndexError'>: index 20 out of bounds 0<=index<9
```

Generally speaking, what is returned when index arrays are used is an array with the same shape as the index array, but with the type and values of the array being indexed. As an example, we can use a multidimensional index array instead:

```
>>> x[np.array([[1,1],[2,3]])]
array([[9, 9],
       [8, 7]])
```

3.4.5 Indexing Multi-dimensional arrays

Things become more complex when multidimensional arrays are indexed, particularly with multidimensional index arrays. These tend to be more unusual uses, but they are permitted, and they are useful for some problems. We'll start with the simplest multidimensional case (using the array y from the previous examples):

```
>>> y[np.array([0,2,4]), np.array([0,1,2])]
array([ 0, 15, 30])
```

In this case, if the index arrays have a matching shape, and there is an index array for each dimension of the array being indexed, the resultant array has the same shape as the index arrays, and the values correspond to the index set for each position in the index arrays. In this example, the first index value is 0 for both index arrays, and thus the first value of the resultant array is y[0,0]. The next value is y[2,1], and the last is y[4,2].

If the index arrays do not have the same shape, there is an attempt to broadcast them to the same shape. If they cannot be broadcast to the same shape, an exception is raised:

```
>>> y[np.array([0,2,4]), np.array([0,1])]
<type 'exceptions.ValueError'>: shape mismatch: objects cannot be
broadcast to a single shape
```

The broadcasting mechanism permits index arrays to be combined with scalars for other indices. The effect is that the scalar value is used for all the corresponding values of the index arrays:

```
>>> y[np.array([0,2,4]), 1]
array([ 1, 15, 29])
```

Jumping to the next level of complexity, it is possible to only partially index an array with index arrays. It takes a bit of thought to understand what happens in such cases. For example if we just use one index array with y:

```
>>> y[np.array([0,2,4])]
array([[ 0,  1,  2,  3,  4,  5,  6],
       [14, 15, 16, 17, 18, 19, 20],
       [28, 29, 30, 31, 32, 33, 34]])
```

What results is the construction of a new array where each value of the index array selects one row from the array being indexed and the resultant array has the resulting shape (number of index elements, size of row).

An example of where this may be useful is for a color lookup table where we want to map the values of an image into RGB triples for display. The lookup table could have a shape (nlookup, 3). Indexing such an array with an image with shape (ny, nx) with dtype=np.uint8 (or any integer type so long as values are with the bounds of the lookup table) will result in an array of shape (ny, nx, 3) where a triple of RGB values is associated with each pixel location.

In general, the shape of the resultant array will be the concatenation of the shape of the index array (or the shape that all the index arrays were broadcast to) with the shape of any unused dimensions (those not indexed) in the array being indexed.

3.4.6 Boolean or "mask" index arrays

Boolean arrays used as indices are treated in a different manner entirely than index arrays. Boolean arrays must be of the same shape as the initial dimensions of the array being indexed. In the most straightforward case, the boolean array has the same shape:

```
>>> b = y>20
>>> y[b]
array([21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34])
```

Unlike in the case of integer index arrays, in the boolean case, the result is a 1-D array containing all the elements in the indexed array corresponding to all the true elements in the boolean array. The elements in the indexed array are always iterated and returned in row-major (C-style) order. The result is also identical to `y[np.nonzero(b)]`. As with index arrays, what is returned is a copy of the data, not a view as one gets with slices.

The result will be multidimensional if y has more dimensions than b. For example:

```
>>> b[:,5] # use a 1-D boolean whose first dim agrees with the first dim of y
array([False, False, False,  True,  True])
>>> y[b[:,5]]
array([[21, 22, 23, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])
```

Here the 4th and 5th rows are selected from the indexed array and combined to make a 2-D array.

In general, when the boolean array has fewer dimensions than the array being indexed, this is equivalent to y[b, …], which means y is indexed by b followed by as many : as are needed to fill out the rank of y. Thus the shape of the result is one dimension containing the number of True elements of the boolean array, followed by the remaining dimensions of the array being indexed.

For example, using a 2-D boolean array of shape (2,3) with four True elements to select rows from a 3-D array of shape (2,3,5) results in a 2-D result of shape (4,5):

```
>>> x = np.arange(30).reshape(2,3,5)
>>> x
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, 24],
        [25, 26, 27, 28, 29]]])
>>> b = np.array([[True, True, False], [False, True, True]])
>>> x[b]
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29]])
```

For further details, consult the numpy reference documentation on array indexing.

3.4.7 Combining index arrays with slices

Index arrays may be combined with slices. For example:

```
>>> y[np.array([0,2,4]),1:3]
array([[ 1,  2],
       [15, 16],
       [29, 30]])
```

In effect, the slice is converted to an index array np.array([[1,2]]) (shape (1,2)) that is broadcast with the index array to produce a resultant array of shape (3,2).

Likewise, slicing can be combined with broadcasted boolean indices:

```
>>> b = y > 20
>>> b
array([[False, False, False, False, False, False, False],
      [False, False, False, False, False, False, False],
      [False, False, False, False, False, False, False],
      [ True,  True,  True,  True,  True,  True,  True],
      [ True,  True,  True,  True,  True,  True,  True]])
>>> y[b[:,5],1:3]
array([[22, 23],
       [29, 30]])
```

3.4.8 Structural indexing tools

To facilitate easy matching of array shapes with expressions and in assignments, the np.newaxis object can be used within array indices to add new dimensions with a size of 1. For example:

```
>>> y.shape
(5, 7)
```

(continues on next page)

(continued from previous page)

```
>>> y[:,np.newaxis,:].shape
(5, 1, 7)
```

Note that there are no new elements in the array, just that the dimensionality is increased. This can be handy to combine two arrays in a way that otherwise would require explicitly reshaping operations. For example:

```
>>> x = np.arange(5)
>>> x[:,np.newaxis] + x[np.newaxis,:]
array([[0, 1, 2, 3, 4],
       [1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7],
       [4, 5, 6, 7, 8]])
```

The ellipsis syntax maybe used to indicate selecting in full any remaining unspecified dimensions. For example:

```
>>> z = np.arange(81).reshape(3,3,3,3)
>>> z[1,...,2]
array([[29, 32, 35],
       [38, 41, 44],
       [47, 50, 53]])
```

This is equivalent to:

```
>>> z[1,:,:,2]
array([[29, 32, 35],
       [38, 41, 44],
       [47, 50, 53]])
```

3.4.9 Assigning values to indexed arrays

As mentioned, one can select a subset of an array to assign to using a single index, slices, and index and mask arrays. The value being assigned to the indexed array must be shape consistent (the same shape or broadcastable to the shape the index produces). For example, it is permitted to assign a constant to a slice:

```
>>> x = np.arange(10)
>>> x[2:7] = 1
```

or an array of the right size:

```
>>> x[2:7] = np.arange(5)
```

Note that assignments may result in changes if assigning higher types to lower types (like floats to ints) or even exceptions (assigning complex to floats or ints):

```
>>> x[1] = 1.2
>>> x[1]
1
>>> x[1] = 1.2j
<type 'exceptions.TypeError'>: can't convert complex to long; use
long(abs(z))
```

Unlike some of the references (such as array and mask indices) assignments are always made to the original data in the array (indeed, nothing else would make sense!). Note though, that some actions may not work as one may naively expect. This particular example is often surprising to people:

```
>>> x = np.arange(0, 50, 10)
>>> x
array([ 0, 10, 20, 30, 40])
>>> x[np.array([1, 1, 3, 1])] += 1
>>> x
array([ 0, 11, 20, 31, 40])
```

Where people expect that the 1st location will be incremented by 3. In fact, it will only be incremented by 1. The reason is because a new array is extracted from the original (as a temporary) containing the values at 1, 1, 3, 1, then the value 1 is added to the temporary, and then the temporary is assigned back to the original array. Thus the value of the array at x[1]+1 is assigned to x[1] three times, rather than being incremented 3 times.

3.4.10 Dealing with variable numbers of indices within programs

The index syntax is very powerful but limiting when dealing with a variable number of indices. For example, if you want to write a function that can handle arguments with various numbers of dimensions without having to write special case code for each number of possible dimensions, how can that be done? If one supplies to the index a tuple, the tuple will be interpreted as a list of indices. For example (using the previous definition for the array z):

```
>>> indices = (1,1,1,1)
>>> z[indices]
40
```

So one can use code to construct tuples of any number of indices and then use these within an index.

Slices can be specified within programs by using the slice() function in Python. For example:

```
>>> indices = (1,1,1,slice(0,2)) # same as [1,1,1,0:2]
>>> z[indices]
array([39, 40])
```

Likewise, ellipsis can be specified by code by using the Ellipsis object:

```
>>> indices = (1, Ellipsis, 1) # same as [1,...,1]
>>> z[indices]
array([[28, 31, 34],
       [37, 40, 43],
       [46, 49, 52]])
```

For this reason it is possible to use the output from the np.nonzero() function directly as an index since it always returns a tuple of index arrays.

Because the special treatment of tuples, they are not automatically converted to an array as a list would be. As an example:

```
>>> z[[1,1,1,1]] # produces a large array
array([[[[27, 28, 29],
         [30, 31, 32], ...
>>> z[(1,1,1,1)] # returns a single value
40
```

3.5 Broadcasting

See also:

`numpy.broadcast`

array-broadcasting-in-numpy An introduction to the concepts discussed here

Note: See this article for illustrations of broadcasting concepts.

The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is "broadcast" across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python. It does this without making needless copies of data and usually leads to efficient algorithm implementations. There are, however, cases where broadcasting is a bad idea because it leads to inefficient use of memory that slows computation.

NumPy operations are usually done on pairs of arrays on an element-by-element basis. In the simplest case, the two arrays must have exactly the same shape, as in the following example:

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = np.array([2.0, 2.0, 2.0])
>>> a * b
array([ 2.,  4.,  6.])
```

NumPy's broadcasting rule relaxes this constraint when the arrays' shapes meet certain constraints. The simplest broadcasting example occurs when an array and a scalar value are combined in an operation:

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([ 2.,  4.,  6.])
```

The result is equivalent to the previous example where `b` was an array. We can think of the scalar `b` being *stretched* during the arithmetic operation into an array with the same shape as `a`. The new elements in `b` are simply copies of the original scalar. The stretching analogy is only conceptual. NumPy is smart enough to use the original scalar value without actually making copies, so that broadcasting operations are as memory and computationally efficient as possible.

The code in the second example is more efficient than that in the first because broadcasting moves less memory around during the multiplication (`b` is a scalar rather than an array).

3.5.1 General Broadcasting Rules

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward. Two dimensions are compatible when

1) they are equal, or
2) one of them is 1

If these conditions are not met, a `ValueError: operands could not be broadcast together` exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the maximum size along each dimension of the input arrays.

Arrays do not need to have the same *number* of dimensions. For example, if you have a `256x256x3` array of RGB values, and you want to scale each color in the image by a different value, you can multiply the image by a one-dimensional array with 3 values. Lining up the sizes of the trailing axes of these arrays according to the broadcast rules, shows that they are compatible:

```
Image  (3d array): 256 x 256 x 3
Scale  (1d array):             3
Result (3d array): 256 x 256 x 3
```

When either of the dimensions compared is one, the other is used. In other words, dimensions with size 1 are stretched or "copied" to match the other.

In the following example, both the `A` and `B` arrays have axes with length one that are expanded to a larger size during the broadcast operation:

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

Here are some more examples:

```
A      (2d array):  5 x 4
B      (1d array):      1
Result (2d array):  5 x 4

A      (2d array):  5 x 4
B      (1d array):      4
Result (2d array):  5 x 4

A      (3d array):  15 x 3 x 5
B      (3d array):  15 x 1 x 5
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 5
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 1
Result (3d array):  15 x 3 x 5
```

Here are examples of shapes that do not broadcast:

```
A      (1d array):  3
B      (1d array):  4 # trailing dimensions do not match

A      (2d array):      2 x 1
B      (3d array):  8 x 4 x 3 # second from last dimensions mismatched
```

An example of broadcasting in practice:

```
>>> x = np.arange(4)
>>> xx = x.reshape(4,1)
>>> y = np.ones(5)
>>> z = np.ones((3,4))

>>> x.shape
(4,)

>>> y.shape
(5,)
```

(continues on next page)