
NumPy User Guide

Release 1.23.0

Written by the NumPy community

June 22, 2022

CONTENTS

1	What is NumPy?	3
2	NumPy quickstart	5
3	NumPy: the absolute basics for beginners	29
4	NumPy fundamentals	63
5	Miscellaneous	157
6	NumPy for MATLAB users	161
7	Building from source	173
8	Using NumPy C-API	179
9	NumPy How Tos	221
10	For downstream package authors	235
11	F2PY user guide and reference manual	239
12	Glossary	301
13	Under-the-hood Documentation for developers	309
14	Reporting bugs	321
15	Release notes	323
16	NumPy license	555
	Python Module Index	557
	Index	559

This guide is an overview and explains the important features; details are found in reference.

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 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 [Broadcasting](#).

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.

NUMPY QUICKSTART

2.1 Prerequisites

You'll need to know a bit of Python. For a refresher, see the [Python tutorial](#).

To work the examples, you'll need `matplotlib` installed in addition to NumPy.

Learner profile

This is a quick overview of arrays in NumPy. It demonstrates how n-dimensional ($n \geq 2$) arrays are represented and can be manipulated. In particular, if you don't know how to apply common functions to n-dimensional arrays (without using for-loops), or if you want to understand axis and shape properties for n-dimensional arrays, this article might be of help.

Learning Objectives

After reading, you should be able to:

- Understand the difference between one-, two- and n-dimensional arrays in NumPy;
- Understand how to apply some linear algebra operations to n-dimensional arrays without using for-loops;
- Understand axis and shape properties for n-dimensional arrays.

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 non-negative integers. In NumPy dimensions are called *axes*.

For example, the array for 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)
<class 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<class '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 arguments, rather than providing a single sequence as an argument.

```
>>> a = np.array(1, 2, 3, 4)      # WRONG
Traceback (most recent call last):
...
TypeError: array() takes from 1 to 2 positional arguments but 4 were given
>>> 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`, but it can be specified via the key word argument `dtype`.

```
>>> 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)
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]]])
```

(continues on next page)

(continued from previous page)

```

        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
>>> np.empty((2, 3))
array([[3.73603959e-262, 6.02658058e-154, 6.55490914e-260], # may vary
       [5.30498948e-313, 3.14673309e-307, 1.00000000e+000]])

```

To create sequences of numbers, NumPy provides the `arange` function which is analogous to the Python built-in `range`, but returns an array.

```

>>> 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.Generator.rand`, `numpy.random.Generator.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]

```

(continues on next page)

(continued from previous page)

```
[ 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=sys.maxsize) # sys module should be imported
```

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
>>> 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],
```

(continues on next page)

(continued from previous page)

```

    [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.

```

>>> rg = np.random.default_rng(1) # create instance of default random number_
↳generator
>>> a = np.ones((2, 3), dtype=int)
>>> b = rg.random((2, 3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[3.51182162, 3.9504637 , 3.14415961],
       [3.94864945, 3.31183145, 3.42332645]])
>>> a += b # b is not automatically converted to integer type
Traceback (most recent call last):
...
numpy.core._exceptions._UFuncOutputCastingError: 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'
>>> 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 = rg.random((2, 3))
>>> a
array([[0.82770259, 0.40919914, 0.54959369],
       [0.02755911, 0.75351311, 0.53814331]])
>>> a.sum()
3.1057109529998157

```

(continues on next page)

(continued from previous page)

```
>>> a.min()
0.027559113243068367
>>> a.max()
0.8277025938204418
```

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.])
>>> 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`, `invert`, `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])
>>> # equivalent to a[0:6:2] = 1000;
>>> # from start to position 6, exclusive, set every 2nd element to 1000
>>> a[6:2] = 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.))
...
9.999999999999998
1.0
9.999999999999998
3.0
9.999999999999998
4.999999999999999
5.999999999999999
6.999999999999999
7.999999999999999
8.999999999999998
```

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
>>> 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
3
10
11
12
13
20
21
22
23
30
31
32
33
40
```

(continues on next page)

(continued from previous page)

```
41
42
43
```

See also:

Indexing on ndarrays, `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 * rg.random((3, 4)))
>>> a
array([[3., 7., 3., 4.],
       [1., 4., 2., 2.],
       [7., 2., 4., 9.]])
>>> 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([3., 7., 3., 4., 1., 4., 2., 2., 7., 2., 4., 9.])
>>> a.reshape(6, 2) # returns the array with a modified shape
array([[3., 7.],
       [3., 4.],
       [1., 4.],
       [2., 2.],
       [7., 2.],
       [4., 9.]])
>>> a.T # returns the array, transposed
array([[3., 1., 7.],
       [7., 4., 2.],
       [3., 2., 4.],
       [4., 2., 9.]])
>>> a.T.shape
(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([[3., 7., 3., 4.],
       [1., 4., 2., 2.],
       [7., 2., 4., 9.]])
>>> a.resize((2, 6))
>>> a
array([[3., 7., 3., 4., 1., 4.],
       [2., 2., 7., 2., 4., 9.]])
```

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

```
>>> a.reshape(3, -1)
array([[3., 7., 3., 4.],
       [1., 4., 2., 2.],
       [7., 2., 4., 9.]])
```

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 * rg.random((2, 2)))
>>> a
array([[9., 7.],
       [5., 2.]])
>>> b = np.floor(10 * rg.random((2, 2)))
>>> b
array([[1., 9.],
       [5., 1.]])
>>> np.vstack((a, b))
array([[9., 7.],
       [5., 2.],
       [1., 9.],
       [5., 1.]])
>>> np.hstack((a, b))
array([[9., 7., 1., 9.],
       [5., 2., 5., 1.]])
```

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([[9., 7., 1., 9.],
       [5., 2., 5., 1.]])
>>> 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] # view `a` as a 2D column vector
```

(continues on next page)

(continued from previous page)

```

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 fact, `row_stack` is an alias for `vstack`:

```

>>> np.column_stack is np.hstack
False
>>> np.row_stack is np.vstack
True

```

In general, for arrays 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 * rg.random((2, 12)))
>>> a
array([[6., 7., 6., 9., 0., 5., 4., 0., 6., 8., 5., 2.],
       [8., 5., 5., 7., 1., 8., 6., 7., 1., 8., 1., 0.]])
>>> # Split `a` into 3
>>> np.hsplit(a, 3)
[array([[6., 7., 6., 9.],
       [8., 5., 5., 7.]]) , array([[0., 5., 4., 0.],
       [1., 8., 6., 7.]]) , array([[6., 8., 5., 2.],
       [1., 8., 1., 0.]])]
>>> # Split `a` after the third and the fourth column
>>> np.hsplit(a, (3, 4))
[array([[6., 7., 6.],
       [8., 5., 5.]]) , array([[9.],

```

(continues on next page)

(continued from previous page)

```
[7.]]), array([[0., 5., 4., 0., 6., 8., 5., 2.],
               [1., 8., 6., 7., 1., 8., 1., 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 objects or their data.

```
>>> a = np.array([[ 0,  1,  2,  3],
...               [ 4,  5,  6,  7],
...               [ 8,  9, 10, 11]])
>>> b = a           # no new object is created
>>> b is a          # a and b are two names for the same ndarray object
True
```

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           # may vary
>>> f(a)
148293216           # may vary
```

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 = c.reshape((2, 6)) # a's shape doesn't change
>>> a.shape
(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]
>>> 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 Advanced 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]
array([[ 2,  1],
       [ 3,  3],

       [ 6,  5],
       [ 7,  7],

       [10,  9],
       [11, 11]])

```

In Python, `arr[i, j]` is exactly the same as `arr[(i, j)]`—so we can put `i` and `j` in a tuple and then do the indexing with that.

```

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

```

(continues on next page)

(continued from previous page)

```
[ 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])
>>> # not what we want
>>> a[s]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 3 is out of bounds for axis 0 with size 3
>>> # same as `a[i, j]`
>>> a[tuple(s)]
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]])
>>> # index of the maxima for each series
>>> ind = data.argmax(axis=0)
>>> ind
array([2, 0, 3, 1])
>>> # times corresponding to the maxima
>>> time_max = time[ind]
>>>
>>> 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, r=2):
...     """Returns an image of the Mandelbrot fractal of size (h,w)."""
...     x = np.linspace(-2.5, 1.5, 4*h+1)
...     y = np.linspace(-1.5, 1.5, 3*w+1)
...     A, B = np.meshgrid(x, y)
...     C = A + B*1j
...     z = np.zeros_like(C)
...     divtime = maxit + np.zeros(z.shape, dtype=int)
...
...     for i in range(maxit):
...         z = z**2 + C
...         diverge = abs(z) > r # who is diverging
```

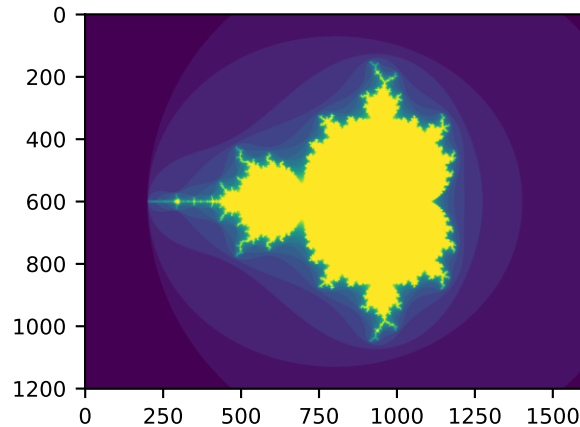
(continues on next page)

(continued from previous page)

```

...     div_now = diverge & (divtime == maxit) # who is diverging now
...     divtime[div_now] = i                   # note when
...     z[diverge] = r                         # avoid diverging too much
...
...     return divtime
>>> plt.clf()
>>> plt.imshow(mandelbrot(400, 400))

```



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])
>>> 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 Tricks and Tips

Here we give a list of short and useful tips.

2.7.1 “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)
>>> b = a.reshape((2, -1, 3)) # -1 means "whatever is needed"
>>> b.shape
(2, 5, 3)
>>> b
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]])
```

2.7.2 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)
>>> y = np.arange(5)
>>> m = np.vstack([x, y])
>>> m
array([[0, 2, 4, 6, 8],
       [0, 1, 2, 3, 4]])
>>> xy = np.hstack([x, y])
>>> xy
array([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

2.7.3 Histograms

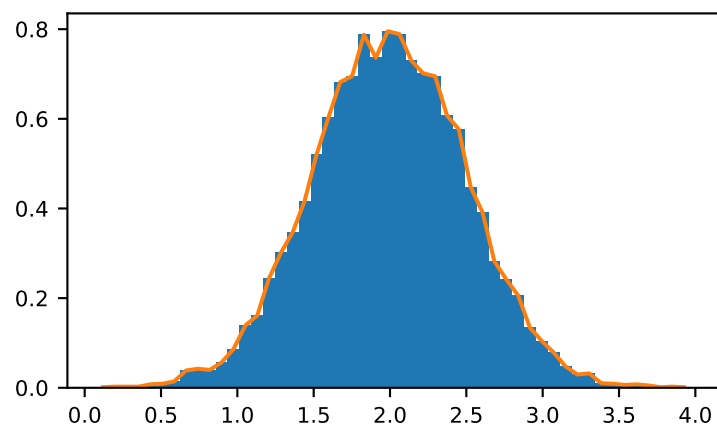
The NumPy histogram function applied to an array returns a pair of vectors: the histogram of the array and a vector of the bin edges. Beware: `matplotlib` also has a function to build histograms (called `hist`, as in Matlab) that differs from the one in NumPy. The main difference is that `pylab.hist`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

```
>>> import numpy as np
>>> rg = np.random.default_rng(1)
>>> 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 = rg.normal(mu, sigma, 10000)
>>> # Plot a normalized histogram with 50 bins
>>> plt.hist(v, bins=50, density=True) # matplotlib version (plot)
(array...)
>>> # 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)
```

With Matplotlib `>=3.4` you can also use `plt.stairs(n, bins)`.

2.8 Further reading

- [The Python tutorial](#)
- [reference](#)
- [SciPy Tutorial](#)
- [SciPy Lecture Notes](#)
- [A matlab, R, IDL, NumPy/SciPy dictionary](#)
- [tutorial-svd](#)



NUMPY: THE ABSOLUTE BASICS FOR BEGINNERS

Welcome to the absolute beginner's guide to NumPy! If you have comments or suggestions, please don't hesitate to [reach out](#)!

3.1 Welcome to NumPy!

NumPy (**N**umerical **P**ython) is an open source Python library that's used in almost every field of science and engineering. It's the universal standard for working with numerical data in Python, and it's at the core of the scientific Python and PyData ecosystems. NumPy users include everyone from beginning coders to experienced researchers doing state-of-the-art scientific and industrial research and development. The NumPy API is used extensively in Pandas, SciPy, Matplotlib, scikit-learn, scikit-image and most other data science and scientific Python packages.

The NumPy library contains multidimensional array and matrix data structures (you'll find more information about this in later sections). It provides **ndarray**, a homogeneous n-dimensional array object, with methods to efficiently operate on it. NumPy can be used to perform a wide variety of mathematical operations on arrays. It adds powerful data structures to Python that guarantee efficient calculations with arrays and matrices and it supplies an enormous library of high-level mathematical functions that operate on these arrays and matrices.

Learn more about *NumPy* [here](#)!

3.2 Installing NumPy

To install NumPy, we strongly recommend using a scientific Python distribution. If you're looking for the full instructions for installing NumPy on your operating system, see [Installing NumPy](#).

If you already have Python, you can install NumPy with:

```
conda install numpy
```

or

```
pip install numpy
```

If you don't have Python yet, you might want to consider using [Anaconda](#). It's the easiest way to get started. The good thing about getting this distribution is the fact that you don't need to worry too much about separately installing NumPy or any of the major packages that you'll be using for your data analyses, like pandas, Scikit-Learn, etc.

3.3 How to import NumPy

To access NumPy and its functions import it in your Python code like this:

```
import numpy as np
```

We shorten the imported name to `np` for better readability of code using NumPy. This is a widely adopted convention that you should follow so that anyone working with your code can easily understand it.

3.4 Reading the example code

If you aren't already comfortable with reading tutorials that contain a lot of code, you might not know how to interpret a code block that looks like this:

```
>>> a = np.arange(6)
>>> a2 = a[np.newaxis, :]
>>> a2.shape
(1, 6)
```

If you aren't familiar with this style, it's very easy to understand. If you see `>>>`, you're looking at **input**, or the code that you would enter. Everything that doesn't have `>>>` in front of it is **output**, or the results of running your code. This is the style you see when you run `python` on the command line, but if you're using IPython, you might see a different style. Note that it is not part of the code and will cause an error if typed or pasted into the Python shell. It can be safely typed or pasted into the IPython shell; the `>>>` is ignored.

3.5 What's the difference between a Python list and a NumPy array?

NumPy gives you an enormous range of fast and efficient ways of creating arrays and manipulating numerical data inside them. While a Python list can contain different data types within a single list, all of the elements in a NumPy array should be homogeneous. The mathematical operations that are meant to be performed on arrays would be extremely inefficient if the arrays weren't homogeneous.

Why use NumPy?

NumPy arrays are faster and more compact than Python lists. An array consumes less memory and is convenient to use. NumPy uses much less memory to store data and it provides a mechanism of specifying the data types. This allows the code to be optimized even further.

3.6 What is an array?

An array is a central data structure of the NumPy library. An array is a grid of values and it contains information about the raw data, how to locate an element, and how to interpret an element. It has a grid of elements that can be indexed in *various ways*. The elements are all of the same type, referred to as the array `dtype`.

An array can be indexed by a tuple of nonnegative integers, by booleans, by another array, or by integers. The `rank` of the array is the number of dimensions. The `shape` of the array is a tuple of integers giving the size of the array along each dimension.

One way we can initialize NumPy arrays is from Python lists, using nested lists for two- or higher-dimensional data.

For example:

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
```

or:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

We can access the elements in the array using square brackets. When you're accessing elements, remember that indexing in NumPy starts at 0. That means that if you want to access the first element in your array, you'll be accessing element "0".

```
>>> print(a[0])
[1 2 3 4]
```

3.7 More information about arrays

This section covers 1D array, 2D array, ndarray, vector, matrix

You might occasionally hear an array referred to as a "ndarray," which is shorthand for "N-dimensional array." An N-dimensional array is simply an array with any number of dimensions. You might also hear **1-D**, or one-dimensional array, **2-D**, or two-dimensional array, and so on. The NumPy `ndarray` class is used to represent both matrices and vectors. A **vector** is an array with a single dimension (there's no difference between row and column vectors), while a **matrix** refers to an array with two dimensions. For **3-D** or higher dimensional arrays, the term **tensor** is also commonly used.

What are the attributes of an array?

An array is usually a fixed-size container of items of the same type and size. The number of dimensions and items in an array is defined by its shape. The shape of an array is a tuple of non-negative integers that specify the sizes of each dimension.

In NumPy, dimensions are called **axes**. This means that if you have a 2D array that looks like this:

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

Your array has 2 axes. The first axis has a length of 2 and the second axis has a length of 3.

Just like in other Python container objects, the contents of an array can be accessed and modified by indexing or slicing the array. Unlike the typical container objects, different arrays can share the same data, so changes made on one array might be visible in another.

Array **attributes** reflect information intrinsic to the array itself. If you need to get, or even set, properties of an array without creating a new array, you can often access an array through its attributes.

Read more about array attributes [here](#) and learn about array objects [here](#).

3.8 How to create a basic array

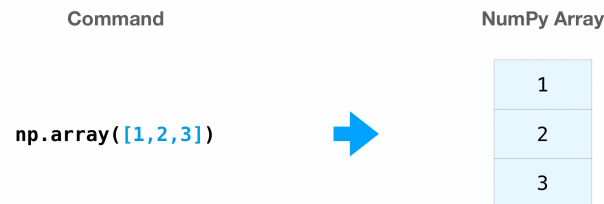
This section covers `np.array()`, `np.zeros()`, `np.ones()`, `np.empty()`, `np.arange()`, `np.linspace()`, `dtype`

To create a NumPy array, you can use the function `np.array()`.

All you need to do to create a simple array is pass a list to it. If you choose to, you can also specify the type of data in your list. You can find more information about data types [here](#).

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
```

You can visualize your array this way:



Be aware that these visualizations are meant to simplify ideas and give you a basic understanding of NumPy concepts and mechanics. Arrays and array operations are much more complicated than are captured here!

Besides creating an array from a sequence of elements, you can easily create an array filled with 0's:

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

Or an array filled with 1's:

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

Or even an empty array! The function `empty` creates an array whose initial content is random and depends on the state of the memory. The reason to use `empty` over `zeros` (or something similar) is speed - just make sure to fill every element afterwards!

```
>>> # Create an empty array with 2 elements
>>> np.empty(2)
array([3.14, 42.  ]) # may vary
```

You can create an array with a range of elements:

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

And even an array that contains a range of evenly spaced intervals. To do this, you will specify the **first number**, **last number**, and the **step size**.

```
>>> np.arange(2, 9, 2)
array([2, 4, 6, 8])
```

You can also use `np.linspace()` to create an array with values that are spaced linearly in a specified interval:

```
>>> np.linspace(0, 10, num=5)
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

Specifying your data type

While the default data type is floating point (`np.float64`), you can explicitly specify which data type you want using the `dtype` keyword.

```
>>> x = np.ones(2, dtype=np.int64)
>>> x
array([1, 1])
```

[Learn more about creating arrays here](#)

3.9 Adding, removing, and sorting elements

This section covers `np.sort()`, `np.concatenate()`

Sorting an element is simple with `np.sort()`. You can specify the axis, kind, and order when you call the function.

If you start with this array:

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

You can quickly sort the numbers in ascending order with:

```
>>> np.sort(arr)
array([1, 2, 3, 4, 5, 6, 7, 8])
```

In addition to `sort`, which returns a sorted copy of an array, you can use:

- `argsort`, which is an indirect sort along a specified axis,
- `lexsort`, which is an indirect stable sort on multiple keys,
- `searchsorted`, which will find elements in a sorted array, and
- `partition`, which is a partial sort.

To read more about sorting an array, see: `sort`.

If you start with these arrays:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([5, 6, 7, 8])
```

You can concatenate them with `np.concatenate()`.

```
>>> np.concatenate((a, b))
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Or, if you start with these arrays:

```
>>> x = np.array([[1, 2], [3, 4]])
>>> y = np.array([[5, 6]])
```

You can concatenate them with:

```
>>> np.concatenate((x, y), axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In order to remove elements from an array, it's simple to use indexing to select the elements that you want to keep.

To read more about concatenate, see: `concatenate`.

3.10 How do you know the shape and size of an array?

This section covers `ndarray.ndim`, `ndarray.size`, `ndarray.shape`

`ndarray.ndim` will tell you the number of axes, or dimensions, of the array.

`ndarray.size` will tell you the total number of elements of the array. This is the *product* of the elements of the array's shape.

`ndarray.shape` will display a tuple of integers that indicate the number of elements stored along each dimension of the array. If, for example, you have a 2-D array with 2 rows and 3 columns, the shape of your array is `(2, 3)`.

For example, if you create this array:

```
>>> array_example = np.array([[0, 1, 2, 3],
...                           [4, 5, 6, 7]],
...                           [[0, 1, 2, 3],
...                           [4, 5, 6, 7]],
...                           [[0, 1, 2, 3],
...                           [4, 5, 6, 7]])
```

To find the number of dimensions of the array, run:

```
>>> array_example.ndim
3
```

To find the total number of elements in the array, run:

```
>>> array_example.size
24
```

And to find the shape of your array, run:

```
>>> array_example.shape
(3, 2, 4)
```

3.11 Can you reshape an array?

This section covers `arr.reshape()`

Yes!

Using `arr.reshape()` will give a new shape to an array without changing the data. Just remember that when you use the reshape method, the array you want to produce needs to have the same number of elements as the original array. If you start with an array with 12 elements, you'll need to make sure that your new array also has a total of 12 elements.

If you start with this array:

```
>>> a = np.arange(6)
>>> print(a)
[0 1 2 3 4 5]
```

You can use `reshape()` to reshape your array. For example, you can reshape this array to an array with three rows and two columns:

```
>>> b = a.reshape(3, 2)
>>> print(b)
[[0 1]
 [2 3]
 [4 5]]
```

With `np.reshape`, you can specify a few optional parameters:

```
>>> np.reshape(a, newshape=(1, 6), order='C')
array([[0, 1, 2, 3, 4, 5]])
```

`a` is the array to be reshaped.

`newshape` is the new shape you want. You can specify an integer or a tuple of integers. If you specify an integer, the result will be an array of that length. The shape should be compatible with the original shape.

`order`: C means to read/write the elements using C-like index order, F means to read/write the elements using Fortran-like index order, A means to read/write the elements in Fortran-like index order if `a` is Fortran contiguous in memory, C-like order otherwise. (This is an optional parameter and doesn't need to be specified.)

If you want to learn more about C and Fortran order, you can [read more about the internal organization of NumPy arrays here](#). Essentially, C and Fortran orders have to do with how indices correspond to the order the array is stored in memory. In Fortran, when moving through the elements of a two-dimensional array as it is stored in memory, the **first** index is the most rapidly varying index. As the first index moves to the next row as it changes, the matrix is stored one column at a time. This is why Fortran is thought of as a **Column-major language**. In C on the other hand, the **last** index changes the most rapidly. The matrix is stored by rows, making it a **Row-major language**. What you do for C or Fortran depends on whether it's more important to preserve the indexing convention or not reorder the data.

[Learn more about shape manipulation here.](#)

3.12 How to convert a 1D array into a 2D array (how to add a new axis to an array)

This section covers `np.newaxis`, `np.expand_dims`

You can use `np.newaxis` and `np.expand_dims` to increase the dimensions of your existing array.

Using `np.newaxis` will increase the dimensions of your array by one dimension when used once. This means that a **1D** array will become a **2D** array, a **2D** array will become a **3D** array, and so on.

For example, if you start with this array:

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
>>> a.shape
(6,)
```

You can use `np.newaxis` to add a new axis:

```
>>> a2 = a[np.newaxis, :]
>>> a2.shape
(1, 6)
```

You can explicitly convert a 1D array with either a row vector or a column vector using `np.newaxis`. For example, you can convert a 1D array to a row vector by inserting an axis along the first dimension:

```
>>> row_vector = a[np.newaxis, :]
>>> row_vector.shape
(1, 6)
```

Or, for a column vector, you can insert an axis along the second dimension:

```
>>> col_vector = a[:, np.newaxis]
>>> col_vector.shape
(6, 1)
```

You can also expand an array by inserting a new axis at a specified position with `np.expand_dims`.

For example, if you start with this array:

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
>>> a.shape
(6,)
```

You can use `np.expand_dims` to add an axis at index position 1 with:

```
>>> b = np.expand_dims(a, axis=1)
>>> b.shape
(6, 1)
```

You can add an axis at index position 0 with:

```
>>> c = np.expand_dims(a, axis=0)
>>> c.shape
(1, 6)
```

Find more information about `newaxis` here and `expand_dims` at `expand_dims`.

3.13 Indexing and slicing

You can index and slice NumPy arrays in the same ways you can slice Python lists.

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

>>> data[1]
2
>>> data[0:2]
array([1, 2])
>>> data[1:]
array([2, 3])
>>> data[-2:]
array([2, 3])
```

You can visualize it this way:

	data	data[0]	data[1]	data[0:2]	data[1:]	data[-2:]		data
0	1	1		1			0	1
1	2		2	2	2	2	1	2
2	3				3	3	2	3
							3	

You may want to take a section of your array or specific array elements to use in further analysis or additional operations. To do that, you'll need to subset, slice, and/or index your arrays.

If you want to select values from your array that fulfill certain conditions, it's straightforward with NumPy.

For example, if you start with this array:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

You can easily print all of the values in the array that are less than 5.

```
>>> print(a[a < 5])
[1 2 3 4]
```

You can also select, for example, numbers that are equal to or greater than 5, and use that condition to index an array.

```
>>> five_up = (a >= 5)
>>> print(a[five_up])
[ 5  6  7  8  9 10 11 12]
```

You can select elements that are divisible by 2:

```
>>> divisible_by_2 = a[a%2==0]
>>> print(divisible_by_2)
[ 2  4  6  8 10 12]
```

Or you can select elements that satisfy two conditions using the & and | operators:

```
>>> c = a[(a > 2) & (a < 11)]
>>> print(c)
[ 3  4  5  6  7  8  9 10]
```

You can also make use of the logical operators **&** and **|** in order to return boolean values that specify whether or not the values in an array fulfill a certain condition. This can be useful with arrays that contain names or other categorical values.

```
>>> five_up = (a > 5) | (a == 5)
>>> print(five_up)
[[False False False False]
 [ True  True  True  True]
 [ True  True  True True]]
```

You can also use `np.nonzero()` to select elements or indices from an array.

Starting with this array:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

You can use `np.nonzero()` to print the indices of elements that are, for example, less than 5:

```
>>> b = np.nonzero(a < 5)
>>> print(b)
(array([0, 0, 0, 0]), array([0, 1, 2, 3]))
```

In this example, a tuple of arrays was returned: one for each dimension. The first array represents the row indices where these values are found, and the second array represents the column indices where the values are found.

If you want to generate a list of coordinates where the elements exist, you can zip the arrays, iterate over the list of coordinates, and print them. For example:

```
>>> list_of_coordinates= list(zip(b[0], b[1]))

>>> for coord in list_of_coordinates:
...     print(coord)
(0, 0)
(0, 1)
(0, 2)
(0, 3)
```

You can also use `np.nonzero()` to print the elements in an array that are less than 5 with:

```
>>> print(a[b])
[1 2 3 4]
```

If the element you're looking for doesn't exist in the array, then the returned array of indices will be empty. For example:

```
>>> not_there = np.nonzero(a == 42)
>>> print(not_there)
(array([], dtype=int64), array([], dtype=int64))
```

Learn more about *[indexing and slicing](#)* [here](#) and [here](#).

Read more about using the nonzero function at: `nonzero`.

3.14 How to create an array from existing data

This section covers slicing and indexing, `np.vstack()`, `np.hstack()`, `np.hsplit()`, `.view()`, `copy()`

You can easily create a new array from a section of an existing array.

Let's say you have this array:

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

You can create a new array from a section of your array any time by specifying where you want to slice your array.

```
>>> arr1 = a[3:8]
>>> arr1
array([4, 5, 6, 7, 8])
```

Here, you grabbed a section of your array from index position 3 through index position 8.

You can also stack two existing arrays, both vertically and horizontally. Let's say you have two arrays, `a1` and `a2`:

```
>>> a1 = np.array([[1, 1],
...               [2, 2]])
>>> a2 = np.array([[3, 3],
...               [4, 4]])
```

You can stack them vertically with `vstack`:

```
>>> np.vstack((a1, a2))
array([[1, 1],
       [2, 2],
       [3, 3],
       [4, 4]])
```

Or stack them horizontally with `hstack`:

```
>>> np.hstack((a1, a2))
array([[1, 1, 3, 3],
       [2, 2, 4, 4]])
```

You can split an array into several smaller arrays using `hsplit`. You can specify either the number of equally shaped arrays to return or the columns *after* which the division should occur.

Let's say you have this array:

```
>>> x = np.arange(1, 25).reshape(2, 12)
>>> x
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]])
```

If you wanted to split this array into three equally shaped arrays, you would run:

```
>>> np.hsplit(x, 3)
(array([[ 1,  2,  3,  4],
       [13, 14, 15, 16]]), array([[ 5,  6,  7,  8],
       [17, 18, 19, 20]]), array([[ 9, 10, 11, 12],
       [21, 22, 23, 24]]))
```

(continues on next page)

(continued from previous page)

```
[17, 18, 19, 20]], array([[ 9, 10, 11, 12],
[21, 22, 23, 24]])]
```

If you wanted to split your array after the third and fourth column, you'd run:

```
>>> np.hsplit(x, (3, 4))
[array([[ 1,  2,  3],
       [13, 14, 15]]), array([[ 4],
       [16]]), array([[ 5,  6,  7,  8,  9, 10, 11, 12],
       [17, 18, 19, 20, 21, 22, 23, 24]])]
```

Learn more about stacking and splitting arrays [here](#).

You can use the `view` method to create a new array object that looks at the same data as the original array (a *shallow copy*).

Views are an important NumPy concept! NumPy functions, as well as operations like indexing and slicing, will return views whenever possible. This saves memory and is faster (no copy of the data has to be made). However it's important to be aware of this - modifying data in a view also modifies the original array!

Let's say you create this array:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

Now we create an array `b1` by slicing `a` and modify the first element of `b1`. This will modify the corresponding element in `a` as well!

```
>>> b1 = a[0, :]
>>> b1
array([1, 2, 3, 4])
>>> b1[0] = 99
>>> b1
array([99,  2,  3,  4])
>>> a
array([[99,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Using the `copy` method will make a complete copy of the array and its data (a *deep copy*). To use this on your array, you could run:

```
>>> b2 = a.copy()
```

Learn more about copies and views [here](#).

3.15 Basic array operations

This section covers addition, subtraction, multiplication, division, and more

Once you've created your arrays, you can start to work with them. Let's say, for example, that you've created two arrays, one called "data" and one called "ones"

$$\text{data} = \text{np.array}([1, 2]) \quad \begin{array}{|c|} \hline \text{data} \\ \hline 1 \\ \hline 2 \\ \hline \end{array} \quad \text{ones} = \text{np.ones}(2) \quad \begin{array}{|c|} \hline \text{ones} \\ \hline 1 \\ \hline 1 \\ \hline \end{array}$$

You can add the arrays together with the plus sign.

```
>>> data = np.array([1, 2])
>>> ones = np.ones(2, dtype=int)
>>> data + ones
array([2, 3])
```

$$\text{data} + \text{ones} = \begin{array}{|c|} \hline \text{data} \\ \hline 1 \\ \hline 2 \\ \hline \end{array} + \begin{array}{|c|} \hline \text{ones} \\ \hline 1 \\ \hline 1 \\ \hline \end{array} = \begin{array}{|c|} \hline 2 \\ \hline 3 \\ \hline \end{array}$$

You can, of course, do more than just addition!

```
>>> data - ones
array([0, 1])
>>> data * data
array([1, 4])
>>> data / data
array([1., 1.])
```

$$\begin{array}{|c|} \hline \text{data} \\ \hline 1 \\ \hline 2 \\ \hline \end{array} - \begin{array}{|c|} \hline \text{ones} \\ \hline 1 \\ \hline 1 \\ \hline \end{array} = \begin{array}{|c|} \hline 0 \\ \hline 1 \\ \hline \end{array}$$

$$\begin{array}{|c|} \hline \text{data} \\ \hline 1 \\ \hline 2 \\ \hline \end{array} * \begin{array}{|c|} \hline \text{data} \\ \hline 1 \\ \hline 2 \\ \hline \end{array} = \begin{array}{|c|} \hline 1 \\ \hline 4 \\ \hline \end{array}$$

$$\begin{array}{|c|} \hline \text{data} \\ \hline 1 \\ \hline 2 \\ \hline \end{array} / \begin{array}{|c|} \hline \text{data} \\ \hline 1 \\ \hline 2 \\ \hline \end{array} = \begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline \end{array}$$

Basic operations are simple with NumPy. If you want to find the sum of the elements in an array, you'd use `sum()`. This works for 1D arrays, 2D arrays, and arrays in higher dimensions.

```
>>> a = np.array([1, 2, 3, 4])
>>> a.sum()
10
```

To add the rows or the columns in a 2D array, you would specify the axis.

If you start with this array:

```
>>> b = np.array([[1, 1], [2, 2]])
```

You can sum over the axis of rows with:

```
>>> b.sum(axis=0)
array([3, 3])
```

You can sum over the axis of columns with:

```
>>> b.sum(axis=1)
array([2, 4])
```

Learn more about basic operations [here](#).

3.16 Broadcasting

There are times when you might want to carry out an operation between an array and a single number (also called *an operation between a vector and a scalar*) or between arrays of two different sizes. For example, your array (we'll call it “data”) might contain information about distance in miles but you want to convert the information to kilometers. You can perform this operation with:

```
>>> data = np.array([1.0, 2.0])
>>> data * 1.6
array([1.6, 3.2])
```

<table><tr><td>1</td></tr><tr><td>2</td></tr></table>	1	2	*	1.6	=	<table><tr><td>1</td></tr><tr><td>2</td></tr></table>	1	2	*	<table><tr><td>1.6</td></tr><tr><td>1.6</td></tr></table>	1.6	1.6	=	<table><tr><td>1.6</td></tr><tr><td>3.2</td></tr></table>	1.6	3.2
1																
2																
1																
2																
1.6																
1.6																
1.6																
3.2																

NumPy understands that the multiplication should happen with each cell. That concept is called **broadcasting**. Broadcasting is a mechanism that allows NumPy to perform operations on arrays of different shapes. The dimensions of your array must be compatible, for example, when the dimensions of both arrays are equal or when one of them is 1. If the dimensions are not compatible, you will get a `ValueError`.

Learn more about broadcasting [here](#).

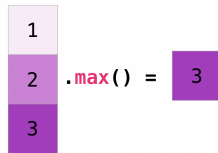
3.17 More useful array operations

This section covers maximum, minimum, sum, mean, product, standard deviation, and more

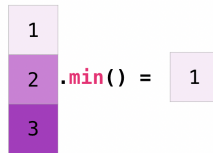
NumPy also performs aggregation functions. In addition to `min`, `max`, and `sum`, you can easily run `mean` to get the average, `prod` to get the result of multiplying the elements together, `std` to get the standard deviation, and more.

```
>>> data.max()
2.0
>>> data.min()
1.0
>>> data.sum()
3.0
```

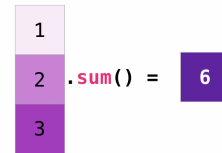
data



data



data



Let's start with this array, called "a"

```
>>> a = np.array([[0.45053314, 0.17296777, 0.34376245, 0.5510652],
...               [0.54627315, 0.05093587, 0.40067661, 0.55645993],
...               [0.12697628, 0.82485143, 0.26590556, 0.56917101]])
```

It's very common to want to aggregate along a row or column. By default, every NumPy aggregation function will return the aggregate of the entire array. To find the sum or the minimum of the elements in your array, run:

```
>>> a.sum()
4.8595784
```

Or:

```
>>> a.min()
0.05093587
```

You can specify on which axis you want the aggregation function to be computed. For example, you can find the minimum value within each column by specifying `axis=0`.

```
>>> a.min(axis=0)
array([0.12697628, 0.05093587, 0.26590556, 0.5510652 ])
```

The four values listed above correspond to the number of columns in your array. With a four-column array, you will get four values as your result.

Read more about array methods [here](#).

3.18 Creating matrices

You can pass Python lists of lists to create a 2-D array (or "matrix") to represent them in NumPy.

```
>>> data = np.array([[1, 2], [3, 4], [5, 6]])
>>> data
array([[1, 2],
       [3, 4],
       [5, 6]])
```

```
np.array([[1,2],[3,4],[5,6]])
```



1	2
3	4
5	6

Indexing and slicing operations are useful when you're manipulating matrices:

```
>>> data[0, 1]
2
>>> data[1:3]
array([[3, 4],
       [5, 6]])
>>> data[0:2, 0]
array([1, 3])
```

data		data[0,1]		data[1:3]		data[0:2,0]	
0	1	0	1	0	1	0	1
0	1	2	0	1	2	0	1
1	3	4	1	3	4	1	3
2	5	6	2	5	6	2	5

You can aggregate matrices the same way you aggregated vectors:

```
>>> data.max()
6
>>> data.min()
1
>>> data.sum()
21
```

data		data		data	
1	2	1	2	1	2
3	4	3	4	3	4
5	6	5	6	5	6

.max() = 6

.min() = 1

.sum() = 21

You can aggregate all the values in a matrix and you can aggregate them across columns or rows using the `axis` parameter. To illustrate this point, let's look at a slightly modified dataset:

```
>>> data = np.array([[1, 2], [5, 3], [4, 6]])
>>> data
array([[1, 2],
       [5, 3],
       [4, 6]])
```

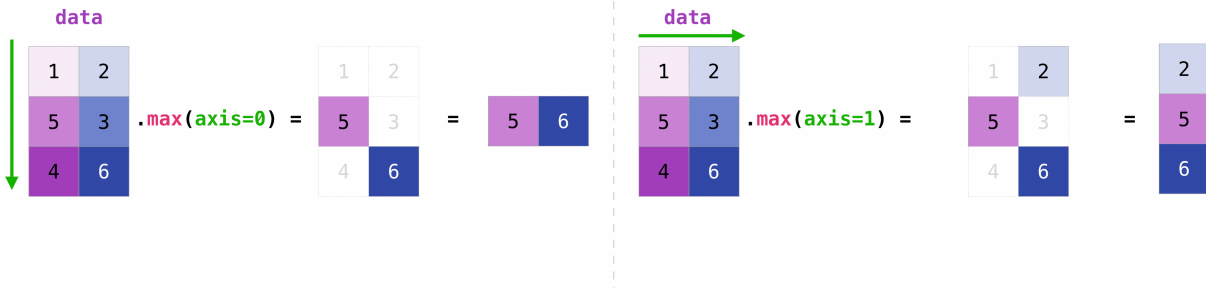
(continues on next page)

(continued from previous page)

```

    [5, 3],
    [4, 6]])
>>> data.max(axis=0)
array([5, 6])
>>> data.max(axis=1)
array([2, 5, 6])

```

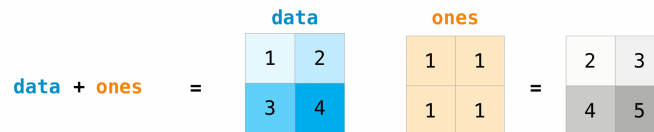


Once you've created your matrices, you can add and multiply them using arithmetic operators if you have two matrices that are the same size.

```

>>> data = np.array([[1, 2], [3, 4]])
>>> ones = np.array([[1, 1], [1, 1]])
>>> data + ones
array([[2, 3],
       [4, 5]])

```



You can do these arithmetic operations on matrices of different sizes, but only if one matrix has only one column or one row. In this case, NumPy will use its broadcast rules for the operation.

```

>>> data = np.array([[1, 2], [3, 4], [5, 6]])
>>> ones_row = np.array([[1, 1]])
>>> data + ones_row
array([[2, 3],
       [4, 5],
       [6, 7]])

```

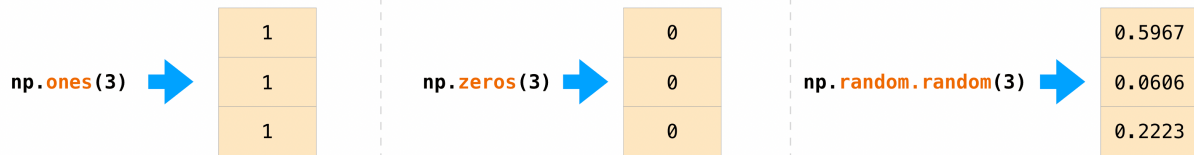
$$\text{data} + \text{ones_row} = \begin{array}{|c|c|} \hline \text{data} & \\ \hline 1 & 2 \\ \hline 3 & 4 \\ \hline 5 & 6 \\ \hline \end{array} + \begin{array}{|c|c|} \hline \text{ones_row} & \\ \hline 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|} \hline \text{data} & \text{ones_row} \\ \hline 1 & 2 \\ \hline 3 & 4 \\ \hline 5 & 6 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 1 & 1 \\ \hline 1 & 1 \\ \hline 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2 & 3 \\ \hline 4 & 5 \\ \hline 6 & 7 \\ \hline \end{array}$$

Be aware that when NumPy prints N-dimensional arrays, the last axis is looped over the fastest while the first axis is the slowest. For instance:

```
>>> np.ones((4, 3, 2))
array([[1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.]])
```

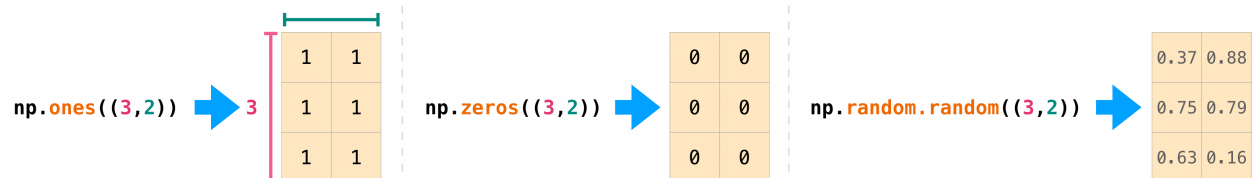
There are often instances where we want NumPy to initialize the values of an array. NumPy offers functions like `ones()` and `zeros()`, and the `random.Generator` class for random number generation for that. All you need to do is pass in the number of elements you want it to generate:

```
>>> np.ones(3)
array([1., 1., 1.])
>>> np.zeros(3)
array([0., 0., 0.])
>>> rng = np.random.default_rng() # the simplest way to generate random numbers
>>> rng.random(3)
array([0.63696169, 0.26978671, 0.04097352])
```



You can also use `ones()`, `zeros()`, and `random()` to create a 2D array if you give them a tuple describing the dimensions of the matrix:

```
>>> np.ones((3, 2))
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
>>> np.zeros((3, 2))
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
>>> rng.random((3, 2))
array([[0.01652764, 0.81327024],
       [0.91275558, 0.60663578],
       [0.72949656, 0.54362499]]) # may vary
```



Read more about creating arrays, filled with 0's, 1's, other values or uninitialized, at array creation routines.

3.19 Generating random numbers

The use of random number generation is an important part of the configuration and evaluation of many numerical and machine learning algorithms. Whether you need to randomly initialize weights in an artificial neural network, split data into random sets, or randomly shuffle your dataset, being able to generate random numbers (actually, repeatable pseudo-random numbers) is essential.

With `Generator.integers`, you can generate random integers from low (remember that this is inclusive with NumPy) to high (exclusive). You can set `endpoint=True` to make the high number inclusive.

You can generate a 2 x 4 array of random integers between 0 and 4 with:

```
>>> rng.integers(5, size=(2, 4))
array([[2, 1, 1, 0],
       [0, 0, 0, 4]]) # may vary
```

Read more about random number generation [here](#).

3.20 How to get unique items and counts

This section covers `np.unique()`

You can find the unique elements in an array easily with `np.unique`.

For example, if you start with this array:

```
>>> a = np.array([11, 11, 12, 13, 14, 15, 16, 17, 12, 13, 11, 14, 18, 19, 20])
```

you can use `np.unique` to print the unique values in your array:

```
>>> unique_values = np.unique(a)
>>> print(unique_values)
[11 12 13 14 15 16 17 18 19 20]
```

To get the indices of unique values in a NumPy array (an array of first index positions of unique values in the array), just pass the `return_index` argument in `np.unique()` as well as your array.

```
>>> unique_values, indices_list = np.unique(a, return_index=True)
>>> print(indices_list)
[ 0  2  3  4  5  6  7 12 13 14]
```

You can pass the `return_counts` argument in `np.unique()` along with your array to get the frequency count of unique values in a NumPy array.

```
>>> unique_values, occurrence_count = np.unique(a, return_counts=True)
>>> print(occurrence_count)
[3 2 2 2 1 1 1 1 1 1]
```

This also works with 2D arrays! If you start with this array:

```
>>> a_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [1, 2, 3, 4]])
```

You can find unique values with:

```
>>> unique_values = np.unique(a_2d)
>>> print(unique_values)
[ 1  2  3  4  5  6  7  8  9 10 11 12]
```

If the `axis` argument isn't passed, your 2D array will be flattened.

If you want to get the unique rows or columns, make sure to pass the `axis` argument. To find the unique rows, specify `axis=0` and for columns, specify `axis=1`.

```
>>> unique_rows = np.unique(a_2d, axis=0)
>>> print(unique_rows)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

To get the unique rows, index position, and occurrence count, you can use:

```
>>> unique_rows, indices, occurrence_count = np.unique(
...     a_2d, axis=0, return_counts=True, return_index=True)
>>> print(unique_rows)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
>>> print(indices)
[0 1 2]
>>> print(occurrence_count)
[2 1 1]
```

To learn more about finding the unique elements in an array, see `unique`.

3.21 Transposing and reshaping a matrix

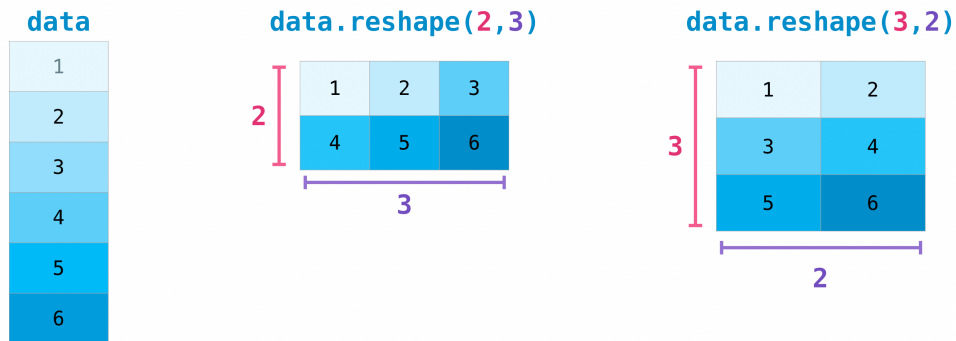
This section covers `arr.reshape()`, `arr.transpose()`, `arr.T`

It's common to need to transpose your matrices. NumPy arrays have the property `T` that allows you to transpose a matrix.

data		data.T		
1	2	1	3	5
3	4	2	4	6
5	6			

You may also need to switch the dimensions of a matrix. This can happen when, for example, you have a model that expects a certain input shape that is different from your dataset. This is where the `reshape` method can be useful. You simply need to pass in the new dimensions that you want for the matrix.

```
>>> data.reshape(2, 3)
array([[1, 2, 3],
       [4, 5, 6]])
>>> data.reshape(3, 2)
array([[1, 2],
       [3, 4],
       [5, 6]])
```



You can also use `.transpose()` to reverse or change the axes of an array according to the values you specify.

If you start with this array:

```
>>> arr = np.arange(6).reshape((2, 3))
>>> arr
array([[0, 1, 2],
       [3, 4, 5]])
```

You can transpose your array with `arr.transpose()`.

```
>>> arr.transpose()
array([[0, 3],
       [1, 4],
       [2, 5]])
```

You can also use `arr.T`:

```
>>> arr.T
array([[0, 3],
       [1, 4],
       [2, 5]])
```

To learn more about transposing and reshaping arrays, see `transpose` and `reshape`.

3.22 How to reverse an array

This section covers `np.flip()`

NumPy's `np.flip()` function allows you to flip, or reverse, the contents of an array along an axis. When using `np.flip()`, specify the array you would like to reverse and the axis. If you don't specify the axis, NumPy will reverse the contents along all of the axes of your input array.

Reversing a 1D array

If you begin with a 1D array like this one:

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

You can reverse it with:

```
>>> reversed_arr = np.flip(arr)
```

If you want to print your reversed array, you can run:

```
>>> print('Reversed Array: ', reversed_arr)
Reversed Array:  [8 7 6 5 4 3 2 1]
```

Reversing a 2D array

A 2D array works much the same way.

If you start with this array:

```
>>> arr_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

You can reverse the content in all of the rows and all of the columns with:

```
>>> reversed_arr = np.flip(arr_2d)
>>> print(reversed_arr)
[[12 11 10  9]
 [ 8  7  6  5]
 [ 4  3  2  1]]
```

You can easily reverse only the *rows* with:

```
>>> reversed_arr_rows = np.flip(arr_2d, axis=0)
>>> print(reversed_arr_rows)
[[ 9 10 11 12]
 [ 5  6  7  8]
 [ 1  2  3  4]]
```

Or reverse only the *columns* with:

```
>>> reversed_arr_columns = np.flip(arr_2d, axis=1)
>>> print(reversed_arr_columns)
[[ 4  3  2  1]
 [ 8  7  6  5]
 [12 11 10  9]]
```

You can also reverse the contents of only one column or row. For example, you can reverse the contents of the row at index position 1 (the second row):

```
>>> arr_2d[1] = np.flip(arr_2d[1])
>>> print(arr_2d)
[[ 1  2  3  4]
 [ 8  7  6  5]
 [ 9 10 11 12]]
```

You can also reverse the column at index position 1 (the second column):

```
>>> arr_2d[:,1] = np.flip(arr_2d[:,1])
>>> print(arr_2d)
[[ 1 10  3  4]
 [ 8  7  6  5]
 [ 9  2 11 12]]
```

Read more about reversing arrays at `flip`.

3.23 Reshaping and flattening multidimensional arrays

This section covers `.flatten()`, `ravel()`

There are two popular ways to flatten an array: `.flatten()` and `.ravel()`. The primary difference between the two is that the new array created using `ravel()` is actually a reference to the parent array (i.e., a “view”). This means that any changes to the new array will affect the parent array as well. Since `ravel` does not create a copy, it’s memory efficient.

If you start with this array:

```
>>> x = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

You can use `flatten` to flatten your array into a 1D array.

```
>>> x.flatten()
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

When you use `flatten`, changes to your new array won’t change the parent array.

For example:

```
>>> a1 = x.flatten()
>>> a1[0] = 99
>>> print(x) # Original array
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
>>> print(a1) # New array
[99  2  3  4  5  6  7  8  9 10 11 12]
```

But when you use `ravel`, the changes you make to the new array will affect the parent array.

For example:

```
>>> a2 = x.ravel()
>>> a2[0] = 98
>>> print(x) # Original array
```

(continues on next page)

(continued from previous page)

```
[[98  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
>>> print(a2) # New array
[98  2  3  4  5  6  7  8  9 10 11 12]
```

Read more about `flatten` at `ndarray.flatten` and `ravel` at `ravel`.

3.24 How to access the docstring for more information

This section covers `help()`, `?`, `??`

When it comes to the data science ecosystem, Python and NumPy are built with the user in mind. One of the best examples of this is the built-in access to documentation. Every object contains the reference to a string, which is known as the **docstring**. In most cases, this docstring contains a quick and concise summary of the object and how to use it. Python has a built-in `help()` function that can help you access this information. This means that nearly any time you need more information, you can use `help()` to quickly find the information that you need.

For example:

```
>>> help(max)
Help on built-in function max in module builtins:

max(...)
    max(iterable, *[, default=obj, key=func]) -> value
    max(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its biggest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the largest argument.
```

Because access to additional information is so useful, IPython uses the `?` character as a shorthand for accessing this documentation along with other relevant information. IPython is a command shell for interactive computing in multiple languages. [You can find more information about IPython here.](#)

For example:

```
In [0]: max?
max(iterable, *[, default=obj, key=func]) -> value
max(arg1, arg2, *args, *[, key=func]) -> value

With a single iterable argument, return its biggest item. The
default keyword-only argument specifies an object to return if
the provided iterable is empty.
With two or more arguments, return the largest argument.
Type:          builtin_function_or_method
```

You can even use this notation for object methods and objects themselves.

Let's say you create this array:

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
```

Then you can obtain a lot of useful information (first details about a itself, followed by the docstring of `ndarray` of which `a` is an instance):

```
In [1]: a?
Type: ndarray
String form: [1 2 3 4 5 6]
Length: 6
File: ~/anaconda3/lib/python3.9/site-packages/numpy/__init__.py
Docstring: <no docstring>
Class docstring:
ndarray(shape, dtype=float, buffer=None, offset=0,
        strides=None, order=None)

An array object represents a multidimensional, homogeneous array
of fixed-size items. An associated data-type object describes the
format of each element in the array (its byte-order, how many bytes it
occupies in memory, whether it is an integer, a floating point number,
or something else, etc.)

Arrays should be constructed using `array`, `zeros` or `empty` (refer
to the See Also section below). The parameters given here refer to
a low-level method (`ndarray(...)`) for instantiating an array.

For more information, refer to the `numpy` module and examine the
methods and attributes of an array.

Parameters
-----
(for the __new__ method; see Notes below)

shape : tuple of ints
        Shape of created array.
...
```

This also works for functions and other objects that **you** create. Just remember to include a docstring with your function using a string literal (`""" """` or `''' '''` around your documentation).

For example, if you create this function:

```
>>> def double(a):
...     '''Return a * 2'''
...     return a * 2
```

You can obtain information about the function:

```
In [2]: double?
Signature: double(a)
Docstring: Return a * 2
File: ~/Desktop/<ipython-input-23-b5adf20be596>
Type: function
```

You can reach another level of information by reading the source code of the object you're interested in. Using a double question mark (`??`) allows you to access the source code.

For example:

```
In [3]: double??
Signature: double(a)
```

(continues on next page)

(continued from previous page)

```
Source:
def double(a):
    '''Return a * 2'''
    return a * 2
File:      ~/Desktop/<ipython-input-23-b5adf20be596>
Type:      function
```

If the object in question is compiled in a language other than Python, using `??` will return the same information as `?`. You'll find this with a lot of built-in objects and types, for example:

```
In [4]: len?
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

and :

```
In [5]: len??
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

have the same output because they were compiled in a programming language other than Python.

3.25 Working with mathematical formulas

The ease of implementing mathematical formulas that work on arrays is one of the things that make NumPy so widely used in the scientific Python community.

For example, this is the mean square error formula (a central formula used in supervised machine learning models that deal with regression):

$$\text{MeanSquareError} = \frac{1}{n} \sum_{i=1}^n (Y_{\text{prediction}_i} - Y_i)^2$$

Implementing this formula is simple and straightforward in NumPy:

```
error = (1/n) * np.sum(np.square(predictions - labels))
```

What makes this work so well is that `predictions` and `labels` can contain one or a thousand values. They only need to be the same size.

You can visualize it this way:

$$\text{error} = (1/3) * \text{np.sum}(\text{np.square}(\begin{array}{|c|} \hline \text{predictions} \\ \hline 1 \\ 1 \\ 1 \\ \hline \end{array} - \begin{array}{|c|} \hline \text{labels} \\ \hline 1 \\ 2 \\ 3 \\ \hline \end{array}))$$

In this example, both the predictions and labels vectors contain three values, meaning n has a value of three. After we carry out subtractions the values in the vector are squared. Then NumPy sums the values, and your result is the error value for that prediction and a score for the quality of the model.

$$\begin{aligned} \text{error} &= (1/3) * \text{np.sum}(\text{np.square}(\begin{array}{|c|} \hline 0 \\ -1 \\ -2 \\ \hline \end{array})) \\ \text{error} &= (1/3) * \text{np.sum}(\begin{array}{|c|} \hline 0 \\ 1 \\ 4 \\ \hline \end{array}) \\ \text{error} &= (1/3) * 5 \end{aligned}$$

3.26 How to save and load NumPy objects

This section covers `np.save`, `np.savez`, `np.savetxt`, `np.load`, `np.loadtxt`

You will, at some point, want to save your arrays to disk and load them back without having to re-run the code. Fortunately, there are several ways to save and load objects with NumPy. The ndarray objects can be saved to and loaded from the disk files with `loadtxt` and `savetxt` functions that handle normal text files, `load` and `save` functions that handle NumPy binary files with a **.npy** file extension, and a `savez` function that handles NumPy files with a **.npz** file extension.

The **.npy** and **.npz** files store data, shape, dtype, and other information required to reconstruct the ndarray in a way that allows the array to be correctly retrieved, even when the file is on another machine with different architecture.

If you want to store a single ndarray object, store it as a **.npy** file using `np.save`. If you want to store more than one ndarray object in a single file, save it as a **.npz** file using `np.savez`. You can also save several arrays into a single file in compressed npz format with `savez_compressed`.

It's easy to save and load an array with `np.save()`. Just make sure to specify the array you want to save and a file name. For example, if you create this array:

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
```

You can save it as “filename.npy” with:

```
>>> np.save('filename', a)
```

You can use `np.load()` to reconstruct your array.

```
>>> b = np.load('filename.npy')
```

If you want to check your array, you can run::

```
>>> print(b)
[1 2 3 4 5 6]
```

You can save a NumPy array as a plain text file like a `.csv` or `.txt` file with `np.savetxt`.

For example, if you create this array:

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

You can easily save it as a `.csv` file with the name “new_file.csv” like this:

```
>>> np.savetxt('new_file.csv', csv_arr)
```

You can quickly and easily load your saved text file using `loadtxt()`:

```
>>> np.loadtxt('new_file.csv')
array([1., 2., 3., 4., 5., 6., 7., 8.])
```

The `savetxt()` and `loadtxt()` functions accept additional optional parameters such as header, footer, and delimiter. While text files can be easier for sharing, `.npy` and `.npz` files are smaller and faster to read. If you need more sophisticated handling of your text file (for example, if you need to work with lines that contain missing values), you will want to use the `genfromtxt` function.

With `savetxt`, you can specify headers, footers, comments, and more.

Learn more about input and output routines [here](#).

3.27 Importing and exporting a CSV

It’s simple to read in a CSV that contains existing information. The best and easiest way to do this is to use [Pandas](#).

```
>>> import pandas as pd

>>> # If all of your columns are the same type:
>>> x = pd.read_csv('music.csv', header=0).values
>>> print(x)
[['Billie Holiday' 'Jazz' 1300000 27000000]
 ['Jimmie Hendrix' 'Rock' 2700000 70000000]
 ['Miles Davis' 'Jazz' 1500000 48000000]
 ['SIA' 'Pop' 2000000 74000000]]

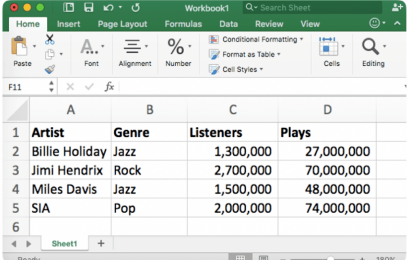
>>> # You can also simply select the columns you need:
>>> x = pd.read_csv('music.csv', usecols=['Artist', 'Plays']).values
>>> print(x)
[['Billie Holiday' 27000000]
 ['Jimmie Hendrix' 70000000]]
```

(continues on next page)

(continued from previous page)

```
['Miles Davis' 48000000]
['SIA' 74000000]]
```

music.csv



The image shows a transition from a spreadsheet to a pandas DataFrame. On the left, a screenshot of a spreadsheet titled 'music.csv' shows columns: Artist, Genre, Listeners, and Plays. The data rows are: Billie Holiday (Jazz, 1,300,000 Listeners, 27,000,000 Plays), Jimi Hendrix (Rock, 2,700,000 Listeners, 70,000,000 Plays), Miles Davis (Jazz, 1,500,000 Listeners, 48,000,000 Plays), and SIA (Pop, 2,000,000 Listeners, 74,000,000 Plays). An arrow points to the right, where the pandas command `pandas.read_csv('music.csv')` is shown above a DataFrame table with the same data.

	Artist	Genre	Listeners	Plays
0	Billie Holiday	Jazz	1,300,000	27,000,000
1	Jimi Hendrix	Rock	2,700,000	70,000,000
2	Miles Davis	Jazz	1,500,000	48,000,000
3	SIA	Pop	2,000,000	74,000,000

It's simple to use Pandas in order to export your array as well. If you are new to NumPy, you may want to create a Pandas dataframe from the values in your array and then write the data frame to a CSV file with Pandas.

If you created this array “a”

```
>>> a = np.array([[-2.58289208,  0.43014843, -1.24082018,  1.59572603],
...               [ 0.99027828,  1.17150989,  0.94125714, -0.14692469],
...               [ 0.76989341,  0.81299683, -0.95068423,  0.11769564],
...               [ 0.20484034,  0.34784527,  1.96979195,  0.51992837]])
```

You could create a Pandas dataframe

```
>>> df = pd.DataFrame(a)
>>> print(df)
           0          1          2          3
0 -2.582892  0.430148 -1.240820  1.595726
1  0.990278  1.171510  0.941257 -0.146925
2  0.769893  0.812997 -0.950684  0.117696
3  0.204840  0.347845  1.969792  0.519928
```

You can easily save your dataframe with:

```
>>> df.to_csv('pd.csv')
```

And read your CSV with:

```
>>> data = pd.read_csv('pd.csv')
```

	Unnamed: 0	0	1	2	3
0	0	-2.582892	0.430148	-1.240820	1.595726
1	1	0.990278	1.171510	0.941257	-0.146925
2	2	0.769893	0.812997	-0.950684	0.117696
3	3	0.204840	0.347845	1.969792	0.519928

You can also save your array with the NumPy `savetxt` method.

```
>>> np.savetxt('np.csv', a, fmt='%.2f', delimiter=',', header='1, 2, 3, 4')
```

If you're using the command line, you can read your saved CSV any time with a command such as:

```
$ cat np.csv
# 1, 2, 3, 4
-2.58,0.43,-1.24,1.60
0.99,1.17,0.94,-0.15
0.77,0.81,-0.95,0.12
0.20,0.35,1.97,0.52
```

Or you can open the file any time with a text editor!

If you're interested in learning more about Pandas, take a look at the [official Pandas documentation](#). Learn how to install Pandas with the [official Pandas installation information](#).

3.28 Plotting arrays with Matplotlib

If you need to generate a plot for your values, it's very simple with [Matplotlib](#).

For example, you may have an array like this one:

```
>>> a = np.array([2, 1, 5, 7, 4, 6, 8, 14, 10, 9, 18, 20, 22])
```

If you already have Matplotlib installed, you can import it with:

```
>>> import matplotlib.pyplot as plt

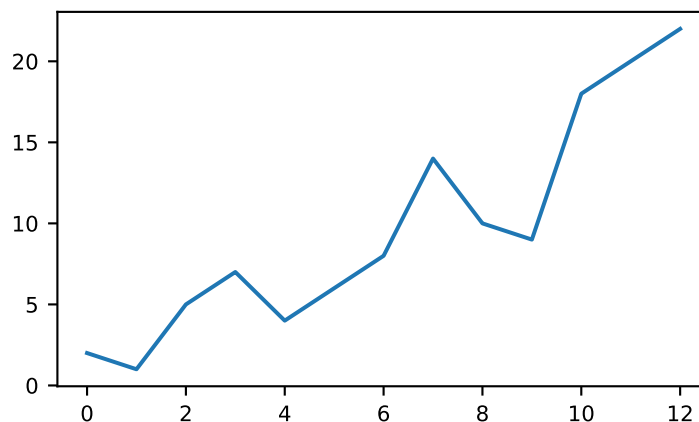
# If you're using Jupyter Notebook, you may also want to run the following
# line of code to display your code in the notebook:

%matplotlib inline
```

All you need to do to plot your values is run:

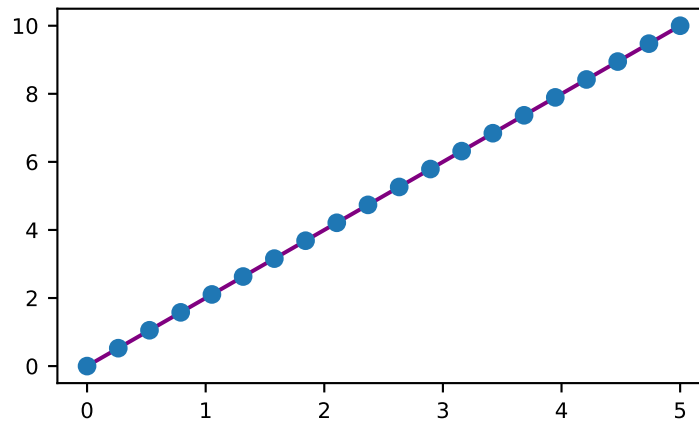
```
>>> plt.plot(a)

# If you are running from a command line, you may need to do this:
# >>> plt.show()
```



For example, you can plot a 1D array like this:

```
>>> x = np.linspace(0, 5, 20)
>>> y = np.linspace(0, 10, 20)
>>> plt.plot(x, y, 'purple') # line
>>> plt.plot(x, y, 'o')      # dots
```



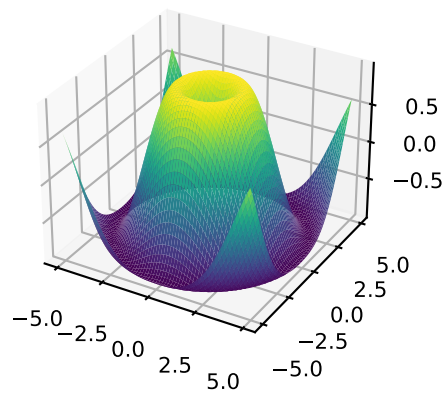
With Matplotlib, you have access to an enormous number of visualization options.

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(projection='3d')
>>> X = np.arange(-5, 5, 0.15)
>>> Y = np.arange(-5, 5, 0.15)
>>> X, Y = np.meshgrid(X, Y)
>>> R = np.sqrt(X**2 + Y**2)
>>> Z = np.sin(R)

>>> ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='viridis')
```

To read more about Matplotlib and what it can do, take a look at [the official documentation](#). For directions regarding installing Matplotlib, see the official [installation section](#).

Image credits: Jay Alammar <http://jalammar.github.io/>



NUMPY FUNDAMENTALS

These documents clarify concepts, design decisions, and technical constraints in NumPy. This is a great place to understand the fundamental NumPy ideas and philosophy.

4.1 Array creation

See also:

Array creation routines

4.1.1 Introduction

There are 6 general mechanisms for creating arrays:

- 1) Conversion from other Python structures (i.e. lists and tuples)
- 2) Intrinsic NumPy array creation functions (e.g. `arange`, `ones`, `zeros`, etc.)
- 3) Replicating, joining, or mutating existing arrays
- 4) Reading arrays from disk, either from standard or custom formats
- 5) Creating arrays from raw bytes through the use of strings or buffers
- 6) Use of special library functions (e.g., `random`)

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

4.1.2 1) Converting Python sequences to NumPy Arrays

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

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

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

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

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

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

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

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

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

4.1.3 2) Intrinsic NumPy array creation functions

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

- 1) 1D arrays
- 2) 2D arrays
- 3) ndarrays

1 - 1D array creation functions

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

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

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

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

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

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

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

2 - 2D array creation functions

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

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

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

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

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

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

```
>>> np.vander(np.linspace(0, 2, 5), 2)
array([[0. , 1. ],
       [0.5, 1. ],
       [1. , 1. ],
       [1.5, 1. ],
```

(continues on next page)

(continued from previous page)

```

[2. , 1. ]])
>>> np.vander([1, 2, 3, 4], 2)
array([[1, 1],
       [2, 1],
       [3, 1],
       [4, 1]])
>>> np.vander((1, 2, 3, 4), 4)
array([[ 1,  1,  1,  1],
       [ 8,  4,  2,  1],
       [27,  9,  3,  1],
       [64, 16,  4,  1]])

```

3 - general ndarray creation functions

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

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

```

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

```

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

```

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

```

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

```

>>> from numpy.random import default_rng
>>> default_rng(42).random((2, 3))
array([[0.77395605, 0.43887844, 0.85859792],
       [0.69736803, 0.09417735, 0.97562235]])
>>> default_rng(42).random((2, 3, 2))

```

(continues on next page)

(continued from previous page)

```
array([[0.77395605, 0.43887844],
       [0.85859792, 0.69736803],
       [0.09417735, 0.97562235]],
      [[0.7611397 , 0.78606431],
       [0.12811363, 0.45038594],
       [0.37079802, 0.92676499]])
```

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

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

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

4.1.4 3) Replicating, joining, or mutating existing arrays

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

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

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

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

For more information and examples look at [Copies and Views](#).

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

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

(continues on next page)

(continued from previous page)

```
[ 0.,  0., -3.,  0.],  
[ 0.,  0.,  0., -4.]])
```

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

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

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

Standard Binary Formats

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

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

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

Common ASCII Formats

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

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

Importing `simple.csv` is accomplished using `loadtxt`:

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

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

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

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

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

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

4.2 Indexing on `ndarrays`

See also:

Indexing routines

`ndarrays` can be indexed using the standard Python `x[obj]` syntax, where `x` is the array and `obj` the selection. There are different kinds of indexing available depending on `obj`: basic indexing, advanced indexing and field access.

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 [Assigning values to indexed arrays](#) for specific examples and explanations on how assignments work.

Note that in Python, `x[(exp1, exp2, ..., expN)]` is equivalent to `x[exp1, exp2, ..., expN]`; the latter is just syntactic sugar for the former.

4.2.1 Basic indexing

Single element indexing

Single element indexing works 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
```

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 a *view*, i.e., it 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: 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.

Slicing and striding

Basic slicing extends Python's basic concept of slicing to N dimensions. Basic slicing occurs when *obj* is a `slice` object (constructed by `start:stop:step` notation inside of brackets), an integer, or a tuple of slice objects and integers. `Ellipsis` and `newaxis` objects can be interspersed with these as well.

The simplest case of indexing with *N* integers returns an array scalar representing the corresponding item. As in Python, all indices are zero-based: for the *i*-th index n_i , the valid range is $0 \leq n_i < d_i$ where d_i is the *i*-th element of the shape of the array. Negative indices are interpreted as counting from the end of the array (i.e., if $n_i < 0$, it means $n_i + d_i$).

All arrays generated by basic slicing are always *views* of the original array.

Note: NumPy slicing creates a *view* instead of a copy as in the case of built-in Python sequences such as string, tuple and list. Care must be taken when extracting a small portion from a large array which becomes useless after the extraction, because the small portion extracted contains a reference to the large original array whose memory will not be released until all arrays derived from it are garbage-collected. In such cases an explicit `copy()` is recommended.

The standard rules of sequence slicing apply to basic slicing on a per-dimension basis (including using a step index). Some useful concepts to remember include:

- The basic slice syntax is `i:j:k` where *i* is the starting index, *j* is the stopping index, and *k* is the step ($k \neq 0$). This selects the *m* elements (in the corresponding dimension) with index values *i*, *i* + *k*, ..., *i* + (*m* - 1) *k* where $m = q + (r \neq 0)$ and *q* and *r* are the quotient and remainder obtained by dividing *j* - *i* by *k*: $j - i = qk + r$, so that $i + (m - 1)k < j$. For example:

```
>>> x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x[1:7:2]
array([1, 3, 5])
```

- Negative *i* and *j* are interpreted as *n* + *i* and *n* + *j* where *n* is the number of elements in the corresponding dimension. Negative *k* makes stepping go towards smaller indices. From the above example:

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

- Assume n is the number of elements in the dimension being sliced. Then, if i is not given it defaults to 0 for $k > 0$ and $n - 1$ for $k < 0$. If j is not given it defaults to n for $k > 0$ and $-n - 1$ for $k < 0$. If k is not given it defaults to 1. Note that $::$ is the same as $:$ and means select all indices along this axis. From the above example:

```
>>> x[5:]
array([5, 6, 7, 8, 9])
```

- If the number of objects in the selection tuple is less than N , then $:$ is assumed for any subsequent dimensions. For example:

```
>>> x = np.array([[[1],[2],[3]], [[4],[5],[6]]])
>>> x.shape
(2, 3, 1)
>>> x[1:2]
array([[4],
       [5],
       [6]])
```

- An integer, i , returns the same values as $i:i+1$ **except** the dimensionality of the returned object is reduced by 1. In particular, a selection tuple with the p -th element an integer (and all other entries $:$) returns the corresponding sub-array with dimension $N - 1$. If $N = 1$ then the returned object is an array scalar. These objects are explained in `arrays.scalars`.
- If the selection tuple has all entries $:$ except the p -th entry which is a slice object $i:j:k$, then the returned array has dimension N formed by concatenating the sub-arrays returned by integer indexing of elements $i, i+k, \dots, i + (m - 1)k < j$,
- Basic slicing with more than one non- $:$ entry in the slicing tuple, acts like repeated application of slicing using a single non- $:$ entry, where the non- $:$ entries are successively taken (with all other non- $:$ entries replaced by $:$). Thus, `x[ind1, ..., ind2, :]` acts like `x[ind1][..., ind2, :]` under basic slicing.

Warning: The above is **not** true for advanced indexing.

- You may use slicing to set values in the array, but (unlike lists) you can never grow the array. The size of the value to be set in `x[obj] = value` must be (broadcastable) to the same shape as `x[obj]`.
- A slicing tuple can always be constructed as *obj* and used in the `x[obj]` notation. Slice objects can be used in the construction in place of the `[start:stop:step]` notation. For example, `x[1:10:5, :-1]` can also be implemented as `obj = (slice(1, 10, 5), slice(None, None, -1)); x[obj]`. This can be useful for constructing generic code that works on arrays of arbitrary dimensions. See [Dealing with variable numbers of indices within programs](#) for more information.

Dimensional indexing tools

There are some tools to facilitate the easy matching of array shapes with expressions and in assignments.

`Ellipsis` expands to the number of `:` objects needed for the selection tuple to index all dimensions. In most cases, this means that the length of the expanded selection tuple is `x.ndim`. There may only be a single ellipsis present. From the above example:

```
>>> x[..., 0]
array([[1, 2, 3],
       [4, 5, 6]])
```

This is equivalent to:

```
>>> x[:, :, 0]
array([[1, 2, 3],
       [4, 5, 6]])
```

Each `newaxis` object in the selection tuple serves to expand the dimensions of the resulting selection by one unit-length dimension. The added dimension is the position of the `newaxis` object in the selection tuple. `newaxis` is an alias for `None`, and `None` can be used in place of this with the same result. From the above example:

```
>>> x[:, np.newaxis, :, :].shape
(2, 1, 3, 1)
>>> x[:, None, :, :].shape
(2, 1, 3, 1)
```

This can be handy to combine two arrays in a way that otherwise would require explicit 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]])
```

4.2.2 Advanced indexing

Advanced indexing is triggered when the selection object, *obj*, is a non-tuple sequence object, an `ndarray` (of data type integer or bool), or a tuple with at least one sequence object or `ndarray` (of data type integer or bool). There are two types of advanced indexing: integer and Boolean.

Advanced indexing always returns a *copy* of the data (contrast with basic slicing that returns a *view*).

Warning: The definition of advanced indexing means that `x[(1, 2, 3),]` is fundamentally different than `x[(1, 2, 3)]`. The latter is equivalent to `x[1, 2, 3]` which will trigger basic selection while the former will trigger advanced indexing. Be sure to understand why this occurs.

Also recognize that `x[[1, 2, 3]]` will trigger advanced indexing, whereas due to the deprecated Numeric compatibility mentioned above, `x[[1, 2, slice(None)]]` will trigger basic slicing.

Integer array indexing

Integer array indexing allows selection of arbitrary items in the array based on their N -dimensional index. Each integer array represents a number of indices into that dimension.

Negative values are permitted in the index arrays and work as they do with single indices or slices:

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

If the index values are out of bounds then an `IndexError` is thrown:

```
>>> x = np.array([[1, 2], [3, 4], [5, 6]])
>>> x[np.array([1, -1])]
array([[3, 4],
       [5, 6]])
>>> x[np.array([3, 4])]
Traceback (most recent call last):
...
IndexError: index 3 is out of bounds for axis 0 with size 3
```

When the index consists of as many integer arrays as dimensions of the array being indexed, the indexing is straightforward, but different from slicing.

Advanced indices always are *broadcast* and iterated as *one*:

```
result[i_1, ..., i_M] == x[ind_1[i_1, ..., i_M], ind_2[i_1, ..., i_M],
                           ..., ind_N[i_1, ..., i_M]]
```

Note that the resulting shape is identical to the (broadcast) indexing array shapes `ind_1, ..., ind_N`. If the indices cannot be broadcast to the same shape, an exception `IndexError: shape mismatch: indexing arrays could not be broadcast together with shapes...` is raised.

Indexing with multidimensional index arrays 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:

```
>>> y = np.arange(35).reshape(5, 7)
>>> y
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, 30, 31, 32, 33, 34]])
>>> 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])]
Traceback (most recent call last):
...
IndexError: shape mismatch: indexing arrays could not be broadcast
together with shapes (3,) (2,)
```

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]])
```

It results in 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).

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.

Example

From each row, a specific element should be selected. The row index is just `[0, 1, 2]` and the column index specifies the element to choose for the corresponding row, here `[0, 1, 0]`. Using both together the task can be solved using advanced indexing:

```
>>> x = np.array([[1, 2], [3, 4], [5, 6]])
>>> x[[0, 1, 2], [0, 1, 0]]
array([1, 4, 5])
```

To achieve a behaviour similar to the basic slicing above, broadcasting can be used. The function `ix_` can help with this broadcasting. This is best understood with an example.

Example

From a 4x3 array the corner elements should be selected using advanced indexing. Thus all elements for which the column is one of `[0, 2]` and the row is one of `[0, 3]` need to be selected. To use advanced indexing one needs to select all elements *explicitly*. Using the method explained previously one could write:

```
>>> x = np.array([[ 0,  1,  2],
...               [ 3,  4,  5],
...               [ 6,  7,  8],
...               [ 9, 10, 11]])
>>> rows = np.array([0, 0,
...                  [3, 3]], dtype=np.intp)
>>> columns = np.array([0, 2],
...                    [0, 2]], dtype=np.intp)
```

(continues on next page)

(continued from previous page)

```
>>> x[rows, columns]
array([[ 0,  2],
       [ 9, 11]])
```

However, since the indexing arrays above just repeat themselves, broadcasting can be used (compare operations such as `rows[:, np.newaxis] + columns`) to simplify this:

```
>>> rows = np.array([0, 3], dtype=np.intp)
>>> columns = np.array([0, 2], dtype=np.intp)
>>> rows[:, np.newaxis]
array([[0],
       [3]])
>>> x[rows[:, np.newaxis], columns]
array([[ 0,  2],
       [ 9, 11]])
```

This broadcasting can also be achieved using the function `ix_`:

```
>>> x[np.ix_(rows, columns)]
array([[ 0,  2],
       [ 9, 11]])
```

Note that without the `np.ix_` call, only the diagonal elements would be selected:

```
>>> x[rows, columns]
array([ 0, 11])
```

This difference is the most important thing to remember about indexing with multiple advanced indices.

Example

A real-life example of where advanced indexing 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 within 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.

Boolean array indexing

This advanced indexing occurs when *obj* is an array object of Boolean type, such as may be returned from comparison operators. A single boolean index array is practically identical to `x[obj.nonzero()]` where, as described above, `obj.nonzero()` returns a tuple (of length `obj.ndim`) of integer index arrays showing the `True` elements of *obj*. However, it is faster when `obj.shape == x.shape`.

If `obj.ndim == x.ndim`, `x[obj]` returns a 1-dimensional array filled with the elements of *x* corresponding to the `True` values of *obj*. The search order will be *row-major*, C-style. If *obj* has `True` values at entries that are outside of the bounds of *x*, then an index error will be raised. If *obj* is smaller than *x* it is identical to filling it with `False`.

A common use case for this is filtering for desired element values. For example, one may wish to select all entries from an array which are not NaN:

```
>>> x = np.array([[1., 2.], [np.nan, 3.], [np.nan, np.nan]])
>>> x[~np.isnan(x)]
array([1., 2., 3.])
```

Or wish to add a constant to all negative elements:

```
>>> x = np.array([1., -1., -2., 3])
>>> x[x < 0] += 20
>>> x
array([ 1., 19., 18., 3.]
```

In general if an index includes a Boolean array, the result will be identical to inserting `obj.nonzero()` into the same position and using the integer array indexing mechanism described above. `x[ind_1, boolean_array, ind_2]` is equivalent to `x[(ind_1,) + boolean_array.nonzero() + (ind_2,)]`.

If there is only one Boolean array and no integer indexing array present, this is straightforward. Care must only be taken to make sure that the boolean index has *exactly* as many dimensions as it is supposed to work with.

In general, when the boolean array has fewer dimensions than the array being indexed, this is equivalent to `x[b, ...]`, which means `x` is indexed by `b` followed by as many `:` as are needed to fill out the rank of `x`. 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:

```
>>> x = np.arange(35).reshape(5, 7)
>>> b = x > 20
>>> b[:, 5]
array([False, False, False,  True,  True])
>>> x[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.

Example

From an array, select all rows which sum up to less or equal two:

```
>>> x = np.array([[0, 1], [1, 1], [2, 2]])
>>> rowsum = x.sum(-1)
>>> x[rowsum <= 2, :]
array([[0, 1],
       [1, 1]])
```

Combining multiple Boolean indexing arrays or a Boolean with an integer indexing array can best be understood with the `obj.nonzero()` analogy. The function `ix_` also supports boolean arrays and will work without any surprises.

Example

Use boolean indexing to select all rows adding up to an even number. At the same time columns 0 and 2 should be selected with an advanced integer index. Using the `ix_` function this can be done with:

```
>>> x = np.array([[ 0,  1,  2],
...               [ 3,  4,  5],
...               [ 6,  7,  8],
...               [ 9, 10, 11]])
>>> rows = (x.sum(-1) % 2) == 0
>>> rows
array([False,  True, False,  True])
>>> columns = [0, 2]
>>> x[np.ix_(rows, columns)]
```

(continues on next page)

(continued from previous page)

```
array([[ 3,  5],
       [ 9, 11]])
```

Without the `np.ix_` call, only the diagonal elements would be selected.

Or without `np.ix_` (compare the integer array examples):

```
>>> rows = rows.nonzero()[0]
>>> x[rows[:, np.newaxis], columns]
array([[ 3,  5],
       [ 9, 11]])
```

Example

Use 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]])
```

Combining advanced and basic indexing

When there is at least one slice (:), ellipsis (...) or `newaxis` in the index (or the array has more dimensions than there are advanced indices), then the behaviour can be more complicated. It is like concatenating the indexing result for each advanced index element.

In the simplest case, there is only a *single* advanced index combined with a slice. For example:

```
>>> y = np.arange(35).reshape(5, 7)
>>> y[np.array([0, 2, 4]), 1:3]
array([[ 1,  2],
       [15, 16],
       [29, 30]])
```

In effect, the slice and index array operation are independent. The slice operation extracts columns with index 1 and 2, (i.e. the 2nd and 3rd columns), followed by the index array operation which extracts rows with index 0, 2 and 4 (i.e. the first, third and fifth rows). This is equivalent to:

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

A single advanced index can, for example, replace a slice and the result array will be the same. However, it is a copy and may have a different memory layout. A slice is preferable when it is possible. For example:

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

The easiest way to understand a combination of *multiple* advanced indices may be to think in terms of the resulting shape. There are two parts to the indexing operation, the subspace defined by the basic indexing (excluding integers) and the subspace from the advanced indexing part. Two cases of index combination need to be distinguished:

- The advanced indices are separated by a slice, `Ellipsis` or `newaxis`. For example `x[arr1, :, arr2]`.
- The advanced indices are all next to each other. For example `x[..., arr1, arr2, :]` but *not* `x[arr1, :, 1]` since 1 is an advanced index in this regard.

In the first case, the dimensions resulting from the advanced indexing operation come first in the result array, and the subspace dimensions after that. In the second case, the dimensions from the advanced indexing operations are inserted into the result array at the same spot as they were in the initial array (the latter logic is what makes simple advanced indexing behave just like slicing).

Example

Suppose `x.shape` is (10, 20, 30) and `ind` is a (2, 3, 4)-shaped indexing `intp` array, then `result = x[..., ind, :]` has shape (10, 2, 3, 4, 30) because the (20,)-shaped subspace has been replaced with a (2, 3, 4)-shaped broadcasted indexing subspace. If we let `i, j, k` loop over the (2, 3, 4)-shaped subspace then `result[..., i, j, k, :] = x[..., ind[i, j, k], :]`. This example produces the same result as `x.take(ind, axis=-2)`.

Example

Let `x.shape` be (10, 20, 30, 40, 50) and suppose `ind_1` and `ind_2` can be broadcast to the shape (2, 3, 4). Then `x[:, ind_1, ind_2]` has shape (10, 2, 3, 4, 40, 50) because the (20, 30)-shaped subspace from `X` has been replaced with the (2, 3, 4) subspace from the indices. However, `x[:, ind_1, :, ind_2]` has shape (2, 3, 4, 10, 30, 50) because there is no unambiguous place to drop in the indexing subspace, thus it is tacked-on to the beginning. It is always possible to use `.transpose()` to move the subspace anywhere desired. Note that this example cannot be replicated using `take`.

Example

Slicing can be combined with broadcasted boolean indices:

```
>>> x = np.arange(35).reshape(5, 7)
>>> b = x > 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]])
```

(continues on next page)

(continued from previous page)

```
>>> x[b[:, 5], 1:3]
array([[22, 23],
       [29, 30]])
```

4.2.3 Field access

See also:

Structured arrays

If the `ndarray` object is a structured array the *fields* of the array can be accessed by indexing the array with strings, dictionary-like.

Indexing `x['field-name']` returns a new *view* to the array, which is of the same shape as `x` (except when the field is a sub-array) but of data type `x.dtype['field-name']` and contains only the part of the data in the specified field. Also, record array scalars can be “indexed” this way.

Indexing into a structured array can also be done with a list of field names, e.g. `x[['field-name1', 'field-name2']]`. As of NumPy 1.16, this returns a view containing only those fields. In older versions of NumPy, it returned a copy. See the user guide section on *Structured arrays* for more information on multifield indexing.

If the accessed field is a sub-array, the dimensions of the sub-array are appended to the shape of the result. For example:

```
>>> x = np.zeros((2, 2), dtype=[('a', np.int32), ('b', np.float64, (3, 3))])
>>> x['a'].shape
(2, 2)
>>> x['a'].dtype
dtype('int32')
>>> x['b'].shape
(2, 2, 3, 3)
>>> x['b'].dtype
dtype('float64')
```

4.2.4 Flat Iterator indexing

`x.flat` returns an iterator that will iterate over the entire array (in C-contiguous style with the last index varying the fastest). This iterator object can also be indexed using basic slicing or advanced indexing as long as the selection object is not a tuple. This should be clear from the fact that `x.flat` is a 1-dimensional view. It can be used for integer indexing with 1-dimensional C-style-flat indices. The shape of any returned array is therefore the shape of the integer indexing object.

4.2.5 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
Traceback (most recent call last):
...
TypeError: can't convert complex to int
```

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 that 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.

4.2.6 Dealing with variable numbers of indices within programs

The indexing 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:

```
>>> z = np.arange(81).reshape(3, 3, 3, 3)
>>> 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 of 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
```

4.2.7 Detailed notes

These are some detailed notes, which are not of importance for day to day indexing (in no particular order):

- The native NumPy indexing type is `intp` and may differ from the default integer array type. `intp` is the smallest data type sufficient to safely index any array; for advanced indexing it may be faster than other types.
- For advanced assignments, there is in general no guarantee for the iteration order. This means that if an element is set more than once, it is not possible to predict the final result.
- An empty (tuple) index is a full scalar index into a zero-dimensional array. `x[()]` returns a *scalar* if `x` is zero-dimensional and a view otherwise. On the other hand, `x[...]` always returns a view.
- If a zero-dimensional array is present in the index *and* it is a full integer index the result will be a *scalar* and not a zero-dimensional array. (Advanced indexing is not triggered.)
- When an ellipsis (`...`) is present but has no size (i.e. replaces zero `:`) the result will still always be an array. A view if no advanced index is present, otherwise a copy.
- The `nonzero` equivalence for Boolean arrays does not hold for zero dimensional boolean arrays.
- When the result of an advanced indexing operation has no elements but an individual index is out of bounds, whether or not an `IndexError` is raised is undefined (e.g. `x[:, [123]]` with 123 being out of bounds).
- When a *casting* error occurs during assignment (for example updating a numerical array using a sequence of strings), the array being assigned to may end up in an unpredictable partially updated state. However, if any other error (such as an out of bounds index) occurs, the array will remain unchanged.
- The memory layout of an advanced indexing result is optimized for each indexing operation and no particular memory order can be assumed.
- When using a subclass (especially one which manipulates its shape), the default `ndarray.__setitem__` behaviour will call `__getitem__` for *basic* indexing but not for *advanced* indexing. For such a subclass it may be preferable to call `ndarray.__setitem__` with a *base class* `ndarray` view on the data. This *must* be done if the subclasses `__getitem__` does not return views.

4.3 I/O with NumPy

4.3.1 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, a generator or an open file-like object with a `read` method, for example, a file or `io.StringIO` object. If a single string is provided, it is assumed to be the name of a local or remote file. 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=",")
array([[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., 3.), (4., 6.)], dtype=[('a', '<f8'), ('c', '<f8')])
>>> np.genfromtxt(StringIO(data),
...               names="a, b, c", usecols="a, c")
array([(1., 3.), (4., 6.)], 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.

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., 2., 3.), (4., 5., 6.)],
      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., 2., 3.), (4., 5., 6.)],
      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., 3), (4, 5., 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., 3), (4, 5., 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., 3), (4, 5., 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., 3), (4, 5., 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 a string representing a percentage to a float between 0 and 1:

```
>>> convertfunc = lambda x: float(x.strip(b"%"))/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., nan, 45.), (6., nan, 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., 0.023, 45.), (6., 0.789, 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., 0.023, 45.), (6., 0.789, 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 kinds 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	<code>np.nan</code>
complex	<code>np.nan+0j</code>
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,
```

(continues on next page)

(continued from previous page)

```

...         names="a,b,c",
...         missing_values={0:"N/A", 'b':" ", 2:"???"},
...         filling_values={0:0, 'b':0, 2:-999})
>>> np.genfromtxt(StringIO(data), **kwargs)
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.npyio` module provides several convenience functions derived from `genfromtxt`. These functions work the same way as the original, but they have different default values.

numpy.lib.npyio.recfromtxt

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

numpy.lib.npyio.recfromcsv

Like `numpy.lib.npyio.recfromtxt`, but with a default `delimiter=","`.

4.4 Data types

See also:

Data type objects

4.4.1 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
<code>numpy.bool_</code>	<code>bool</code>	Boolean (True or False) stored as a byte
<code>numpy.byte</code>	<code>signed char</code>	Platform-defined
<code>numpy.ubyte</code>	<code>unsigned char</code>	Platform-defined
<code>numpy.short</code>	<code>short</code>	Platform-defined
<code>numpy.ushort</code>	<code>unsigned short</code>	Platform-defined
<code>numpy.intc</code>	<code>int</code>	Platform-defined
<code>numpy.uintc</code>	<code>unsigned int</code>	Platform-defined
<code>numpy.int_</code>	<code>long</code>	Platform-defined
<code>numpy.uint</code>	<code>unsigned long</code>	Platform-defined
<code>numpy.longlong</code>	<code>long long</code>	Platform-defined
<code>numpy.ulonglong</code>	<code>unsigned long long</code>	Platform-defined
<code>numpy.half</code> / <code>numpy.float16</code>		Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
<code>numpy.single</code>	<code>float</code>	Platform-defined single precision float: typically sign bit, 8 bits exponent, 23 bits mantissa
<code>numpy.double</code>	<code>double</code>	Platform-defined double precision float: typically sign bit, 11 bits exponent, 52 bits mantissa.
<code>numpy.longdouble</code>	<code>long double</code>	Platform-defined extended-precision float
<code>numpy.csingle</code>	<code>float complex</code>	Complex number, represented by two single-precision floats (real and imaginary components)
<code>numpy.cdouble</code>	<code>double complex</code>	Complex number, represented by two double-precision floats (real and imaginary components).
<code>numpy.clongdouble</code>	<code>long double complex</code>	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 (See sized-aliases).

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 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:

```
>>> 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)
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

4.4.2 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.

4.4.3 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 ** 8` correctly for 64-bit integers, but gives 1874919424 (incorrect) for a 32-bit integer.

```
>>> np.power(100, 8, dtype=np.int64)
1000000000000000000
>>> 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(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
```

4.4.4 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's `long double`; 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`.

4.5 Broadcasting

See also:

`numpy.broadcast`

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`, as shown in [Figure 1](#), 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.

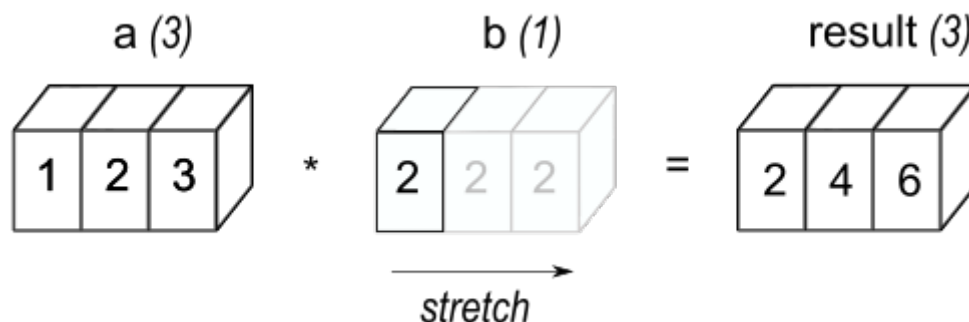


Fig. 1: *Figure 1*

In the simplest example of broadcasting, the scalar `b` is stretched to become an array of same shape as `a` so the shapes are compatible for element-by-element multiplication.

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).

4.5.1 General Broadcasting Rules

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimensions and works its way left. 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 size that is not 1 along each axis of the inputs.

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
```

4.5.2 Broadcastable arrays

A set of arrays is called “broadcastable” to the same shape if the above rules produce a valid result.

For example, if `a.shape` is `(5,1)`, `b.shape` is `(1,6)`, `c.shape` is `(6,)` and `d.shape` is `()` so that `d` is a scalar, then `a`, `b`, `c`, and `d` are all broadcastable to dimension `(5,6)`; and

- `a` acts like a `(5,6)` array where `a[:, 0]` is broadcast to the other columns,
- `b` acts like a `(5,6)` array where `b[0, :]` is broadcast to the other rows,
- `c` acts like a `(1,6)` array and therefore like a `(5,6)` array where `c[:]` is broadcast to every row, and finally,
- `d` acts like a `(5,6)` array where the single value is repeated.

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
```

(continues on next page)

(continued from previous page)

```

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 when a 1-d array is added to a 2-d array:

```

>>> a = np.array([[ 0.0,  0.0,  0.0],
...               [10.0, 10.0, 10.0],
...               [20.0, 20.0, 20.0],
...               [30.0, 30.0, 30.0]])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a + b
array([[ 1.,  2.,  3.],
       [11., 12., 13.],
       [21., 22., 23.],
       [31., 32., 33.]])
>>> b = np.array([1.0, 2.0, 3.0, 4.0])
>>> a + b
Traceback (most recent call last):
ValueError: operands could not be broadcast together with shapes (4,3) (4,)

```

As shown in [Figure 2](#), `b` is added to each row of `a`. In [Figure 3](#), an exception is raised because of the incompatible shapes.

Broadcasting provides a convenient way of taking the outer product (or any other outer operation) of two arrays. The following example shows an outer addition operation of two 1-d arrays:

```

>>> a = np.array([0.0, 10.0, 20.0, 30.0])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a[:, np.newaxis] + b
array([[ 1.,  2.,  3.],
       [11., 12., 13.],
       [21., 22., 23.],
       [31., 32., 33.]])

```

Here the `newaxis` index operator inserts a new axis into `a`, making it a two-dimensional 4×1 array. Combining the 4×1 array with `b`, which has shape $(3,)$, yields a 4×3 array.

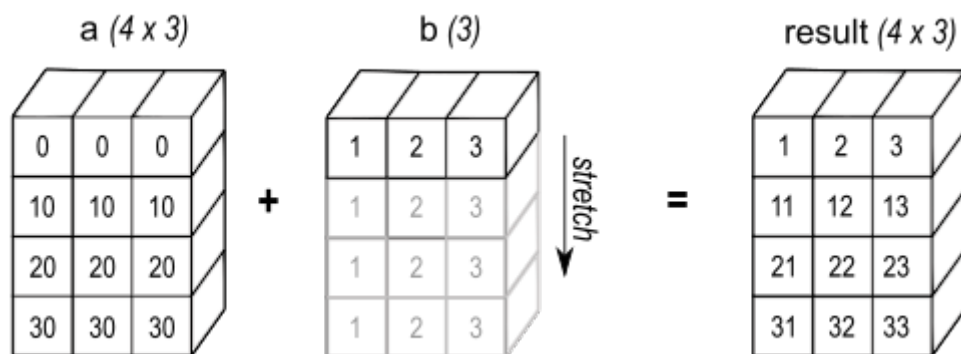


Fig. 2: Figure 2

A one dimensional array added to a two dimensional array results in broadcasting if number of 1-d array elements matches the number of 2-d array columns.

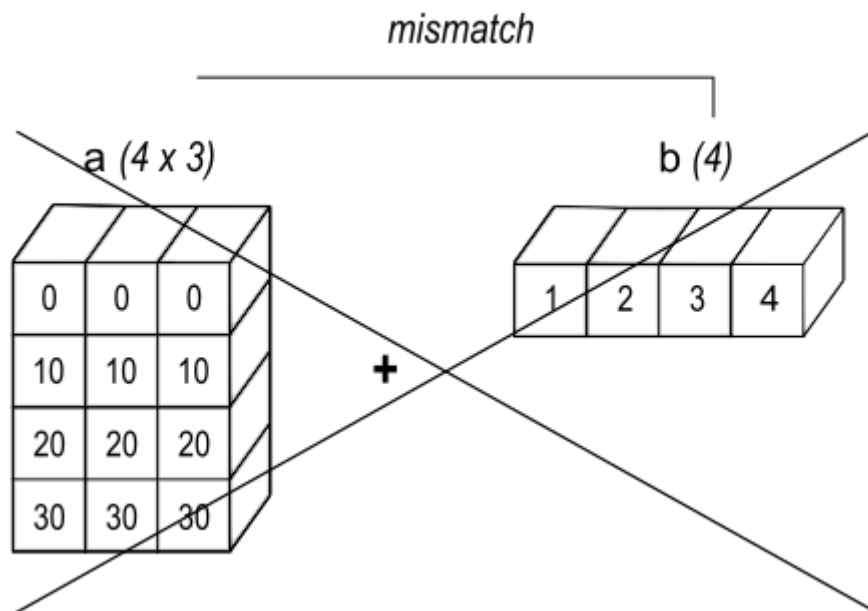


Fig. 3: Figure 3

When the trailing dimensions of the arrays are unequal, broadcasting fails because it is impossible to align the values in the rows of the 1st array with the elements of the 2nd arrays for element-by-element addition.

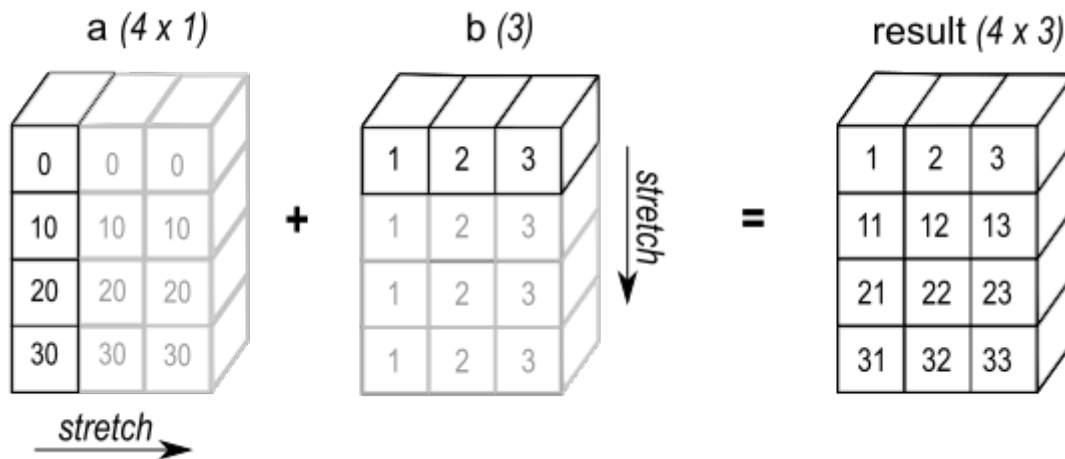


Fig. 4: Figure 4

In some cases, broadcasting stretches both arrays to form an output array larger than either of the initial arrays.

4.5.3 A Practical Example: Vector Quantization

Broadcasting comes up quite often in real world problems. A typical example occurs in the vector quantization (VQ) algorithm used in information theory, classification, and other related areas. The basic operation in VQ finds the closest point in a set of points, called `codes` in VQ jargon, to a given point, called the `observation`. In the very simple, two-dimensional case shown below, the values in `observation` describe the weight and height of an athlete to be classified. The `codes` represent different classes of athletes.¹ Finding the closest point requires calculating the distance between `observation` and each of the `codes`. The shortest distance provides the best match. In this example, `codes[0]` is the closest class indicating that the athlete is likely a basketball player.

```
>>> from numpy import array, argmin, sqrt, sum
>>> observation = array([111.0, 188.0])
>>> codes = array([[102.0, 203.0],
...               [132.0, 193.0],
...               [45.0, 155.0],
...               [57.0, 173.0]])
>>> diff = codes - observation # the broadcast happens here
>>> dist = sqrt(sum(diff**2,axis=-1))
>>> argmin(dist)
0
```

In this example, the `observation` array is stretched to match the shape of the `codes` array:

```
Observation    (1d array):      2
Codes          (2d array):    4 x 2
Diff           (2d array):    4 x 2
```

Typically, a large number of `observations`, perhaps read from a database, are compared to a set of `codes`. Consider this scenario:

```
Observation    (2d array):    10 x 3
Codes          (2d array):    5 x 3
```

(continues on next page)

¹ In this example, weight has more impact on the distance calculation than height because of the larger values. In practice, it is important to normalize the height and weight, often by their standard deviation across the data set, so that both have equal influence on the distance calculation.

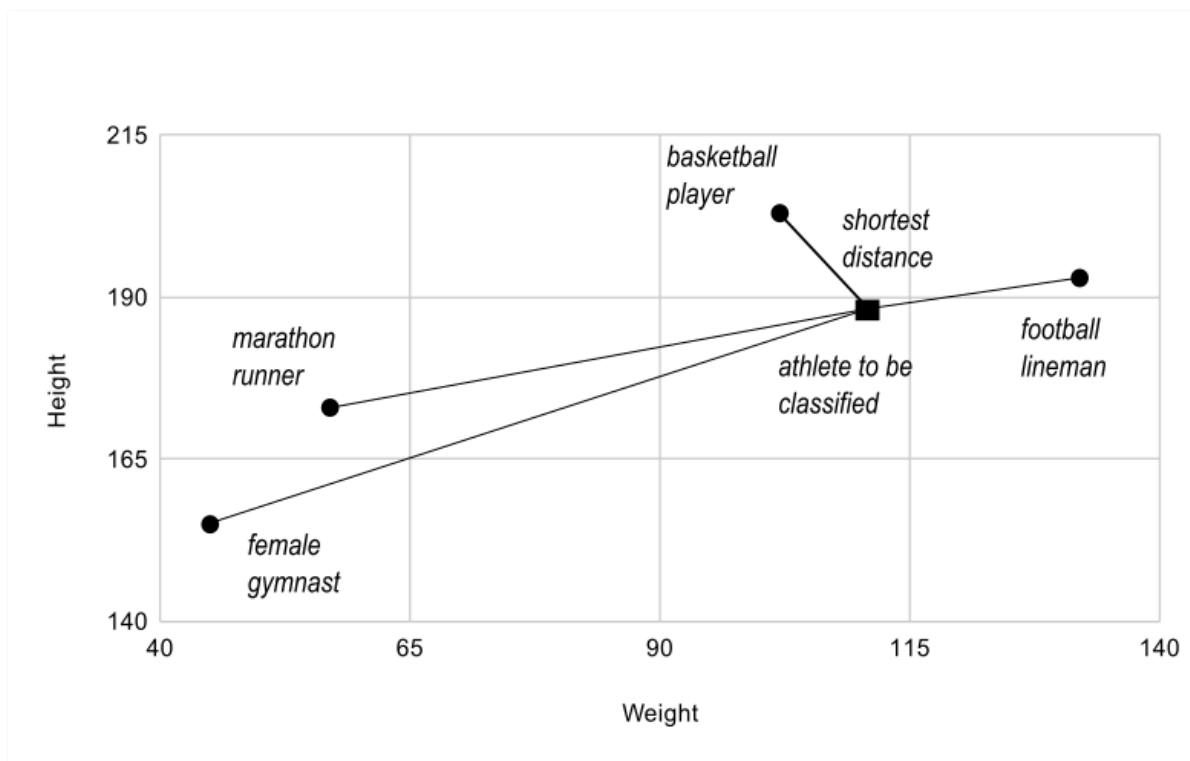


Fig. 5: Figure 5

The basic operation of vector quantization calculates the distance between an object to be classified, the dark square, and multiple known codes, the gray circles. In this simple case, the codes represent individual classes. More complex cases use multiple codes per class.

(continued from previous page)

```
Diff          (3d array):  5 x 10 x 3
```

The three-dimensional array, `diff`, is a consequence of broadcasting, not a necessity for the calculation. Large data sets will generate a large intermediate array that is computationally inefficient. Instead, if each observation is calculated individually using a Python loop around the code in the two-dimensional example above, a much smaller array is used.

Broadcasting is a powerful tool for writing short and usually intuitive code that does its computations very efficiently in C. However, there are cases when broadcasting uses unnecessarily large amounts of memory for a particular algorithm. In these cases, it is better to write the algorithm's outer loop in Python. This may also produce more readable code, as algorithms that use broadcasting tend to become more difficult to interpret as the number of dimensions in the broadcast increases.

4.6 Byte-swapping

4.6.1 Introduction to byte ordering and ndarrays

The `ndarray` is an object that provide a python array interface to data in memory.

It often happens that the memory that you want to view with an array is not of the same byte ordering as the computer on which you are running Python.

For example, I might be working on a computer with a little-endian CPU - such as an Intel Pentium, but I have loaded some data from a file written by a computer that is big-endian. Let's say I have loaded 4 bytes from a file written by a Sun (big-endian) computer. I know that these 4 bytes represent two 16-bit integers. On a big-endian machine, a two-byte integer is stored with the Most Significant Byte (MSB) first, and then the Least Significant Byte (LSB). Thus the bytes are, in memory order:

1. MSB integer 1
2. LSB integer 1
3. MSB integer 2
4. LSB integer 2

Let's say the two integers were in fact 1 and 770. Because $770 = 256 * 3 + 2$, the 4 bytes in memory would contain respectively: 0, 1, 3, 2. The bytes I have loaded from the file would have these contents:

```
>>> big_end_buffer = bytearray([0,1,3,2])
>>> big_end_buffer
bytearray(b'\x00\x01\x03\x02')
```

We might want to use an `ndarray` to access these integers. In that case, we can create an array around this memory, and tell numpy that there are two integers, and that they are 16 bit and big-endian:

```
>>> import numpy as np
>>> big_end_arr = np.ndarray(shape=(2,), dtype='>i2', buffer=big_end_buffer)
>>> big_end_arr[0]
1
>>> big_end_arr[1]
770
```

Note the array `dtype` above of `>i2`. The `>` means 'big-endian' (`<` is little-endian) and `i2` means 'signed 2-byte integer'. For example, if our data represented a single unsigned 4-byte little-endian integer, the `dtype` string would be `<u4`.

In fact, why don't we try that?

```
>>> little_end_u4 = np.ndarray(shape=(1,), dtype='<u4', buffer=big_end_buffer)
>>> little_end_u4[0] == 1 * 256**1 + 3 * 256**2 + 2 * 256**3
True
```

Returning to our `big_end_arr` - in this case our underlying data is big-endian (data endianness) and we've set the dtype to match (the dtype is also big-endian). However, sometimes you need to flip these around.

Warning: Scalars currently do not include byte order information, so extracting a scalar from an array will return an integer in native byte order. Hence:

```
>>> big_end_arr[0].dtype.byteorder == little_end_u4[0].dtype.byteorder
True
```

4.6.2 Changing byte ordering

As you can imagine from the introduction, there are two ways you can affect the relationship between the byte ordering of the array and the underlying memory it is looking at:

- Change the byte-ordering information in the array dtype so that it interprets the underlying data as being in a different byte order. This is the role of `arr.newbyteorder()`
- Change the byte-ordering of the underlying data, leaving the dtype interpretation as it was. This is what `arr.byteswap()` does.

The common situations in which you need to change byte ordering are:

1. Your data and dtype endianness don't match, and you want to change the dtype so that it matches the data.
2. Your data and dtype endianness don't match, and you want to swap the data so that they match the dtype
3. Your data and dtype endianness match, but you want the data swapped and the dtype to reflect this

Data and dtype endianness don't match, change dtype to match data

We make something where they don't match:

```
>>> wrong_end_dtype_arr = np.ndarray(shape=(2,), dtype='<i2', buffer=big_end_buffer)
>>> wrong_end_dtype_arr[0]
256
```

The obvious fix for this situation is to change the dtype so it gives the correct endianness:

```
>>> fixed_end_dtype_arr = wrong_end_dtype_arr.newbyteorder()
>>> fixed_end_dtype_arr[0]
1
```

Note the array has not changed in memory:

```
>>> fixed_end_dtype_arr.tobytes() == big_end_buffer
True
```


Data and type endianness don't match, change data to match dtype

You might want to do this if you need the data in memory to be a certain ordering. For example you might be writing the memory out to a file that needs a certain byte ordering.

```
>>> fixed_end_mem_arr = wrong_end_dtype_arr.byteswap()
>>> fixed_end_mem_arr[0]
1
```

Now the array *has* changed in memory:

```
>>> fixed_end_mem_arr.tobytes() == big_end_buffer
False
```

Data and dtype endianness match, swap data and dtype

You may have a correctly specified array dtype, but you need the array to have the opposite byte order in memory, and you want the dtype to match so the array values make sense. In this case you just do both of the previous operations:

```
>>> swapped_end_arr = big_end_arr.byteswap().newbyteorder()
>>> swapped_end_arr[0]
1
>>> swapped_end_arr.tobytes() == big_end_buffer
False
```

An easier way of casting the data to a specific dtype and byte ordering can be achieved with the ndarray astype method:

```
>>> swapped_end_arr = big_end_arr.astype('<i2')
>>> swapped_end_arr[0]
1
>>> swapped_end_arr.tobytes() == big_end_buffer
False
```

4.7 Structured arrays

4.7.1 Introduction

Structured arrays are ndarrays whose datatype is a composition of simpler datatypes organized as a sequence of named *fields*. For example,

```
>>> x = np.array([('Rex', 9, 81.0), ('Fido', 3, 27.0)],
...              dtype=[('name', 'U10'), ('age', 'i4'), ('weight', 'f4')])
>>> x
array([('Rex', 9, 81.), ('Fido', 3, 27.)],
      dtype=[('name', '<U10'), ('age', '<i4'), ('weight', '<f4')])
```

Here *x* is a one-dimensional array of length two whose datatype is a structure with three fields: 1. A string of length 10 or less named 'name', 2. a 32-bit integer named 'age', and 3. a 32-bit float named 'weight'.

If you index *x* at position 1 you get a structure:

```
>>> x[1]
('Fido', 3, 27.)
```

You can access and modify individual fields of a structured array by indexing with the field name:

```
>>> x['age']
array([9, 3], dtype=int32)
>>> x['age'] = 5
>>> x
array([( 'Rex', 5, 81.), ( 'Fido', 5, 27.)],
      dtype=[('name', '<U10'), ('age', '<i4'), ('weight', '<f4')])
```

Structured datatypes are designed to be able to mimic ‘structs’ in the C language, and share a similar memory layout. They are meant for interfacing with C code and for low-level manipulation of structured buffers, for example for interpreting binary blobs. For these purposes they support specialized features such as subarrays, nested datatypes, and unions, and allow control over the memory layout of the structure.

Users looking to manipulate tabular data, such as stored in csv files, may find other pydata projects more suitable, such as xarray, pandas, or DataArray. These provide a high-level interface for tabular data analysis and are better optimized for that use. For instance, the C-struct-like memory layout of structured arrays in numpy can lead to poor cache behavior in comparison.

4.7.2 Structured Datatypes

A structured datatype can be thought of as a sequence of bytes of a certain length (the structure’s *itemsize*) which is interpreted as a collection of fields. Each field has a name, a datatype, and a byte offset within the structure. The datatype of a field may be any numpy datatype including other structured datatypes, and it may also be a *subarray data type* which behaves like an ndarray of a specified shape. The offsets of the fields are arbitrary, and fields may even overlap. These offsets are usually determined automatically by numpy, but can also be specified.

Structured Datatype Creation

Structured datatypes may be created using the function `numpy.dtype`. There are 4 alternative forms of specification which vary in flexibility and conciseness. These are further documented in the Data Type Objects reference page, and in summary they are:

1. A list of tuples, one tuple per field

Each tuple has the form (fieldname, datatype, shape) where shape is optional. fieldname is a string (or tuple if titles are used, see *Field Titles* below), datatype may be any object convertible to a datatype, and shape is a tuple of integers specifying subarray shape.

```
>>> np.dtype([( 'x', 'f4'), ('y', np.float32), ('z', 'f4', (2, 2))])
dtype([( 'x', '<f4'), ('y', '<f4'), ('z', '<f4', (2, 2))])
```

If fieldname is the empty string ' ', the field will be given a default name of the form f#, where # is the integer index of the field, counting from 0 from the left:

```
>>> np.dtype([( 'x', 'f4'), ('', 'i4'), ('z', 'i8')])
dtype([( 'x', '<f4'), ('f1', '<i4'), ('z', '<i8')])
```

The byte offsets of the fields within the structure and the total structure itemsize are determined automatically.

2. A string of comma-separated dtype specifications

In this shorthand notation any of the string dtype specifications may be used in a string and separated by commas. The itemsize and byte offsets of the fields are determined automatically, and the field names are given the default names f0, f1, etc.

```
>>> np.dtype('i8, f4, S3')
dtype([('f0', '<i8'), ('f1', '<f4'), ('f2', 'S3')])
>>> np.dtype('3int8, float32, (2, 3)float64')
dtype([('f0', 'i1', (3,)), ('f1', '<f4'), ('f2', '<f8', (2, 3))])
```

3. A dictionary of field parameter arrays

This is the most flexible form of specification since it allows control over the byte-offsets of the fields and the itemsize of the structure.

The dictionary has two required keys, ‘names’ and ‘formats’, and four optional keys, ‘offsets’, ‘itemsize’, ‘aligned’ and ‘titles’. The values for ‘names’ and ‘formats’ should respectively be a list of field names and a list of dtype specifications, of the same length. The optional ‘offsets’ value should be a list of integer byte-offsets, one for each field within the structure. If ‘offsets’ is not given the offsets are determined automatically. The optional ‘itemsize’ value should be an integer describing the total size in bytes of the dtype, which must be large enough to contain all the fields.

```
>>> np.dtype({'names': ['col1', 'col2'], 'formats': ['i4', 'f4']})
dtype([('col1', '<i4'), ('col2', '<f4')])
>>> np.dtype({'names': ['col1', 'col2'],
...           'formats': ['i4', 'f4'],
...           'offsets': [0, 4],
...           'itemsize': 12})
dtype({'names': ['col1', 'col2'], 'formats': ['<i4', '<f4'], 'offsets': [0, 4],
      ↪ 'itemsize': 12})
```

Offsets may be chosen such that the fields overlap, though this will mean that assigning to one field may clobber any overlapping field’s data. As an exception, fields of `numpy.object_` type cannot overlap with other fields, because of the risk of clobbering the internal object pointer and then dereferencing it.

The optional ‘aligned’ value can be set to `True` to make the automatic offset computation use aligned offsets (see [Automatic Byte Offsets and Alignment](#)), as if the ‘align’ keyword argument of `numpy.dtype` had been set to `True`.

The optional ‘titles’ value should be a list of titles of the same length as ‘names’, see [Field Titles](#) below.

4. A dictionary of field names

The keys of the dictionary are the field names and the values are tuples specifying type and offset:

```
>>> np.dtype({'col1': ('i1', 0), 'col2': ('f4', 1)})
dtype([('col1', 'i1'), ('col2', '<f4')])
```

This form was discouraged because Python dictionaries did not preserve order in Python versions before Python 3.6. [Field Titles](#) may be specified by using a 3-tuple, see below.

Manipulating and Displaying Structured Datatypes

The list of field names of a structured datatype can be found in the `names` attribute of the dtype object:

```
>>> d = np.dtype([('x', 'i8'), ('y', 'f4')])
>>> d.names
('x', 'y')
```

The field names may be modified by assigning to the `names` attribute using a sequence of strings of the same length.

The dtype object also has a dictionary-like attribute, `fields`, whose keys are the field names (and [Field Titles](#), see below) and whose values are tuples containing the dtype and byte offset of each field.

```
>>> d.fields
mappingproxy({'x': (dtype('int64'), 0), 'y': (dtype('float32'), 8)})
```

Both the `names` and `fields` attributes will equal `None` for unstructured arrays. The recommended way to test if a dtype is structured is with *if `dt.names` is not `None`* rather than *if `dt.names`*, to account for dtypes with 0 fields.

The string representation of a structured datatype is shown in the “list of tuples” form if possible, otherwise numpy falls back to using the more general dictionary form.

Automatic Byte Offsets and Alignment

Numpy uses one of two methods to automatically determine the field byte offsets and the overall itemsize of a structured datatype, depending on whether `align=True` was specified as a keyword argument to `numpy.dtype`.

By default (`align=False`), numpy will pack the fields together such that each field starts at the byte offset the previous field ended, and the fields are contiguous in memory.

```
>>> def print_offsets(d):
...     print("offsets:", [d.fields[name][1] for name in d.names])
...     print("itemsize:", d.itemsize)
>>> print_offsets(np.dtype('u1, u1, i4, u1, i8, u2'))
offsets: [0, 1, 2, 6, 7, 15]
itemsize: 17
```

If `align=True` is set, numpy will pad the structure in the same way many C compilers would pad a C-struct. Aligned structures can give a performance improvement in some cases, at the cost of increased datatype size. Padding bytes are inserted between fields such that each field’s byte offset will be a multiple of that field’s alignment, which is usually equal to the field’s size in bytes for simple datatypes, see `PyArray_Descr.alignment`. The structure will also have trailing padding added so that its itemsize is a multiple of the largest field’s alignment.

```
>>> print_offsets(np.dtype('u1, u1, i4, u1, i8, u2', align=True))
offsets: [0, 1, 4, 8, 16, 24]
itemsize: 32
```

Note that although almost all modern C compilers pad in this way by default, padding in C structs is C-implementation-dependent so this memory layout is not guaranteed to exactly match that of a corresponding struct in a C program. Some work may be needed, either on the numpy side or the C side, to obtain exact correspondence.

If offsets were specified using the optional `offsets` key in the dictionary-based dtype specification, setting `align=True` will check that each field’s offset is a multiple of its size and that the itemsize is a multiple of the largest field size, and raise an exception if not.

If the offsets of the fields and itemsize of a structured array satisfy the alignment conditions, the array will have the `ALIGNED` flag set.

A convenience function `numpy.lib.recfunctions.repack_fields` converts an aligned dtype or array to a packed one and vice versa. It takes either a dtype or structured ndarray as an argument, and returns a copy with fields re-packed, with or without padding bytes.

Field Titles

In addition to field names, fields may also have an associated *title*, an alternate name, which is sometimes used as an additional description or alias for the field. The title may be used to index an array, just like a field name.

To add titles when using the list-of-tuples form of dtype specification, the field name may be specified as a tuple of two strings instead of a single string, which will be the field's title and field name respectively. For example:

```
>>> np.dtype([(('my title', 'name'), 'f4')])
dtype([(('my title', 'name'), '<f4')])
```

When using the first form of dictionary-based specification, the titles may be supplied as an extra 'titles' key as described above. When using the second (discouraged) dictionary-based specification, the title can be supplied by providing a 3-element tuple (datatype, offset, title) instead of the usual 2-element tuple:

```
>>> np.dtype({'name': ('i4', 0, 'my title')})
dtype([(('my title', 'name'), '<i4')])
```

The `dtype.fields` dictionary will contain titles as keys, if any titles are used. This means effectively that a field with a title will be represented twice in the fields dictionary. The tuple values for these fields will also have a third element, the field title. Because of this, and because the `names` attribute preserves the field order while the `fields` attribute may not, it is recommended to iterate through the fields of a dtype using the `names` attribute of the dtype, which will not list titles, as in:

```
>>> for name in d.names:
...     print(d.fields[name][:2])
(dtype('int64'), 0)
(dtype('float32'), 8)
```

Union types

Structured datatypes are implemented in numpy to have base type `numpy.void` by default, but it is possible to interpret other numpy types as structured types using the (base_dtype, dtype) form of dtype specification described in Data Type Objects. Here, `base_dtype` is the desired underlying dtype, and fields and flags will be copied from `dtype`. This dtype is similar to a 'union' in C.

4.7.3 Indexing and Assignment to Structured arrays

Assigning data to a Structured Array

There are a number of ways to assign values to a structured array: Using python tuples, using scalar values, or using other structured arrays.

Assignment from Python Native Types (Tuples)

The simplest way to assign values to a structured array is using python tuples. Each assigned value should be a tuple of length equal to the number of fields in the array, and not a list or array as these will trigger numpy's broadcasting rules. The tuple's elements are assigned to the successive fields of the array, from left to right:

```
>>> x = np.array([(1, 2, 3), (4, 5, 6)], dtype='i8, f4, f8')
>>> x[1] = (7, 8, 9)
>>> x
array([(1, 2., 3.), (7, 8., 9.)],
      dtype=[('f0', '<i8'), ('f1', '<f4'), ('f2', '<f8')])
```

Assignment from Scalars

A scalar assigned to a structured element will be assigned to all fields. This happens when a scalar is assigned to a structured array, or when an unstructured array is assigned to a structured array:

```
>>> x = np.zeros(2, dtype='i8, f4, ?, S1')
>>> x[:] = 3
>>> x
array([(3, 3., True, b'3'), (3, 3., True, b'3')],
      dtype=[('f0', '<i8'), ('f1', '<f4'), ('f2', '?'), ('f3', 'S1')])
>>> x[:] = np.arange(2)
>>> x
array([(0, 0., False, b'0'), (1, 1., True, b'1')],
      dtype=[('f0', '<i8'), ('f1', '<f4'), ('f2', '?'), ('f3', 'S1')])
```

Structured arrays can also be assigned to unstructured arrays, but only if the structured datatype has just a single field:

```
>>> twofield = np.zeros(2, dtype=[('A', 'i4'), ('B', 'i4')])
>>> onefield = np.zeros(2, dtype=[('A', 'i4')])
>>> nostruct = np.zeros(2, dtype='i4')
>>> nostruct[:] = twofield
Traceback (most recent call last):
...
TypeError: Cannot cast array data from dtype([('A', '<i4'), ('B', '<i4')]) to dtype(
↪ 'int32') according to the rule 'unsafe'
```

Assignment from other Structured Arrays

Assignment between two structured arrays occurs as if the source elements had been converted to tuples and then assigned to the destination elements. That is, the first field of the source array is assigned to the first field of the destination array, and the second field likewise, and so on, regardless of field names. Structured arrays with a different number of fields cannot be assigned to each other. Bytes of the destination structure which are not included in any of the fields are unaffected.

```
>>> a = np.zeros(3, dtype=[('a', 'i8'), ('b', 'f4'), ('c', 'S3')])
>>> b = np.ones(3, dtype=[('x', 'f4'), ('y', 'S3'), ('z', 'O')])
>>> b[:] = a
>>> b
array([(0., b'0.0', b''), (0., b'0.0', b''), (0., b'0.0', b'')],
      dtype=[('x', '<f4'), ('y', 'S3'), ('z', 'O')])
```

Assignment involving subarrays

When assigning to fields which are subarrays, the assigned value will first be broadcast to the shape of the subarray.

Indexing Structured Arrays

Accessing Individual Fields

Individual fields of a structured array may be accessed and modified by indexing the array with the field name.

```
>>> x = np.array([(1, 2), (3, 4)], dtype=[('foo', 'i8'), ('bar', 'f4')])
>>> x['foo']
array([1, 3])
>>> x['foo'] = 10
>>> x
array([(10, 2.), (10, 4.)],
      dtype=[('foo', '<i8'), ('bar', '<f4')])
```

The resulting array is a view into the original array. It shares the same memory locations and writing to the view will modify the original array.

```
>>> y = x['bar']
>>> y[:] = 11
>>> x
array([(10, 11.), (10, 11.)],
      dtype=[('foo', '<i8'), ('bar', '<f4')])
```

This view has the same dtype and itemsize as the indexed field, so it is typically a non-structured array, except in the case of nested structures.

```
>>> y.dtype, y.shape, y.strides
(dtype('float32'), (2,), (12,))
```

If the accessed field is a subarray, the dimensions of the subarray are appended to the shape of the result:

```
>>> x = np.zeros((2, 2), dtype=[('a', np.int32), ('b', np.float64, (3, 3))])
>>> x['a'].shape
(2, 2)
>>> x['b'].shape
(2, 2, 3, 3)
```

Accessing Multiple Fields

One can index and assign to a structured array with a multi-field index, where the index is a list of field names.

Warning: The behavior of multi-field indexes changed from Numpy 1.15 to Numpy 1.16.

The result of indexing with a multi-field index is a view into the original array, as follows:

```
>>> a = np.zeros(3, dtype=[('a', 'i4'), ('b', 'i4'), ('c', 'f4')])
>>> a[['a', 'c']]
array([(0, 0.), (0, 0.), (0, 0.)],
      dtype={'names': ['a', 'c'], 'formats': ['<i4', '<f4'], 'offsets': [0, 8],
      ↪ 'itemsize': 12})
```

Assignment to the view modifies the original array. The view's fields will be in the order they were indexed. Note that unlike for single-field indexing, the dtype of the view has the same itemsize as the original array, and has fields at the same offsets as in the original array, and unindexed fields are merely missing.

Warning: In Numpy 1.15, indexing an array with a multi-field index returned a copy of the result above, but with fields packed together in memory as if passed through `numpy.lib.recfunctions.repack_fields`.

The new behavior as of Numpy 1.16 leads to extra “padding” bytes at the location of unindexed fields compared to 1.15. You will need to update any code which depends on the data having a “packed” layout. For instance code such as:

```
>>> a[['a', 'c']].view('i8') # Fails in Numpy 1.16
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: When changing to a smaller dtype, its size must be a divisor of the
↪ size of original dtype
```

will need to be changed. This code has raised a `FutureWarning` since Numpy 1.12, and similar code has raised `FutureWarning` since 1.7.

In 1.16 a number of functions have been introduced in the `numpy.lib.recfunctions` module to help users account for this change. These are `numpy.lib.recfunctions.repack_fields`, `numpy.lib.recfunctions.structured_to_unstructured`, `numpy.lib.recfunctions.unstructured_to_structured`, `numpy.lib.recfunctions.apply_along_fields`, `numpy.lib.recfunctions.assign_fields_by_name`, and `numpy.lib.recfunctions.require_fields`.

The function `numpy.lib.recfunctions.repack_fields` can always be used to reproduce the old behavior, as it will return a packed copy of the structured array. The code above, for example, can be replaced with:

```
>>> from numpy.lib.recfunctions import repack_fields
>>> repack_fields(a[['a', 'c']]).view('i8') # supported in 1.16
array([0, 0, 0])
```

Furthermore, `numpy` now provides a new function `numpy.lib.recfunctions.structured_to_unstructured` which is a safer and more efficient alternative for users who wish to convert structured arrays to unstructured arrays, as the view above is often intended to do. This function allows safe conversion to an unstructured type taking into account padding, often avoids a copy, and also casts the datatypes as needed, unlike the view. Code such as:

```
>>> b = np.zeros(3, dtype=[('x', 'f4'), ('y', 'f4'), ('z', 'f4')])
>>> b[['x', 'z']].view('f4')
array([0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

can be made safer by replacing with:

```
>>> from numpy.lib.recfunctions import structured_to_unstructured
>>> structured_to_unstructured(b[['x', 'z']])
array([[0., 0.],
       [0., 0.],
       [0., 0.]], dtype=float32)
```

Assignment to an array with a multi-field index modifies the original array:

```
>>> a[['a', 'c']] = (2, 3)
>>> a
array([(2, 0, 3.), (2, 0, 3.), (2, 0, 3.)],
      dtype=[('a', '<i4'), ('b', '<i4'), ('c', '<f4')])
```

This obeys the structured array assignment rules described above. For example, this means that one can swap the values of two fields using appropriate multi-field indexes:

```
>>> a[['a', 'c']] = a[['c', 'a']]
```

Indexing with an Integer to get a Structured Scalar

Indexing a single element of a structured array (with an integer index) returns a structured scalar:

```
>>> x = np.array([(1, 2., 3.)], dtype='i, f, f')
>>> scalar = x[0]
>>> scalar
(1, 2., 3.)
>>> type(scalar)
<class 'numpy.void'>
```

Unlike other `numpy` scalars, structured scalars are mutable and act like views into the original array, such that modifying the scalar will modify the original array. Structured scalars also support access and assignment by field name:


```
>>> x = np.array([(1, 2), (3, 4)], dtype=[('foo', 'i8'), ('bar', 'f4')])
>>> s = x[0]
>>> s['bar'] = 100
>>> x
array([(1, 100.), (3, 4.)],
      dtype=[('foo', '<i8'), ('bar', '<f4')])
```

Similarly to tuples, structured scalars can also be indexed with an integer:

```
>>> scalar = np.array([(1, 2., 3.)], dtype='i, f, f')[0]
>>> scalar[0]
1
>>> scalar[1] = 4
```

Thus, tuples might be thought of as the native Python equivalent to numpy's structured types, much like native python integers are the equivalent to numpy's integer types. Structured scalars may be converted to a tuple by calling `numpy.ndarray.item`:

```
>>> scalar.item(), type(scalar.item())
((1, 4.0, 3.0), <class 'tuple'>)
```

Viewing Structured Arrays Containing Objects

In order to prevent clobbering object pointers in fields of `object` type, numpy currently does not allow views of structured arrays containing objects.

Structure Comparison and Promotion

If the dtypes of two void structured arrays are equal, testing the equality of the arrays will result in a boolean array with the dimensions of the original arrays, with elements set to `True` where all fields of the corresponding structures are equal:

```
>>> a = np.array([(1, 1), (2, 2)], dtype=[('a', 'i4'), ('b', 'i4')])
>>> b = np.array([(1, 1), (2, 3)], dtype=[('a', 'i4'), ('b', 'i4')])
>>> a == b
array([True, False])
```

NumPy will promote individual field datatypes to perform the comparison. So the following is also valid (note the `'f4'` dtype for the `'a'` field):

```
>>> b = np.array([(1.0, 1), (2.5, 2)], dtype=[("a", "f4"), ("b", "i4")])
>>> a == b
array([True, False])
```

To compare two structured arrays, it must be possible to promote them to a common dtype as returned by `numpy.result_type` and `np.promote_types`. This enforces that the number of fields, the field names, and the field titles must match precisely. When promotion is not possible, for example due to mismatching field names, NumPy will raise an error. Promotion between two structured dtypes results in a canonical dtype that ensures native byte-order for all fields:

```
>>> np.result_type(np.dtype("i,>i"))
dtype([('f0', '<i4'), ('f1', '<i4')])
>>> np.result_type(np.dtype("i,>i"), np.dtype("i,i"))
dtype([('f0', '<i4'), ('f1', '<i4')])
```

The resulting dtype from promotion is also guaranteed to be packed, meaning that all fields are ordered contiguously and any unnecessary padding is removed:

```
>>> dt = np.dtype("i1,V3,i4,V1")(["f0", "f2"])
>>> dt
dtype({'names': ['f0', 'f2'], 'formats': ['i1', '<i4'], 'offsets': [0, 4], 'itemsize': 9})
>>> np.result_type(dt)
dtype([('f0', 'i1'), ('f2', '<i4')])
```

Note that the result prints without offsets or itemsize indicating no additional padding. If a structured dtype is created with `align=True` ensuring that `dtype.isalignedstruct` is true, this property is preserved:

```
>>> dt = np.dtype("i1,V3,i4,V1", align=True)(["f0", "f2"])
>>> dt
dtype({'names': ['f0', 'f2'], 'formats': ['i1', '<i4'], 'offsets': [0, 4], 'itemsize': 12},
      ↪ align=True)
>>> np.result_type(dt)
dtype([('f0', 'i1'), ('f2', '<i4')], align=True)
>>> np.result_type(dt).isalignedstruct
True
```

When promoting multiple dtypes, the result is aligned if any of the inputs is:

```
>>> np.result_type(np.dtype("i,i"), np.dtype("i,i", align=True))
dtype([('f0', '<i4'), ('f1', '<i4')], align=True)
```

The `<` and `>` operators always return `False` when comparing void structured arrays, and arithmetic and bitwise operations are not supported.

Changed in version 1.23: Before NumPy 1.23, a warning was given and `False` returned when promotion to a common dtype failed. Further, promotion was much more restrictive: It would reject the mixed float/integer comparison example above.

4.7.4 Record Arrays

As an optional convenience numpy provides an ndarray subclass, `numpy.recarray` that allows access to fields of structured arrays by attribute instead of only by index. Record arrays use a special datatype, `numpy.record`, that allows field access by attribute on the structured scalars obtained from the array. The `numpy.rec` module provides functions for creating recarrays from various objects. Additional helper functions for creating and manipulating structured arrays can be found in `numpy.lib.recfunctions`.

The simplest way to create a record array is with `numpy.rec.array`:

```
>>> recordarr = np.rec.array([(1, 2., 'Hello'), (2, 3., "World")],
...                          dtype=[('foo', 'i4'), ('bar', 'f4'), ('baz', 'S10')])
>>> recordarr.bar
array([2., 3.], dtype=float32)
>>> recordarr[1:2]
rec.array([(2, 3., b'World')],
          dtype=[('foo', '<i4'), ('bar', '<f4'), ('baz', 'S10')])
>>> recordarr[1:2].foo
array([2], dtype=int32)
>>> recordarr.foo[1:2]
array([2], dtype=int32)
>>> recordarr[1].baz
b'World'
```

`numpy.rec.array` can convert a wide variety of arguments into record arrays, including structured arrays:

```
>>> arr = np.array([(1, 2., 'Hello'), (2, 3., "World")],
...                 dtype=[('foo', 'i4'), ('bar', 'f4'), ('baz', 'S10')])
>>> recordarr = np.rec.array(arr)
```

The `numpy.rec` module provides a number of other convenience functions for creating record arrays, see record array creation routines.

A record array representation of a structured array can be obtained using the appropriate [view](#):

```
>>> arr = np.array([(1, 2., 'Hello'), (2, 3., "World")],
...                 dtype=[('foo', 'i4'), ('bar', 'f4'), ('baz', 'a10')])
>>> recordarr = arr.view(dtype=np.dtype((np.record, arr.dtype)),
...                       type=np.recarray)
```

For convenience, viewing an ndarray as type `numpy.recarray` will automatically convert to `numpy.record` datatype, so the dtype can be left out of the view:

```
>>> recordarr = arr.view(np.recarray)
>>> recordarr.dtype
dtype((numpy.record, [('foo', '<i4'), ('bar', '<f4'), ('baz', 'S10')]))
```

To get back to a plain ndarray both the dtype and type must be reset. The following view does so, taking into account the unusual case that the recordarr was not a structured type:

```
>>> arr2 = recordarr.view(recordarr.dtype.fields or recordarr.dtype, np.ndarray)
```

Record array fields accessed by index or by attribute are returned as a record array if the field has a structured type but as a plain ndarray otherwise.

```
>>> recordarr = np.rec.array([('Hello', (1, 2)), ("World", (3, 4))],
...                           dtype=[('foo', 'S6'), ('bar', [(('A', int), ('B', int))])])
>>> type(recordarr.foo)
<class 'numpy.ndarray'>
>>> type(recordarr.bar)
<class 'numpy.recarray'>
```

Note that if a field has the same name as an ndarray attribute, the ndarray attribute takes precedence. Such fields will be inaccessible by attribute but will still be accessible by index.

Recarray Helper Functions

Collection of utilities to manipulate structured arrays.

Most of these functions were initially implemented by John Hunter for matplotlib. They have been rewritten and extended for convenience.

`numpy.lib.recfunctions.append_fields` (*base, names, data, dtypes=None, fill_value=-1, usemask=True, asrecarray=False*)

Add new fields to an existing array.

The names of the fields are given with the *names* arguments, the corresponding values with the *data* arguments. If a single field is appended, *names*, *data* and *dtypes* do not have to be lists but just values.

Parameters

base

[array] Input array to extend.

names

[string, sequence] String or sequence of strings corresponding to the names of the new fields.

data

[array or sequence of arrays] Array or sequence of arrays storing the fields to add to the base.

dtypes

[sequence of datatypes, optional] Datatype or sequence of datatypes. If None, the datatypes are estimated from the *data*.

fill_value

[{float}, optional] Filling value used to pad missing data on the shorter arrays.

usemask

[{False, True}, optional] Whether to return a masked array or not.

asrecarray

[{False, True}, optional] Whether to return a recarray (MaskedRecords) or not.

`numpy.lib.recfunctions.apply_along_fields(func, arr)`

Apply function ‘func’ as a reduction across fields of a structured array.

This is similar to *apply_along_axis*, but treats the fields of a structured array as an extra axis. The fields are all first cast to a common type following the type-promotion rules from `numpy.result_type` applied to the field’s dtypes.

Parameters**func**

[function] Function to apply on the “field” dimension. This function must support an *axis* argument, like `np.mean`, `np.sum`, etc.

arr

[ndarray] Structured array for which to apply func.

Returns**out**

[ndarray] Result of the recution operation

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> b = np.array([(1, 2, 5), (4, 5, 7), (7, 8, 11), (10, 11, 12)],
...              dtype=[('x', 'i4'), ('y', 'f4'), ('z', 'f8')])
>>> rfn.apply_along_fields(np.mean, b)
array([ 2.66666667,  5.33333333,  8.66666667, 11.        ])
>>> rfn.apply_along_fields(np.mean, b[['x', 'z']])
array([ 3. ,  5.5,  9. , 11. ])
```

`numpy.lib.recfunctions.assign_fields_by_name(dst, src, zero_unassigned=True)`

Assigns values from one structured array to another by field name.

Normally in numpy ≥ 1.14 , assignment of one structured array to another copies fields “by position”, meaning that the first field from the *src* is copied to the first field of the *dst*, and so on, regardless of field name.

This function instead copies “by field name”, such that fields in the *dst* are assigned from the identically named field in the *src*. This applies recursively for nested structures. This is how structure assignment worked in numpy ≥ 1.6 to ≤ 1.13 .

Parameters

dst

[ndarray]

src

[ndarray] The source and destination arrays during assignment.

zero_unassigned

[bool, optional] If True, fields in the *dst* for which there was no matching field in the *src* are filled with the value 0 (zero). This was the behavior of numpy ≤ 1.13 . If False, those fields are not modified.

`numpy.lib.recfunctions.drop_fields(base, drop_names, usemask=True, asrecarray=False)`

Return a new array with fields in *drop_names* dropped.

Nested fields are supported.

Changed in version 1.18.0: `drop_fields` returns an array with 0 fields if all fields are dropped, rather than returning `None` as it did previously.

Parameters

base

[array] Input array

drop_names

[string or sequence] String or sequence of strings corresponding to the names of the fields to drop.

usemask

[{False, True}, optional] Whether to return a masked array or not.

asrecarray

[string or sequence, optional] Whether to return a recarray or a mrecarray (`asrecarray=True`) or a plain ndarray or masked array with flexible dtype. The default is False.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> a = np.array([(1, (2, 3.0)), (4, (5, 6.0))],
...              dtype=[('a', np.int64), ('b', [('ba', np.double), ('bb', np.int64)])])
>>> rfn.drop_fields(a, 'a')
array([(2., 3), (5., 6)],
      dtype=[('b', [('ba', '<f8'), ('bb', '<i8')])])
>>> rfn.drop_fields(a, 'ba')
array([(1, (3,)), (4, (6,))], dtype=[('a', '<i8'), ('b', [('bb', '<i8')])])
>>> rfn.drop_fields(a, ['ba', 'bb'])
array([(1,), (4,)], dtype=[('a', '<i8')])
```

`numpy.lib.recfunctions.find_duplicates` (*a*, *key=None*, *ignoremask=True*, *return_index=False*)

Find the duplicates in a structured array along a given key

Parameters

a

[array-like] Input array

key

[{string, None}, optional] Name of the fields along which to check the duplicates. If None, the search is performed by records

ignoremask

[{True, False}, optional] Whether masked data should be discarded or considered as duplicates.

return_index

[{False, True}, optional] Whether to return the indices of the duplicated values.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> ndtype = [('a', int)]
>>> a = np.ma.array([1, 1, 1, 2, 2, 3, 3],
...                 mask=[0, 0, 1, 0, 0, 0, 1]).view(ndtype)
>>> rfn.find_duplicates(a, ignoremask=True, return_index=True)
(masked_array(data=[(1,), (1,), (2,), (2,)],
               mask=[(False,), (False,), (False,), (False,)],
               fill_value=999999),
 dtype=[('a', '<i8')]), array([0, 1, 3, 4]))
```

`numpy.lib.recfunctions.flatten_descr` (*ndtype*)

Flatten a structured data-type description.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> ndtype = np.dtype([('a', '<i4'), ('b', [('ba', '<f8'), ('bb', '<i4')])])
>>> rfn.flatten_descr(ndtype)
(('a', dtype('int32')), ('ba', dtype('float64')), ('bb', dtype('int32')))
```

`numpy.lib.recfunctions.get_fieldstructure` (*adtype*, *lastname=None*, *parents=None*)

Returns a dictionary with fields indexing lists of their parent fields.

This function is used to simplify access to fields nested in other fields.

Parameters

adtype

[`np.dtype`] Input datatype

lastname

[optional] Last processed field name (used internally during recursion).

parents

[dictionary] Dictionary of parent fields (used internally during recursion).

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> ndtype = np.dtype([('A', int),
...                     ('B', [('BA', int),
...                             ('BB', [('BBA', int), ('BBB', int)]))]])
>>> rfn.get_fieldstructure(ndtype)
... # XXX: possible regression, order of BBA and BBB is swapped
{'A': [], 'B': [], 'BA': ['B'], 'BB': ['B'], 'BBA': ['B', 'BB'], 'BBB': ['B', 'BB
↪']}
```

`numpy.lib.recfunctions.get_names` (*adtype*)

Returns the field names of the input datatype as a tuple. Input datatype must have fields otherwise error is raised.

Parameters

adtype

[`dtype`] Input datatype

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> rfn.get_names(np.empty((1,), dtype=[('A', int)]).dtype)
('A',)
>>> rfn.get_names(np.empty((1,), dtype=[('A', int), ('B', float)]).dtype)
('A', 'B')
>>> adtype = np.dtype([('a', int), ('b', [('ba', int), ('bb', int)])])
>>> rfn.get_names(adtype)
('a', ('b', ('ba', 'bb')))
```

`numpy.lib.recfunctions.get_names_flat` (*adtype*)

Returns the field names of the input datatype as a tuple. Input datatype must have fields otherwise error is raised. Nested structure are flattened beforehand.

Parameters

adtype

[dtype] Input datatype

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> rfn.get_names_flat(np.empty((1,), dtype=[('A', int)]).dtype) is None
False
>>> rfn.get_names_flat(np.empty((1,), dtype=[('A', int), ('B', str)]).dtype)
('A', 'B')
>>> adtype = np.dtype([('a', int), ('b', [('ba', int), ('bb', int)])])
>>> rfn.get_names_flat(adtype)
('a', 'b', 'ba', 'bb')
```

`numpy.lib.recfunctions.join_by` (*key*, *r1*, *r2*, *jointype*='inner', *r1postfix*='1', *r2postfix*='2', *defaults*=None, *usemask*=True, *asrecarray*=False)

Join arrays *r1* and *r2* on key *key*.

The key should be either a string or a sequence of string corresponding to the fields used to join the array. An exception is raised if the *key* field cannot be found in the two input arrays. Neither *r1* nor *r2* should have any duplicates along *key*: the presence of duplicates will make the output quite unreliable. Note that duplicates are not looked for by the algorithm.

Parameters

key

[{string, sequence}] A string or a sequence of strings corresponding to the fields used for comparison.

r1, r2

[arrays] Structured arrays.

jointype

[{'inner', 'outer', 'leftouter'}, optional] If 'inner', returns the elements common to both *r1* and *r2*. If 'outer', returns the common elements as well as the elements of *r1* not in *r2* and the elements of not in *r2*. If 'leftouter', returns the common elements and the elements of *r1* not in *r2*.

r1postfix

[string, optional] String appended to the names of the fields of *r1* that are present in *r2* but absent of the key.

r2postfix

[string, optional] String appended to the names of the fields of *r2* that are present in *r1* but absent of the key.

defaults

[{dictionary}, optional] Dictionary mapping field names to the corresponding default values.

usemask

[{True, False}, optional] Whether to return a MaskedArray (or MaskedRecords if *asrecarray==True*) or a ndarray.

asrecarray

[{False, True}, optional] Whether to return a recarray (or MaskedRecords if *usemask==True*) or just a flexible-type ndarray.

Notes

- The output is sorted along the key.
- A temporary array is formed by dropping the fields not in the key for the two arrays and concatenating the result. This array is then sorted, and the common entries selected. The output is constructed by filling the fields with the selected entries. Matching is not preserved if there are some duplicates...

`numpy.lib.recfunctions.merge_arrays` (*seqarrays*, *fill_value=-1*, *flatten=False*, *usemask=False*, *asrecarray=False*)

Merge arrays field by field.

Parameters**seqarrays**

[sequence of ndarrays] Sequence of arrays

fill_value

[{float}, optional] Filling value used to pad missing data on the shorter arrays.

flatten

[{False, True}, optional] Whether to collapse nested fields.

usemask

[{False, True}, optional] Whether to return a masked array or not.

asrecarray

[{False, True}, optional] Whether to return a recarray (MaskedRecords) or not.

Notes

- Without a mask, the missing value will be filled with something, depending on what its corresponding type:
 - -1 for integers
 - -1.0 for floating point numbers
 - '-' for characters
 - '-1' for strings
 - True for boolean values
- XXX: I just obtained these values empirically

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> rfn.merge_arrays((np.array([1, 2]), np.array([10., 20., 30.])))
array([( 1, 10.), ( 2, 20.), (-1, 30.)],
      dtype=[('f0', '<i8'), ('f1', '<f8')])
```

```
>>> rfn.merge_arrays((np.array([1, 2], dtype=np.int64),
...                   np.array([10., 20., 30.])), usemask=False)
array([(1, 10.0), (2, 20.0), (-1, 30.0)],
      dtype=[('f0', '<i8'), ('f1', '<f8')])
>>> rfn.merge_arrays((np.array([1, 2]).view([('a', np.int64)]),
...                   np.array([10., 20., 30.])),
...                   usemask=False, asrecarray=True)
rec.array([( 1, 10.), ( 2, 20.), (-1, 30.)],
          dtype=[('a', '<i8'), ('f1', '<f8')])
```

`numpy.lib.recfunctions.rec_append_fields` (*base*, *names*, *data*, *dtypes=None*)

Add new fields to an existing array.

The names of the fields are given with the *names* arguments, the corresponding values with the *data* arguments. If a single field is appended, *names*, *data* and *dtypes* do not have to be lists but just values.

Parameters

base

[array] Input array to extend.

names

[string, sequence] String or sequence of strings corresponding to the names of the new fields.

data

[array or sequence of arrays] Array or sequence of arrays storing the fields to add to the base.

dtypes

[sequence of datatypes, optional] Datatype or sequence of datatypes. If None, the datatypes are estimated from the *data*.

Returns

appended_array

[np.recarray]

See also:

[*append_fields*](#)

`numpy.lib.recfunctions.rec_drop_fields` (*base*, *drop_names*)

Returns a new `numpy.recarray` with fields in *drop_names* dropped.

`numpy.lib.recfunctions.rec_join` (*key*, *r1*, *r2*, *jointype='inner'*, *r1postfix='1'*, *r2postfix='2'*, *defaults=None*)

Join arrays *r1* and *r2* on keys. Alternative to `join_by`, that always returns a `np.recarray`.

See also:

join_by

equivalent function

`numpy.lib.recfunctions.recursive_fill_fields(input, output)`

Fills fields from output with fields from input, with support for nested structures.

Parameters**input**

[ndarray] Input array.

output

[ndarray] Output array.

Notes

- *output* should be at least the same size as *input*

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> a = np.array([(1, 10.), (2, 20.)], dtype=[('A', np.int64), ('B', np.float64)])
>>> b = np.zeros((3,), dtype=a.dtype)
>>> rfn.recursive_fill_fields(a, b)
array([(1, 10.), (2, 20.), (0, 0.)], dtype=[('A', '<i8'), ('B', '<f8')])
```

`numpy.lib.recfunctions.rename_fields(base, namemapper)`

Rename the fields from a flexible-datatype ndarray or recarray.

Nested fields are supported.

Parameters**base**

[ndarray] Input array whose fields must be modified.

namemapper

[dictionary] Dictionary mapping old field names to their new version.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> a = np.array([(1, (2, [3.0, 30.])), (4, (5, [6.0, 60.]))],
... dtype=[('a', int), ('b', [('ba', float), ('bb', (float, 2))])])
>>> rfn.rename_fields(a, {'a': 'A', 'bb': 'BB'})
array([(1, (2., [ 3., 30.])), (4, (5., [ 6., 60.])),
      dtype=[('A', '<i8'), ('b', [('ba', '<f8'), ('BB', '<f8', (2,))])])
```

`numpy.lib.recfunctions.repack_fields(a, align=False, recurse=False)`

Re-pack the fields of a structured array or dtype in memory.

The memory layout of structured datatypes allows fields at arbitrary byte offsets. This means the fields can be separated by padding bytes, their offsets can be non-monotonically increasing, and they can overlap.

This method removes any overlaps and reorders the fields in memory so they have increasing byte offsets, and adds or removes padding bytes depending on the *align* option, which behaves like the *align* option to `numpy.dtype`.

If *align=False*, this method produces a “packed” memory layout in which each field starts at the byte the previous field ended, and any padding bytes are removed.

If *align=True*, this method produces an “aligned” memory layout in which each field’s offset is a multiple of its alignment, and the total itemsize is a multiple of the largest alignment, by adding padding bytes as needed.

Parameters

a

[ndarray or dtype] array or dtype for which to repack the fields.

align

[boolean] If true, use an “aligned” memory layout, otherwise use a “packed” layout.

recurse

[boolean] If True, also repack nested structures.

Returns

repacked

[ndarray or dtype] Copy of *a* with fields repacked, or *a* itself if no repacking was needed.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> def print_offsets(d):
...     print("offsets:", [d.fields[name][1] for name in d.names])
...     print("itemsize:", d.itemsize)
...
>>> dt = np.dtype('u1, <i8, <f8', align=True)
>>> dt
dtype({'names': ['f0', 'f1', 'f2'], 'formats': ['u1', '<i8', '<f8'], 'offsets': [0, 8, 16], 'itemsize': 24}, align=True)
>>> print_offsets(dt)
offsets: [0, 8, 16]
itemsize: 24
>>> packed_dt = rfn.repack_fields(dt)
>>> packed_dt
dtype([('f0', 'u1'), ('f1', '<i8'), ('f2', '<f8')])
>>> print_offsets(packed_dt)
offsets: [0, 1, 9]
itemsize: 17
```

`numpy.lib.recfunctions.require_fields(array, required_dtype)`

Cast a structured array to a new dtype using assignment by field-name.

This function assigns from the old to the new array by name, so the value of a field in the output array is the value of the field with the same name in the source array. This has the effect of creating a new ndarray containing only the fields “required” by the `required_dtype`.

If a field name in the `required_dtype` does not exist in the input array, that field is created and set to 0 in the output array.

Parameters

a

[ndarray] array to cast

required_dtype

[dtype] datatype for output array

Returns

out

[ndarray] array with the new dtype, with field values copied from the fields in the input array with the same name

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> a = np.ones(4, dtype=[('a', 'i4'), ('b', 'f8'), ('c', 'u1')])
>>> rfn.require_fields(a, [('b', 'f4'), ('c', 'u1')])
array([(1., 1), (1., 1), (1., 1), (1., 1)],
      dtype=[('b', '<f4'), ('c', 'u1')])
>>> rfn.require_fields(a, [('b', 'f4'), ('newf', 'u1')])
array([(1., 0), (1., 0), (1., 0), (1., 0)],
      dtype=[('b', '<f4'), ('newf', 'u1')])
```

`numpy.lib.recfunctions.stack_arrays` (*arrays*, *defaults=None*, *usemask=True*, *asrecarray=False*, *autoconvert=False*)

Superposes arrays fields by fields

Parameters

arrays

[array or sequence] Sequence of input arrays.

defaults

[dictionary, optional] Dictionary mapping field names to the corresponding default values.

usemask

[{True, False}, optional] Whether to return a MaskedArray (or MaskedRecords if *asrecarray==True*) or a ndarray.

asrecarray

[{False, True}, optional] Whether to return a recarray (or MaskedRecords if *usemask==True*) or just a flexible-type ndarray.

autoconvert

[{False, True}, optional] Whether automatically cast the type of the field to the maximum.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> x = np.array([1, 2,])
>>> rfn.stack_arrays(x) is x
True
>>> z = np.array([('A', 1), ('B', 2)], dtype=[('A', '<S3'), ('B', float)])
>>> zz = np.array([('a', 10., 100.), ('b', 20., 200.), ('c', 30., 300.)],
...               dtype=[('A', '<S3'), ('B', np.double), ('C', np.double)])
>>> test = rfn.stack_arrays((z, zz))
>>> test
masked_array(data=[(b'A', 1.0, --), (b'B', 2.0, --), (b'a', 10.0, 100.0),
                   (b'b', 20.0, 200.0), (b'c', 30.0, 300.0)],
              mask=[(False, False,  True), (False, False,  True),
                    (False, False, False), (False, False, False),
                    (False, False, False)],
              fill_value=(b'N/A', 1.e+20, 1.e+20),
              dtype=[('A', '<S3'), ('B', '<f8'), ('C', '<f8')])
```

`numpy.lib.recfunctions.structured_to_unstructured` (*arr*, *dtype=None*, *copy=False*, *casting='unsafe'*)

Converts an n-D structured array into an (n+1)-D unstructured array.

The new array will have a new last dimension equal in size to the number of field-elements of the input array. If not supplied, the output datatype is determined from the numpy type promotion rules applied to all the field datatypes.

Nested fields, as well as each element of any subarray fields, all count as a single field-elements.

Parameters**arr**

[ndarray] Structured array or dtype to convert. Cannot contain object datatype.

dtype

[dtype, optional] The dtype of the output unstructured array.

copy

[bool, optional] See `copy` argument to `numpy.ndarray.astype`. If true, always return a copy. If false, and *dtype* requirements are satisfied, a view is returned.

casting

[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] See `casting` argument of `numpy.ndarray.astype`. Controls what kind of data casting may occur.

Returns**unstructured**

[ndarray] Unstructured array with one more dimension.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> a = np.zeros(4, dtype=[('a', 'i4'), ('b', 'f4,u2'), ('c', 'f4', 2)])
>>> a
array([(0, (0., 0), [0., 0.]), (0, (0., 0), [0., 0.]),
      (0, (0., 0), [0., 0.]), (0, (0., 0), [0., 0.])],
      dtype=[('a', '<i4'), ('b', [('f0', '<f4'), ('f1', '<u2')]), ('c', '<f4', (2,
      ↪))])
>>> rfn.structured_to_unstructured(a)
array([[0., 0., 0., 0., 0.],
      [0., 0., 0., 0., 0.],
      [0., 0., 0., 0., 0.],
      [0., 0., 0., 0., 0.]])
```

```
>>> b = np.array([(1, 2, 5), (4, 5, 7), (7, 8, 11), (10, 11, 12)],
...              dtype=[('x', 'i4'), ('y', 'f4'), ('z', 'f8')])
>>> np.mean(rfn.structured_to_unstructured(b[['x', 'z']]), axis=-1)
array([ 3. ,  5.5,  9. , 11. ])
```

`numpy.lib.recfunctions.structured_to_unstructured` (*arr*, *dtype=None*, *names=None*,
align=False, *copy=False*, *casting='unsafe'*)

Converts an n-D unstructured array into an (n-1)-D structured array.

The last dimension of the input array is converted into a structure, with number of field-elements equal to the size of the last dimension of the input array. By default all output fields have the input array's dtype, but an output structured dtype with an equal number of fields-elements can be supplied instead.

Nested fields, as well as each element of any subarray fields, all count towards the number of field-elements.

Parameters

arr

[ndarray] Unstructured array or dtype to convert.

dtype

[dtype, optional] The structured dtype of the output array

names

[list of strings, optional] If dtype is not supplied, this specifies the field names for the output dtype, in order. The field dtypes will be the same as the input array.

align

[boolean, optional] Whether to create an aligned memory layout.

copy

[bool, optional] See copy argument to `numpy.ndarray.astype`. If true, always return a copy. If false, and dtype requirements are satisfied, a view is returned.

casting

[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] See casting argument of `numpy.ndarray.astype`. Controls what kind of data casting may occur.

Returns

structured

[ndarray] Structured array with fewer dimensions.

Examples

```

>>> from numpy.lib import recfunctions as rfn
>>> dt = np.dtype([('a', 'i4'), ('b', 'f4,u2'), ('c', 'f4', 2)])
>>> a = np.arange(20).reshape((4,5))
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> rfn.unstructured_to_structured(a, dt)
array([( 0, ( 1., 2), [ 3., 4.]), ( 5, ( 6., 7), [ 8., 9.]),
      (10, (11., 12), [13., 14.]), (15, (16., 17), [18., 19.])],
      dtype=[('a', '<i4'), ('b', [('f0', '<f4'), ('f1', '<u2')]), ('c', '<f4', (2,
↪))])

```

4.8 Writing custom array containers

NumPy's dispatch mechanism, introduced in numpy version v1.16 is the recommended approach for writing custom N-dimensional array containers that are compatible with the numpy API and provide custom implementations of numpy functionality. Applications include [dask](#) arrays, an N-dimensional array distributed across multiple nodes, and [cupy](#) arrays, an N-dimensional array on a GPU.

To get a feel for writing custom array containers, we'll begin with a simple example that has rather narrow utility but illustrates the concepts involved.

```

>>> import numpy as np
>>> class DiagonalArray:
...     def __init__(self, N, value):
...         self._N = N
...         self._i = value
...     def __repr__(self):
...         return f"{self.__class__.__name__}(N={self._N}, value={self._i})"
...     def __array__(self, dtype=None):
...         return self._i * np.eye(self._N, dtype=dtype)

```

Our custom array can be instantiated like:

```

>>> arr = DiagonalArray(5, 1)
>>> arr
DiagonalArray(N=5, value=1)

```

We can convert to a numpy array using `numpy.array` or `numpy.asarray`, which will call its `__array__` method to obtain a standard `numpy.ndarray`.

```

>>> np.asarray(arr)
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])

```


If we operate on `arr` with a numpy function, numpy will again use the `__array__` interface to convert it to an array and then apply the function in the usual way.

```
>>> np.multiply(arr, 2)
array([[2., 0., 0., 0., 0.],
       [0., 2., 0., 0., 0.],
       [0., 0., 2., 0., 0.],
       [0., 0., 0., 2., 0.],
       [0., 0., 0., 0., 2.]])
```

Notice that the return type is a standard `numpy.ndarray`.

```
>>> type(np.multiply(arr, 2))
<class 'numpy.ndarray'>
```

How can we pass our custom array type through this function? Numpy allows a class to indicate that it would like to handle computations in a custom-defined way through the interfaces `__array_ufunc__` and `__array_function__`. Let's take one at a time, starting with `__array_ufunc__`. This method covers ufuncs, a class of functions that includes, for example, `numpy.multiply` and `numpy.sin`.

The `__array_ufunc__` receives:

- ufunc, a function like `numpy.multiply`
- method, a string, differentiating between `numpy.multiply(...)` and variants like `numpy.multiply.outer`, `numpy.multiply.accumulate`, and so on. For the common case, `numpy.multiply(...)`, `method == '__call__'`.
- inputs, which could be a mixture of different types
- kwargs, keyword arguments passed to the function

For this example we will only handle the method `__call__`

```
>>> from numbers import Number
>>> class DiagonalArray:
...     def __init__(self, N, value):
...         self._N = N
...         self._i = value
...     def __repr__(self):
...         return f"{self.__class__.__name__}(N={self._N}, value={self._i})"
...     def __array__(self, dtype=None):
...         return self._i * np.eye(self._N, dtype=dtype)
...     def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
...         if method == '__call__':
...             N = None
...             scalars = []
...             for input in inputs:
...                 if isinstance(input, Number):
...                     scalars.append(input)
...                 elif isinstance(input, self.__class__):
...                     scalars.append(input._i)
...                     if N is not None:
...                         if N != self._N:
...                             raise TypeError("inconsistent sizes")
...                     else:
...                         N = self._N
...             else:
...                 return NotImplemented
...             return self.__class__(N, ufunc(*scalars, **kwargs))
```

(continues on next page)

(continued from previous page)

```
...         else:
...             return NotImplemented
```

Now our custom array type passes through numpy functions.

```
>>> arr = DiagonalArray(5, 1)
>>> np.multiply(arr, 3)
DiagonalArray(N=5, value=3)
>>> np.add(arr, 3)
DiagonalArray(N=5, value=4)
>>> np.sin(arr)
DiagonalArray(N=5, value=0.8414709848078965)
```

At this point `arr + 3` does not work.

```
>>> arr + 3
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'DiagonalArray' and 'int'
```

To support it, we need to define the Python interfaces `__add__`, `__lt__`, and so on to dispatch to the corresponding ufunc. We can achieve this conveniently by inheriting from the mixin `NDArrayOperatorsMixin`.

```
>>> import numpy.lib.mixins
>>> class DiagonalArray(numpy.lib.mixins.NDArrayOperatorsMixin):
...     def __init__(self, N, value):
...         self._N = N
...         self._i = value
...     def __repr__(self):
...         return f"{self.__class__.__name__}(N={self._N}, value={self._i})"
...     def __array__(self, dtype=None):
...         return self._i * np.eye(self._N, dtype=dtype)
...     def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
...         if method == '__call__':
...             N = None
...             scalars = []
...             for input in inputs:
...                 if isinstance(input, Number):
...                     scalars.append(input)
...                 elif isinstance(input, self.__class__):
...                     scalars.append(input._i)
...                     if N is not None:
...                         if N != self._N:
...                             raise TypeError("inconsistent sizes")
...                     else:
...                         N = self._N
...             else:
...                 return NotImplemented
...             return self.__class__(N, ufunc(*scalars, **kwargs))
...         else:
...             return NotImplemented
```

```
>>> arr = DiagonalArray(5, 1)
>>> arr + 3
DiagonalArray(N=5, value=4)
>>> arr > 0
```

(continues on next page)

(continued from previous page)

```
DiagonalArray(N=5, value=True)
```

Now let's tackle `__array_function__`. We'll create dict that maps numpy functions to our custom variants.

```
>>> HANDLED_FUNCTIONS = {}
>>> class DiagonalArray(numpy.lib.mixins.NDArrayOperatorsMixin):
...     def __init__(self, N, value):
...         self._N = N
...         self._i = value
...     def __repr__(self):
...         return f"{self.__class__.__name__}(N={self._N}, value={self._i})"
...     def __array__(self, dtype=None):
...         return self._i * np.eye(self._N, dtype=dtype)
...     def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
...         if method == '__call__':
...             N = None
...             scalars = []
...             for input in inputs:
...                 # In this case we accept only scalar numbers or DiagonalArrays.
...                 if isinstance(input, Number):
...                     scalars.append(input)
...                 elif isinstance(input, self.__class__):
...                     scalars.append(input._i)
...                     if N is not None:
...                         if N != self._N:
...                             raise TypeError("inconsistent sizes")
...                     else:
...                         N = self._N
...                 else:
...                     return NotImplemented
...             return self.__class__(N, ufunc(*scalars, **kwargs))
...         else:
...             return NotImplemented
...     def __array_function__(self, func, types, args, kwargs):
...         if func not in HANDLED_FUNCTIONS:
...             return NotImplemented
...         # Note: this allows subclasses that don't override
...         # __array_function__ to handle DiagonalArray objects.
...         if not all(issubclass(t, self.__class__) for t in types):
...             return NotImplemented
...         return HANDLED_FUNCTIONS[func](*args, **kwargs)
```

A convenient pattern is to define a decorator implements that can be used to add functions to `HANDLED_FUNCTIONS`.

```
>>> def implements(np_function):
...     "Register an __array_function__ implementation for DiagonalArray objects."
...     def decorator(func):
...         HANDLED_FUNCTIONS[np_function] = func
...         return func
...     return decorator
```

Now we write implementations of numpy functions for `DiagonalArray`. For completeness, to support the usage `arr.sum()` add a method `sum` that calls `numpy.sum(self)`, and the same for `mean`.

```
>>> @implements(np.sum)
... def sum(arr):
...     "Implementation of np.sum for DiagonalArray objects"
...     return arr._i * arr._N
...
>>> @implements(np.mean)
... def mean(arr):
...     "Implementation of np.mean for DiagonalArray objects"
...     return arr._i / arr._N
...
>>> arr = DiagonalArray(5, 1)
>>> np.sum(arr)
5
>>> np.mean(arr)
0.2
```

If the user tries to use any numpy functions not included in `HANDLED_FUNCTIONS`, a `TypeError` will be raised by numpy, indicating that this operation is not supported. For example, concatenating two `DiagonalArrays` does not produce another diagonal array, so it is not supported.

```
>>> np.concatenate([arr, arr])
Traceback (most recent call last):
...
TypeError: no implementation found for 'numpy.concatenate' on types that implement __
↳array_function__: [<class '__main__.DiagonalArray'>]
```

Additionally, our implementations of `sum` and `mean` do not accept the optional arguments that numpy's implementation does.

```
>>> np.sum(arr, axis=0)
Traceback (most recent call last):
...
TypeError: sum() got an unexpected keyword argument 'axis'
```

The user always has the option of converting to a normal `numpy.ndarray` with `numpy.asarray` and using standard numpy from there.

```
>>> np.concatenate([np.asarray(arr), np.asarray(arr)])
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

Refer to the [dask source code](#) and [cupy source code](#) for more fully-worked examples of custom array containers.

See also [NEP 18](#).

4.9 Subclassing ndarray

4.9.1 Introduction

Subclassing ndarray is relatively simple, but it has some complications compared to other Python objects. On this page we explain the machinery that allows you to subclass ndarray, and the implications for implementing a subclass.

ndarrays and object creation

Subclassing ndarray is complicated by the fact that new instances of ndarray classes can come about in three different ways. These are:

1. Explicit constructor call - as in `MySubClass(params)`. This is the usual route to Python instance creation.
2. View casting - casting an existing ndarray as a given subclass
3. New from template - creating a new instance from a template instance. Examples include returning slices from a subclassed array, creating return types from ufuncs, and copying arrays. See [Creating new from template](#) for more details

The last two are characteristics of ndarrays - in order to support things like array slicing. The complications of subclassing ndarray are due to the mechanisms numpy has to support these latter two routes of instance creation.

When to use subclassing

Besides the additional complexities of subclassing a NumPy array, subclasses can run into unexpected behaviour because some functions may convert the subclass to a baseclass and “forget” any additional information associated with the subclass. This can result in surprising behavior if you use NumPy methods or functions you have not explicitly tested.

On the other hand, compared to other interoperability approaches, subclassing can be a useful because many thing will “just work”.

This means that subclassing can be a convenient approach and for a long time it was also often the only available approach. However, NumPy now provides additional interoperability protocols described in [“Interoperability with NumPy”](#). For many use-cases these interoperability protocols may now be a better fit or supplement the use of subclassing.

Subclassing can be a good fit if:

- you are less worried about maintainability or users other than yourself: Subclass will be faster to implement and additional interoperability can be added “as-needed”. And with few users, possible surprises are not an issue.
- you do not think it is problematic if the subclass information is ignored or lost silently. An example is `np.memmap` where “forgetting” about data being memory mapped cannot lead to a wrong result. An example of a subclass that sometimes confuses users are NumPy’s masked arrays. When they were introduced, subclassing was the only approach for implementation. However, today we would possibly try to avoid subclassing and rely only on interoperability protocols.

Note that also subclass authors may wish to study [Interoperability with NumPy](#) to support more complex use-cases or work around the surprising behavior.

`astropy.units.Quantity` and `xarray` are examples for array-like objects that interoperate well with NumPy. Astropy’s `Quantity` is an example which uses a dual approach of both subclassing and interoperability protocols.

4.9.2 View casting

View casting is the standard ndarray mechanism by which you take an ndarray of any subclass, and return a view of the array as another (specified) subclass:

```
>>> import numpy as np
>>> # create a completely useless ndarray subclass
>>> class C(np.ndarray): pass
>>> # create a standard ndarray
>>> arr = np.zeros((3,))
>>> # take a view of it, as our useless subclass
>>> c_arr = arr.view(C)
>>> type(c_arr)
<class '__main__.C'>
```

4.9.3 Creating new from template

New instances of an ndarray subclass can also come about by a very similar mechanism to *View casting*, when numpy finds it needs to create a new instance from a template instance. The most obvious place this has to happen is when you are taking slices of subclassed arrays. For example:

```
>>> v = c_arr[1:]
>>> type(v) # the view is of type 'C'
<class '__main__.C'>
>>> v is c_arr # but it's a new instance
False
```

The slice is a *view* onto the original `c_arr` data. So, when we take a view from the ndarray, we return a new ndarray, of the same class, that points to the data in the original.

There are other points in the use of ndarrays where we need such views, such as copying arrays (`c_arr.copy()`), creating ufunc output arrays (see also `__array_wrap__` for ufuncs and other functions), and reducing methods (like `c_arr.mean()`).

4.9.4 Relationship of view casting and new-from-template

These paths both use the same machinery. We make the distinction here, because they result in different input to your methods. Specifically, *View casting* means you have created a new instance of your array type from any potential subclass of ndarray. *Creating new from template* means you have created a new instance of your class from a pre-existing instance, allowing you - for example - to copy across attributes that are particular to your subclass.

4.9.5 Implications for subclassing

If we subclass ndarray, we need to deal not only with explicit construction of our array type, but also *View casting* or *Creating new from template*. NumPy has the machinery to do this, and it is this machinery that makes subclassing slightly non-standard.

There are two aspects to the machinery that ndarray uses to support views and new-from-template in subclasses.

The first is the use of the ndarray.`__new__` method for the main work of object initialization, rather than the more usual `__init__` method. The second is the use of the `__array_finalize__` method to allow subclasses to clean up after the creation of views and new instances from templates.

A brief Python primer on `__new__` and `__init__`

`__new__` is a standard Python method, and, if present, is called before `__init__` when we create a class instance. See the [python `__new__` documentation](#) for more detail.

For example, consider the following Python code:

```
>>> class C:
>>>     def __new__(cls, *args):
>>>         print('Cls in __new__:', cls)
>>>         print('Args in __new__:', args)
>>>         # The `object` type __new__ method takes a single argument.
>>>         return object.__new__(cls)
>>>     def __init__(self, *args):
>>>         print('type(self) in __init__:', type(self))
>>>         print('Args in __init__:', args)
```

meaning that we get:

```
>>> c = C('hello')
Cls in __new__: <class 'C'>
Args in __new__: ('hello',)
type(self) in __init__: <class 'C'>
Args in __init__: ('hello',)
```

When we call `C('hello')`, the `__new__` method gets its own class as first argument, and the passed argument, which is the string `'hello'`. After python calls `__new__`, it usually (see below) calls our `__init__` method, with the output of `__new__` as the first argument (now a class instance), and the passed arguments following.

As you can see, the object can be initialized in the `__new__` method or the `__init__` method, or both, and in fact `ndarray` does not have an `__init__` method, because all the initialization is done in the `__new__` method.

Why use `__new__` rather than just the usual `__init__`? Because in some cases, as for `ndarray`, we want to be able to return an object of some other class. Consider the following:

```
class D(C):
    def __new__(cls, *args):
        print('D cls is:', cls)
        print('D args in __new__:', args)
        return C.__new__(C, *args)

    def __init__(self, *args):
        # we never get here
        print('In D __init__')
```

meaning that:

```
>>> obj = D('hello')
D cls is: <class 'D'>
D args in __new__: ('hello',)
Cls in __new__: <class 'C'>
Args in __new__: ('hello',)
>>> type(obj)
<class 'C'>
```

The definition of `C` is the same as before, but for `D`, the `__new__` method returns an instance of class `C` rather than `D`. Note that the `__init__` method of `D` does not get called. In general, when the `__new__` method returns an object of class other than the class in which it is defined, the `__init__` method of that class is not called.

This is how subclasses of the ndarray class are able to return views that preserve the class type. When taking a view, the standard ndarray machinery creates the new ndarray object with something like:

```
obj = ndarray.__new__(subtype, shape, ...
```

where subtype is the subclass. Thus the returned view is of the same class as the subclass, rather than being of class ndarray.

That solves the problem of returning views of the same type, but now we have a new problem. The machinery of ndarray can set the class this way, in its standard methods for taking views, but the ndarray `__new__` method knows nothing of what we have done in our own `__new__` method in order to set attributes, and so on. (Aside - why not call `obj = subtype.__new__(...)` then? Because we may not have a `__new__` method with the same call signature).

The role of `__array_finalize__`

`__array_finalize__` is the mechanism that numpy provides to allow subclasses to handle the various ways that new instances get created.

Remember that subclass instances can come about in these three ways:

1. explicit constructor call (`obj = MySubClass(params)`). This will call the usual sequence of `MySubClass.__new__` then (if it exists) `MySubClass.__init__`.
2. *View casting*
3. *Creating new from template*

Our `MySubClass.__new__` method only gets called in the case of the explicit constructor call, so we can't rely on `MySubClass.__new__` or `MySubClass.__init__` to deal with the view casting and new-from-template. It turns out that `MySubClass.__array_finalize__` *does* get called for all three methods of object creation, so this is where our object creation housekeeping usually goes.

- For the explicit constructor call, our subclass will need to create a new ndarray instance of its own class. In practice this means that we, the authors of the code, will need to make a call to `ndarray.__new__(MySubClass, ...)`, a class-hierarchy prepared call to `super().__new__(cls, ...)`, or do view casting of an existing array (see below)
- For view casting and new-from-template, the equivalent of `ndarray.__new__(MySubClass, ...)` is called, at the C level.

The arguments that `__array_finalize__` receives differ for the three methods of instance creation above.

The following code allows us to look at the call sequences and arguments:

```
import numpy as np

class C(np.ndarray):
    def __new__(cls, *args, **kwargs):
        print('In __new__ with class %s' % cls)
        return super().__new__(cls, *args, **kwargs)

    def __init__(self, *args, **kwargs):
        # in practice you probably will not need or want an __init__
        # method for your subclass
        print('In __init__ with class %s' % self.__class__)

    def __array_finalize__(self, obj):
        print('In array_finalize:')
        print('    self type is %s' % type(self))
        print('    obj type is %s' % type(obj))
```


Now:

```
>>> # Explicit constructor
>>> c = C((10,))
In __new__ with class <class 'C'>
In array_finalize:
    self type is <class 'C'>
    obj type is <type 'NoneType'>
In __init__ with class <class 'C'>
>>> # View casting
>>> a = np.arange(10)
>>> cast_a = a.view(C)
In array_finalize:
    self type is <class 'C'>
    obj type is <type 'numpy.ndarray'>
>>> # Slicing (example of new-from-template)
>>> cv = c[:1]
In array_finalize:
    self type is <class 'C'>
    obj type is <class 'C'>
```

The signature of `__array_finalize__` is:

```
def __array_finalize__(self, obj):
```

One sees that the super call, which goes to `ndarray.__new__`, passes `__array_finalize__` the new object, of our own class (`self`) as well as the object from which the view has been taken (`obj`). As you can see from the output above, the `self` is always a newly created instance of our subclass, and the type of `obj` differs for the three instance creation methods:

- When called from the explicit constructor, `obj` is `None`
- When called from view casting, `obj` can be an instance of any subclass of `ndarray`, including our own.
- When called in new-from-template, `obj` is another instance of our own subclass, that we might use to update the new `self` instance.

Because `__array_finalize__` is the only method that always sees new instances being created, it is the sensible place to fill in instance defaults for new object attributes, among other tasks.

This may be clearer with an example.

4.9.6 Simple example - adding an extra attribute to ndarray

```
import numpy as np

class InfoArray(np.ndarray):

    def __new__(subtype, shape, dtype=float, buffer=None, offset=0,
                strides=None, order=None, info=None):
        # Create the ndarray instance of our type, given the usual
        # ndarray input arguments. This will call the standard
        # ndarray constructor, but return an object of our type.
        # It also triggers a call to InfoArray.__array_finalize__
        obj = super().__new__(subtype, shape, dtype,
                               buffer, offset, strides, order)
        # set the new 'info' attribute to the value passed
        obj.info = info
```

(continues on next page)

(continued from previous page)

```

    # Finally, we must return the newly created object:
    return obj

def __array_finalize__(self, obj):
    # ``self`` is a new object resulting from
    # ndarray.__new__(InfoArray, ...), therefore it only has
    # attributes that the ndarray.__new__ constructor gave it -
    # i.e. those of a standard ndarray.
    #
    # We could have got to the ndarray.__new__ call in 3 ways:
    # From an explicit constructor - e.g. InfoArray():
    #     obj is None
    #     (we're in the middle of the InfoArray.__new__
    #     constructor, and self.info will be set when we return to
    #     InfoArray.__new__)
    if obj is None: return
    # From view casting - e.g. arr.view(InfoArray):
    #     obj is arr
    #     (type(obj) can be InfoArray)
    # From new-from-template - e.g. infoarr[:3]
    #     type(obj) is InfoArray
    #
    # Note that it is here, rather than in the __new__ method,
    # that we set the default value for 'info', because this
    # method sees all creation of default objects - with the
    # InfoArray.__new__ constructor, but also with
    # arr.view(InfoArray).
    self.info = getattr(obj, 'info', None)
    # We do not need to return anything

```

Using the object looks like this:

```

>>> obj = InfoArray(shape=(3,)) # explicit constructor
>>> type(obj)
<class 'InfoArray'>
>>> obj.info is None
True
>>> obj = InfoArray(shape=(3,), info='information')
>>> obj.info
'information'
>>> v = obj[1:] # new-from-template - here - slicing
>>> type(v)
<class 'InfoArray'>
>>> v.info
'information'
>>> arr = np.arange(10)
>>> cast_arr = arr.view(InfoArray) # view casting
>>> type(cast_arr)
<class 'InfoArray'>
>>> cast_arr.info is None
True

```

This class isn't very useful, because it has the same constructor as the bare ndarray object, including passing in buffers and shapes and so on. We would probably prefer the constructor to be able to take an already formed ndarray from the usual numpy calls to `np.array` and return an object.

4.9.7 Slightly more realistic example - attribute added to existing array

Here is a class that takes a standard ndarray that already exists, casts as our type, and adds an extra attribute.

```
import numpy as np

class RealisticInfoArray(np.ndarray):

    def __new__(cls, input_array, info=None):
        # Input array is an already formed ndarray instance
        # We first cast to be our class type
        obj = np.asarray(input_array).view(cls)
        # add the new attribute to the created instance
        obj.info = info
        # Finally, we must return the newly created object:
        return obj

    def __array_finalize__(self, obj):
        # see InfoArray.__array_finalize__ for comments
        if obj is None: return
        self.info = getattr(obj, 'info', None)
```

So:

```
>>> arr = np.arange(5)
>>> obj = RealisticInfoArray(arr, info='information')
>>> type(obj)
<class 'RealisticInfoArray'>
>>> obj.info
'information'
>>> v = obj[1:]
>>> type(v)
<class 'RealisticInfoArray'>
>>> v.info
'information'
```

4.9.8 __array_ufunc__ for ufuncs

New in version 1.13.

A subclass can override what happens when executing numpy ufuncs on it by overriding the default ndarray. `__array_ufunc__` method. This method is executed *instead* of the ufunc and should return either the result of the operation, or `NotImplemented` if the operation requested is not implemented.

The signature of `__array_ufunc__` is:

```
def __array_ufunc__(ufunc, method, *inputs, **kwargs):

    - *ufunc* is the ufunc object that was called.
    - *method* is a string indicating how the Ufunc was called, either
      ``__call__`` to indicate it was called directly, or one of its
      :ref:`methods<ufuncs.methods>`: ``"reduce"`` , ``"accumulate"`` ,
      ``"reduceat"`` , ``"outer"`` , or ``"at"`` .
    - *inputs* is a tuple of the input arguments to the ``ufunc``
    - *kwargs* contains any optional or keyword arguments passed to the
      function. This includes any ``out`` arguments, which are always
      contained in a tuple.
```

A typical implementation would convert any inputs or outputs that are instances of one's own class, pass everything on to a superclass using `super()`, and finally return the results after possible back-conversion. An example, taken from the test case `test_ufunc_override_with_super` in `core/tests/test_umath.py`, is the following.

```
input numpy as np

class A(np.ndarray):
    def __array_ufunc__(self, ufunc, method, *inputs, out=None, **kwargs):
        args = []
        in_no = []
        for i, input_ in enumerate(inputs):
            if isinstance(input_, A):
                in_no.append(i)
                args.append(input_.view(np.ndarray))
            else:
                args.append(input_)

        outputs = out
        out_no = []
        if outputs:
            out_args = []
            for j, output in enumerate(outputs):
                if isinstance(output, A):
                    out_no.append(j)
                    out_args.append(output.view(np.ndarray))
                else:
                    out_args.append(output)
            kwargs['out'] = tuple(out_args)
        else:
            outputs = (None,) * ufunc.nout

        info = {}
        if in_no:
            info['inputs'] = in_no
        if out_no:
            info['outputs'] = out_no

        results = super().__array_ufunc__(ufunc, method, *args, **kwargs)
        if results is NotImplemented:
            return NotImplemented

        if method == 'at':
            if isinstance(inputs[0], A):
                inputs[0].info = info
            return

        if ufunc.nout == 1:
            results = (results,)

        results = tuple((np.asarray(result).view(A)
                        if output is None else output)
                        for result, output in zip(results, outputs))
        if results and isinstance(results[0], A):
            results[0].info = info

        return results[0] if len(results) == 1 else results
```

So, this class does not actually do anything interesting: it just converts any instances of its own to regular ndarray (other-

wise, we'd get infinite recursion!), and adds an `info` dictionary that tells which inputs and outputs it converted. Hence, e.g.,

```
>>> a = np.arange(5.).view(A)
>>> b = np.sin(a)
>>> b.info
{'inputs': [0]}
>>> b = np.sin(np.arange(5.), out=(a,))
>>> b.info
{'outputs': [0]}
>>> a = np.arange(5.).view(A)
>>> b = np.ones(1).view(A)
>>> c = a + b
>>> c.info
{'inputs': [0, 1]}
>>> a += b
>>> a.info
{'inputs': [0, 1], 'outputs': [0]}
```

Note that another approach would be to use `getattr(ufunc, methods)(*inputs, **kwargs)` instead of the `super` call. For this example, the result would be identical, but there is a difference if another operand also defines `__array_ufunc__`. E.g., lets assume that we evaluate `np.add(a, b)`, where `b` is an instance of another class `B` that has an override. If you use `super` as in the example, `ndarray.__array_ufunc__` will notice that `b` has an override, which means it cannot evaluate the result itself. Thus, it will return *NotImplemented* and so will our class `A`. Then, control will be passed over to `b`, which either knows how to deal with us and produces a result, or does not and returns *NotImplemented*, raising a `TypeError`.

If instead, we replace our `super` call with `getattr(ufunc, method)`, we effectively do `np.add(a.view(np.ndarray), b)`. Again, `B.__array_ufunc__` will be called, but now it sees an `ndarray` as the other argument. Likely, it will know how to handle this, and return a new instance of the `B` class to us. Our example class is not set up to handle this, but it might well be the best approach if, e.g., one were to re-implement `MaskedArray` using `__array_ufunc__`.

As a final note: if the `super` route is suited to a given class, an advantage of using it is that it helps in constructing class hierarchies. E.g., suppose that our other class `B` also used the `super` in its `__array_ufunc__` implementation, and we created a class `C` that depended on both, i.e., `class C(A, B)` (with, for simplicity, not another `__array_ufunc__` override). Then any ufunc on an instance of `C` would pass on to `A.__array_ufunc__`, the `super` call in `A` would go to `B.__array_ufunc__`, and the `super` call in `B` would go to `ndarray.__array_ufunc__`, thus allowing `A` and `B` to collaborate.

4.9.9 `__array_wrap__` for ufuncs and other functions

Prior to numpy 1.13, the behaviour of ufuncs could only be tuned using `__array_wrap__` and `__array_prepare__`. These two allowed one to change the output type of a ufunc, but, in contrast to `__array_ufunc__`, did not allow one to make any changes to the inputs. It is hoped to eventually deprecate these, but `__array_wrap__` is also used by other numpy functions and methods, such as `squeeze`, so at the present time is still needed for full functionality.

Conceptually, `__array_wrap__` “wraps up the action” in the sense of allowing a subclass to set the type of the return value and update attributes and metadata. Let's show how this works with an example. First we return to the simpler example subclass, but with a different name and some print statements:

```
import numpy as np

class MySubClass(np.ndarray):
```

(continues on next page)

(continued from previous page)

```

def __new__(cls, input_array, info=None):
    obj = np.asarray(input_array).view(cls)
    obj.info = info
    return obj

def __array_finalize__(self, obj):
    print('In __array_finalize__:')
    print('    self is %s' % repr(self))
    print('    obj is %s' % repr(obj))
    if obj is None: return
    self.info = getattr(obj, 'info', None)

def __array_wrap__(self, out_arr, context=None):
    print('In __array_wrap__:')
    print('    self is %s' % repr(self))
    print('    arr is %s' % repr(out_arr))
    # then just call the parent
    return super().__array_wrap__(self, out_arr, context)

```

We run a ufunc on an instance of our new array:

```

>>> obj = MySubClass(np.arange(5), info='spam')
In __array_finalize__:
    self is MySubClass([0, 1, 2, 3, 4])
    obj is array([0, 1, 2, 3, 4])
>>> arr2 = np.arange(5)+1
>>> ret = np.add(arr2, obj)
In __array_wrap__:
    self is MySubClass([0, 1, 2, 3, 4])
    arr is array([1, 3, 5, 7, 9])
In __array_finalize__:
    self is MySubClass([1, 3, 5, 7, 9])
    obj is MySubClass([0, 1, 2, 3, 4])
>>> ret
MySubClass([1, 3, 5, 7, 9])
>>> ret.info
'spam'

```

Note that the ufunc (`np.add`) has called the `__array_wrap__` method with arguments `self` as `obj`, and `out_arr` as the (`ndarray`) result of the addition. In turn, the default `__array_wrap__` (`ndarray.__array_wrap__`) has cast the result to class `MySubClass`, and called `__array_finalize__` - hence the copying of the `info` attribute. This has all happened at the C level.

But, we could do anything we wanted:

```

class SillySubClass(np.ndarray):

    def __array_wrap__(self, arr, context=None):
        return 'I lost your data'

```

```

>>> arr1 = np.arange(5)
>>> obj = arr1.view(SillySubClass)
>>> arr2 = np.arange(5)
>>> ret = np.multiply(obj, arr2)
>>> ret
'I lost your data'

```

So, by defining a specific `__array_wrap__` method for our subclass, we can tweak the output from ufuncs. The `__array_wrap__` method requires `self`, then an argument - which is the result of the ufunc - and an optional parameter `context`. This parameter is returned by ufuncs as a 3-element tuple: (name of the ufunc, arguments of the ufunc, domain of the ufunc), but is not set by other numpy functions. Though, as seen above, it is possible to do otherwise, `__array_wrap__` should return an instance of its containing class. See the masked array subclass for an implementation.

In addition to `__array_wrap__`, which is called on the way out of the ufunc, there is also an `__array_prepare__` method which is called on the way into the ufunc, after the output arrays are created but before any computation has been performed. The default implementation does nothing but pass through the array. `__array_prepare__` should not attempt to access the array data or resize the array, it is intended for setting the output array type, updating attributes and metadata, and performing any checks based on the input that may be desired before computation begins. Like `__array_wrap__`, `__array_prepare__` must return an ndarray or subclass thereof or raise an error.

4.9.10 Extra gotchas - custom `__del__` methods and `ndarray.base`

One of the problems that ndarray solves is keeping track of memory ownership of ndarrays and their views. Consider the case where we have created an ndarray, `arr` and have taken a slice with `v = arr[1:]`. The two objects are looking at the same memory. NumPy keeps track of where the data came from for a particular array or view, with the `base` attribute:

```
>>> # A normal ndarray, that owns its own data
>>> arr = np.zeros((4,))
>>> # In this case, base is None
>>> arr.base is None
True
>>> # We take a view
>>> v1 = arr[1:]
>>> # base now points to the array that it derived from
>>> v1.base is arr
True
>>> # Take a view of a view
>>> v2 = v1[1:]
>>> # base points to the original array that it was derived from
>>> v2.base is arr
True
```

In general, if the array owns its own memory, as for `arr` in this case, then `arr.base` will be `None` - there are some exceptions to this - see the numpy book for more details.

The `base` attribute is useful in being able to tell whether we have a view or the original array. This in turn can be useful if we need to know whether or not to do some specific cleanup when the subclassed array is deleted. For example, we may only want to do the cleanup if the original array is deleted, but not the views. For an example of how this can work, have a look at the `memmap` class in `numpy.core`.

4.9.11 Subclassing and Downstream Compatibility

When sub-classing `ndarray` or creating duck-types that mimic the `ndarray` interface, it is your responsibility to decide how aligned your APIs will be with those of `numpy`. For convenience, many `numpy` functions that have a corresponding `ndarray` method (e.g., `sum`, `mean`, `take`, `reshape`) work by checking if the first argument to a function has a method of the same name. If it exists, the method is called instead of coercing the arguments to a `numpy` array.

For example, if you want your sub-class or duck-type to be compatible with `numpy`'s `sum` function, the method signature for this object's `sum` method should be the following:

```
def sum(self, axis=None, dtype=None, out=None, keepdims=False):
    ...
```

This is the exact same method signature for `np.sum`, so now if a user calls `np.sum` on this object, `numpy` will call the object's own `sum` method and pass in these arguments enumerated above in the signature, and no errors will be raised because the signatures are completely compatible with each other.

If, however, you decide to deviate from this signature and do something like this:

```
def sum(self, axis=None, dtype=None):
    ...
```

This object is no longer compatible with `np.sum` because if you call `np.sum`, it will pass in unexpected arguments `out` and `keepdims`, causing a `TypeError` to be raised.

If you wish to maintain compatibility with `numpy` and its subsequent versions (which might add new keyword arguments) but do not want to surface all of `numpy`'s arguments, your function's signature should accept `**kwargs`. For example:

```
def sum(self, axis=None, dtype=None, **unused_kwargs):
    ...
```

This object is now compatible with `np.sum` again because any extraneous arguments (i.e. keywords that are not `axis` or `dtype`) will be hidden away in the `**unused_kwargs` parameter.

4.10 Universal functions (ufunc) basics

See also:

`ufuncs`

A universal function (or *ufunc* for short) is a function that operates on `ndarrays` in an element-by-element fashion, supporting *array broadcasting*, *type casting*, and several other standard features. That is, a *ufunc* is a “*vectorized*” wrapper for a function that takes a fixed number of specific inputs and produces a fixed number of specific outputs.

In NumPy, universal functions are instances of the `numpy.ufunc` class. Many of the built-in functions are implemented in compiled C code. The basic *ufuncs* operate on scalars, but there is also a generalized kind for which the basic elements are sub-arrays (vectors, matrices, etc.), and broadcasting is done over other dimensions. The simplest example is the addition operator:

```
>>> np.array([0,2,3,4]) + np.array([1,1,-1,2])
array([1, 3, 2, 6])
```

One can also produce custom `numpy.ufunc` instances using the `numpy.frompyfunc` factory function.

4.10.1 Ufunc methods

All ufuncs have four methods. They can be found at `ufuncs.methods`. However, these methods only make sense on scalar ufuncs that take two input arguments and return one output argument. Attempting to call these methods on other ufuncs will cause a `ValueError`.

The reduce-like methods all take an *axis* keyword, a *dtype* keyword, and an *out* keyword, and the arrays must all have dimension ≥ 1 . The *axis* keyword specifies the axis of the array over which the reduction will take place (with negative values counting backwards). Generally, it is an integer, though for `numpy.ufunc.reduce`, it can also be a tuple of `int` to reduce over several axes at once, or `None`, to reduce over all axes. For example:

```
>>> x = np.arange(9).reshape(3,3)
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.add.reduce(x, 1)
array([ 3, 12, 21])
>>> np.add.reduce(x, (0, 1))
36
```

The *dtype* keyword allows you to manage a very common problem that arises when naively using `ufunc.reduce`. Sometimes you may have an array of a certain data type and wish to add up all of its elements, but the result does not fit into the data type of the array. This commonly happens if you have an array of single-byte integers. The *dtype* keyword allows you to alter the data type over which the reduction takes place (and therefore the type of the output). Thus, you can ensure that the output is a data type with precision large enough to handle your output. The responsibility of altering the reduce type is mostly up to you. There is one exception: if no *dtype* is given for a reduction on the “add” or “multiply” operations, then if the input type is an integer (or Boolean) data-type and smaller than the size of the `numpy.int_` data type, it will be internally upcast to the `int_` (or `numpy.uint`) data-type. In the previous example:

```
>>> x.dtype
dtype('int64')
>>> np.multiply.reduce(x, dtype=float)
array([ 0., 28., 80.])
```

Finally, the *out* keyword allows you to provide an output array (for single-output ufuncs, which are currently the only ones supported; for future extension, however, a tuple with a single argument can be passed in). If *out* is given, the *dtype* argument is ignored. Considering `x` from the previous example:

```
>>> y = np.zeros(3, dtype=int)
>>> y
array([0, 0, 0])
>>> np.multiply.reduce(x, dtype=float, out=y)
array([ 0, 28, 80])
```

Ufuncs also have a fifth method, `numpy.ufunc.at`, that allows in place operations to be performed using advanced indexing. No *buffering* is used on the dimensions where advanced indexing is used, so the advanced index can list an item more than once and the operation will be performed on the result of the previous operation for that item.

4.10.2 Output type determination

The output of the ufunc (and its methods) is not necessarily an `ndarray`, if all input arguments are not `ndarrays`. Indeed, if any input defines an `__array_ufunc__` method, control will be passed completely to that function, i.e., the ufunc is *overridden*.

If none of the inputs overrides the ufunc, then all output arrays will be passed to the `__array_prepare__` and `__array_wrap__` methods of the input (besides `ndarrays`, and scalars) that defines it **and** has the highest `__array_priority__` of any other input to the universal function. The default `__array_priority__` of the `ndarray` is 0.0, and the default `__array_priority__` of a subtype is 0.0. Matrices have `__array_priority__` equal to 10.0.

All ufuncs can also take output arguments. If necessary, output will be cast to the data-type(s) of the provided output array(s). If a class with an `__array__` method is used for the output, results will be written to the object returned by `__array__`. Then, if the class also has an `__array_prepare__` method, it is called so metadata may be determined based on the context of the ufunc (the context consisting of the ufunc itself, the arguments passed to the ufunc, and the ufunc domain.) The array object returned by `__array_prepare__` is passed to the ufunc for computation. Finally, if the class also has an `__array_wrap__` method, the returned `ndarray` result will be passed to that method just before passing control back to the caller.

4.10.3 Broadcasting

See also:

Broadcasting basics

Each universal function takes array inputs and produces array outputs by performing the core function element-wise on the inputs (where an element is generally a scalar, but can be a vector or higher-order sub-array for generalized ufuncs). Standard *broadcasting rules* are applied so that inputs not sharing exactly the same shapes can still be usefully operated on.

By these rules, if an input has a dimension size of 1 in its shape, the first data entry in that dimension will be used for all calculations along that dimension. In other words, the stepping machinery of the *ufunc* will simply not step along that dimension (the stride will be 0 for that dimension).

4.10.4 Type casting rules

Note: In NumPy 1.6.0, a type promotion API was created to encapsulate the mechanism for determining output types. See the functions `numpy.result_type`, `numpy.promote_types`, and `numpy.min_scalar_type` for more details.

At the core of every ufunc is a one-dimensional strided loop that implements the actual function for a specific type combination. When a ufunc is created, it is given a static list of inner loops and a corresponding list of type signatures over which the ufunc operates. The ufunc machinery uses this list to determine which inner loop to use for a particular case. You can inspect the `.types` attribute for a particular ufunc to see which type combinations have a defined inner loop and which output type they produce (character codes are used in said output for brevity).

Casting must be done on one or more of the inputs whenever the ufunc does not have a core loop implementation for the input types provided. If an implementation for the input types cannot be found, then the algorithm searches for an implementation with a type signature to which all of the inputs can be cast “safely.” The first one it finds in its internal list of loops is selected and performed, after all necessary type casting. Recall that internal copies during ufuncs (even for casting) are limited to the size of an internal buffer (which is user settable).

Note: Universal functions in NumPy are flexible enough to have mixed type signatures. Thus, for example, a universal function could be defined that works with floating-point and integer values. See `numpy.1dexp` for an example.

By the above description, the casting rules are essentially implemented by the question of when a data type can be cast “safely” to another data type. The answer to this question can be determined in Python with a function call: `can_cast(fromtype, totype)`. The example below shows the results of this call for the 24 internally supported types on the author’s 64-bit system. You can generate this table for your system with the code given in the example.

Example

Code segment showing the “can cast safely” table for a 64-bit system. Generally the output depends on the system; your system might result in a different table.

```
>>> mark = {False: '-', True: 'Y'}
>>> def print_table(ntypes):
...     print('X ' + ' '.join(ntypes))
...     for row in ntypes:
...         print(row, end='')
...         for col in ntypes:
...             print(mark[np.can_cast(row, col)], end='')
...         print()
...
>>> print_table(np.typecodes['All'])
X ? b h i l q p B H I L Q P e f d g F D G S U V O M m
? Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y - Y
b - Y Y Y Y Y Y - - - - - Y Y Y Y Y Y Y Y Y Y - Y
h - - Y Y Y Y Y - - - - - - Y Y Y Y Y Y Y Y Y Y - Y
i - - - Y Y Y Y - - - - - - - Y Y - Y Y Y Y Y Y - Y
l - - - - Y Y Y - - - - - - - - Y Y - Y Y Y Y Y Y - Y
q - - - - Y Y Y - - - - - - - - Y Y - Y Y Y Y Y Y - Y
p - - - - Y Y Y - - - - - - - - Y Y - Y Y Y Y Y Y - Y
B - - Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y - Y
H - - - Y Y Y Y - Y Y Y Y Y - Y Y Y Y Y Y Y Y Y - Y
I - - - - Y Y Y - - Y Y Y Y - - Y Y - Y Y Y Y Y Y - Y
L - - - - - - - - - Y Y Y - - Y Y - Y Y Y Y Y Y - -
Q - - - - - - - - - Y Y Y - - Y Y - Y Y Y Y Y Y - -
P - - - - - - - - - Y Y Y - - Y Y - Y Y Y Y Y Y - -
e - - - - - - - - - - - - - Y Y Y Y Y Y Y Y Y Y - -
f - - - - - - - - - - - - - Y Y Y Y Y Y Y Y Y Y - -
d - - - - - - - - - - - - - Y Y - Y Y Y Y Y Y - -
g - - - - - - - - - - - - - Y - - Y Y Y Y Y Y - -
F - - - - - - - - - - - - - - - Y Y Y Y Y Y Y - -
D - - - - - - - - - - - - - - - Y Y Y Y Y Y Y - -
G - - - - - - - - - - - - - - - Y Y Y Y Y Y - -
S - - - - - - - - - - - - - - - Y Y Y Y - -
U - - - - - - - - - - - - - - - Y Y Y - -
V - - - - - - - - - - - - - - - Y Y - -
O - - - - - - - - - - - - - - - Y - -
M - - - - - - - - - - - - - - - Y Y Y -
m - - - - - - - - - - - - - - - Y Y - Y
```

You should note that, while included in the table for completeness, the ‘S’, ‘U’, and ‘V’ types cannot be operated on by ufuncs. Also, note that on a 32-bit system the integer types may have different sizes, resulting in a slightly altered table.

Mixed scalar-array operations use a different set of casting rules that ensure that a scalar cannot “upcast” an array unless the scalar is of a fundamentally different kind of data (i.e., under a different hierarchy in the data-type hierarchy) than

the array. This rule enables you to use scalar constants in your code (which, as Python types, are interpreted accordingly in ufuncs) without worrying about whether the precision of the scalar constant will cause upcasting on your large (small precision) array.

4.10.5 Use of internal buffers

Internally, buffers are used for misaligned data, swapped data, and data that has to be converted from one data type to another. The size of internal buffers is settable on a per-thread basis. There can be up to $2(n_{\text{inputs}} + n_{\text{outputs}})$ buffers of the specified size created to handle the data from all the inputs and outputs of a ufunc. The default size of a buffer is 10,000 elements. Whenever buffer-based calculation would be needed, but all input arrays are smaller than the buffer size, those misbehaved or incorrectly-typed arrays will be copied before the calculation proceeds. Adjusting the size of the buffer may therefore alter the speed at which ufunc calculations of various sorts are completed. A simple interface for setting this variable is accessible using the function `numpy.setbufsize`.

4.10.6 Error handling

Universal functions can trip special floating-point status registers in your hardware (such as divide-by-zero). If available on your platform, these registers will be regularly checked during calculation. Error handling is controlled on a per-thread basis, and can be configured using the functions `numpy.seterr` and `numpy.seterrcall`.

4.10.7 Overriding ufunc behavior

Classes (including ndarray subclasses) can override how ufuncs act on them by defining certain special methods. For details, see `arrays.classes`.

4.11 Copies and views

When operating on NumPy arrays, it is possible to access the internal data buffer directly using a *view* without copying data around. This ensures good performance but can also cause unwanted problems if the user is not aware of how this works. Hence, it is important to know the difference between these two terms and to know which operations return copies and which return views.

The NumPy array is a data structure consisting of two parts: the *contiguous* data buffer with the actual data elements and the metadata that contains information about the data buffer. The metadata includes data type, strides, and other important information that helps manipulate the ndarray easily. See the *Internal organization of NumPy arrays* section for a detailed look.

4.11.1 View

It is possible to access the array differently by just changing certain metadata like *stride* and *dtype* without changing the data buffer. This creates a new way of looking at the data and these new arrays are called views. The data buffer remains the same, so any changes made to a view reflects in the original copy. A view can be forced through the `ndarray.view` method.

4.11.2 Copy

When a new array is created by duplicating the data buffer as well as the metadata, it is called a copy. Changes made to the copy do not reflect on the original array. Making a copy is slower and memory-consuming but sometimes necessary. A copy can be forced by using `ndarray.copy`.

4.11.3 Indexing operations

See also:

Indexing on ndarrays

Views are created when elements can be addressed with offsets and strides in the original array. Hence, basic indexing always creates views. For example:

```
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> y = x[1:3] # creates a view
>>> y
array([1, 2])
>>> x[1:3] = [10, 11]
>>> x
array([ 0, 10, 11,  3,  4,  5,  6,  7,  8,  9])
>>> y
array([10, 11])
```

Here, `y` gets changed when `x` is changed because it is a view.

Advanced indexing, on the other hand, always creates copies. For example:

```
>>> x = np.arange(9).reshape(3, 3)
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> y = x[[1, 2]]
>>> y
array([[3, 4, 5],
       [6, 7, 8]])
>>> y.base is None
True
```

Here, `y` is a copy, as signified by the `base` attribute. We can also confirm this by assigning new values to `x[[1, 2]]` which in turn will not affect `y` at all:

```
>>> x[[1, 2]] = [[10, 11, 12], [13, 14, 15]]
>>> x
array([[ 0,  1,  2],
       [10, 11, 12],
       [13, 14, 15]])
>>> y
array([[3, 4, 5],
       [6, 7, 8]])
```

It must be noted here that during the assignment of `x[[1, 2]]` no view or copy is created as the assignment happens in-place.

4.11.4 Other operations

The `numpy.reshape` function creates a view where possible or a copy otherwise. In most cases, the strides can be modified to reshape the array with a view. However, in some cases where the array becomes non-contiguous (perhaps after a `ndarray.transpose` operation), the reshaping cannot be done by modifying strides and requires a copy. In these cases, we can raise an error by assigning the new shape to the shape attribute of the array. For example:

```
>>> x = np.ones((2, 3))
>>> y = x.T # makes the array non-contiguous
>>> y
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
>>> z = y.view()
>>> z.shape = 6
Traceback (most recent call last):
...
AttributeError: Incompatible shape for in-place modification. Use
`.reshape()` to make a copy with the desired shape.
```

Taking the example of another operation, `ravel` returns a contiguous flattened view of the array wherever possible. On the other hand, `ndarray.flatten` always returns a flattened copy of the array. However, to guarantee a view in most cases, `x.reshape(-1)` may be preferable.

4.11.5 How to tell if the array is a view or a copy

The `base` attribute of the `ndarray` makes it easy to tell if an array is a view or a copy. The `base` attribute of a view returns the original array while it returns `None` for a copy.

```
>>> x = np.arange(9)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> y = x.reshape(3, 3)
>>> y
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> y.base # .reshape() creates a view
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> z = y[[2, 1]]
>>> z
array([[6, 7, 8],
       [3, 4, 5]])
>>> z.base is None # advanced indexing creates a copy
True
```

Note that the `base` attribute should not be used to determine if an `ndarray` object is *new*; only if it is a view or a copy of another `ndarray`.

4.12 Interoperability with NumPy

NumPy's `ndarray` objects provide both a high-level API for operations on array-structured data and a concrete implementation of the API based on strided in-RAM storage. While this API is powerful and fairly general, its concrete implementation has limitations. As datasets grow and NumPy becomes used in a variety of new environments and architectures, there are cases where the strided in-RAM storage strategy is inappropriate, which has caused different libraries to reimplement this API for their own uses. This includes GPU arrays (`CuPy`), Sparse arrays (`scipy.sparse`, `PyData/Sparse`) and parallel arrays (`Dask` arrays) as well as various NumPy-like implementations in deep learning frameworks, like `TensorFlow` and `PyTorch`. Similarly, there are many projects that build on top of the NumPy API for labeled and indexed arrays (`XArray`), automatic differentiation (`JAX`), masked arrays (`numpy.ma`), physical units (`astropy.units`, `pint`, `unyt`), among others that add additional functionality on top of the NumPy API.

Yet, users still want to work with these arrays using the familiar NumPy API and re-use existing code with minimal (ideally zero) porting overhead. With this goal in mind, various protocols are defined for implementations of multi-dimensional arrays with high-level APIs matching NumPy.

Broadly speaking, there are three groups of features used for interoperability with NumPy:

1. Methods of turning a foreign object into an `ndarray`;
2. Methods of deferring execution from a NumPy function to another array library;
3. Methods that use NumPy functions and return an instance of a foreign object.

We describe these features below.

4.12.1 1. Using arbitrary objects in NumPy

The first set of interoperability features from the NumPy API allows foreign objects to be treated as NumPy arrays whenever possible. When NumPy functions encounter a foreign object, they will try (in order):

1. The buffer protocol, described [in the Python C-API documentation](#).
2. The `__array_interface__` protocol, described in this page. A precursor to Python's buffer protocol, it defines a way to access the contents of a NumPy array from other C extensions.
3. The `__array__()` method, which asks an arbitrary object to convert itself into an array.

For both the buffer and the `__array_interface__` protocols, the object describes its memory layout and NumPy does everything else (zero-copy if possible). If that's not possible, the object itself is responsible for returning a `ndarray` from `__array__()`.

`DLPack` is yet another protocol to convert foreign objects to NumPy arrays in a language and device agnostic manner. NumPy doesn't implicitly convert objects to `ndarrays` using `DLPack`. It provides the function `numpy.from_dlpack` that accepts any object implementing the `__dlpack__` method and outputs a NumPy `ndarray` (which is generally a view of the input object's data buffer). The [Python Specification for DLPack](#) page explains the `__dlpack__` protocol in detail.

The array interface protocol

The array interface protocol defines a way for array-like objects to re-use each other's data buffers. Its implementation relies on the existence of the following attributes or methods:

- `__array_interface__`: a Python dictionary containing the shape, the element type, and optionally, the data buffer address and the strides of an array-like object;
- `__array__()`: a method returning the NumPy ndarray view of an array-like object;

The `__array_interface__` attribute can be inspected directly:

```
>>> import numpy as np
>>> x = np.array([1, 2, 5.0, 8])
>>> x.__array_interface__
{'data': (94708397920832, False), 'strides': None, 'descr': [('', '<f8')], 'typestr':
↳ '<f8', 'shape': (4,), 'version': 3}
```

The `__array_interface__` attribute can also be used to manipulate the object data in place:

```
>>> class wrapper():
...     pass
...
>>> arr = np.array([1, 2, 3, 4])
>>> buf = arr.__array_interface__
>>> buf
{'data': (140497590272032, False), 'strides': None, 'descr': [('', '<i8')], 'typestr':
↳ '<i8', 'shape': (4,), 'version': 3}
>>> buf['shape'] = (2, 2)
>>> w = wrapper()
>>> w.__array_interface__ = buf
>>> new_arr = np.array(w, copy=False)
>>> new_arr
array([[1, 2],
       [3, 4]])
```

We can check that `arr` and `new_arr` share the same data buffer:

```
>>> new_arr[0, 0] = 1000
>>> new_arr
array([[1000, 2],
       [ 3, 4]])
>>> arr
array([1000, 2, 3, 4])
```

The `__array__()` method

The `__array__()` method ensures that any NumPy-like object (an array, any object exposing the array interface, an object whose `__array__()` method returns an array or any nested sequence) that implements it can be used as a NumPy array. If possible, this will mean using `__array__()` to create a NumPy ndarray view of the array-like object. Otherwise, this copies the data into a new ndarray object. This is not optimal, as coercing arrays into ndarrays may cause performance problems or create the need for copies and loss of metadata, as the original object and any attributes/behavior it may have had, is lost.

To see an example of a custom array implementation including the use of `__array__()`, see [Writing custom array containers](#).

The DLPack Protocol

The **DLPack** protocol defines a memory-layout of strided n-dimensional array objects. It offers the following syntax for data exchange:

1. A `numpy.from_dlpack` function, which accepts (array) objects with a `__dlpack__` method and uses that method to construct a new array containing the data from `x`.
2. `__dlpack__(self, stream=None)` and `__dlpack_device__` methods on the array object, which will be called from within `from_dlpack`, to query what device the array is on (may be needed to pass in the correct stream, e.g. in the case of multiple GPUs) and to access the data.

Unlike the buffer protocol, DLPack allows exchanging arrays containing data on devices other than the CPU (e.g. Vulkan or GPU). Since NumPy only supports CPU, it can only convert objects whose data exists on the CPU. But other libraries, like **PyTorch** and **CuPy**, may exchange data on GPU using this protocol.

4.12.2 2. Operating on foreign objects without converting

A second set of methods defined by the NumPy API allows us to defer the execution from a NumPy function to another array library.

Consider the following function.

```
>>> import numpy as np
>>> def f(x):
...     return np.mean(np.exp(x))
```

Note that `np.exp` is a *ufunc*, which means that it operates on ndarrays in an element-by-element fashion. On the other hand, `np.mean` operates along one of the array's axes.

We can apply `f` to a NumPy ndarray object directly:

```
>>> x = np.array([1, 2, 3, 4])
>>> f(x)
21.1977562209304
```

We would like this function to work equally well with any NumPy-like array object.

NumPy allows a class to indicate that it would like to handle computations in a custom-defined way through the following interfaces:

- `__array_ufunc__`: allows third-party objects to support and override *ufuncs*.
- `__array_function__`: a catch-all for NumPy functionality that is not covered by the `__array_ufunc__` protocol for universal functions.

As long as foreign objects implement the `__array_ufunc__` or `__array_function__` protocols, it is possible to operate on them without the need for explicit conversion.

The `__array_ufunc__` protocol

A *universal function* (or *ufunc* for short) is a “vectorized” wrapper for a function that takes a fixed number of specific inputs and produces a fixed number of specific outputs. The output of the ufunc (and its methods) is not necessarily a ndarray, if not all input arguments are ndarrays. Indeed, if any input defines an `__array_ufunc__` method, control will be passed completely to that function, i.e., the ufunc is overridden. The `__array_ufunc__` method defined on that (non-ndarray) object has access to the NumPy ufunc. Because ufuncs have a well-defined structure, the foreign `__array_ufunc__` method may rely on ufunc attributes like `.at()`, `.reduce()`, and others.

A subclass can override what happens when executing NumPy ufuncs on it by overriding the default `ndarray.__array_ufunc__` method. This method is executed instead of the ufunc and should return either the result of the operation, or `NotImplemented` if the operation requested is not implemented.

The `__array_function__` protocol

To achieve enough coverage of the NumPy API to support downstream projects, there is a need to go beyond `__array_ufunc__` and implement a protocol that allows arguments of a NumPy function to take control and divert execution to another function (for example, a GPU or parallel implementation) in a way that is safe and consistent across projects.

The semantics of `__array_function__` are very similar to `__array_ufunc__`, except the operation is specified by an arbitrary callable object rather than a ufunc instance and method. For more details, see [NEP 18 — A dispatch mechanism for NumPy’s high level array functions](#).

4.12.3 3. Returning foreign objects

A third type of feature set is meant to use the NumPy function implementation and then convert the return value back into an instance of the foreign object. The `__array_finalize__` and `__array_wrap__` methods act behind the scenes to ensure that the return type of a NumPy function can be specified as needed.

The `__array_finalize__` method is the mechanism that NumPy provides to allow subclasses to handle the various ways that new instances get created. This method is called whenever the system internally allocates a new array from an object which is a subclass (subtype) of the ndarray. It can be used to change attributes after construction, or to update meta-information from the “parent.”

The `__array_wrap__` method “wraps up the action” in the sense of allowing any object (such as user-defined functions) to set the type of its return value and update attributes and metadata. This can be seen as the opposite of the `__array__` method. At the end of every object that implements `__array_wrap__`, this method is called on the input object with the highest *array priority*, or the output object if one was specified. The `__array_priority__` attribute is used to determine what type of object to return in situations where there is more than one possibility for the Python type of the returned object. For example, subclasses may opt to use this method to transform the output array into an instance of the subclass and update metadata before returning the array to the user.

For more information on these methods, see [Subclassing ndarray](#) and [Specific features of ndarray sub-typing](#).

4.12.4 Interoperability examples

Example: Pandas Series objects

Consider the following:

```
>>> import pandas as pd
>>> ser = pd.Series([1, 2, 3, 4])
>>> type(ser)
pandas.core.series.Series
```

Now, `ser` is **not** a ndarray, but because it implements the `__array_ufunc__` protocol, we can apply ufuncs to it as if it were a ndarray:

```
>>> np.exp(ser)
0    2.718282
1    7.389056
2   20.085537
3   54.598150
dtype: float64
>>> np.sin(ser)
0    0.841471
1    0.909297
2    0.141120
3   -0.756802
dtype: float64
```

We can even do operations with other ndarrays:

```
>>> np.add(ser, np.array([5, 6, 7, 8]))
0     6
1     8
2    10
3    12
dtype: int64
>>> f(ser)
21.1977562209304
>>> result = ser.__array__()
>>> type(result)
numpy.ndarray
```

Example: PyTorch tensors

PyTorch is an optimized tensor library for deep learning using GPUs and CPUs. PyTorch arrays are commonly called *tensors*. Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs or other hardware accelerators. In fact, tensors and NumPy arrays can often share the same underlying memory, eliminating the need to copy data.

```
>>> import torch
>>> data = [[1, 2], [3, 4]]
>>> x_np = np.array(data)
>>> x_tensor = torch.tensor(data)
```

Note that `x_np` and `x_tensor` are different kinds of objects:

```
>>> x_np
array([[1, 2],
       [3, 4]])
>>> x_tensor
tensor([[1, 2],
        [3, 4]])
```

However, we can treat PyTorch tensors as NumPy arrays without the need for explicit conversion:

```
>>> np.exp(x_tensor)
tensor([[ 2.7183,  7.3891],
        [20.0855, 54.5982]], dtype=torch.float64)
```

Also, note that the return type of this function is compatible with the initial data type.

Warning

While this mixing of ndarrays and tensors may be convenient, it is not recommended. It will not work for non-CPU tensors, and will have unexpected behavior in corner cases. Users should prefer explicitly converting the ndarray to a tensor.

Note: PyTorch does not implement `__array_function__` or `__array_ufunc__`. Under the hood, the `Tensor.__array__()` method returns a NumPy ndarray as a view of the tensor data buffer. See [this issue](#) and the [__torch_function__ implementation](#) for details.

Note also that we can see `__array_wrap__` in action here, even though `torch.Tensor` is not a subclass of ndarray:

```
>>> import torch
>>> t = torch.arange(4)
>>> np.abs(t)
tensor([0, 1, 2, 3])
```

PyTorch implements `__array_wrap__` to be able to get tensors back from NumPy functions, and we can modify it directly to control which type of objects are returned from these functions.

Example: CuPy arrays

CuPy is a NumPy/SciPy-compatible array library for GPU-accelerated computing with Python. CuPy implements a subset of the NumPy interface by implementing `cupy.ndarray`, a counterpart to NumPy ndarrays.

```
>>> import cupy as cp
>>> x_gpu = cp.array([1, 2, 3, 4])
```

The `cupy.ndarray` object implements the `__array_ufunc__` interface. This enables NumPy ufuncs to be applied to CuPy arrays (this will defer operation to the matching CuPy CUDA/ROCm implementation of the ufunc):

```
>>> np.mean(np.exp(x_gpu))
array(21.19775622)
```

Note that the return type of these operations is still consistent with the initial type:

```
>>> arr = cp.random.randn(1, 2, 3, 4).astype(cp.float32)
>>> result = np.sum(arr)
>>> print(type(result))
<class 'cupy._core.core.ndarray'>
```

See [this page in the CuPy documentation](#) for details.

`cupy.ndarray` also implements the `__array_function__` interface, meaning it is possible to do operations such as

```
>>> a = np.random.randn(100, 100)
>>> a_gpu = cp.asarray(a)
>>> qr_gpu = np.linalg.qr(a_gpu)
```

CuPy implements many NumPy functions on `cupy.ndarray` objects, but not all. See [the CuPy documentation](#) for details.

Example: Dask arrays

Dask is a flexible library for parallel computing in Python. Dask Array implements a subset of the NumPy ndarray interface using blocked algorithms, cutting up the large array into many small arrays. This allows computations on larger-than-memory arrays using multiple cores.

Dask supports `__array__()` and `__array_ufunc__`.

```
>>> import dask.array as da
>>> x = da.random.normal(1, 0.1, size=(20, 20), chunks=(10, 10))
>>> np.mean(np.exp(x))
dask.array<mean_agg-aggregate, shape=(), dtype=float64, chunksize=(), chunktype=numpy.
↳ndarray>
>>> np.mean(np.exp(x)).compute()
5.090097550553843
```

Note: Dask is lazily evaluated, and the result from a computation isn't computed until you ask for it by invoking `compute()`.

See [the Dask array documentation](#) and [the scope of Dask arrays interoperability with NumPy arrays](#) for details.

Example: DLPack

Several Python data science libraries implement the `__dlpack__` protocol. Among them are [PyTorch](#) and [CuPy](#). A full list of libraries that implement this protocol can be found on [this page of DLPack documentation](#).

Convert a PyTorch CPU tensor to NumPy array:

```
>>> import torch
>>> x_torch = torch.arange(5)
>>> x_torch
tensor([0, 1, 2, 3, 4])
>>> x_np = np.from_dlpack(x_torch)
>>> x_np
array([0, 1, 2, 3, 4])
>>> # note that x_np is a view of x_torch
```

(continues on next page)

(continued from previous page)

```
>>> x_torch[1] = 100
>>> x_torch
tensor([ 0, 100,  2,  3,  4])
>>> x_np
array([ 0, 100,  2,  3,  4])
```

The imported arrays are read-only so writing or operating in-place will fail:

```
>>> x.flags.writeable
False
>>> x_np[1] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: assignment destination is read-only
```

A copy must be created in order to operate on the imported arrays in-place, but will mean duplicating the memory. Do not do this for very large arrays:

```
>>> x_np_copy = x_np.copy()
>>> x_np_copy.sort() # works
```

Note: Note that GPU tensors can't be converted to NumPy arrays since NumPy doesn't support GPU devices:

```
>>> x_torch = torch.arange(5, device='cuda')
>>> np.from_dlpack(x_torch)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: Unsupported device in DLTensor.
```

But, if both libraries support the device the data buffer is on, it is possible to use the `__dlpack__` protocol (e.g. [PyTorch](#) and [CuPy](#)):

```
>>> x_torch = torch.arange(5, device='cuda')
>>> x_cupy = cupy.from_dlpack(x_torch)
```

Similarly, a NumPy array can be converted to a PyTorch tensor:

```
>>> x_np = np.arange(5)
>>> x_torch = torch.from_dlpack(x_np)
```

Read-only arrays cannot be exported:

```
>>> x_np = np.arange(5)
>>> x_np.flags.writeable = False
>>> torch.from_dlpack(x_np)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../site-packages/torch/utils/dlpack.py", line 63, in from_dlpack
    dlpack = ext_tensor.__dlpack__()
TypeError: NumPy currently only supports dlpack for writeable arrays
```

4.12.5 Further reading

- [arrays.interface](#)
- [Writing custom array containers](#)
- [special-attributes-and-methods](#) (details on the `__array_ufunc__` and `__array_function__` protocols)
- [Subclassing ndarray](#) (details on the `__array_wrap__` and `__array_finalize__` methods)
- [Specific features of ndarray sub-typing](#) (more details on the implementation of `__array_finalize__`, `__array_wrap__` and `__array_priority__`)
- [NumPy roadmap: interoperability](#)
- [PyTorch documentation on the Bridge with NumPy](#)

MISCELLANEOUS

5.1 IEEE 754 Floating Point Special Values

Special values defined in numpy: nan, inf,

NaNs can be used as a poor-man's mask (if you don't care what the original value was)

Note: cannot use equality to test NaNs. E.g.:

```
>>> myarr = np.array([1., 0., np.nan, 3.])
>>> np.nonzero(myarr == np.nan)
(array([], dtype=int64),)
>>> np.nan == np.nan  # is always False! Use special numpy functions instead.
False
>>> myarr[myarr == np.nan] = 0. # doesn't work
>>> myarr
array([ 1.,  0., nan,  3.])
>>> myarr[np.isnan(myarr)] = 0. # use this instead find
>>> myarr
array([1.,  0.,  0.,  3.])
```

Other related special value functions:

```
isinf():      True if value is inf
isfinite():   True if not nan or inf
nan_to_num(): Map nan to 0, inf to max float, -inf to min float
```

The following corresponds to the usual functions except that nans are excluded from the results:

```
nansum()
nanmax()
nanmin()
nanargmax()
nanargmin()

>>> x = np.arange(10.)
>>> x[3] = np.nan
>>> x.sum()
nan
>>> np.nansum(x)
42.0
```