
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
```

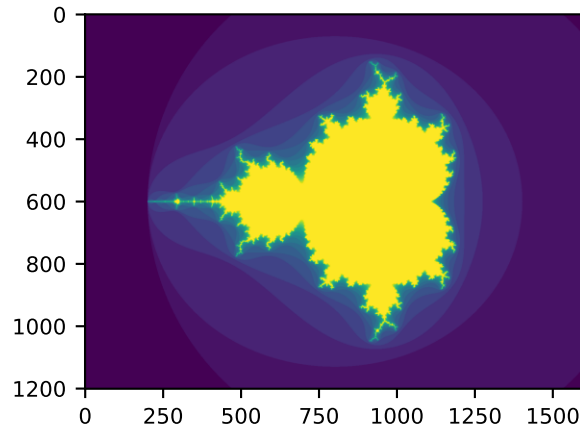
(continues on next page)

(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

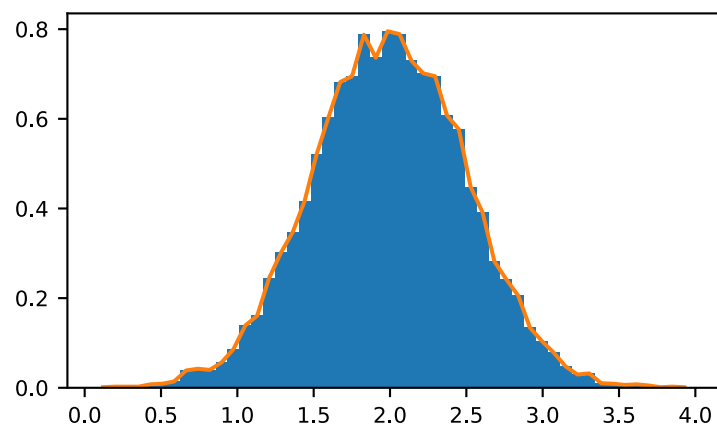
The NumPy histogram function applied to an array returns a pair of vectors: the histogram of the array and a vector of the bin edges. Beware: `matplotlib` also has a function to build histograms (called `hist`, as in Matlab) that differs from the one in NumPy. The main difference is that `pylab.hist`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

(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](#).

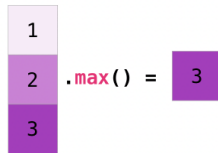
3.17 More useful array operations

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

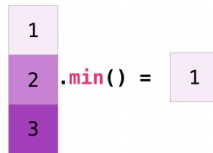
NumPy also performs aggregation functions. In addition to `min`, `max`, and `sum`, you can easily run `mean` to get the average, `prod` to get the result of multiplying the elements together, `std` to get the standard deviation, and more.

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

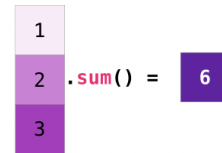
data



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

You can aggregate matrices the same way you aggregated vectors:

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

data			
1	2		
3	4	.max()	6
5	6		

data			
1	2		
3	4	.min()	1
5	6		

data			
1	2		
3	4	.sum()	21
5	6		

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]])
```

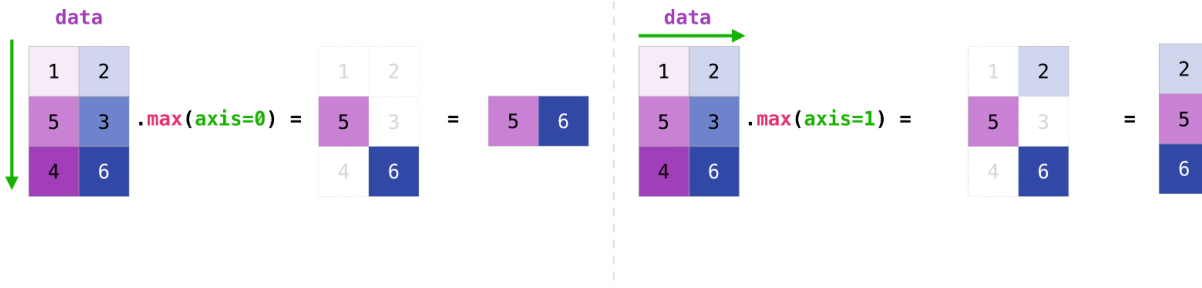
(continues on next page)

(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]])

```

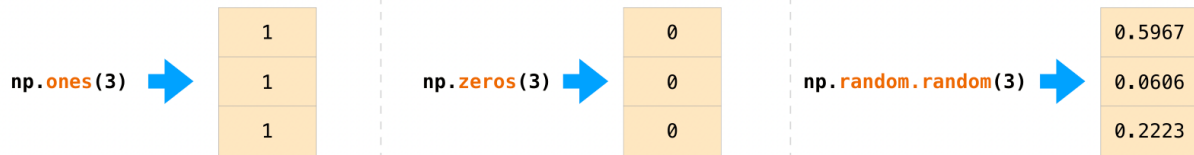
$$\text{data} + \text{ones_row} = \begin{array}{|c|c|} \hline \text{data} & \\ \hline 1 & 2 \\ \hline 3 & 4 \\ \hline 5 & 6 \\ \hline \end{array} + \begin{array}{|c|c|} \hline \text{ones_row} & \\ \hline 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|} \hline \text{data} & \text{ones_row} \\ \hline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

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.]])
```

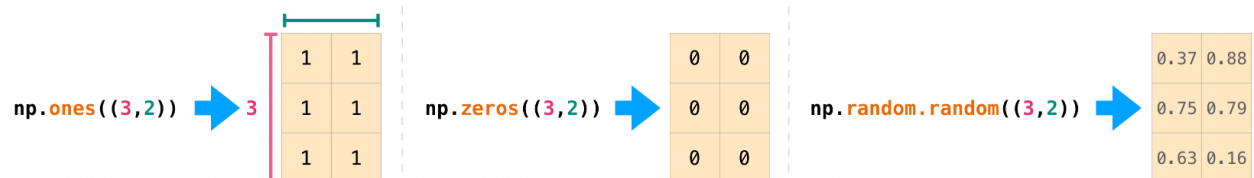
There are often instances where we want NumPy to initialize the values of an array. NumPy offers functions like `ones()` and `zeros()`, and the `random.Generator` class for random number generation for that. All you need to do is pass in the number of elements you want it to generate:

```
>>> 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

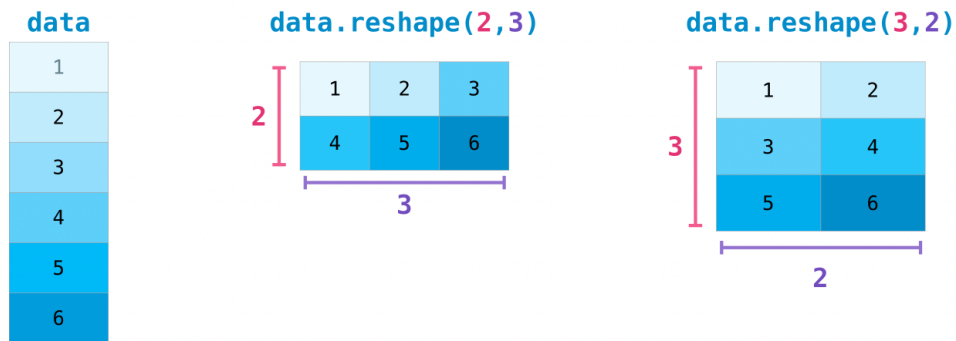
This section covers `arr.reshape()`, `arr.transpose()`, `arr.T`

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):

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

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]]

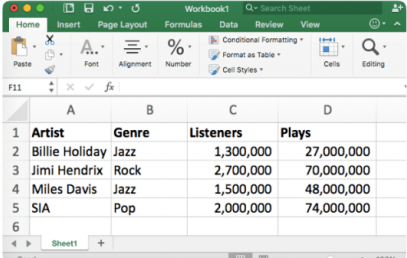
>>> # You can also simply select the columns you need:
>>> x = pd.read_csv('music.csv', usecols=['Artist', 'Plays']).values
>>> print(x)
[['Billie Holiday' 27000000]
 ['Jimmie Hendrix' 70000000]]
```

(continues on next page)

(continued from previous page)

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

music.csv



pandas.read_csv('music.csv')

	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

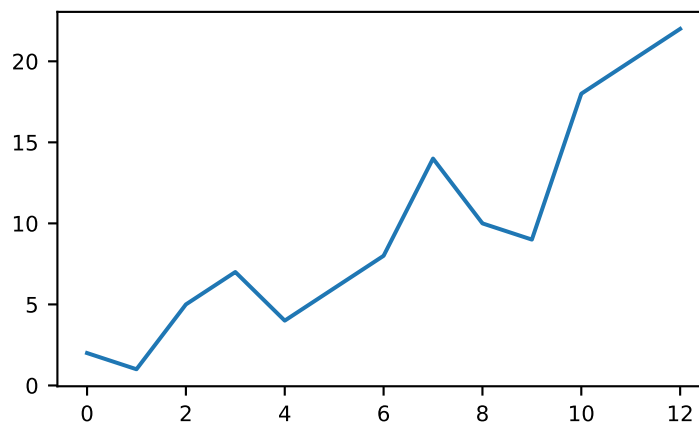
# If you're using Jupyter Notebook, you may also want to run the following
# line of code to display your code in the notebook:

%matplotlib inline
```

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

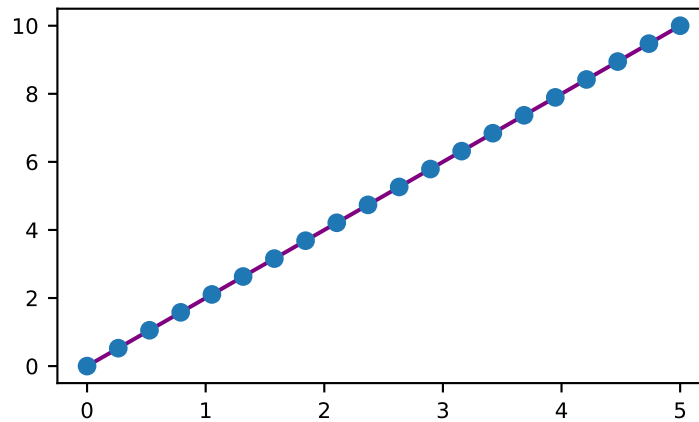
```
>>> plt.plot(a)

# 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
```



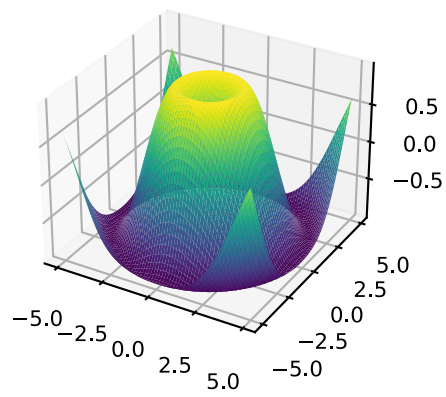
With Matplotlib, you have access to an enormous number of visualization options.

```
>>> 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 $i, i + k, \dots, i + (m - 1)k$ 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.

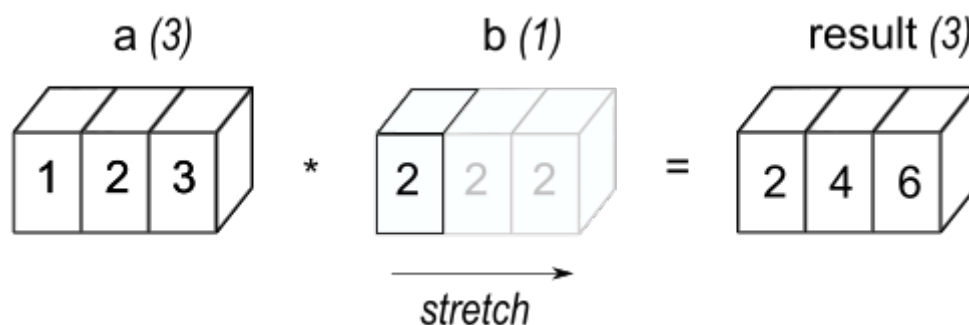


Fig. 1: *Figure 1*

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.

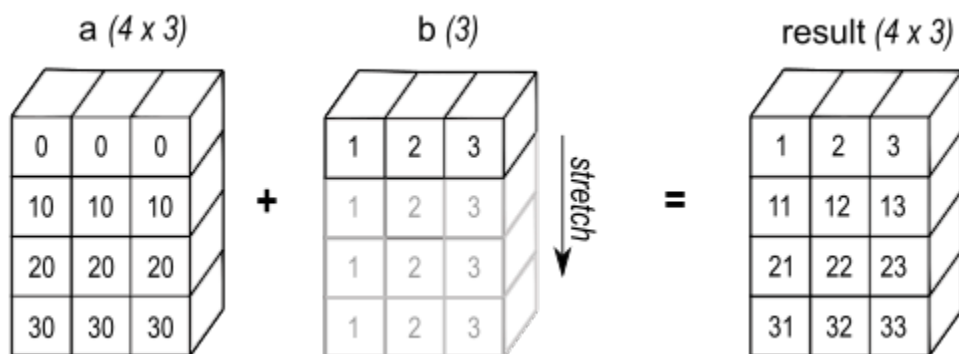


Fig. 2: Figure 2

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.

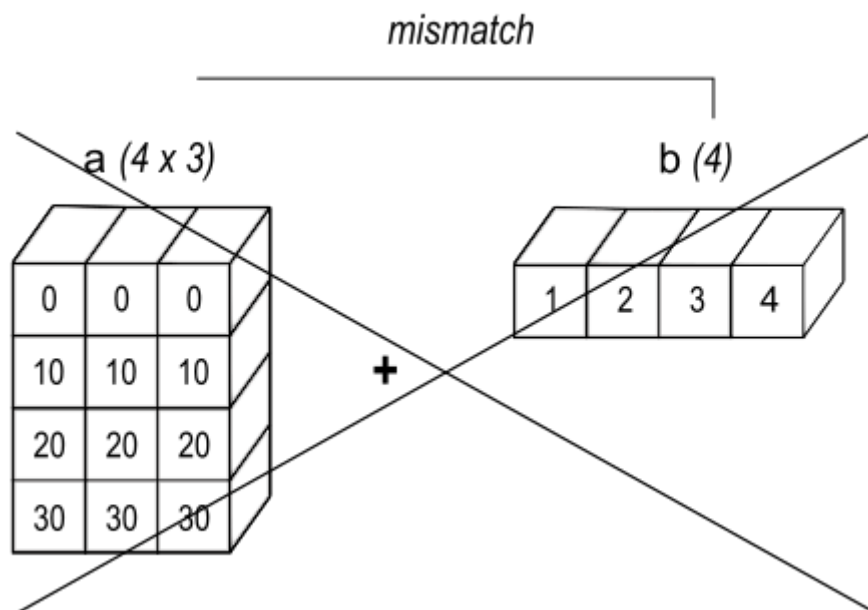


Fig. 3: Figure 3

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.

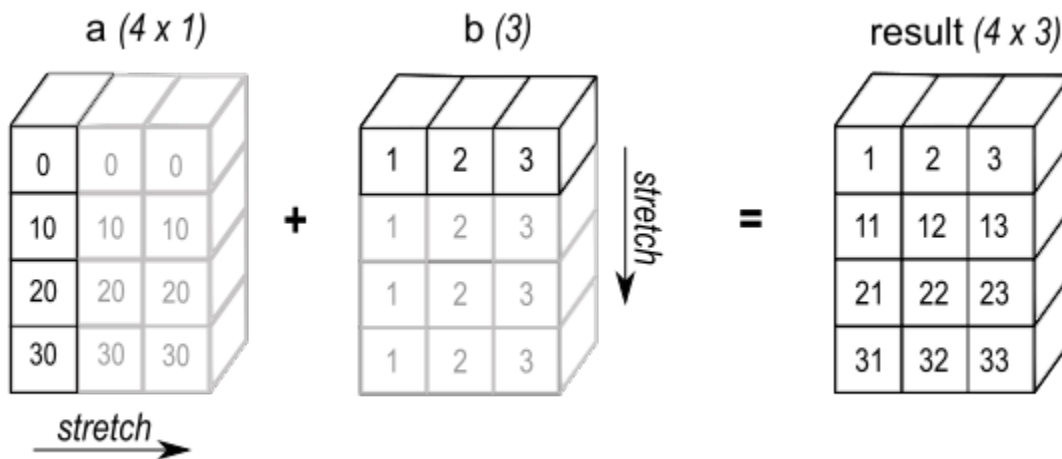


Fig. 4: Figure 4

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.

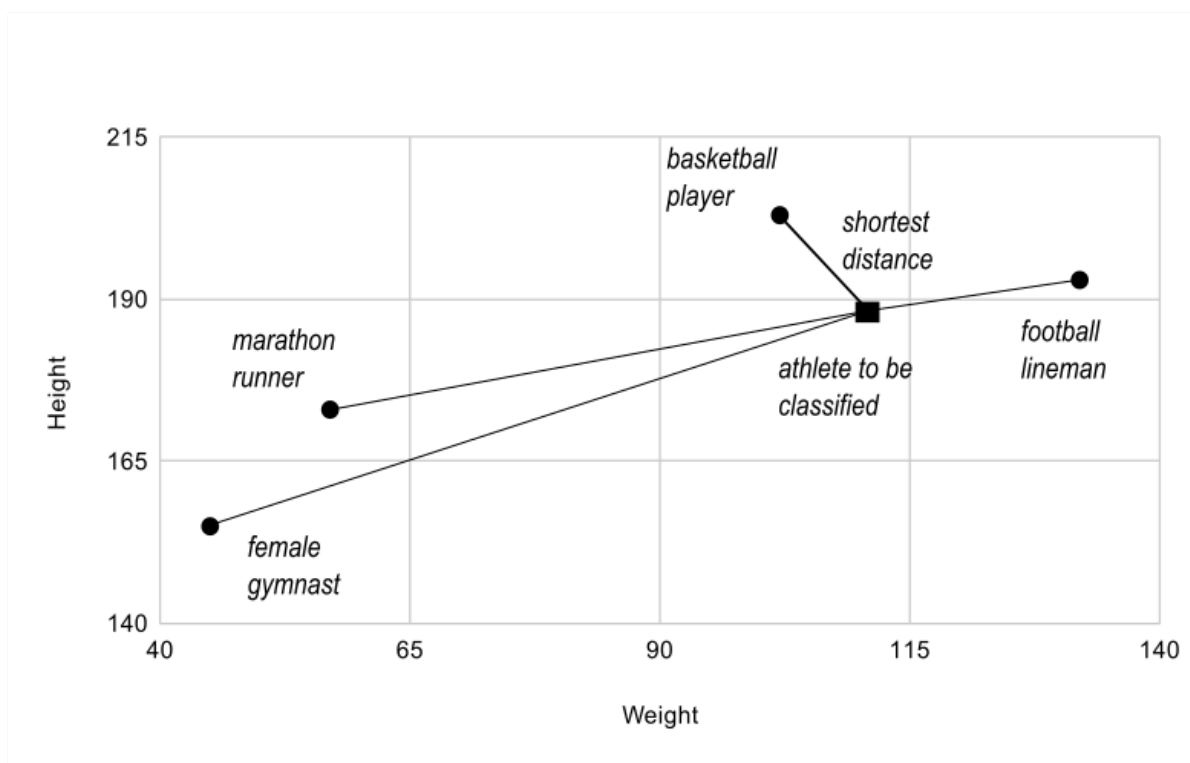


Fig. 5: Figure 5

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 Y - Y
h - - 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
l - - - - Y 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 - Y
p - - - - 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 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 Y - Y
I - - - - Y 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 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 - -
e - - - - - - - - - - - - - Y Y Y Y Y Y Y Y Y Y Y - -
f - - - - - - - - - - - - - Y Y Y Y Y Y Y Y Y Y Y - -
d - - - - - - - - - - - - - Y Y - Y Y Y Y Y Y Y - -
g - - - - - - - - - - - - - Y - - Y Y Y Y Y Y - -
F - - - - - - - - - - - - - - - Y Y Y Y Y Y Y Y - -
D - - - - - - - - - - - - - - - Y 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
```

5.2 How numpy handles numerical exceptions

The default is to 'warn' for invalid, divide, and overflow and 'ignore' for underflow. But this can be changed, and it can be set individually for different kinds of exceptions. The different behaviors are:

- 'ignore' : Take no action when the exception occurs.
- 'warn' : Print a *RuntimeWarning* (via the Python `warnings` module).
- 'raise' : Raise a *FloatingPointError*.
- 'call' : Call a function specified using the *seterrcall* function.
- 'print' : Print a warning directly to `stdout`.
- 'log' : Record error in a Log object specified by *seterrcall*.

These behaviors can be set for all kinds of errors or specific ones:

- all : apply to all numeric exceptions
- invalid : when NaNs are generated
- divide : divide by zero (for integers as well!)
- overflow : floating point overflows
- underflow : floating point underflows

Note that integer divide-by-zero is handled by the same machinery. These behaviors are set on a per-thread basis.

5.3 Examples

```
>>> oldsettings = np.seterr(all='warn')
>>> np.zeros(5, dtype=np.float32)/0.
Traceback (most recent call last):
...
RuntimeWarning: invalid value encountered in divide
>>> j = np.seterr(under='ignore')
>>> np.array([1.e-100])**10
array([0.])
>>> j = np.seterr(invalid='raise')
>>> np.sqrt(np.array([-1.]))
Traceback (most recent call last):
...
FloatingPointError: invalid value encountered in sqrt
>>> def errorhandler(errstr, errflag):
...     print("saw stupid error!")
>>> np.seterrcall(errorhandler)
>>> j = np.seterr(all='call')
>>> np.zeros(5, dtype=np.int32)/0
saw stupid error!
array([nan, nan, nan, nan, nan])
>>> j = np.seterr(**oldsettings) # restore previous
...                             # error-handling settings
```

5.4 Interfacing to C

Only a survey of the choices. Little detail on how each works.

1) Bare metal, wrap your own C-code manually.

- Plusses:
 - Efficient
 - No dependencies on other tools
- Minuses:
 - Lots of learning overhead:
 - * need to learn basics of Python C API
 - * need to learn basics of numpy C API
 - * need to learn how to handle reference counting and love it.
 - Reference counting often difficult to get right.
 - * getting it wrong leads to memory leaks, and worse, segfaults

2) Cython

- Plusses:
 - avoid learning C API's
 - no dealing with reference counting
 - can code in pseudo python and generate C code
 - can also interface to existing C code
 - should shield you from changes to Python C api
 - has become the de-facto standard within the scientific Python community
 - fast indexing support for arrays
- Minuses:
 - Can write code in non-standard form which may become obsolete
 - Not as flexible as manual wrapping

3) ctypes

- Plusses:
 - part of Python standard library
 - good for interfacing to existing shareable libraries, particularly Windows DLLs
 - avoids API/reference counting issues
 - good numpy support: arrays have all these in their ctypes attribute:

```
a.ctypes.data
a.ctypes.data_as
a.ctypes.shape
a.ctypes.shape_as
a.ctypes.strides
a.ctypes.strides_as
```

- Minuses:
 - can't use for writing code to be turned into C extensions, only a wrapper tool.

4) SWIG (automatic wrapper generator)

- Plusses:
 - around a long time
 - multiple scripting language support
 - C++ support
 - Good for wrapping large (many functions) existing C libraries
- Minuses:
 - generates lots of code between Python and the C code
 - can cause performance problems that are nearly impossible to optimize out
 - interface files can be hard to write
 - doesn't necessarily avoid reference counting issues or needing to know API's

5) Psyco

- Plusses:
 - Turns pure python into efficient machine code through jit-like optimizations
 - very fast when it optimizes well
- Minuses:
 - Only on intel (windows?)
 - Doesn't do much for numpy?

5.5 Interfacing to Fortran:

The clear choice to wrap Fortran code is `f2py`.

Pyfort is an older alternative, but not supported any longer. Fwrap is a newer project that looked promising but isn't being developed any longer.

5.6 Interfacing to C++:

- 1) Cython
- 2) CXX
- 3) Boost.python
- 4) SWIG
- 5) SIP (used mainly in PyQt)

NUMPY FOR MATLAB USERS

6.1 Introduction

MATLAB® and NumPy have a lot in common, but NumPy was created to work with Python, not to be a MATLAB clone. This guide will help MATLAB users get started with NumPy.

6.2 Some key differences

In MATLAB, the basic type, even for scalars, is a multidimensional array. Array assignments in MATLAB are stored as 2D arrays of double precision floating point numbers, unless you specify the number of dimensions and type. Operations on the 2D instances of these arrays are modeled on matrix operations in linear algebra.	In NumPy, the basic type is a multidimensional <code>array</code> . Array assignments in NumPy are usually stored as n-dimensional arrays with the minimum type required to hold the objects in sequence, unless you specify the number of dimensions and type. NumPy performs operations element-by-element, so multiplying 2D arrays with <code>*</code> is not a matrix multiplication – it’s an element-by-element multiplication. (The <code>@</code> operator, available since Python 3.5, can be used for conventional matrix multiplication.)
MATLAB numbers indices from 1; <code>a(1)</code> is the first element. <i>See note INDEXING</i>	NumPy, like Python, numbers indices from 0; <code>a[0]</code> is the first element.
MATLAB’s scripting language was created for linear algebra so the syntax for some array manipulations is more compact than NumPy’s. On the other hand, the API for adding GUIs and creating full-fledged applications is more or less an afterthought.	NumPy is based on Python, a general-purpose language. The advantage to NumPy is access to Python libraries including: SciPy , Matplotlib , Pandas , OpenCV , and more. In addition, Python is often embedded as a scripting language in other software, allowing NumPy to be used there too.
MATLAB array slicing uses pass-by-value semantics, with a lazy copy-on-write scheme to prevent creating copies until they are needed. Slicing operations copy parts of the array.	NumPy array slicing uses pass-by-reference, that does not copy the arguments. Slicing operations are views into an array.

6.3 Rough equivalents

The table below gives rough equivalents for some common MATLAB expressions. These are similar expressions, not equivalents. For details, see the documentation.

In the table below, it is assumed that you have executed the following commands in Python:

```
import numpy as np
from scipy import io, integrate, linalg, signal
from scipy.sparse.linalg import eigs
```

Also assume below that if the Notes talk about “matrix” that the arguments are two-dimensional entities.

6.3.1 General purpose equivalents

MATLAB	NumPy	Notes
<code>help func</code>	<code>info(func)</code> or <code>help(func)</code> or <code>func?</code> (in IPython)	get help on the function <i>func</i>
<code>which func</code>	see note HELP	find out where <i>func</i> is defined
<code>type func</code>	<code>np.source(func)</code> or <code>func??</code> (in IPython)	print source for <i>func</i> (if not a native function)
<code>% comment</code>	<code># comment</code>	comment a line of code with the text <code>comment</code>
<pre>for i=1:3 fprintf('%i\n',i) end</pre>	<pre>for i in range(1, 4): print(i)</pre>	use a for-loop to print the numbers 1, 2, and 3 using <code>range</code>
<code>a && b</code>	<code>a and b</code>	short-circuiting logical AND operator (Python native operator); scalar arguments only
<code>a b</code>	<code>a or b</code>	short-circuiting logical OR operator (Python native operator); scalar arguments only
<pre>>> 4 == 4 ans = 1 >> 4 == 5 ans = 0</pre>	<pre>>>> 4 == 4 True >>> 4 == 5 False</pre>	The boolean objects in Python are <code>True</code> and <code>False</code> , as opposed to MATLAB logical types of 1 and 0.
<pre>a=4 if a==4 fprintf('a = 4\n') elseif a==5 fprintf('a = 5\n') end</pre>	<pre>a = 4 if a == 4: print('a = 4') elif a == 5: print('a = 5')</pre>	create an if-else statement to check if <code>a</code> is 4 or 5 and print result
<code>1*i, 1*j, 1i, 1j</code>	<code>1j</code>	complex numbers
<code>eps</code>	<code>np.finfo(float).eps</code> or <code>np.spacing(1)</code>	Upper bound to relative error due to rounding in 64-bit floating point arithmetic.
<code>load data.mat</code>	<code>io.loadmat('data.mat')</code>	Load MATLAB variables saved to the file <code>data.mat</code> . (Note: When saving arrays to <code>data.mat</code> in MATLAB/Octave, use a recent binary format. scipy.io.loadmat will create a dictionary with the saved arrays and further information.)
<code>ode45</code>	<code>integrate.solve_ivp(f)</code>	integrate an ODE with Runge-Kutta 4,5
<code>ode15s</code>	<code>integrate.solve_ivp(f, method='BDF')</code>	integrate an ODE with BDF method

6.3.2 Linear algebra equivalents

MATLAB	NumPy	Notes
<code>ndims(a)</code>	<code>np.ndim(a)</code> or <code>a.ndim</code>	number of dimensions of array <code>a</code>
<code>numel(a)</code>	<code>np.size(a)</code> or <code>a.size</code>	number of elements of array <code>a</code>
<code>size(a)</code>	<code>np.shape(a)</code> or <code>a.shape</code>	“size” of array <code>a</code>
<code>size(a,n)</code>	<code>a.shape[n-1]</code>	get the number of elements of the <code>n</code> -th dimension of array <code>a</code> . (Note that MATLAB uses 1 based indexing while Python uses 0 based indexing. See note INDEXING)
<code>[1 2 3; 4 5 6]</code>	<code>np.array([[1. ,2. ,3.],[4. ,5. ,6.]])</code>	define a 2x3 2D array
<code>[a b; c d]</code>	<code>np.block([[a, b], [c, d]])</code>	construct a matrix from blocks <code>a</code> , <code>b</code> , <code>c</code> , and <code>d</code>
<code>a(end)</code>	<code>a[-1]</code>	access last element in MATLAB vector (1xn or nx1) or 1D NumPy array <code>a</code> (length <code>n</code>)
<code>a(2,5)</code>	<code>a[1, 4]</code>	access element in second row, fifth column in 2D array <code>a</code>
<code>a(2,:)</code>	<code>a[1]</code> or <code>a[1, :]</code>	entire second row of 2D array <code>a</code>
<code>a(1:5,:)</code>	<code>a[0:5]</code> or <code>a[:5]</code> or <code>a[0:5, :]</code>	first 5 rows of 2D array <code>a</code>
<code>a(end-4:end,:)</code>	<code>a[-5:]</code>	last 5 rows of 2D array <code>a</code>
<code>a(1:3,5:9)</code>	<code>a[0:3, 4:9]</code>	The first through third rows and fifth through ninth columns of a 2D array, <code>a</code> .
<code>a([2,4,5],[1,3])</code>	<code>a[np.ix_([1, 3, 4], [0, 2])]</code>	rows 2,4 and 5 and columns 1 and 3. This allows the matrix to be modified, and doesn't require a regular slice.
<code>a(3:2:21,:)</code>	<code>a[2:21:2, :]</code>	every other row of <code>a</code> , starting with the third and going to the twenty-first
<code>a(1:2:end,:)</code>	<code>a[::2, :]</code>	every other row of <code>a</code> , starting with the first
<code>a(end:-1:1,:)</code> or <code>flipud(a)</code>	<code>a[::-1, :]</code>	<code>a</code> with rows in reverse order
<code>a([1:end 1],:)</code>	<code>a[np.r_[:len(a), 0]]</code>	<code>a</code> with copy of the first row appended to the end
<code>a.'</code>	<code>a.transpose()</code> or <code>a.T</code>	transpose of <code>a</code>
<code>a'</code>	<code>a.conj().transpose()</code> or <code>a.conj().T</code>	conjugate transpose of <code>a</code>
<code>a * b</code>	<code>a @ b</code>	matrix multiply
<code>a .* b</code>	<code>a * b</code>	element-wise multiply
<code>a ./ b</code>	<code>a / b</code>	element-wise divide
<code>a.^3</code>	<code>a**3</code>	element-wise exponentiation
<code>(a > 0.5)</code>	<code>(a > 0.5)</code>	matrix whose <code>i,j</code> th element is (<code>a_{ij} > 0.5</code>). The MATLAB result is an array of logical values 0 and 1. The NumPy result is an array of the boolean values <code>False</code> and <code>True</code> .
<code>find(a > 0.5)</code>	<code>np.nonzero(a > 0.5)</code>	find the indices where (<code>a > 0.5</code>)

continues on next page

Table 1 – continued from previous page

MATLAB	NumPy	Notes
<code>a(:, find(v > 0.5))</code>	<code>a[:, np.nonzero(v > 0.5)[0]]</code>	extract the columns of <code>a</code> where vector <code>v > 0.5</code>
<code>a(:, find(v>0.5))</code>	<code>a[:, v.T > 0.5]</code>	extract the columns of <code>a</code> where column vector <code>v > 0.5</code>
<code>a(a<0.5)=0</code>	<code>a[a < 0.5]=0</code>	<code>a</code> with elements less than 0.5 zeroed out
<code>a .* (a>0.5)</code>	<code>a * (a > 0.5)</code>	<code>a</code> with elements less than 0.5 zeroed out
<code>a(:) = 3</code>	<code>a[:] = 3</code>	set all values to the same scalar value
<code>y=x</code>	<code>y = x.copy()</code>	NumPy assigns by reference
<code>y=x(2, :)</code>	<code>y = x[1, :].copy()</code>	NumPy slices are by reference
<code>y=x(:)</code>	<code>y = x.flatten()</code>	turn array into vector (note that this forces a copy). To obtain the same data ordering as in MATLAB, use <code>x.flatten('F')</code> .
<code>1:10</code>	<code>np.arange(1., 11.)</code> or <code>np.r_[1.:11.]</code> or <code>np.r_[1:10:10j]</code>	create an increasing vector (see note RANGES)
<code>0:9</code>	<code>np.arange(10.)</code> or <code>np.r_[10:]</code> or <code>np.r_[9:10j]</code>	create an increasing vector (see note RANGES)
<code>[1:10]'</code>	<code>np.arange(1., 11.)[:, np.newaxis]</code>	create a column vector
<code>zeros(3,4)</code>	<code>np.zeros((3, 4))</code>	3x4 two-dimensional array full of 64-bit floating point zeros
<code>zeros(3,4,5)</code>	<code>np.zeros((3, 4, 5))</code>	3x4x5 three-dimensional array full of 64-bit floating point zeros
<code>ones(3,4)</code>	<code>np.ones((3, 4))</code>	3x4 two-dimensional array full of 64-bit floating point ones
<code>eye(3)</code>	<code>np.eye(3)</code>	3x3 identity matrix
<code>diag(a)</code>	<code>np.diag(a)</code>	returns a vector of the diagonal elements of 2D array, <code>a</code>
<code>diag(v,0)</code>	<code>np.diag(v, 0)</code>	returns a square diagonal matrix whose nonzero values are the elements of vector, <code>v</code>
<code>rng(42, 'twister')</code> <code>rand(3,4)</code>	<code>from numpy.random import _</code> <code>↪ default_rng</code> <code>rng = default_rng(42)</code> <code>rng.random(3, 4)</code> or older version: <code>random.rand((3, 4))</code>	generate a random 3x4 array with default random number generator and seed = 42
<code>linspace(1,3,4)</code>	<code>np.linspace(1,3,4)</code>	4 equally spaced samples between 1 and 3, inclusive
<code>[x,y]=meshgrid(0:8,0:5)</code>	<code>np.mgrid[0:9.,0:6.]</code> or <code>np.meshgrid(r_[0:9.], r_[0:6.])</code>	two 2D arrays: one of <code>x</code> values, the other of <code>y</code> values
	<code>ogrid[0:9.,0:6.]</code> or <code>np.ix_(np.r_[0:9.], np.r_[0:6.])</code>	the best way to eval functions on a grid

continues on next page

Table 1 – continued from previous page

MATLAB	NumPy	Notes
<code>[x,y]=meshgrid([1,2,4],[2,4,5])</code>	<code>np.meshgrid([1,2,4],[2,4,5])</code>	
	<code>ix_([1,2,4],[2,4,5])</code>	the best way to eval functions on a grid
<code>repmat(a, m, n)</code>	<code>np.tile(a, (m, n))</code>	create m by n copies of a
<code>[a b]</code>	<code>np.concatenate((a,b), 1)</code> or <code>np.hstack((a,b))</code> or <code>np.column_stack((a,b))</code> or <code>np.c_[a,b]</code>	concatenate columns of a and b
<code>[a; b]</code>	<code>np.concatenate((a,b))</code> or <code>np.vstack((a,b))</code> or <code>np.r_[a,b]</code>	concatenate rows of a and b
<code>max(max(a))</code>	<code>a.max()</code> or <code>np.nanmax(a)</code>	maximum element of a (with <code>ndims(a)<=2</code> for MATLAB, if there are NaN's, <code>nanmax</code> will ignore these and return largest value)
<code>max(a)</code>	<code>a.max(0)</code>	maximum element of each column of array a
<code>max(a,[],2)</code>	<code>a.max(1)</code>	maximum element of each row of array a
<code>max(a,b)</code>	<code>np.maximum(a, b)</code>	compares a and b element-wise, and returns the maximum value from each pair
<code>norm(v)</code>	<code>np.sqrt(v @ v)</code> or <code>np.linalg.norm(v)</code>	L2 norm of vector v
<code>a & b</code>	<code>logical_and(a,b)</code>	element-by-element AND operator (NumPy ufunc) <i>See note LOGICOPS</i>
<code>a b</code>	<code>np.logical_or(a,b)</code>	element-by-element OR operator (NumPy ufunc) <i>See note LOGICOPS</i>
<code>bitand(a,b)</code>	<code>a & b</code>	bitwise AND operator (Python native and NumPy ufunc)
<code>bitor(a,b)</code>	<code>a b</code>	bitwise OR operator (Python native and NumPy ufunc)
<code>inv(a)</code>	<code>linalg.inv(a)</code>	inverse of square 2D array a
<code>pinv(a)</code>	<code>linalg.pinv(a)</code>	pseudo-inverse of 2D array a
<code>rank(a)</code>	<code>linalg.matrix_rank(a)</code>	matrix rank of a 2D array a
<code>a\b</code>	<code>linalg.solve(a, b)</code> if a is square; <code>linalg.lstsq(a, b)</code> otherwise	solution of a x = b for x
<code>b/a</code>	Solve <code>a.T x.T = b.T</code> instead	solution of x a = b for x
<code>[U,S,V]=svd(a)</code>	<code>U, S, Vh = linalg.svd(a)</code> , <code>V = Vh.T</code>	singular value decomposition of a
<code>c=chol(a)</code> where <code>a==c'*c</code>	<code>c = linalg.cholesky(a)</code> where <code>a == c@c.T</code>	Cholesky factorization of a 2D array (<code>chol(a)</code> in MATLAB returns an upper triangular 2D array, but <code>cholesky</code> returns a lower triangular 2D array)
<code>[V,D]=eig(a)</code>	<code>D,V = linalg.eig(a)</code>	eigenvalues λ and eigenvectors \bar{v} of a, where $\lambda \bar{v} = a \bar{v}$

continues on next page

Table 1 – continued from previous page

MATLAB	NumPy	Notes
<code>[V,D]=eig(a,b)</code>	<code>D,V = linalg.eig(a, b)</code>	eigenvalues λ and eigenvectors \bar{v} of a, b where $\lambda b \bar{v} = a \bar{v}$
<code>[V,D]=eigs(a,3)</code>	<code>D,V = eigs(a, k = 3)</code>	find the $k=3$ largest eigenvalues and eigenvectors of 2D array, a
<code>[Q,R,P]=qr(a,0)</code>	<code>Q,R = linalg.qr(a)</code>	QR decomposition
<code>[L,U,P]=lu(a)</code> <code>a==P'*L*U</code>	<code>P,L,U = linalg.lu(a)</code> where <code>a == P@L@U</code>	LU decomposition (note: <code>P(MATLAB) == transpose(P(NumPy))</code>)
<code>conjgrad</code>	<code>cg</code>	Conjugate gradients solver
<code>fft(a)</code>	<code>np.fft(a)</code>	Fourier transform of a
<code>ifft(a)</code>	<code>np.ifft(a)</code>	inverse Fourier transform of a
<code>sort(a)</code>	<code>np.sort(a)</code> or <code>a.sort(axis=0)</code>	sort each column of a 2D array, a
<code>sort(a, 2)</code>	<code>np.sort(a, axis = 1)</code> or <code>a.sort(axis = 1)</code>	sort the each row of 2D array, a
<code>[b,I]=sortrows(a,1)</code>	<code>I = np.argsort(a[:, 0]);</code> <code>b = a[I, :]</code>	save the array a as array b with rows sorted by the first column
<code>x = Z\y</code>	<code>x = linalg.lstsq(Z, y)</code>	perform a linear regression of the form $Zx = y$
<code>decimate(x, q)</code>	<code>signal.resample(x, np.ceil(len(x)/q))</code>	downsample with low-pass filtering
<code>unique(a)</code>	<code>np.unique(a)</code>	a vector of unique values in array a
<code>squeeze(a)</code>	<code>a.squeeze()</code>	remove singleton dimensions of array a . Note that MATLAB will always return arrays of 2D or higher while NumPy will return arrays of 0D or higher

6.4 Notes

Submatrix: Assignment to a submatrix can be done with lists of indices using the `ix_` command. E.g., for 2D array a , one might do: `ind=[1, 3]; a[np.ix_(ind, ind)] += 100.`

HELP: There is no direct equivalent of MATLAB's `which` command, but the commands `help` and `numpy.` source will usually list the filename where the function is located. Python also has an `inspect` module (do `import inspect`) which provides a `getfile` that often works.

INDEXING: MATLAB uses one based indexing, so the initial element of a sequence has index 1. Python uses zero based indexing, so the initial element of a sequence has index 0. Confusion and flamewars arise because each has advantages and disadvantages. One based indexing is consistent with common human language usage, where the “first” element of a sequence has index 1. Zero based indexing [simplifies indexing](#). See also [a text by prof.dr. Edsger W. Dijkstra](#).

RANGES: In MATLAB, `0:5` can be used as both a range literal and a ‘slice’ index (inside parentheses); however, in Python, constructs like `0:5` can *only* be used as a slice index (inside square brackets). Thus the somewhat quirky `r_` object was created to allow NumPy to have a similarly terse range construction mechanism. Note that `r_` is not called like a function or a constructor, but rather *indexed* using square brackets, which allows the use of Python's slice syntax in the arguments.

LOGICOPS: `&` or `|` in NumPy is bitwise AND/OR, while in MATLAB `&` and `|` are logical AND/OR. The two can appear to work the same, but there are important differences. If you would have used MATLAB's `&` or `|` operators, you should use the NumPy ufuncs `logical_and`/`logical_or`. The notable differences between MATLAB's and

NumPy's `&` and `|` operators are:

- Non-logical `{0,1}` inputs: NumPy's output is the bitwise AND of the inputs. MATLAB treats any non-zero value as 1 and returns the logical AND. For example `(3 & 4)` in NumPy is 0, while in MATLAB both 3 and 4 are considered logical true and `(3 & 4)` returns 1.
- Precedence: NumPy's `&` operator is higher precedence than logical operators like `<` and `>`; MATLAB's is the reverse.

If you know you have boolean arguments, you can get away with using NumPy's bitwise operators, but be careful with parentheses, like this: `z = (x > 1) & (x < 2)`. The absence of NumPy operator forms of `logical_and` and `logical_or` is an unfortunate consequence of Python's design.

RESHAPE and LINEAR INDEXING: MATLAB always allows multi-dimensional arrays to be accessed using scalar or linear indices, NumPy does not. Linear indices are common in MATLAB programs, e.g. `find()` on a matrix returns them, whereas NumPy's `find` behaves differently. When converting MATLAB code it might be necessary to first reshape a matrix to a linear sequence, perform some indexing operations and then reshape back. As `reshape` (usually) produces views onto the same storage, it should be possible to do this fairly efficiently. Note that the scan order used by `reshape` in NumPy defaults to the 'C' order, whereas MATLAB uses the Fortran order. If you are simply converting to a linear sequence and back this doesn't matter. But if you are converting reshapes from MATLAB code which relies on the scan order, then this MATLAB code: `z = reshape(x, 3, 4);` should become `z = x.reshape(3, 4, order='F').copy()` in NumPy.

6.5 'array' or 'matrix'? Which should I use?

Historically, NumPy has provided a special matrix type, `np.matrix`, which is a subclass of `ndarray` which makes binary operations linear algebra operations. You may see it used in some existing code instead of `np.array`. So, which one to use?

6.5.1 Short answer

Use arrays.

- They support multidimensional array algebra that is supported in MATLAB
- They are the standard vector/matrix/tensor type of NumPy. Many NumPy functions return arrays, not matrices.
- There is a clear distinction between element-wise operations and linear algebra operations.
- You can have standard vectors or row/column vectors if you like.

Until Python 3.5 the only disadvantage of using the array type was that you had to use `dot` instead of `*` to multiply (reduce) two tensors (scalar product, matrix vector multiplication etc.). Since Python 3.5 you can use the matrix multiplication `@` operator.

Given the above, we intend to deprecate `matrix` eventually.

6.5.2 Long answer

NumPy contains both an `array` class and a `matrix` class. The `array` class is intended to be a general-purpose n-dimensional array for many kinds of numerical computing, while `matrix` is intended to facilitate linear algebra computations specifically. In practice there are only a handful of key differences between the two.

- Operators `*` and `@`, functions `dot()`, and `multiply()`:
 - For `array`, **“*” means element-wise multiplication**, while **“@” means matrix multiplication**; they have associated functions `multiply()` and `dot()`. (Before Python 3.5, `@` did not exist and one had to use `dot()` for matrix multiplication).
 - For `matrix`, **“*” means matrix multiplication**, and for element-wise multiplication one has to use the `multiply()` function.
- Handling of vectors (one-dimensional arrays)
 - For `array`, the **vector shapes `1xN`, `Nx1`, and `N` are all different things**. Operations like `A[:, 1]` return a one-dimensional array of shape `N`, not a two-dimensional array of shape `Nx1`. Transpose on a one-dimensional array does nothing.
 - For `matrix`, **one-dimensional arrays are always upconverted to `1xN` or `Nx1` matrices** (row or column vectors). `A[:, 1]` returns a two-dimensional matrix of shape `Nx1`.
- Handling of higher-dimensional arrays (`ndim > 2`)
 - `array` objects **can have number of dimensions `> 2`**;
 - `matrix` objects **always have exactly two dimensions**.
- Convenience attributes
 - `array` **has a `.T` attribute**, which returns the transpose of the data.
 - `matrix` **also has `.H`, `.I`, and `.A` attributes**, which return the conjugate transpose, inverse, and `asarray()` of the matrix, respectively.
- Convenience constructor
 - The `array` constructor **takes (nested) Python sequences as initializers**. As in, `array([[1, 2, 3], [4, 5, 6]])`.
 - The `matrix` constructor additionally **takes a convenient string initializer**. As in `matrix("[1 2 3; 4 5 6]")`.

There are pros and cons to using both:

- `array`
 - `:` Element-wise multiplication is easy: `A*B`.
 - `:` You have to remember that matrix multiplication has its own operator, `@`.
 - `:` You can treat one-dimensional arrays as *either* row or column vectors. `A @ v` treats `v` as a column vector, while `v @ A` treats `v` as a row vector. This can save you having to type a lot of transposes.
 - `:` `array` is the “default” NumPy type, so it gets the most testing, and is the type most likely to be returned by 3rd party code that uses NumPy.
 - `:` Is quite at home handling data of any number of dimensions.
 - `:` Closer in semantics to tensor algebra, if you are familiar with that.
 - `:` All operations (`*`, `/`, `+`, `-` etc.) are element-wise.
 - `:` Sparse matrices from `scipy.sparse` do not interact as well with arrays.

- `matrix`
 - `:` Behavior is more like that of MATLAB matrices.
 - `<:` (Maximum of two-dimensional. To hold three-dimensional data you need `array` or perhaps a Python list of `matrix`.
 - `<:` (Minimum of two-dimensional. You cannot have vectors. They must be cast as single-column or single-row matrices.
 - `<:` (Since `array` is the default in NumPy, some functions may return an `array` even if you give them a `matrix` as an argument. This shouldn't happen with NumPy functions (if it does it's a bug), but 3rd party code based on NumPy may not honor type preservation like NumPy does.
 - `:` `A*B` is matrix multiplication, so it looks just like you write it in linear algebra (For Python `>= 3.5` plain arrays have the same convenience with the `@` operator).
 - `<:` (Element-wise multiplication requires calling a function, `multiply(A,B)`.
 - `<:` (The use of operator overloading is a bit illogical: `*` does not work element-wise but `/` does.
 - Interaction with `scipy.sparse` is a bit cleaner.

The `array` is thus much more advisable to use. Indeed, we intend to deprecate `matrix` eventually.

6.6 Customizing your environment

In MATLAB the main tool available to you for customizing the environment is to modify the search path with the locations of your favorite functions. You can put such customizations into a startup script that MATLAB will run on startup.

NumPy, or rather Python, has similar facilities.

- To modify your Python search path to include the locations of your own modules, define the `PYTHONPATH` environment variable.
- To have a particular script file executed when the interactive Python interpreter is started, define the `PYTHONSTARTUP` environment variable to contain the name of your startup script.

Unlike MATLAB, where anything on your path can be called immediately, with Python you need to first do an 'import' statement to make functions in a particular file accessible.

For example you might make a startup script that looks like this (Note: this is just an example, not a statement of "best practices"):

```
# Make all numpy available via shorter 'np' prefix
import numpy as np
#
# Make the SciPy linear algebra functions available as linalg.func()
# e.g. linalg.lu, linalg.eig (for general l*B@u==A@u solution)
from scipy import linalg
#
# Define a Hermitian function
def hermitian(A, **kwargs):
    return np.conj(A, **kwargs).T
# Make a shortcut for hermitian:
#   hermitian(A) --> H(A)
H = hermitian
```

To use the deprecated `matrix` and other `matlib` functions:

```
# Make all matlib functions accessible at the top level via M.func()  
import numpy.matlib as M  
# Make some matlib functions accessible directly at the top level via, e.g. rand(3,3)  
from numpy.matlib import matrix, rand, zeros, ones, empty, eye
```

6.7 Links

Another somewhat outdated MATLAB/NumPy cross-reference can be found at <http://mathesaurus.sf.net/>

An extensive list of tools for scientific work with Python can be found in the [topical software page](#).

See [List of Python software: scripting](#) for a list of software that use Python as a scripting language

MATLAB® and SimuLink® are registered trademarks of The MathWorks, Inc.

BUILDING FROM SOURCE

There are two options for building NumPy- building with Gitpod or locally from source. Your choice depends on your operating system and familiarity with the command line.

7.1 Gitpod

Gitpod is an open-source platform that automatically creates the correct development environment right in your browser, reducing the need to install local development environments and deal with incompatible dependencies.

If you are a Windows user, unfamiliar with using the command line or building NumPy for the first time, it is often faster to build with Gitpod. Here are the in-depth instructions for building NumPy with [building NumPy with Gitpod](#).

7.2 Building locally

Building locally on your machine gives you more granular control. If you are a MacOS or Linux user familiar with using the command line, you can continue with building NumPy locally by following the instructions below.

7.3 Prerequisites

Building NumPy requires the following software installed:

1) Python 3.8.x or newer

Please note that the Python development headers also need to be installed, e.g., on Debian/Ubuntu one needs to install both *python3* and *python3-dev*. On Windows and macOS this is normally not an issue.

2) Compilers

Much of NumPy is written in C. You will need a C compiler that complies with the C99 standard.

Part of Numpy is now written in C++. You will also need a C++ compiler that complies with the C++11 standard.

While a FORTRAN 77 compiler is not necessary for building NumPy, it is needed to run the `numpy.f2py` tests. These tests are skipped if the compiler is not auto-detected.

Note that NumPy is developed mainly using GNU compilers and tested on MSVC and Clang compilers. Compilers from other vendors such as Intel, Absoft, Sun, NAG, Compaq, Vast, Portland, Lahey, HP, IBM are only supported in the form of community feedback, and may not work out of the box. GCC 4.x (and later) compilers are recommended. On ARM64 (aarch64) GCC 8.x (and later) are recommended.

3) Linear Algebra libraries

NumPy does not require any external linear algebra libraries to be installed. However, if these are available, NumPy's setup script can detect them and use them for building. A number of different LAPACK library setups can be used, including optimized LAPACK libraries such as OpenBLAS or MKL. The choice and location of these libraries as well as include paths and other such build options can be specified in a `site.cfg` file located in the NumPy root repository or a `.numpy-site.cfg` file in your home directory. See the `site.cfg.example` example file included in the NumPy repository or `sdist` for documentation, and below for specifying search priority from environmental variables.

4) Cython

For building NumPy, you'll need a recent version of Cython.

7.4 Basic Installation

To install NumPy, run:

```
pip install .
```

To perform an in-place build that can be run from the source folder run:

```
python setup.py build_ext --inplace
```

Note: for build instructions to do development work on NumPy itself, see development-environment.

7.5 Testing

Make sure to test your builds. To ensure everything stays in shape, see if all tests pass.

The test suite requires additional dependencies, which can easily be installed with:

```
$ python -m pip install -r test_requirements.txt
```

Run tests:

```
$ python runtests.py -v -m full
```

For detailed info on testing, see `testing-builds`.

7.5.1 Parallel builds

It's possible to do a parallel build with:

```
python setup.py build -j 4 install --prefix $HOME/.local
```

This will compile numpy on 4 CPUs and install it into the specified prefix. to perform a parallel in-place build, run:

```
python setup.py build_ext --inplace -j 4
```

The number of build jobs can also be specified via the environment variable `NPY_NUM_BUILD_JOBS`.

7.5.2 Choosing the fortran compiler

Compilers are auto-detected; building with a particular compiler can be done with `--fcompiler`. E.g. to select gfortran:

```
python setup.py build --fcompiler=gnu95
```

For more information see:

```
python setup.py build --help-fcompiler
```

7.5.3 How to check the ABI of BLAS/LAPACK libraries

One relatively simple and reliable way to check for the compiler used to build a library is to use `ldd` on the library. If `libg2c.so` is a dependency, this means that `g77` has been used (note: `g77` is no longer supported for building NumPy). If `libgfortran.so` is a dependency, `gfortran` has been used. If both are dependencies, this means both have been used, which is almost always a very bad idea.

7.6 Accelerated BLAS/LAPACK libraries

NumPy searches for optimized linear algebra libraries such as BLAS and LAPACK. There are specific orders for searching these libraries, as described below and in the `site.cfg.example` file.

7.6.1 BLAS

Note that both BLAS and CBLAS interfaces are needed for a properly optimized build of NumPy.

The default order for the libraries are:

1. MKL
2. BLIS
3. OpenBLAS
4. ATLAS
5. BLAS (NetLIB)

The detection of BLAS libraries may be bypassed by defining the environment variable `NPY_BLAS_LIBS`, which should contain the exact linker flags you want to use (interface is assumed to be Fortran 77). Also define `NPY_CBLAS_LIBS` (even empty if CBLAS is contained in your BLAS library) to trigger use of CBLAS and avoid slow fallback code for matrix calculations.

If you wish to build against OpenBLAS but you also have BLIS available one may predefine the order of searching via the environment variable `NPY_BLAS_ORDER` which is a comma-separated list of the above names which is used to determine what to search for, for instance:

```
NPY_BLAS_ORDER=ATLAS,blis,openblas,MKL python setup.py build
```

will prefer to use ATLAS, then BLIS, then OpenBLAS and as a last resort MKL. If neither of these exists the build will fail (names are compared lower case).

Alternatively one may use `!` or `^` to negate all items:

```
NPY_BLAS_ORDER='^blas,atlas' python setup.py build
```

will allow using anything **but** NetLIB BLAS and ATLAS libraries, the order of the above list is retained.

One cannot mix negation and positives, nor have multiple negations, such cases will raise an error.

7.6.2 LAPACK

The default order for the libraries are:

1. MKL
2. OpenBLAS
3. libFLAME
4. ATLAS
5. LAPACK (NetLIB)

The detection of LAPACK libraries may be bypassed by defining the environment variable `NPY_LAPACK_LIBS`, which should contain the exact linker flags you want to use (language is assumed to be Fortran 77).

If you wish to build against OpenBLAS but you also have MKL available one may predefine the order of searching via the environment variable `NPY_LAPACK_ORDER` which is a comma-separated list of the above names, for instance:

```
NPY_LAPACK_ORDER=ATLAS,openblas,MKL python setup.py build
```

will prefer to use ATLAS, then OpenBLAS and as a last resort MKL. If neither of these exists the build will fail (names are compared lower case).

Alternatively one may use `!` or `^` to negate all items:

```
NPY_LAPACK_ORDER='^lapack' python setup.py build
```

will allow using anything **but** the NetLIB LAPACK library, the order of the above list is retained.

One cannot mix negation and positives, nor have multiple negations, such cases will raise an error.

Deprecated since version 1.20: The native libraries on macOS, provided by Accelerate, are not fit for use in NumPy since they have bugs that cause wrong output under easily reproducible conditions. If the vendor fixes those bugs, the library could be reinstated, but until then users compiling for themselves should use another linear algebra library or use the built-in (but slower) default, see the next section.

7.6.3 Disabling ATLAS and other accelerated libraries

Usage of ATLAS and other accelerated libraries in NumPy can be disabled via:

```
NPY_BLAS_ORDER= NPY_LAPACK_ORDER= python setup.py build
```

or:

```
BLAS=None LAPACK=None ATLAS=None python setup.py build
```

7.6.4 64-bit BLAS and LAPACK

You can tell Numpy to use 64-bit BLAS/LAPACK libraries by setting the environment variable:

```
NPY_USE_BLAS_ILP64=1
```

when building Numpy. The following 64-bit BLAS/LAPACK libraries are supported:

1. OpenBLAS ILP64 with 64_ symbol suffix (openblas64_)
2. OpenBLAS ILP64 without symbol suffix (openblas_ilp64)

The order in which they are preferred is determined by `NPY_BLAS_ILP64_ORDER` and `NPY_LAPACK_ILP64_ORDER` environment variables. The default value is `openblas64_`, `openblas_ilp64`.

Note: Using non-symbol-suffixed 64-bit BLAS/LAPACK in a program that also uses 32-bit BLAS/LAPACK can cause crashes under certain conditions (e.g. with embedded Python interpreters on Linux).

The 64-bit OpenBLAS with 64_ symbol suffix is obtained by compiling OpenBLAS with settings:

```
make INTERFACE64=1 SYMBOLSUFFIX=64_
```

The symbol suffix avoids the symbol name clashes between 32-bit and 64-bit BLAS/LAPACK libraries.

7.7 Supplying additional compiler flags

Additional compiler flags can be supplied by setting the `OPT`, `FOPT` (for Fortran), and `CC` environment variables. When providing options that should improve the performance of the code ensure that you also set `-DNDEBUG` so that debugging code is not executed.

7.8 Cross compilation

Although `numpy.distutils` and `setuptools` do not directly support cross compilation, it is possible to build NumPy on one system for different architectures with minor modifications to the build environment. This may be desirable, for example, to use the power of a high-performance desktop to create a NumPy package for a low-power, single-board computer. Because the `setup.py` scripts are unaware of cross-compilation environments and tend to make decisions based on the environment detected on the build system, it is best to compile for the same type of operating system that runs on the builder. Attempting to compile a Mac version of NumPy on Windows, for example, is likely to be met with challenges not considered here.

For the purpose of this discussion, the nomenclature adopted by `meson` will be used: the “build” system is that which will be running the NumPy build process, while the “host” is the platform on which the compiled package will be run. A native Python interpreter, the `setuptools` and `Cython` packages and the desired cross compiler must be available for the build system. In addition, a Python interpreter and its development headers as well as any external linear algebra libraries must be available for the host platform. For convenience, it is assumed that all host software is available under a separate prefix directory, here called `$CROSS_PREFIX`.

When building and installing NumPy for a host system, the `CC` environment variable must provide the path the cross compiler that will be used to build NumPy C extensions. It may also be necessary to set the `LD_SHARED` environment variable to the path to the linker that can link compiled objects for the host system. The compiler must be told where it can find Python libraries and development headers. On Unix-like systems, this generally requires adding, e.g., the following parameters to the `CFLAGS` environment variable:

```
-I${CROSS_PREFIX}/usr/include
-I${CROSS_PREFIX}/usr/include/python3.y
```

for Python version 3.y. (Replace the “y” in this path with the actual minor number of the installed Python runtime.) Likewise, the linker should be told where to find host libraries by adding a parameter to the `LDFLAGS` environment variable:

```
-L${CROSS_PREFIX}/usr/lib
```

To make sure Python-specific system configuration options are provided for the intended host and not the build system, set:

```
_PYTHON_SYSCONFIGDATA_NAME=_sysconfigdata_${ARCH_TRIPLET}
```

where `${ARCH_TRIPLET}` is an architecture-dependent suffix appropriate for the host architecture. (This should be the name of a `_sysconfigdata` file, without the `.py` extension, found in the host Python library directory.)

When using external linear algebra libraries, include and library directories should be provided for the desired libraries in `site.cfg` as described above and in the comments of the `site.cfg.example` file included in the NumPy repository or `sdist`. In this example, set:

```
include_dirs = ${CROSS_PREFIX}/usr/include
library_dirs = ${CROSS_PREFIX}/usr/lib
```

under appropriate sections of the file to allow `numpy.distutils` to find the libraries.

As of NumPy 1.22.0, a vendored copy of SVML will be built on `x86_64` Linux hosts to provide AVX-512 acceleration of floating-point operations. When using an `x86_64` Linux build system to cross compile NumPy for hosts other than `x86_64` Linux, set the environment variable `NPY_DISABLE_SVML` to prevent the NumPy build script from incorrectly attempting to cross-compile this platform-specific library:

```
NPY_DISABLE_SVML=1
```

With the environment configured, NumPy may be built as it is natively:

```
python setup.py build
```

When the `wheel` package is available, the cross-compiled package may be packed into a wheel for installation on the host with:

```
python setup.py bdist_wheel
```

It may be possible to use `pip` to build a wheel, but `pip` configures its own environment; adapting the `pip` environment to cross-compilation is beyond the scope of this guide.

The cross-compiled package may also be installed into the host prefix for cross-compilation of other packages using, *e.g.*, the command:

```
python setup.py install --prefix=${CROSS_PREFIX}
```

When cross compiling other packages that depend on NumPy, the host `numpy-pkg-config` file must be made available. For further discussion, refer to [numpy.distutils documentation](#).

USING NUMPY C-API

8.1 How to extend NumPy

That which is static and repetitive is boring. That which is dynamic and random is confusing. In between lies art.

— *John A. Locke*

Science is a differential equation. Religion is a boundary condition.

— *Alan Turing*

8.1.1 Writing an extension module

While the `ndarray` object is designed to allow rapid computation in Python, it is also designed to be general-purpose and satisfy a wide- variety of computational needs. As a result, if absolute speed is essential, there is no replacement for a well-crafted, compiled loop specific to your application and hardware. This is one of the reasons that numpy includes `f2py` so that an easy-to-use mechanisms for linking (simple) C/C++ and (arbitrary) Fortran code directly into Python are available. You are encouraged to use and improve this mechanism. The purpose of this section is not to document this tool but to document the more basic steps to writing an extension module that this tool depends on.

When an extension module is written, compiled, and installed to somewhere in the Python path (`sys.path`), the code can then be imported into Python as if it were a standard python file. It will contain objects and methods that have been defined and compiled in C code. The basic steps for doing this in Python are well-documented and you can find more information in the documentation for Python itself available online at www.python.org.

In addition to the Python C-API, there is a full and rich C-API for NumPy allowing sophisticated manipulations on a C-level. However, for most applications, only a few API calls will typically be used. For example, if you need to just extract a pointer to memory along with some shape information to pass to another calculation routine, then you will use very different calls than if you are trying to create a new array-like type or add a new data type for `ndarrays`. This chapter documents the API calls and macros that are most commonly used.

8.1.2 Required subroutine

There is exactly one function that must be defined in your C-code in order for Python to use it as an extension module. The function must be called `init{name}` where `{name}` is the name of the module from Python. This function must be declared so that it is visible to code outside of the routine. Besides adding the methods and constants you desire, this subroutine must also contain calls like `import_array()` and/or `import_ufunc()` depending on which C-API is needed. Forgetting to place these commands will show itself as an ugly segmentation fault (crash) as soon as any C-API subroutine is actually called. It is actually possible to have multiple `init{name}` functions in a single file in which case multiple modules will be defined by that file. However, there are some tricks to get that to work correctly and it is not covered here.

A minimal `init{name}` method looks like:

```
PyMODINIT_FUNC
init{name}(void)
{
    (void)Py_InitModule({name}, mymethods);
    import_array();
}
```

The `mymethods` must be an array (usually statically declared) of `PyMethodDef` structures which contain method names, actual C-functions, a variable indicating whether the method uses keyword arguments or not, and docstrings. These are explained in the next section. If you want to add constants to the module, then you store the returned value from `Py_InitModule` which is a module object. The most general way to add items to the module is to get the module dictionary using `PyModule_GetDict(module)`. With the module dictionary, you can add whatever you like to the module manually. An easier way to add objects to the module is to use one of three additional Python C-API calls that do not require a separate extraction of the module dictionary. These are documented in the Python documentation, but repeated here for convenience:

```
int PyModule_AddObject (PyObject *module, char *name, PyObject *value)
```

```
int PyModule_AddIntConstant (PyObject *module, char *name, long value)
```

```
int PyModule_AddStringConstant (PyObject *module, char *name, char *value)
```

All three of these functions require the *module* object (the return value of `Py_InitModule`). The *name* is a string that labels the value in the module. Depending on which function is called, the *value* argument is either a general object (`PyModule_AddObject` steals a reference to it), an integer constant, or a string constant.

8.1.3 Defining functions

The second argument passed in to the `Py_InitModule` function is a structure that makes it easy to define functions in the module. In the example given above, the `mymethods` structure would have been defined earlier in the file (usually right before the `init{name}` subroutine) to:

```
static PyMethodDef mymethods[] = {
    { nokeywordfunc, nokeyword_cfunc,
      METH_VARARGS,
      Doc string},
    { keywordfunc, keyword_cfunc,
      METH_VARARGS|METH_KEYWORDS,
      Doc string},
    {NULL, NULL, 0, NULL} /* Sentinel */
}
```

Each entry in the `mymethods` array is a `PyMethodDef` structure containing 1) the Python name, 2) the C-function that implements the function, 3) flags indicating whether or not keywords are accepted for this function, and 4) The docstring

for the function. Any number of functions may be defined for a single module by adding more entries to this table. The last entry must be all NULL as shown to act as a sentinel. Python looks for this entry to know that all of the functions for the module have been defined.

The last thing that must be done to finish the extension module is to actually write the code that performs the desired functions. There are two kinds of functions: those that don't accept keyword arguments, and those that do.

Functions without keyword arguments

Functions that don't accept keyword arguments should be written as:

```
static PyObject*
nokeyword_cfunc (PyObject *dummy, PyObject *args)
{
    /* convert Python arguments */
    /* do function */
    /* return something */
}
```

The dummy argument is not used in this context and can be safely ignored. The *args* argument contains all of the arguments passed in to the function as a tuple. You can do anything you want at this point, but usually the easiest way to manage the input arguments is to call `PyArg_ParseTuple` (*args*, *format_string*, *addresses_to_C_variables*...) or `PyArg_UnpackTuple` (*tuple*, "name", min, max, ...). A good description of how to use the first function is contained in the Python C-API reference manual under section 5.5 (Parsing arguments and building values). You should pay particular attention to the "O&" format which uses converter functions to go between the Python object and the C object. All of the other format functions can be (mostly) thought of as special cases of this general rule. There are several converter functions defined in the NumPy C-API that may be of use. In particular, the `PyArray_DescrConverter` function is very useful to support arbitrary data-type specification. This function transforms any valid data-type Python object into a `PyArray_Descr*` object. Remember to pass in the address of the C-variables that should be filled in.

There are lots of examples of how to use `PyArg_ParseTuple` throughout the NumPy source code. The standard usage is like this:

```
PyObject *input;
PyArray_Descr *dtype;
if (!PyArg_ParseTuple(args, "OO&", &input,
                      PyArray_DescrConverter,
                      &dtype)) return NULL;
```

It is important to keep in mind that you get a *borrowed* reference to the object when using the "O" format string. However, the converter functions usually require some form of memory handling. In this example, if the conversion is successful, *dtype* will hold a new reference to a `PyArray_Descr*` object, while *input* will hold a borrowed reference. Therefore, if this conversion were mixed with another conversion (say to an integer) and the data-type conversion was successful but the integer conversion failed, then you would need to release the reference count to the data-type object before returning. A typical way to do this is to set *dtype* to NULL before calling `PyArg_ParseTuple` and then use `Py_XDECREF` on *dtype* before returning.

After the input arguments are processed, the code that actually does the work is written (likely calling other functions as needed). The final step of the C-function is to return something. If an error is encountered then NULL should be returned (making sure an error has actually been set). If nothing should be returned then increment `Py_None` and return it. If a single object should be returned then it is returned (ensuring that you own a reference to it first). If multiple objects should be returned then you need to return a tuple. The `Py_BuildValue` (*format_string*, *c_variables*...) function makes it easy to build tuples of Python objects from C variables. Pay special attention to the difference between 'N' and 'O' in the format string or you can easily create memory leaks. The 'O' format string increments the reference count of the `PyObject*` C-variable it corresponds to, while the 'N' format string steals a reference to the corresponding `PyObject*` C-variable.

You should use 'N' if you have already created a reference for the object and just want to give that reference to the tuple. You should use 'O' if you only have a borrowed reference to an object and need to create one to provide for the tuple.

Functions with keyword arguments

These functions are very similar to functions without keyword arguments. The only difference is that the function signature is:

```
static PyObject*
keyword_cfunc (PyObject *dummy, PyObject *args, PyObject *kwds)
{
    ...
}
```

The `kwds` argument holds a Python dictionary whose keys are the names of the keyword arguments and whose values are the corresponding keyword-argument values. This dictionary can be processed however you see fit. The easiest way to handle it, however, is to replace the `PyArg_ParseTuple` (`args`, `format_string`, `addresses...`) function with a call to `PyArg_ParseTupleAndKeywords` (`args`, `kwds`, `format_string`, `char *kwlist[]`, `addresses...`). The `kwlist` parameter to this function is a `NULL`-terminated array of strings providing the expected keyword arguments. There should be one string for each entry in the `format_string`. Using this function will raise a `TypeError` if invalid keyword arguments are passed in.

For more help on this function please see section 1.8 (Keyword Parameters for Extension Functions) of the Extending and Embedding tutorial in the Python documentation.

Reference counting

The biggest difficulty when writing extension modules is reference counting. It is an important reason for the popularity of `f2py`, `weave`, `Cython`, `ctypes`, etc.... If you mis-handle reference counts you can get problems from memory-leaks to segmentation faults. The only strategy I know of to handle reference counts correctly is blood, sweat, and tears. First, you force it into your head that every Python variable has a reference count. Then, you understand exactly what each function does to the reference count of your objects, so that you can properly use `DECREF` and `INCREf` when you need them. Reference counting can really test the amount of patience and diligence you have towards your programming craft. Despite the grim depiction, most cases of reference counting are quite straightforward with the most common difficulty being not using `DECREF` on objects before exiting early from a routine due to some error. In second place, is the common error of not owning the reference on an object that is passed to a function or macro that is going to steal the reference (e.g. `PyTuple_SET_ITEM`, and most functions that take `PyArray_Descr` objects).

Typically you get a new reference to a variable when it is created or is the return value of some function (there are some prominent exceptions, however — such as getting an item out of a tuple or a dictionary). When you own the reference, you are responsible to make sure that `Py_DECREF` (`var`) is called when the variable is no longer necessary (and no other function has “stolen” its reference). Also, if you are passing a Python object to a function that will “steal” the reference, then you need to make sure you own it (or use `Py_INCREF` to get your own reference). You will also encounter the notion of borrowing a reference. A function that borrows a reference does not alter the reference count of the object and does not expect to “hold on” to the reference. It’s just going to use the object temporarily. When you use `PyArg_ParseTuple` or `PyArg_UnpackTuple` you receive a borrowed reference to the objects in the tuple and should not alter their reference count inside your function. With practice, you can learn to get reference counting right, but it can be frustrating at first.

One common source of reference-count errors is the `Py_BuildValue` function. Pay careful attention to the difference between the 'N' format character and the 'O' format character. If you create a new object in your subroutine (such as an output array), and you are passing it back in a tuple of return values, then you should most-likely use the 'N' format character in `Py_BuildValue`. The 'O' character will increase the reference count by one. This will leave the caller with two reference counts for a brand-new array. When the variable is deleted and the reference count decremented by one, there will still be that extra reference count, and the array will never be deallocated. You will have a reference-counting

induced memory leak. Using the ‘N’ character will avoid this situation as it will return to the caller an object (inside the tuple) with a single reference count.

8.1.4 Dealing with array objects

Most extension modules for NumPy will need to access the memory for an ndarray object (or one of its sub-classes). The easiest way to do this doesn’t require you to know much about the internals of NumPy. The method is to

1. Ensure you are dealing with a well-behaved array (aligned, in machine byte-order and single-segment) of the correct type and number of dimensions.
 1. By converting it from some Python object using `PyArray_FromAny` or a macro built on it.
 2. By constructing a new ndarray of your desired shape and type using `PyArray_NewFromDescr` or a simpler macro or function based on it.
2. Get the shape of the array and a pointer to its actual data.
3. Pass the data and shape information on to a subroutine or other section of code that actually performs the computation.
4. If you are writing the algorithm, then I recommend that you use the stride information contained in the array to access the elements of the array (the `PyArray_GetPtr` macros make this painless). Then, you can relax your requirements so as not to force a single-segment array and the data-copying that might result.

Each of these sub-topics is covered in the following sub-sections.

Converting an arbitrary sequence object

The main routine for obtaining an array from any Python object that can be converted to an array is `PyArray_FromAny`. This function is very flexible with many input arguments. Several macros make it easier to use the basic function. `PyArray_FROM_OTF` is arguably the most useful of these macros for the most common uses. It allows you to convert an arbitrary Python object to an array of a specific builtin data-type (e.g. float), while specifying a particular set of requirements (e.g. contiguous, aligned, and writeable). The syntax is

`PyArray_FROM_OTF`

Return an ndarray from any Python object, *obj*, that can be converted to an array. The number of dimensions in the returned array is determined by the object. The desired data-type of the returned array is provided in *typenum* which should be one of the enumerated types. The *requirements* for the returned array can be any combination of standard array flags. Each of these arguments is explained in more detail below. You receive a new reference to the array on success. On failure, NULL is returned and an exception is set.

obj

The object can be any Python object convertible to an ndarray. If the object is already (a subclass of) the ndarray that satisfies the requirements then a new reference is returned. Otherwise, a new array is constructed. The contents of *obj* are copied to the new array unless the array interface is used so that data does not have to be copied. Objects that can be converted to an array include: 1) any nested sequence object, 2) any object exposing the array interface, 3) any object with an `__array__` method (which should return an ndarray), and 4) any scalar object (becomes a zero-dimensional array). Sub-classes of the ndarray that otherwise fit the requirements will be passed through. If you want to ensure a base-class ndarray, then use `NPY_ARRAY_ENSUREARRAY` in the requirements flag. A copy is made only if necessary. If you want to guarantee a copy, then pass in `NPY_ARRAY_ENSURECOPY` to the requirements flag.

typenum

One of the enumerated types or `NPY_NOTYPE` if the data-type should be determined from the object itself. The C-based names can be used:

```
NPY_BOOL, NPY_BYTE, NPY_UBYTE, NPY_SHORT, NPY_USHORT, NPY_INT,
NPY_UINT, NPY_LONG, NPY_ULONG, NPY_LONGLONG, NPY_ULONGLONG, NPY_DOUBLE,
NPY_LONGDOUBLE, NPY_CFLOAT, NPY_CDOUBLE, NPY_CLONGDOUBLE, NPY_OBJECT.
```

Alternatively, the bit-width names can be used as supported on the platform. For example:

```
NPY_INT8, NPY_INT16, NPY_INT32, NPY_INT64, NPY_UINT8, NPY_UINT16,
NPY_UINT32, NPY_UINT64, NPY_FLOAT32, NPY_FLOAT64, NPY_COMPLEX64,
NPY_COMPLEX128.
```

The object will be converted to the desired type only if it can be done without losing precision. Otherwise `NULL` will be returned and an error raised. Use `NPY_ARRAY_FORCECAST` in the requirements flag to override this behavior.

requirements

The memory model for an ndarray admits arbitrary strides in each dimension to advance to the next element of the array. Often, however, you need to interface with code that expects a C-contiguous or a Fortran-contiguous memory layout. In addition, an ndarray can be misaligned (the address of an element is not at an integral multiple of the size of the element) which can cause your program to crash (or at least work more slowly) if you try and dereference a pointer into the array data. Both of these problems can be solved by converting the Python object into an array that is more “well-behaved” for your specific usage.

The requirements flag allows specification of what kind of array is acceptable. If the object passed in does not satisfy this requirements then a copy is made so that the returned object will satisfy the requirements. these ndarray can use a very generic pointer to memory. This flag allows specification of the desired properties of the returned array object. All of the flags are explained in the detailed API chapter. The flags most commonly needed are `NPY_ARRAY_IN_ARRAY`, `NPY_OUT_ARRAY`, and `NPY_ARRAY_INOUT_ARRAY`:

NPY_ARRAY_IN_ARRAY

This flag is useful for arrays that must be in C-contiguous order and aligned. These kinds of arrays are usually input arrays for some algorithm.

NPY_ARRAY_OUT_ARRAY

This flag is useful to specify an array that is in C-contiguous order, is aligned, and can be written to as well. Such an array is usually returned as output (although normally such output arrays are created from scratch).

NPY_ARRAY_INOUT_ARRAY

This flag is useful to specify an array that will be used for both input and output. `PyArray_ResolveWritebackIfCopy` must be called before `Py_DECREF` at the end of the interface routine to write back the temporary data into the original array passed in. Use of the `NPY_ARRAY_WRITEBACKIFCOPY` flag requires that the input object is already an array (because other objects cannot be automatically updated in this fashion). If an error occurs use `PyArray_DiscardWritebackIfCopy (obj)` on an array with these flags set. This will set the underlying base array writable without causing the contents to be copied back into the original array.

Other useful flags that can be OR’d as additional requirements are:

NPY_ARRAY_FORCECAST

Cast to the desired type, even if it can’t be done without losing information.

NPY_ARRAY_ENSURECOPY

Make sure the resulting array is a copy of the original.

NPY_ARRAY_ENSUREARRAY

Make sure the resulting object is an actual ndarray and not a sub- class.

Note: Whether or not an array is byte-swapped is determined by the data-type of the array. Native byte-order arrays are always requested by `PyArray_FROM_OTF` and so there is no need for a `NPY_ARRAY_NOTSWAPPED` flag in the requirements argument. There is also no way to get a byte-swapped array from this routine.

Creating a brand-new ndarray

Quite often, new arrays must be created from within extension-module code. Perhaps an output array is needed and you don't want the caller to have to supply it. Perhaps only a temporary array is needed to hold an intermediate calculation. Whatever the need there are simple ways to get an ndarray object of whatever data-type is needed. The most general function for doing this is `PyArray_NewFromDescr`. All array creation functions go through this heavily re-used code. Because of its flexibility, it can be somewhat confusing to use. As a result, simpler forms exist that are easier to use. These forms are part of the `PyArray_SimpleNew` family of functions, which simplify the interface by providing default values for common use cases.

Getting at ndarray memory and accessing elements of the ndarray

If `obj` is an ndarray (`PyArrayObject*`), then the data-area of the ndarray is pointed to by the `void*` pointer `PyArray_DATA(obj)` or the `char*` pointer `PyArray_BYTES(obj)`. Remember that (in general) this data-area may not be aligned according to the data-type, it may represent byte-swapped data, and/or it may not be writeable. If the data area is aligned and in native byte-order, then how to get at a specific element of the array is determined only by the array of `numpy.intp` variables, `PyArray_STRIDES(obj)`. In particular, this c-array of integers shows how many **bytes** must be added to the current element pointer to get to the next element in each dimension. For arrays less than 4-dimensions there are `PyArray_GETPTR{k}(obj, ...)` macros where `{k}` is the integer 1, 2, 3, or 4 that make using the array strides easier. The arguments `....` represent `{k}` non-negative integer indices into the array. For example, suppose `E` is a 3-dimensional ndarray. A (`void*`) pointer to the element `E[i, j, k]` is obtained as `PyArray_GETPTR3(E, i, j, k)`.

As explained previously, C-style contiguous arrays and Fortran-style contiguous arrays have particular striding patterns. Two array flags (`NPY_ARRAY_C_CONTIGUOUS` and `NPY_ARRAY_F_CONTIGUOUS`) indicate whether or not the striding pattern of a particular array matches the C-style contiguous or Fortran-style contiguous or neither. Whether or not the striding pattern matches a standard C or Fortran one can be tested Using `PyArray_IS_C_CONTIGUOUS(obj)` and `PyArray_ISFORTRAN(obj)` respectively. Most third-party libraries expect contiguous arrays. But, often it is not difficult to support general-purpose striding. I encourage you to use the striding information in your own code whenever possible, and reserve single-segment requirements for wrapping third-party code. Using the striding information provided with the ndarray rather than requiring a contiguous striding reduces copying that otherwise must be made.

8.1.5 Example

The following example shows how you might write a wrapper that accepts two input arguments (that will be converted to an array) and an output argument (that must be an array). The function returns `None` and updates the output array. Note the updated use of `WRITEBACKIFCOPY` semantics for NumPy v1.14 and above

```
static PyObject *
example_wrapper(PyObject *dummy, PyObject *args)
{
    PyObject *arg1=NULL, *arg2=NULL, *out=NULL;
    PyObject *arr1=NULL, *arr2=NULL, *oarr=NULL;

    if (!PyArg_ParseTuple(args, "OOO!", &arg1, &arg2,
        &PyArray_Type, &out)) return NULL;
```

(continues on next page)

(continued from previous page)

```

arr1 = PyArray_FROM_OTF(arg1, NPY_DOUBLE, NPY_ARRAY_IN_ARRAY);
if (arr1 == NULL) return NULL;
arr2 = PyArray_FROM_OTF(arg2, NPY_DOUBLE, NPY_ARRAY_IN_ARRAY);
if (arr2 == NULL) goto fail;
#if NPY_API_VERSION >= 0x0000000c
oarr = PyArray_FROM_OTF(out, NPY_DOUBLE, NPY_ARRAY_INOUT_ARRAY2);
#else
oarr = PyArray_FROM_OTF(out, NPY_DOUBLE, NPY_ARRAY_INOUT_ARRAY);
#endif
if (oarr == NULL) goto fail;

/* code that makes use of arguments */
/* You will probably need at least
   nd = PyArray_NDIM(<..>)    -- number of dimensions
   dims = PyArray_DIMS(<..>) -- npy_intp array of length nd
                               showing length in each dim.
   dptra = (double *)PyArray_DATA(<..>) -- pointer to data.

   If an error occurs goto fail.
*/

Py_DECREF(arr1);
Py_DECREF(arr2);
#if NPY_API_VERSION >= 0x0000000c
PyArray_ResolveWritebackIfCopy(oarr);
#endif
Py_DECREF(oarr);
Py_INCREF(Py_None);
return Py_None;

fail:
Py_XDECREF(arr1);
Py_XDECREF(arr2);
#if NPY_API_VERSION >= 0x0000000c
PyArray_DiscardWritebackIfCopy(oarr);
#endif
Py_XDECREF(oarr);
return NULL;
}

```

8.2 Using Python as glue

There is no conversation more boring than the one where everybody agrees.

— Michel de Montaigne

Duct tape is like the force. It has a light side, and a dark side, and it holds the universe together.

— Carl Zwanzig

Many people like to say that Python is a fantastic glue language. Hopefully, this Chapter will convince you that this is

true. The first adopters of Python for science were typically people who used it to glue together large application codes running on super-computers. Not only was it much nicer to code in Python than in a shell script or Perl, in addition, the ability to easily extend Python made it relatively easy to create new classes and types specifically adapted to the problems being solved. From the interactions of these early contributors, Numeric emerged as an array-like object that could be used to pass data between these applications.

As Numeric has matured and developed into NumPy, people have been able to write more code directly in NumPy. Often this code is fast-enough for production use, but there are still times that there is a need to access compiled code. Either to get that last bit of efficiency out of the algorithm or to make it easier to access widely-available codes written in C/C++ or Fortran.

This chapter will review many of the tools that are available for the purpose of accessing code written in other compiled languages. There are many resources available for learning to call other compiled libraries from Python and the purpose of this Chapter is not to make you an expert. The main goal is to make you aware of some of the possibilities so that you will know what to “Google” in order to learn more.

8.2.1 Calling other compiled libraries from Python

While Python is a great language and a pleasure to code in, its dynamic nature results in overhead that can cause some code (*i.e.* raw computations inside of for loops) to be up 10-100 times slower than equivalent code written in a static compiled language. In addition, it can cause memory usage to be larger than necessary as temporary arrays are created and destroyed during computation. For many types of computing needs, the extra slow-down and memory consumption can often not be spared (at least for time- or memory- critical portions of your code). Therefore one of the most common needs is to call out from Python code to a fast, machine-code routine (e.g. compiled using C/C++ or Fortran). The fact that this is relatively easy to do is a big reason why Python is such an excellent high-level language for scientific and engineering programming.

There are two basic approaches to calling compiled code: writing an extension module that is then imported to Python using the import command, or calling a shared-library subroutine directly from Python using the `ctypes` module. Writing an extension module is the most common method.

Warning: Calling C-code from Python can result in Python crashes if you are not careful. None of the approaches in this chapter are immune. You have to know something about the way data is handled by both NumPy and by the third-party library being used.

8.2.2 Hand-generated wrappers

Extension modules were discussed in *Writing an extension module*. The most basic way to interface with compiled code is to write an extension module and construct a module method that calls the compiled code. For improved readability, your method should take advantage of the `PyArg_ParseTuple` call to convert between Python objects and C data-types. For standard C data-types there is probably already a built-in converter. For others you may need to write your own converter and use the “O&” format string which allows you to specify a function that will be used to perform the conversion from the Python object to whatever C-structures are needed.

Once the conversions to the appropriate C-structures and C data-types have been performed, the next step in the wrapper is to call the underlying function. This is straightforward if the underlying function is in C or C++. However, in order to call Fortran code you must be familiar with how Fortran subroutines are called from C/C++ using your compiler and platform. This can vary somewhat platforms and compilers (which is another reason `f2py` makes life much simpler for interfacing Fortran code) but generally involves underscore mangling of the name and the fact that all variables are passed by reference (*i.e.* all arguments are pointers).

The advantage of the hand-generated wrapper is that you have complete control over how the C-library gets used and called which can lead to a lean and tight interface with minimal over-head. The disadvantage is that you have to write,

debug, and maintain C-code, although most of it can be adapted using the time-honored technique of “cutting-pasting-and-modifying” from other extension modules. Because, the procedure of calling out to additional C-code is fairly regimented, code-generation procedures have been developed to make this process easier. One of these code-generation techniques is distributed with NumPy and allows easy integration with Fortran and (simple) C code. This package, f2py, will be covered briefly in the next section.

8.2.3 f2py

F2py allows you to automatically construct an extension module that interfaces to routines in Fortran 77/90/95 code. It has the ability to parse Fortran 77/90/95 code and automatically generate Python signatures for the subroutines it encounters, or you can guide how the subroutine interfaces with Python by constructing an interface-definition-file (or modifying the f2py-produced one).

See the *F2PY documentation* for more information and examples.

The f2py method of linking compiled code is currently the most sophisticated and integrated approach. It allows clean separation of Python with compiled code while still allowing for separate distribution of the extension module. The only draw-back is that it requires the existence of a Fortran compiler in order for a user to install the code. However, with the existence of the free-compilers g77, gfortran, and g95, as well as high-quality commercial compilers, this restriction is not particularly onerous. In our opinion, Fortran is still the easiest way to write fast and clear code for scientific computing. It handles complex numbers, and multi-dimensional indexing in the most straightforward way. Be aware, however, that some Fortran compilers will not be able to optimize code as well as good hand-written C-code.

8.2.4 Cython

Cython is a compiler for a Python dialect that adds (optional) static typing for speed, and allows mixing C or C++ code into your modules. It produces C or C++ extensions that can be compiled and imported in Python code.

If you are writing an extension module that will include quite a bit of your own algorithmic code as well, then Cython is a good match. Among its features is the ability to easily and quickly work with multidimensional arrays.

Notice that Cython is an extension-module generator only. Unlike f2py, it includes no automatic facility for compiling and linking the extension module (which must be done in the usual fashion). It does provide a modified distutils class called `build_ext` which lets you build an extension module from a `.pyx` source. Thus, you could write in a `setup.py` file:

```
from Cython.Distutils import build_ext
from distutils.extension import Extension
from distutils.core import setup
import numpy

setup(name='mine', description='Nothing',
      ext_modules=[Extension('filter', ['filter.pyx'],
                             include_dirs=[numpy.get_include()])],
      cmdclass = {'build_ext':build_ext})
```

Adding the NumPy include directory is, of course, only necessary if you are using NumPy arrays in the extension module (which is what we assume you are using Cython for). The distutils extensions in NumPy also include support for automatically producing the extension-module and linking it from a `.pyx` file. It works so that if the user does not have Cython installed, then it looks for a file with the same file-name but a `.c` extension which it then uses instead of trying to produce the `.c` file again.

If you just use Cython to compile a standard Python module, then you will get a C extension module that typically runs a bit faster than the equivalent Python module. Further speed increases can be gained by using the `cdef` keyword to statically define C variables.

Let's look at two examples we've seen before to see how they might be implemented using Cython. These examples were compiled into extension modules using Cython 0.21.1.

Complex addition in Cython

Here is part of a Cython module named `add.pyx` which implements the complex addition functions we previously implemented using `f2py`:

```
cimport cython
cimport numpy as np
import numpy as np

# We need to initialize NumPy.
np.import_array()

#@cython.boundscheck(False)
def zadd(in1, in2):
    cdef double complex[:] a = in1.ravel()
    cdef double complex[:] b = in2.ravel()

    out = np.empty(a.shape[0], np.complex64)
    cdef double complex[:] c = out.ravel()

    for i in range(c.shape[0]):
        c[i].real = a[i].real + b[i].real
        c[i].imag = a[i].imag + b[i].imag

    return out
```

This module shows use of the `cimport` statement to load the definitions from the `numpy.pxd` header that ships with Cython. It looks like NumPy is imported twice; `cimport` only makes the NumPy C-API available, while the regular `import` causes a Python-style import at runtime and makes it possible to call into the familiar NumPy Python API.

The example also demonstrates Cython's "typed memoryviews", which are like NumPy arrays at the C level, in the sense that they are shaped and strided arrays that know their own extent (unlike a C array addressed through a bare pointer). The syntax `double complex[:]` denotes a one-dimensional array (vector) of doubles, with arbitrary strides. A contiguous array of ints would be `int[:,1]`, while a matrix of floats would be `float[:, :]`.

Shown commented is the `cython.boundscheck` decorator, which turns bounds-checking for memory view accesses on or off on a per-function basis. We can use this to further speed up our code, at the expense of safety (or a manual check prior to entering the loop).

Other than the view syntax, the function is immediately readable to a Python programmer. Static typing of the variable `i` is implicit. Instead of the view syntax, we could also have used Cython's special NumPy array syntax, but the view syntax is preferred.

Image filter in Cython

The two-dimensional example we created using Fortran is just as easy to write in Cython:

```
cimport numpy as np
import numpy as np

np.import_array()

def filter(img):
    cdef double[:, :] a = np.asarray(img, dtype=np.double)
    out = np.zeros(img.shape, dtype=np.double)
    cdef double[:, ::1] b = out

    cdef np.npy_intp i, j

    for i in range(1, a.shape[0] - 1):
        for j in range(1, a.shape[1] - 1):
            b[i, j] = (a[i, j]
                       + .5 * ( a[i-1, j] + a[i+1, j]
                              + a[i, j-1] + a[i, j+1]))
            + .25 * ( a[i-1, j-1] + a[i-1, j+1]
                    + a[i+1, j-1] + a[i+1, j+1]))

    return out
```

This 2-d averaging filter runs quickly because the loop is in C and the pointer computations are done only as needed. If the code above is compiled as a module `image`, then a 2-d image, `img`, can be filtered using this code very quickly using:

```
import image
out = image.filter(img)
```

Regarding the code, two things are of note: firstly, it is impossible to return a memory view to Python. Instead, a NumPy array `out` is first created, and then a view `b` onto this array is used for the computation. Secondly, the view `b` is typed `double[:, ::1]`. This means 2-d array with contiguous rows, i.e., C matrix order. Specifying the order explicitly can speed up some algorithms since they can skip stride computations.

Conclusion

Cython is the extension mechanism of choice for several scientific Python libraries, including Scipy, Pandas, SAGE, scikit-image and scikit-learn, as well as the XML processing library LXML. The language and compiler are well-maintained.

There are several disadvantages of using Cython:

1. When coding custom algorithms, and sometimes when wrapping existing C libraries, some familiarity with C is required. In particular, when using C memory management (`malloc` and friends), it's easy to introduce memory leaks. However, just compiling a Python module renamed to `.pyx` can already speed it up, and adding a few type declarations can give dramatic speedups in some code.
2. It is easy to lose a clean separation between Python and C which makes re-using your C-code for other non-Python-related projects more difficult.
3. The C-code generated by Cython is hard to read and modify (and typically compiles with annoying but harmless warnings).

One big advantage of Cython-generated extension modules is that they are easy to distribute. In summary, Cython is a very capable tool for either gluing C code or generating an extension module quickly and should not be over-looked. It is especially useful for people that can't or won't write C or Fortran code.

8.2.5 ctypes

`Ctypes` is a Python extension module, included in the `stdlib`, that allows you to call an arbitrary function in a shared library directly from Python. This approach allows you to interface with C-code directly from Python. This opens up an enormous number of libraries for use from Python. The drawback, however, is that coding mistakes can lead to ugly program crashes very easily (just as can happen in C) because there is little type or bounds checking done on the parameters. This is especially true when array data is passed in as a pointer to a raw memory location. The responsibility is then on you that the subroutine will not access memory outside the actual array area. But, if you don't mind living a little dangerously `ctypes` can be an effective tool for quickly taking advantage of a large shared library (or writing extended functionality in your own shared library).

Because the `ctypes` approach exposes a raw interface to the compiled code it is not always tolerant of user mistakes. Robust use of the `ctypes` module typically involves an additional layer of Python code in order to check the data types and array bounds of objects passed to the underlying subroutine. This additional layer of checking (not to mention the conversion from `ctypes` objects to C-data-types that `ctypes` itself performs), will make the interface slower than a hand-written extension-module interface. However, this overhead should be negligible if the C-routine being called is doing any significant amount of work. If you are a great Python programmer with weak C skills, `ctypes` is an easy way to write a useful interface to a (shared) library of compiled code.

To use `ctypes` you must

1. Have a shared library.
2. Load the shared library.
3. Convert the Python objects to `ctypes`-understood arguments.
4. Call the function from the library with the `ctypes` arguments.

Having a shared library

There are several requirements for a shared library that can be used with `ctypes` that are platform specific. This guide assumes you have some familiarity with making a shared library on your system (or simply have a shared library available to you). Items to remember are:

- A shared library must be compiled in a special way (e.g. using the `-shared` flag with `gcc`).
- On some platforms (e.g. Windows), a shared library requires a `.def` file that specifies the functions to be exported. For example a `mylib.def` file might contain:

```
LIBRARY mylib.dll
EXPORTS
cool_function1
cool_function2
```

Alternatively, you may be able to use the storage-class specifier `__declspec(dllexport)` in the C-definition of the function to avoid the need for this `.def` file.

There is no standard way in Python distutils to create a standard shared library (an extension module is a “special” shared library Python understands) in a cross-platform manner. Thus, a big disadvantage of `ctypes` at the time of writing this book is that it is difficult to distribute in a cross-platform manner a Python extension that uses `ctypes` and includes your own code which should be compiled as a shared library on the users system.

Loading the shared library

A simple, but robust way to load the shared library is to get the absolute path name and load it using the `cdll` object of `ctypes`:

```
lib = ctypes.cdll[<full_path_name>]
```

However, on Windows accessing an attribute of the `cdll` method will load the first DLL by that name found in the current directory or on the `PATH`. Loading the absolute path name requires a little finesse for cross-platform work since the extension of shared libraries varies. There is a `ctypes.util.find_library` utility available that can simplify the process of finding the library to load but it is not foolproof. Complicating matters, different platforms have different default extensions used by shared libraries (e.g. `.dll` – Windows, `.so` – Linux, `.dylib` – Mac OS X). This must also be taken into account if you are using `ctypes` to wrap code that needs to work on several platforms.

NumPy provides a convenience function called `ctypeslib.load_library` (`name`, `path`). This function takes the name of the shared library (including any prefix like `'lib'` but excluding the extension) and a path where the shared library can be located. It returns a `ctypes` library object or raises an `OSError` if the library cannot be found or raises an `ImportError` if the `ctypes` module is not available. (Windows users: the `ctypes` library object loaded using `load_library` is always loaded assuming `cdecl` calling convention. See the `ctypes` documentation under `ctypes.windll` and/or `ctypes.oledll` for ways to load libraries under other calling conventions).

The functions in the shared library are available as attributes of the `ctypes` library object (returned from `ctypeslib.load_library`) or as items using `lib['func_name']` syntax. The latter method for retrieving a function name is particularly useful if the function name contains characters that are not allowable in Python variable names.

Converting arguments

Python ints/longs, strings, and unicode objects are automatically converted as needed to equivalent `ctypes` arguments. The `None` object is also converted automatically to a `NULL` pointer. All other Python objects must be converted to `ctypes`-specific types. There are two ways around this restriction that allow `ctypes` to integrate with other objects.

1. Don't set the `argtypes` attribute of the function object and define an `_as_parameter_` method for the object you want to pass in. The `_as_parameter_` method must return a Python int which will be passed directly to the function.
2. Set the `argtypes` attribute to a list whose entries contain objects with a classmethod named `from_param` that knows how to convert your object to an object that `ctypes` can understand (an int/long, string, unicode, or object with the `_as_parameter_` attribute).

NumPy uses both methods with a preference for the second method because it can be safer. The `ctypes` attribute of the `ndarray` returns an object that has an `_as_parameter_` attribute which returns an integer representing the address of the `ndarray` to which it is associated. As a result, one can pass this `ctypes` attribute object directly to a function expecting a pointer to the data in your `ndarray`. The caller must be sure that the `ndarray` object is of the correct type, shape, and has the correct flags set or risk nasty crashes if the data-pointer to inappropriate arrays are passed in.

To implement the second method, NumPy provides the class-factory function `ndpointer` in the `numpy.ctypeslib` module. This class-factory function produces an appropriate class that can be placed in an `argtypes` attribute entry of a `ctypes` function. The class will contain a `from_param` method which `ctypes` will use to convert any `ndarray` passed in to the function to a `ctypes`-recognized object. In the process, the conversion will perform checking on any properties of the `ndarray` that were specified by the user in the call to `ndpointer`. Aspects of the `ndarray` that can be checked include the data-type, the number-of-dimensions, the shape, and/or the state of the flags on any array passed. The return value of the `from_param` method is the `ctypes` attribute of the array which (because it contains the `_as_parameter_` attribute pointing to the array data area) can be used by `ctypes` directly.

The `ctypes` attribute of an `ndarray` is also endowed with additional attributes that may be convenient when passing additional information about the array into a `ctypes` function. The attributes **data**, **shape**, and **strides** can provide `ctypes` compatible types corresponding to the data-area, the shape, and the strides of the array. The `data` attribute returns a

`c_void_p` representing a pointer to the data area. The shape and strides attributes each return an array of ctypes integers (or None representing a NULL pointer, if a 0-d array). The base ctype of the array is a ctype integer of the same size as a pointer on the platform. There are also methods `data_as(<ctype>)`, `shape_as(<base ctype>)`, and `strides_as(<base ctype>)`. These return the data as a ctype object of your choice and the shape/strides arrays using an underlying base type of your choice. For convenience, the `ctypeslib` module also contains `c_intp` as a ctypes integer data-type whose size is the same as the size of `c_void_p` on the platform (its value is None if ctypes is not installed).

Calling the function

The function is accessed as an attribute of or an item from the loaded shared-library. Thus, if `./mylib.so` has a function named `cool_function1`, it may be accessed either as:

```
lib = numpy.ctypeslib.load_library('mylib', '.')
func1 = lib.cool_function1 # or equivalently
func1 = lib['cool_function1']
```

In ctypes, the return-value of a function is set to be 'int' by default. This behavior can be changed by setting the `restype` attribute of the function. Use None for the `restype` if the function has no return value ('void'):

```
func1.restype = None
```

As previously discussed, you can also set the `argtypes` attribute of the function in order to have ctypes check the types of the input arguments when the function is called. Use the `ndpointer` factory function to generate a ready-made class for data-type, shape, and flags checking on your new function. The `ndpointer` function has the signature

ndpointer (*dtype=None, ndim=None, shape=None, flags=None*)

Keyword arguments with the value None are not checked. Specifying a keyword enforces checking of that aspect of the ndarray on conversion to a ctypes-compatible object. The `dtype` keyword can be any object understood as a data-type object. The `ndim` keyword should be an integer, and the `shape` keyword should be an integer or a sequence of integers. The `flags` keyword specifies the minimal flags that are required on any array passed in. This can be specified as a string of comma separated requirements, an integer indicating the requirement bits OR'd together, or a flags object returned from the `flags` attribute of an array with the necessary requirements.

Using an `ndpointer` class in the `argtypes` method can make it significantly safer to call a C function using ctypes and the data-area of an ndarray. You may still want to wrap the function in an additional Python wrapper to make it user-friendly (hiding some obvious arguments and making some arguments output arguments). In this process, the `requires` function in NumPy may be useful to return the right kind of array from a given input.

Complete example

In this example, we will demonstrate how the addition function and the filter function implemented previously using the other approaches can be implemented using ctypes. First, the C code which implements the algorithms contains the functions `zadd`, `dadd`, `sadd`, `cadd`, and `dfilter2d`. The `zadd` function is:

```
/* Add arrays of contiguous data */
typedef struct {double real; double imag;} cdouble;
typedef struct {float real; float imag;} cfloat;
void zadd(cdouble *a, cdouble *b, cdouble *c, long n)
{
    while (n-- > 0) {
        c->real = a->real + b->real;
        c->imag = a->imag + b->imag;
        a++; b++; c++;
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
}

```

with similar code for `cadd`, `dadd`, and `sadd` that handles complex float, double, and float data-types, respectively:

```

void cadd(cfloat *a, cfloat *b, cfloat *c, long n)
{
    while (n--) {
        c->real = a->real + b->real;
        c->imag = a->imag + b->imag;
        a++; b++; c++;
    }
}

void dadd(double *a, double *b, double *c, long n)
{
    while (n--) {
        *c++ = *a++ + *b++;
    }
}

void sadd(float *a, float *b, float *c, long n)
{
    while (n--) {
        *c++ = *a++ + *b++;
    }
}

```

The code `.c` file also contains the function `dfilter2d`:

```

/*
 * Assumes b is contiguous and has strides that are multiples of
 * sizeof(double)
 */
void
dfilter2d(double *a, double *b, ssize_t *astrides, ssize_t *dims)
{
    ssize_t i, j, M, N, S0, S1;
    ssize_t r, c, rm1, rp1, cp1, cm1;

    M = dims[0]; N = dims[1];
    S0 = astrides[0]/sizeof(double);
    S1 = astrides[1]/sizeof(double);
    for (i = 1; i < M - 1; i++) {
        r = i*S0;
        rp1 = r + S0;
        rm1 = r - S0;
        for (j = 1; j < N - 1; j++) {
            c = j*S1;
            cp1 = j + S1;
            cm1 = j - S1;
            b[i*N + j] = a[r + c] +
                (a[rp1 + c] + a[rm1 + c] +
                 a[r + cp1] + a[r + cm1])*0.5 +
                (a[rp1 + cp1] + a[rp1 + cm1] +
                 a[rm1 + cp1] + a[rm1 + cm1])*0.25;
        }
    }
}

```

A possible advantage this code has over the Fortran-equivalent code is that it takes arbitrarily strided (i.e. non-contiguous arrays) and may also run faster depending on the optimization capability of your compiler. But, it is an obviously more complicated than the simple code in `filter.f`. This code must be compiled into a shared library. On my Linux system this is accomplished using:

```
gcc -o code.so -shared code.c
```

Which creates a shared_library named `code.so` in the current directory. On Windows don't forget to either add `__declspec(dllexport)` in front of `void` on the line preceding each function definition, or write a `code.def` file that lists the names of the functions to be exported.

A suitable Python interface to this shared library should be constructed. To do this create a file named `interface.py` with the following lines at the top:

```
__all__ = ['add', 'filter2d']

import numpy as np
import os

_path = os.path.dirname('__file__')
lib = np.ctypeslib.load_library('code', _path)
_typedict = {'zadd' : complex, 'sadd' : np.single,
             'cadd' : np.csingle, 'dadd' : float}
for name in _typedict.keys():
    val = getattr(lib, name)
    val.restype = None
    _type = _typedict[name]
    val.argtypes = [np.ctypeslib.ndpointer(_type,
                                           flags='aligned, contiguous'),
                   np.ctypeslib.ndpointer(_type,
                                           flags='aligned, contiguous'),
                   np.ctypeslib.ndpointer(_type,
                                           flags='aligned, contiguous, '\
                                           'writeable'),
                   np.ctypeslib.c_intp]
```

This code loads the shared library named `code.{ext}` located in the same path as this file. It then adds a return type of `void` to the functions contained in the library. It also adds argument checking to the functions in the library so that `ndarrays` can be passed as the first three arguments along with an integer (large enough to hold a pointer on the platform) as the fourth argument.

Setting up the filtering function is similar and allows the filtering function to be called with `ndarray` arguments as the first two arguments and with pointers to integers (large enough to handle the strides and shape of an `ndarray`) as the last two arguments.:

```
lib.dfilter2d.restype=None
lib.dfilter2d.argtypes = [np.ctypeslib.ndpointer(float, ndim=2,
                                                  flags='aligned'),
                          np.ctypeslib.ndpointer(float, ndim=2,
                                                  flags='aligned, contiguous, '\
                                                  'writeable'),
                          ctypes.POINTER(np.ctypeslib.c_intp),
                          ctypes.POINTER(np.ctypeslib.c_intp)]
```

Next, define a simple selection function that chooses which addition function to call in the shared library based on the data-type:

```
def select(dtype):
    if dtype.char in ['?bBhHf']:
        return lib.sadd, single
    elif dtype.char in ['F']:
        return lib.cadd, csingle
    elif dtype.char in ['DG']:
        return lib.zadd, complex
    else:
        return lib.dadd, float
    return func, ntype
```

Finally, the two functions to be exported by the interface can be written simply as:

```
def add(a, b):
    requires = ['CONTIGUOUS', 'ALIGNED']
    a = np.asanyarray(a)
    func, dtype = select(a.dtype)
    a = np.require(a, dtype, requires)
    b = np.require(b, dtype, requires)
    c = np.empty_like(a)
    func(a,b,c,a.size)
    return c
```

and:

```
def filter2d(a):
    a = np.require(a, float, ['ALIGNED'])
    b = np.zeros_like(a)
    lib.dfilter2d(a, b, a.ctypes.strides, a.ctypes.shape)
    return b
```

Conclusion

Using ctypes is a powerful way to connect Python with arbitrary C-code. Its advantages for extending Python include

- clean separation of C code from Python code
 - no need to learn a new syntax except Python and C
 - allows re-use of C code
 - functionality in shared libraries written for other purposes can be obtained with a simple Python wrapper and search for the library.
- easy integration with NumPy through the ctypes attribute
- full argument checking with the ndpointer class factory

Its disadvantages include

- It is difficult to distribute an extension module made using ctypes because of a lack of support for building shared libraries in distutils.
- You must have shared-libraries of your code (no static libraries).
- Very little support for C++ code and its different library-calling conventions. You will probably need a C wrapper around C++ code to use with ctypes (or just use Boost.Python instead).

Because of the difficulty in distributing an extension module made using ctypes, f2py and Cython are still the easiest ways to extend Python for package creation. However, ctypes is in some cases a useful alternative. This should bring more features to ctypes that should eliminate the difficulty in extending Python and distributing the extension using ctypes.

8.2.6 Additional tools you may find useful

These tools have been found useful by others using Python and so are included here. They are discussed separately because they are either older ways to do things now handled by f2py, Cython, or ctypes (SWIG, PyFort) or because of a lack of reasonable documentation (SIP, Boost). Links to these methods are not included since the most relevant can be found using Google or some other search engine, and any links provided here would be quickly dated. Do not assume that inclusion in this list means that the package deserves attention. Information about these packages are collected here because many people have found them useful and we'd like to give you as many options as possible for tackling the problem of easily integrating your code.

SWIG

Simplified Wrapper and Interface Generator (SWIG) is an old and fairly stable method for wrapping C/C++-libraries to a large variety of other languages. It does not specifically understand NumPy arrays but can be made usable with NumPy through the use of typemaps. There are some sample typemaps in the `numpy/tools/swig` directory under `numpy.i` together with an example module that makes use of them. SWIG excels at wrapping large C/C++ libraries because it can (almost) parse their headers and auto-produce an interface. Technically, you need to generate a `.i` file that defines the interface. Often, however, this `.i` file can be parts of the header itself. The interface usually needs a bit of tweaking to be very useful. This ability to parse C/C++ headers and auto-generate the interface still makes SWIG a useful approach to adding functionality from C/C++ into Python, despite the other methods that have emerged that are more targeted to Python. SWIG can actually target extensions for several languages, but the typemaps usually have to be language-specific. Nonetheless, with modifications to the Python-specific typemaps, SWIG can be used to interface a library with other languages such as Perl, Tcl, and Ruby.

My experience with SWIG has been generally positive in that it is relatively easy to use and quite powerful. It has been used often before becoming more proficient at writing C-extensions. However, writing custom interfaces with SWIG is often troublesome because it must be done using the concept of typemaps which are not Python specific and are written in a C-like syntax. Therefore, other gluing strategies are preferred and SWIG would be probably considered only to wrap a very-large C/C++ library. Nonetheless, there are others who use SWIG quite happily.

SIP

SIP is another tool for wrapping C/C++ libraries that is Python specific and appears to have very good support for C++. Riverbank Computing developed SIP in order to create Python bindings to the QT library. An interface file must be written to generate the binding, but the interface file looks a lot like a C/C++ header file. While SIP is not a full C++ parser, it understands quite a bit of C++ syntax as well as its own special directives that allow modification of how the Python binding is accomplished. It also allows the user to define mappings between Python types and C/C++ structures and classes.

Boost Python

Boost is a repository of C++ libraries and Boost.Python is one of those libraries which provides a concise interface for binding C++ classes and functions to Python. The amazing part of the Boost.Python approach is that it works entirely in pure C++ without introducing a new syntax. Many users of C++ report that Boost.Python makes it possible to combine the best of both worlds in a seamless fashion. Using Boost to wrap simple C-subroutines is usually over-kill. Its primary purpose is to make C++ classes available in Python. So, if you have a set of C++ classes that need to be integrated cleanly into Python, consider learning about and using Boost.Python.

PyFort

PyFort is a nice tool for wrapping Fortran and Fortran-like C-code into Python with support for Numeric arrays. It was written by Paul Dubois, a distinguished computer scientist and the very first maintainer of Numeric (now retired). It is worth mentioning in the hopes that somebody will update PyFort to work with NumPy arrays as well which now support either Fortran or C-style contiguous arrays.

8.3 Writing your own ufunc

I have the Power!

— *He-Man*

8.3.1 Creating a new universal function

Before reading this, it may help to familiarize yourself with the basics of C extensions for Python by reading/skimming the tutorials in Section 1 of [Extending and Embedding the Python Interpreter](#) and in [How to extend NumPy](#)

The umath module is a computer-generated C-module that creates many ufuncs. It provides a great many examples of how to create a universal function. Creating your own ufunc that will make use of the ufunc machinery is not difficult either. Suppose you have a function that you want to operate element-by-element over its inputs. By creating a new ufunc you will obtain a function that handles

- broadcasting
- N-dimensional looping
- automatic type-conversions with minimal memory usage
- optional output arrays

It is not difficult to create your own ufunc. All that is required is a 1-d loop for each data-type you want to support. Each 1-d loop must have a specific signature, and only ufuncs for fixed-size data-types can be used. The function call used to create a new ufunc to work on built-in data-types is given below. A different mechanism is used to register ufuncs for user-defined data-types.

In the next several sections we give example code that can be easily modified to create your own ufuncs. The examples are successively more complete or complicated versions of the logit function, a common function in statistical modeling. Logit is also interesting because, due to the magic of IEEE standards (specifically IEEE 754), all of the logit functions created below automatically have the following behavior.

```
>>> logit(0)
-inf
>>> logit(1)
inf
```

(continues on next page)

(continued from previous page)

```
>>> logit(2)
nan
>>> logit(-2)
nan
```

This is wonderful because the function writer doesn't have to manually propagate infs or nans.

8.3.2 Example Non-ufunc extension

For comparison and general edification of the reader we provide a simple implementation of a C extension of `logit` that uses no numpy.

To do this we need two files. The first is the C file which contains the actual code, and the second is the `setup.py` file used to create the module.

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <math.h>

/*
 * spammodule.c
 * This is the C code for a non-numpy Python extension to
 * define the logit function, where  $\text{logit}(p) = \log(p/(1-p))$ .
 * This function will not work on numpy arrays automatically.
 * numpy.vectorize must be called in python to generate
 * a numpy-friendly function.
 *
 * Details explaining the Python-C API can be found under
 * 'Extending and Embedding' and 'Python/C API' at
 * docs.python.org .
 */

/* This declares the logit function */
static PyObject *spam_logit(PyObject *self, PyObject *args);

/*
 * This tells Python what methods this module has.
 * See the Python-C API for more information.
 */
static PyMethodDef SpamMethods[] = {
    {"logit",
     spam_logit,
     METH_VARARGS, "compute logit"},
    {NULL, NULL, 0, NULL}
};

/*
 * This actually defines the logit function for
 * input args from Python.
 */

static PyObject *spam_logit(PyObject *self, PyObject *args)
{
    double p;
```

(continues on next page)

(continued from previous page)

```

/* This parses the Python argument into a double */
if(!PyArg_ParseTuple(args, "d", &p)) {
    return NULL;
}

/* THE ACTUAL LOGIT FUNCTION */
p = p/(1-p);
p = log(p);

/*This builds the answer back into a python object */
return Py_BuildValue("d", p);
}

/* This initiates the module using the above definitions. */
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "spam",
    NULL,
    -1,
    SpamMethods,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};

PyMODINIT_FUNC PyInit_spam(void)
{
    PyObject *m;
    m = PyModule_Create(&moduledef);
    if (!m) {
        return NULL;
    }
    return m;
}

```

To use the `setup.py` file, place `setup.py` and `spammodule.c` in the same folder. Then `python setup.py build` will build the module to import, or `python setup.py install` will install the module to your site-packages directory.

```

'''
    setup.py file for spammodule.c

    Calling
    $python setup.py build_ext --inplace
    will build the extension library in the current file.

    Calling
    $python setup.py build
    will build a file that looks like ./build/lib*, where
    lib* is a file that begins with lib. The library will
    be in this file and end with a C library extension,
    such as .so

    Calling

```

(continues on next page)

(continued from previous page)

```

$python setup.py install
will install the module in your site-packages file.

See the distutils section of
'Extending and Embedding the Python Interpreter'
at docs.python.org for more information.
'''

from distutils.core import setup, Extension

module1 = Extension('spam', sources=['spammodule.c'],
                    include_dirs=['/usr/local/lib'])

setup(name = 'spam',
      version='1.0',
      description='This is my spam package',
      ext_modules = [module1])

```

Once the spam module is imported into python, you can call logit via `spam.logit`. Note that the function used above cannot be applied as-is to numpy arrays. To do so we must call `numpy.vectorize` on it. For example, if a python interpreter is opened in the file containing the spam library or spam has been installed, one can perform the following commands:

```

>>> import numpy as np
>>> import spam
>>> spam.logit(0)
-inf
>>> spam.logit(1)
inf
>>> spam.logit(0.5)
0.0
>>> x = np.linspace(0,1,10)
>>> spam.logit(x)
TypeError: only length-1 arrays can be converted to Python scalars
>>> f = np.vectorize(spam.logit)
>>> f(x)
array([-inf, -2.07944154, -1.25276297, -0.69314718, -0.22314355,
        0.22314355,  0.69314718,  1.25276297,  2.07944154,  inf])

```

THE RESULTING LOGIT FUNCTION IS NOT FAST! `numpy.vectorize` simply loops over `spam.logit`. The loop is done at the C level, but the numpy array is constantly being parsed and build back up. This is expensive. When the author compared `numpy.vectorize(spam.logit)` against the logit ufuncs constructed below, the logit ufuncs were almost exactly 4 times faster. Larger or smaller speedups are, of course, possible depending on the nature of the function.

8.3.3 Example NumPy ufunc for one dtype

For simplicity we give a ufunc for a single dtype, the 'f8' double. As in the previous section, we first give the .c file and then the setup.py file used to create the module containing the ufunc.

The place in the code corresponding to the actual computations for the ufunc are marked with `/* BEGIN main ufunc computation */` and `/* END main ufunc computation */`. The code in between those lines is the primary thing that must be changed to create your own ufunc.

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include "numpy/ndarraytypes.h"
#include "numpy/ufuncobject.h"
#include "numpy/npymath.h"
#include <math.h>

/*
 * single_type_logit.c
 * This is the C code for creating your own
 * NumPy ufunc for a logit function.
 *
 * In this code we only define the ufunc for
 * a single dtype. The computations that must
 * be replaced to create a ufunc for
 * a different function are marked with BEGIN
 * and END.
 *
 * Details explaining the Python-C API can be found under
 * 'Extending and Embedding' and 'Python/C API' at
 * docs.python.org .
 */

static PyMethodDef LogitMethods[] = {
    {NULL, NULL, 0, NULL}
};

/* The loop definition must precede the PyMODINIT_FUNC. */

static void double_logit(char **args, const npy_intp *dimensions,
                        const npy_intp *steps, void *data)
{
    npy_intp i;
    npy_intp n = dimensions[0];
    char *in = args[0], *out = args[1];
    npy_intp in_step = steps[0], out_step = steps[1];

    double tmp;

    for (i = 0; i < n; i++) {
        /* BEGIN main ufunc computation */
        tmp = *(double *)in;
        tmp /= 1 - tmp;
        *((double *)out) = log(tmp);
        /* END main ufunc computation */

        in += in_step;
        out += out_step;
    }
}
```

(continues on next page)

(continued from previous page)

```

}

/* This a pointer to the above function */
PyUFuncGenericFunction funcs[1] = {&double_logit};

/* These are the input and return dtypes of logit.*/
static char types[2] = {NPY_DOUBLE, NPY_DOUBLE};

static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "npufunc",
    NULL,
    -1,
    LogitMethods,
    NULL,
    NULL,
    NULL,
    NULL
};

PyMODINIT_FUNC PyInit_npufunc(void)
{
    PyObject *m, *logit, *d;
    m = PyModule_Create(&moduledef);
    if (!m) {
        return NULL;
    }

    import_array();
    import_umath();

    logit = PyUFunc_FromFuncAndData(funcs, NULL, types, 1, 1, 1,
                                    PyUFunc_None, "logit",
                                    "logit_docstring", 0);

    d = PyModule_GetDict(m);

    PyDict_SetItemString(d, "logit", logit);
    Py_DECREF(logit);

    return m;
}

```

This is a `setup.py` file for the above code. As before, the module can be build via calling `python setup.py build` at the command prompt, or installed to site-packages via `python setup.py install`. The module can also be placed into a local folder e.g. `npufunc_directory` below using `python setup.py build_ext --inplace`.

```

'''
    setup.py file for single_type_logit.c
    Note that since this is a numpy extension
    we use numpy.distutils instead of
    distutils from the python standard library.

    Calling
    $python setup.py build_ext --inplace

```

(continues on next page)

(continued from previous page)

```

will build the extension library in the npufunc_directory.

Calling
$python setup.py build
will build a file that looks like ./build/lib*, where
lib* is a file that begins with lib. The library will
be in this file and end with a C library extension,
such as .so

Calling
$python setup.py install
will install the module in your site-packages file.

See the distutils section of
'Extending and Embedding the Python Interpreter'
at docs.python.org and the documentation
on numpy.distutils for more information.
'''

def configuration(parent_package='', top_path=None):
    from numpy.distutils.misc_util import Configuration

    config = Configuration('npufunc_directory',
                           parent_package,
                           top_path)
    config.add_extension('npufunc', ['single_type_logit.c'])

    return config

if __name__ == "__main__":
    from numpy.distutils.core import setup
    setup(configuration=configuration)

```

After the above has been installed, it can be imported and used as follows.

```

>>> import numpy as np
>>> import npufunc
>>> npufunc.logit(0.5)
0.0
>>> a = np.linspace(0,1,5)
>>> npufunc.logit(a)
array([-inf, -1.09861229,  0.          ,  1.09861229,  inf])

```

8.3.4 Example NumPy ufunc with multiple dtypes

We finally give an example of a full ufunc, with inner loops for half-floats, floats, doubles, and long doubles. As in the previous sections we first give the .c file and then the corresponding setup.py file.

The places in the code corresponding to the actual computations for the ufunc are marked with `/* BEGIN main ufunc computation */` and `/* END main ufunc computation */`. The code in between those lines is the primary thing that must be changed to create your own ufunc.

```

#define PY_SSIZE_T_CLEAN
#include <Python.h>

```

(continues on next page)

(continued from previous page)

```

#include "numpy/ndarraytypes.h"
#include "numpy/ufuncobject.h"
#include "numpy/halffloat.h"
#include <math.h>

/*
 * multi_type_logit.c
 * This is the C code for creating your own
 * NumPy ufunc for a logit function.
 *
 * Each function of the form type_logit defines the
 * logit function for a different numpy dtype. Each
 * of these functions must be modified when you
 * create your own ufunc. The computations that must
 * be replaced to create a ufunc for
 * a different function are marked with BEGIN
 * and END.
 *
 * Details explaining the Python-C API can be found under
 * 'Extending and Embedding' and 'Python/C API' at
 * docs.python.org .
 */

static PyMethodDef LogitMethods[] = {
    {NULL, NULL, 0, NULL}
};

/* The loop definitions must precede the PyMODINIT_FUNC. */

static void long_double_logit(char **args, const npy_intp *dimensions,
                              const npy_intp *steps, void *data)
{
    npy_intp i;
    npy_intp n = dimensions[0];
    char *in = args[0], *out = args[1];
    npy_intp in_step = steps[0], out_step = steps[1];

    long double tmp;

    for (i = 0; i < n; i++) {
        /* BEGIN main ufunc computation */
        tmp = *(long double *)in;
        tmp /= 1 - tmp;
        *((long double *)out) = log1(tmp);
        /* END main ufunc computation */

        in += in_step;
        out += out_step;
    }
}

static void double_logit(char **args, const npy_intp *dimensions,
                         const npy_intp *steps, void *data)
{
    npy_intp i;
    npy_intp n = dimensions[0];

```

(continues on next page)

(continued from previous page)

```

char *in = args[0], *out = args[1];
numpy_intp in_step = steps[0], out_step = steps[1];

double tmp;

for (i = 0; i < n; i++) {
    /* BEGIN main ufunc computation */
    tmp = *(double *)in;
    tmp /= 1 - tmp;
    *((double *)out) = log(tmp);
    /* END main ufunc computation */

    in += in_step;
    out += out_step;
}
}

static void float_logit(char **args, const numpy_intp *dimensions,
                       const numpy_intp *steps, void *data)
{
    numpy_intp i;
    numpy_intp n = dimensions[0];
    char *in = args[0], *out = args[1];
    numpy_intp in_step = steps[0], out_step = steps[1];

    float tmp;

    for (i = 0; i < n; i++) {
        /* BEGIN main ufunc computation */
        tmp = *(float *)in;
        tmp /= 1 - tmp;
        *((float *)out) = logf(tmp);
        /* END main ufunc computation */

        in += in_step;
        out += out_step;
    }
}

static void half_float_logit(char **args, const numpy_intp *dimensions,
                             const numpy_intp *steps, void *data)
{
    numpy_intp i;
    numpy_intp n = dimensions[0];
    char *in = args[0], *out = args[1];
    numpy_intp in_step = steps[0], out_step = steps[1];

    float tmp;

    for (i = 0; i < n; i++) {
        /* BEGIN main ufunc computation */
        tmp = numpy_half_to_float(*(numpy_half *)in);
        tmp /= 1 - tmp;
        tmp = logf(tmp);
        *((numpy_half *)out) = numpy_float_to_half(tmp);
    }
}

```

(continues on next page)

(continued from previous page)

```

        /* END main ufunc computation */

        in += in_step;
        out += out_step;
    }
}

/*This gives pointers to the above functions*/
PyUFuncGenericFunction funcs[4] = {&half_float_logit,
                                   &float_logit,
                                   &double_logit,
                                   &long_double_logit};

static char types[8] = {NPY_HALF, NPY_HALF,
                        NPY_FLOAT, NPY_FLOAT,
                        NPY_DOUBLE, NPY_DOUBLE,
                        NPY_LONGDOUBLE, NPY_LONGDOUBLE};

static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "npufunc",
    NULL,
    -1,
    LogitMethods,
    NULL,
    NULL,
    NULL,
    NULL
};

PyMODINIT_FUNC PyInit_npufunc(void)
{
    PyObject *m, *logit, *d;
    m = PyModule_Create(&moduledef);
    if (!m) {
        return NULL;
    }

    import_array();
    import_umath();

    logit = PyUFunc_FromFuncAndData(funcs, NULL, types, 4, 1, 1,
                                    PyUFunc_None, "logit",
                                    "logit_docstring", 0);

    d = PyModule_GetDict(m);

    PyDict_SetItemString(d, "logit", logit);
    Py_DECREF(logit);

    return m;
}

```

This is a `setup.py` file for the above code. As before, the module can be build via calling `python setup.py build` at the command prompt, or installed to site-packages via `python setup.py install`.

```
'''
    setup.py file for multi_type_logit.c
    Note that since this is a numpy extension
    we use numpy.distutils instead of
    distutils from the python standard library.

    Calling
    $python setup.py build_ext --inplace
    will build the extension library in the current file.

    Calling
    $python setup.py build
    will build a file that looks like ./build/lib*, where
    lib* is a file that begins with lib. The library will
    be in this file and end with a C library extension,
    such as .so

    Calling
    $python setup.py install
    will install the module in your site-packages file.

    See the distutils section of
    'Extending and Embedding the Python Interpreter'
    at docs.python.org and the documentation
    on numpy.distutils for more information.
'''

def configuration(parent_package='', top_path=None):
    from numpy.distutils.misc_util import Configuration, get_info

    #Necessary for the half-float d-type.
    info = get_info('npymath')

    config = Configuration('npufunc_directory',
                           parent_package,
                           top_path)
    config.add_extension('npufunc',
                        ['multi_type_logit.c'],
                        extra_info=info)

    return config

if __name__ == "__main__":
    from numpy.distutils.core import setup
    setup(configuration=configuration)
```

After the above has been installed, it can be imported and used as follows.

```
>>> import numpy as np
>>> import npufunc
>>> npufunc.logit(0.5)
0.0
>>> a = np.linspace(0,1,5)
>>> npufunc.logit(a)
array([-inf, -1.09861229,  0.          ,  1.09861229,  inf])
```

8.3.5 Example NumPy ufunc with multiple arguments/return values

Our final example is a ufunc with multiple arguments. It is a modification of the code for a logit ufunc for data with a single dtype. We compute $(A * B, \text{logit}(A * B))$.

We only give the C code as the `setup.py` file is exactly the same as the `setup.py` file in *Example NumPy ufunc for one dtype*, except that the line

```
config.add_extension('npufunc', ['single_type_logit.c'])
```

is replaced with

```
config.add_extension('npufunc', ['multi_arg_logit.c'])
```

The C file is given below. The ufunc generated takes two arguments A and B. It returns a tuple whose first element is $A * B$ and whose second element is $\text{logit}(A * B)$. Note that it automatically supports broadcasting, as well as all other properties of a ufunc.

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include "numpy/ndarraytypes.h"
#include "numpy/ufuncobject.h"
#include "numpy/halffloat.h"
#include <math.h>

/*
 * multi_arg_logit.c
 * This is the C code for creating your own
 * NumPy ufunc for a multiple argument, multiple
 * return value ufunc. The places where the
 * ufunc computation is carried out are marked
 * with comments.
 *
 * Details explaining the Python-C API can be found under
 * 'Extending and Embedding' and 'Python/C API' at
 * docs.python.org.
 */

static PyMethodDef LogitMethods[] = {
    {NULL, NULL, 0, NULL}
};

/* The loop definition must precede the PyMODINIT_FUNC. */

static void double_logitprod(char **args, const npy_intp *dimensions,
                             const npy_intp *steps, void *data)
{
    npy_intp i;
    npy_intp n = dimensions[0];
    char *in1 = args[0], *in2 = args[1];
    char *out1 = args[2], *out2 = args[3];
    npy_intp in1_step = steps[0], in2_step = steps[1];
    npy_intp out1_step = steps[2], out2_step = steps[3];

    double tmp;

    for (i = 0; i < n; i++) {
        /* BEGIN main ufunc computation */
```

(continues on next page)

(continued from previous page)

```

    tmp = *((double *)in1;
    tmp *= *((double *)in2;
    *((double *)out1) = tmp;
    *((double *)out2) = log(tmp / (1 - tmp));
    /* END main ufunc computation */

    in1 += in1_step;
    in2 += in2_step;
    out1 += out1_step;
    out2 += out2_step;
}
}

/*This a pointer to the above function*/
PyUFuncGenericFunction funcs[1] = {&double_logitprod};

/* These are the input and return dtypes of logit.*/

static char types[4] = {NPY_DOUBLE, NPY_DOUBLE,
                        NPY_DOUBLE, NPY_DOUBLE};

static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "npufunc",
    NULL,
    -1,
    LogitMethods,
    NULL,
    NULL,
    NULL,
    NULL
};

PyMODINIT_FUNC PyInit_npufunc(void)
{
    PyObject *m, *logit, *d;
    m = PyModule_Create(&moduledef);
    if (!m) {
        return NULL;
    }

    import_array();
    import_umath();

    logit = PyUFunc_FromFuncAndData(funcs, NULL, types, 1, 2, 2,
                                    PyUFunc_None, "logit",
                                    "logit_docstring", 0);

    d = PyModule_GetDict(m);

    PyDict_SetItemString(d, "logit", logit);
    Py_DECREF(logit);

    return m;
}

```

8.3.6 Example NumPy ufunc with structured array dtype arguments

This example shows how to create a ufunc for a structured array dtype. For the example we show a trivial ufunc for adding two arrays with dtype 'u8,u8,u8'. The process is a bit different from the other examples since a call to `PyUFunc_FromFuncAndData` doesn't fully register ufuncs for custom dtypes and structured array dtypes. We need to also call `PyUFunc_RegisterLoopForDescr` to finish setting up the ufunc.

We only give the C code as the `setup.py` file is exactly the same as the `setup.py` file in [Example NumPy ufunc for one dtype](#), except that the line

```
config.add_extension('npufunc', ['single_type_logit.c'])
```

is replaced with

```
config.add_extension('npufunc', ['add_triplet.c'])
```

The C file is given below.

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include "numpy/ndarraytypes.h"
#include "numpy/ufuncobject.h"
#include "numpy/npymath.h"
#include <math.h>

/*
 * add_triplet.c
 * This is the C code for creating your own
 * NumPy ufunc for a structured array dtype.
 *
 * Details explaining the Python-C API can be found under
 * 'Extending and Embedding' and 'Python/C API' at
 * docs.python.org.
 */

static PyMethodDef StructUfuncTestMethods[] = {
    {NULL, NULL, 0, NULL}
};

/* The loop definition must precede the PyMODINIT_FUNC. */

static void add_uint64_triplet(char **args, const npy_intp *dimensions,
                              const npy_intp *steps, void *data)
{
    npy_intp i;
    npy_intp is1 = steps[0];
    npy_intp is2 = steps[1];
    npy_intp os = steps[2];
    npy_intp n = dimensions[0];
    uint64_t *x, *y, *z;

    char *i1 = args[0];
    char *i2 = args[1];
    char *op = args[2];

    for (i = 0; i < n; i++) {
        x = (uint64_t *)i1;
```

(continues on next page)

(continued from previous page)

```

        y = (uint64_t *)i2;
        z = (uint64_t *)op;

        z[0] = x[0] + y[0];
        z[1] = x[1] + y[1];
        z[2] = x[2] + y[2];

        i1 += is1;
        i2 += is2;
        op += os;
    }
}

/* This a pointer to the above function */
PyUFuncGenericFunction funcs[1] = {&add_uint64_triplet};

/* These are the input and return dtypes of add_uint64_triplet. */
static char types[3] = {NPY_UINT64, NPY_UINT64, NPY_UINT64};

static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "struct_ufunc_test",
    NULL,
    -1,
    StructUfuncTestMethods,
    NULL,
    NULL,
    NULL,
    NULL
};

PyMODINIT_FUNC PyInit_struct_ufunc_test(void)
{
    PyObject *m, *add_triplet, *d;
    PyObject *dtype_dict;
    PyArray_Descr *dtype;
    PyArray_Descr *dtypes[3];

    m = PyModule_Create(&moduledef);

    if (m == NULL) {
        return NULL;
    }

    import_array();
    import_umath();

    /* Create a new ufunc object */
    add_triplet = PyUFunc_FromFuncAndData(NULL, NULL, NULL, 0, 2, 1,
                                           PyUFunc_None, "add_triplet",
                                           "add_triplet_docstring", 0);

    dtype_dict = Py_BuildValue("[(s, s), (s, s), (s, s)]",
                                "f0", "u8", "f1", "u8", "f2", "u8");
    PyArray_DescrConverter(dtype_dict, &dtype);
    Py_DECREF(dtype_dict);

```

(continues on next page)

(continued from previous page)

```

dtypes[0] = dtype;
dtypes[1] = dtype;
dtypes[2] = dtype;

/* Register ufunc for structured dtype */
PyUFunc_RegisterLoopForDescr(add_triplet,
                             dtype,
                             &add_uint64_triplet,
                             dtypes,
                             NULL);

d = PyModule_GetDict(m);

PyDict_SetItemString(d, "add_triplet", add_triplet);
Py_DECREF(add_triplet);
return m;
}

```

The returned ufunc object is a callable Python object. It should be placed in a (module) dictionary under the same name as was used in the name argument to the ufunc-creation routine. The following example is adapted from the umath module

```

static PyUFuncGenericFunction atan2_functions[] = {
    PyUFunc_ff_f, PyUFunc_dd_d,
    PyUFunc_gg_g, PyUFunc_OO_O_method};
static void *atan2_data[] = {
    (void *)atan2f, (void *)atan2,
    (void *)atan2l, (void *)"arctan2"};
static char atan2_signatures[] = {
    NPY_FLOAT, NPY_FLOAT, NPY_FLOAT,
    NPY_DOUBLE, NPY_DOUBLE, NPY_DOUBLE,
    NPY_LONGDOUBLE, NPY_LONGDOUBLE, NPY_LONGDOUBLE,
    NPY_OBJECT, NPY_OBJECT, NPY_OBJECT};
...
/* in the module initialization code */
PyObject *f, *dict, *module;
...
dict = PyModule_GetDict(module);
...
f = PyUFunc_FromFuncAndData(atan2_functions,
    atan2_data, atan2_signatures, 4, 2, 1,
    PyUFunc_None, "arctan2",
    "a safe and correct arctan(x1/x2)", 0);
PyDict_SetItemString(dict, "arctan2", f);
Py_DECREF(f);
...

```

8.4 Beyond the Basics

The voyage of discovery is not in seeking new landscapes but in having new eyes.

— *Marcel Proust*

Discovery is seeing what everyone else has seen and thinking what no one else has thought.

— *Albert Szent-Gyorgi*

8.4.1 Iterating over elements in the array

Basic Iteration

One common algorithmic requirement is to be able to walk over all elements in a multidimensional array. The array iterator object makes this easy to do in a generic way that works for arrays of any dimension. Naturally, if you know the number of dimensions you will be using, then you can always write nested for loops to accomplish the iteration. If, however, you want to write code that works with any number of dimensions, then you can make use of the array iterator. An array iterator object is returned when accessing the `.flat` attribute of an array.

Basic usage is to call `PyArray_IterNew (array)` where `array` is an ndarray object (or one of its sub-classes). The returned object is an array-iterator object (the same object returned by the `.flat` attribute of the ndarray). This object is usually cast to `PyArrayIterObject*` so that its members can be accessed. The only members that are needed are `iter->size` which contains the total size of the array, `iter->index`, which contains the current 1-d index into the array, and `iter->dataptr` which is a pointer to the data for the current element of the array. Sometimes it is also useful to access `iter->ao` which is a pointer to the underlying ndarray object.

After processing data at the current element of the array, the next element of the array can be obtained using the macro `PyArray_ITER_NEXT (iter)`. The iteration always proceeds in a C-style contiguous fashion (last index varying the fastest). The `PyArray_ITER_GOTO (iter, destination)` can be used to jump to a particular point in the array, where `destination` is an array of `numpy_intp` data-type with space to handle at least the number of dimensions in the underlying array. Occasionally it is useful to use `PyArray_ITER_GOTO1D (iter, index)` which will jump to the 1-d index given by the value of `index`. The most common usage, however, is given in the following example.

```
PyObject *obj; /* assumed to be some ndarray object */
PyArrayIterObject *iter;
...
iter = (PyArrayIterObject *)PyArray_IterNew(obj);
if (iter == NULL) goto fail; /* Assume fail has clean-up code */
while (iter->index < iter->size) {
    /* do something with the data at it->dataptr */
    PyArray_ITER_NEXT(it);
}
...
```

You can also use `PyArrayIter_Check (obj)` to ensure you have an iterator object and `PyArray_ITER_RESET (iter)` to reset an iterator object back to the beginning of the array.

It should be emphasized at this point that you may not need the array iterator if your array is already contiguous (using an array iterator will work but will be slower than the fastest code you could write). The major purpose of array iterators is to encapsulate iteration over N-dimensional arrays with arbitrary strides. They are used in many, many places in the NumPy source code itself. If you already know your array is contiguous (Fortran or C), then simply adding the `element-size` to

a running pointer variable will step you through the array very efficiently. In other words, code like this will probably be faster for you in the contiguous case (assuming doubles).

```

numpy_intp size;
double *dptr; /* could make this any variable type */
size = PyArray_SIZE(obj);
dptr = PyArray_DATA(obj);
while(size--) {
    /* do something with the data at dptr */
    dptr++;
}

```

Iterating over all but one axis

A common algorithm is to loop over all elements of an array and perform some function with each element by issuing a function call. As function calls can be time consuming, one way to speed up this kind of algorithm is to write the function so it takes a vector of data and then write the iteration so the function call is performed for an entire dimension of data at a time. This increases the amount of work done per function call, thereby reducing the function-call over-head to a small(er) fraction of the total time. Even if the interior of the loop is performed without a function call it can be advantageous to perform the inner loop over the dimension with the highest number of elements to take advantage of speed enhancements available on micro- processors that use pipelining to enhance fundamental operations.

The `PyArray_IterAllButAxis (array, &dim)` constructs an iterator object that is modified so that it will not iterate over the dimension indicated by `dim`. The only restriction on this iterator object, is that the `PyArray_ITER_GOTO1D (it, ind)` macro cannot be used (thus flat indexing won't work either if you pass this object back to Python — so you shouldn't do this). Note that the returned object from this routine is still usually cast to `PyArrayIterObject *`. All that's been done is to modify the strides and dimensions of the returned iterator to simulate iterating over `array[...0,...]` where 0 is placed on the `dimth` dimension. If `dim` is negative, then the dimension with the largest axis is found and used.

Iterating over multiple arrays

Very often, it is desirable to iterate over several arrays at the same time. The universal functions are an example of this kind of behavior. If all you want to do is iterate over arrays with the same shape, then simply creating several iterator objects is the standard procedure. For example, the following code iterates over two arrays assumed to be the same shape and size (actually `obj1` just has to have at least as many total elements as does `obj2`):

```

/* It is already assumed that obj1 and obj2
   are ndarrays of the same shape and size.
*/
iter1 = (PyArrayIterObject *)PyArray_IterNew(obj1);
if (iter1 == NULL) goto fail;
iter2 = (PyArrayIterObject *)PyArray_IterNew(obj2);
if (iter2 == NULL) goto fail; /* assume iter1 is DECFREF'd at fail */
while (iter2->index < iter2->size) {
    /* process with iter1->dataptr and iter2->dataptr */
    PyArray_ITER_NEXT(iter1);
    PyArray_ITER_NEXT(iter2);
}

```

Broadcasting over multiple arrays

When multiple arrays are involved in an operation, you may want to use the same broadcasting rules that the math operations (*i.e.* the ufuncs) use. This can be done easily using the `PyArrayMultiIterObject`. This is the object returned from the Python command `numpy.broadcast` and it is almost as easy to use from C. The function `PyArray_MultiIterNew(n, ...)` is used (with `n` input objects in place of `...`). The input objects can be arrays or anything that can be converted into an array. A pointer to a `PyArrayMultiIterObject` is returned. Broadcasting has already been accomplished which adjusts the iterators so that all that needs to be done to advance to the next element in each array is for `PyArray_ITER_NEXT` to be called for each of the inputs. This incrementing is automatically performed by `PyArray_MultiIter_NEXT(obj)` macro (which can handle a multiterator `obj` as either a `PyArrayMultiIterObject*` or a `PyObject*`). The data from input number `i` is available using `PyArray_MultiIter_DATA(obj, i)`. An example of using this feature follows.

```
mobj = PyArray_MultiIterNew(2, obj1, obj2);
size = mobj->size;
while(size-->0) {
    ptr1 = PyArray_MultiIter_DATA(mobj, 0);
    ptr2 = PyArray_MultiIter_DATA(mobj, 1);
    /* code using contents of ptr1 and ptr2 */
    PyArray_MultiIter_NEXT(mobj);
}
```

The function `PyArray_RemoveSmallest(multi)` can be used to take a multi-iterator object and adjust all the iterators so that iteration does not take place over the largest dimension (it makes that dimension of size 1). The code being looped over that makes use of the pointers will very-likely also need the strides data for each of the iterators. This information is stored in `multi->iters[i]->strides`.

There are several examples of using the multi-iterator in the NumPy source code as it makes N-dimensional broadcasting-code very simple to write. Browse the source for more examples.

8.4.2 User-defined data-types

NumPy comes with 24 builtin data-types. While this covers a large majority of possible use cases, it is conceivable that a user may have a need for an additional data-type. There is some support for adding an additional data-type into the NumPy system. This additional data-type will behave much like a regular data-type except ufuncs must have 1-d loops registered to handle it separately. Also checking for whether or not other data-types can be cast “safely” to and from this new type or not will always return “can cast” unless you also register which types your new data-type can be cast to and from.

The NumPy source code includes an example of a custom data-type as part of its test suite. The file `_rational_tests.c.src` in the source code directory `numpy/numpy/core/src/umath/` contains an implementation of a data-type that represents a rational number as the ratio of two 32 bit integers.

Adding the new data-type

To begin to make use of the new data-type, you need to first define a new Python type to hold the scalars of your new data-type. It should be acceptable to inherit from one of the array scalars if your new type has a binary compatible layout. This will allow your new data type to have the methods and attributes of array scalars. New data- types must have a fixed memory size (if you want to define a data-type that needs a flexible representation, like a variable-precision number, then use a pointer to the object as the data-type). The memory layout of the object structure for the new Python type must be `PyObject_HEAD` followed by the fixed-size memory needed for the data- type. For example, a suitable structure for the new Python type is:

```
typedef struct {
    PyObject_HEAD;
    some_data_type obval;
    /* the name can be whatever you want */
} PySomeDataTypeObject;
```

After you have defined a new Python type object, you must then define a new `PyArray_Descr` structure whose type-object member will contain a pointer to the data-type you’ve just defined. In addition, the required functions in the “.f” member must be defined: `nonzero`, `copyswap`, `copyswapn`, `setitem`, `getitem`, and `cast`. The more functions in the “.f” member you define, however, the more useful the new data-type will be. It is very important to initialize unused functions to `NULL`. This can be achieved using `PyArray_InitArrFuncs(f)`.

Once a new `PyArray_Descr` structure is created and filled with the needed information and useful functions you call `PyArray_RegisterDataType(new_descr)`. The return value from this call is an integer providing you with a unique `type_number` that specifies your data-type. This type number should be stored and made available by your module so that other modules can use it to recognize your data-type (the other mechanism for finding a user-defined data-type number is to search based on the name of the type-object associated with the data-type using `PyArray_TypeNumFromName()`).

Registering a casting function

You may want to allow builtin (and other user-defined) data-types to be cast automatically to your data-type. In order to make this possible, you must register a casting function with the data-type you want to be able to cast from. This requires writing low-level casting functions for each conversion you want to support and then registering these functions with the data-type descriptor. A low-level casting function has the signature.

void **castfunc** (void *from, void *to, npy_intp n, void *fromarr, void *toarr)

Cast `n` elements from one type to another. The data to cast from is in a contiguous, correctly-swapped and aligned chunk of memory pointed to by `from`. The buffer to cast to is also contiguous, correctly-swapped and aligned. The `fromarr` and `toarr` arguments should only be used for flexible-element-sized arrays (string, unicode, void).

An example castfunc is:

```
static void
double_to_float(double *from, float* to, npy_intp n,
                void* ignore1, void* ignore2) {
    while (n--) {
        (*to++) = (double) *(from++);
    }
}
```

This could then be registered to convert doubles to floats using the code:

```
doub = PyArray_DescrFromType(NPY_DOUBLE);
PyArray_RegisterCastFunc(doub, NPY_FLOAT,
    (PyArray_VectorUnaryFunc *)double_to_float);
Py_DECREF(doub);
```

Registering coercion rules

By default, all user-defined data-types are not presumed to be safely castable to any builtin data-types. In addition builtin data-types are not presumed to be safely castable to user-defined data-types. This situation limits the ability of user-defined data-types to participate in the coercion system used by ufuncs and other times when automatic coercion takes place in NumPy. This can be changed by registering data-types as safely castable from a particular data-type object. The function `PyArray_RegisterCanCast` (from_descr, totype_number, scalarkind) should be used to specify that the data-type object from_descr can be cast to the data-type with type number totype_number. If you are not trying to alter scalar coercion rules, then use `NPY_NOSCALAR` for the scalarkind argument.

If you want to allow your new data-type to also be able to share in the scalar coercion rules, then you need to specify the scalarkind function in the data-type object's ".f" member to return the kind of scalar the new data-type should be seen as (the value of the scalar is available to that function). Then, you can register data-types that can be cast to separately for each scalar kind that may be returned from your user-defined data-type. If you don't register scalar coercion handling, then all of your user-defined data-types will be seen as `NPY_NOSCALAR`.

Registering a ufunc loop

You may also want to register low-level ufunc loops for your data-type so that an ndarray of your data-type can have math applied to it seamlessly. Registering a new loop with exactly the same `arg_types` signature, silently replaces any previously registered loops for that data-type.

Before you can register a 1-d loop for a ufunc, the ufunc must be previously created. Then you call `PyUFunc_RegisterLoopForType(...)` with the information needed for the loop. The return value of this function is 0 if the process was successful and -1 with an error condition set if it was not successful.

8.4.3 Subtyping the ndarray in C

One of the lesser-used features that has been lurking in Python since 2.2 is the ability to sub-class types in C. This facility is one of the important reasons for basing NumPy off of the Numeric code-base which was already in C. A sub-type in C allows much more flexibility with regards to memory management. Sub-typing in C is not difficult even if you have only a rudimentary understanding of how to create new types for Python. While it is easiest to sub-type from a single parent type, sub-typing from multiple parent types is also possible. Multiple inheritance in C is generally less useful than it is in Python because a restriction on Python sub-types is that they have a binary compatible memory layout. Perhaps for this reason, it is somewhat easier to sub-type from a single parent type.

All C-structures corresponding to Python objects must begin with `PyObject_HEAD` (or `PyObject_VAR_HEAD`). In the same way, any sub-type must have a C-structure that begins with exactly the same memory layout as the parent type (or all of the parent types in the case of multiple-inheritance). The reason for this is that Python may attempt to access a member of the sub-type structure as if it had the parent structure (*i.e.* it will cast a given pointer to a pointer to the parent structure and then dereference one of it's members). If the memory layouts are not compatible, then this attempt will cause unpredictable behavior (eventually leading to a memory violation and program crash).

One of the elements in `PyObject_HEAD` is a pointer to a type-object structure. A new Python type is created by creating a new type-object structure and populating it with functions and pointers to describe the desired behavior of the type. Typically, a new C-structure is also created to contain the instance-specific information needed for each object of the type as well. For example, `&PyArray_Type` is a pointer to the type-object table for the ndarray while a `PyArrayObject*` variable is a pointer to a particular instance of an ndarray (one of the members of the ndarray structure is, in turn, a pointer to the type-object table `&PyArray_Type`). Finally `PyType_Ready` (<pointer_to_type_object>) must be called for every new Python type.

Creating sub-types

To create a sub-type, a similar procedure must be followed except only behaviors that are different require new entries in the type- object structure. All other entries can be NULL and will be filled in by `PyType_Ready` with appropriate functions from the parent type(s). In particular, to create a sub-type in C follow these steps:

1. If needed create a new C-structure to handle each instance of your type. A typical C-structure would be:

```
typedef _new_struct {
    PyArrayObject base;
    /* new things here */
} NewArrayObject;
```

Notice that the full `PyArrayObject` is used as the first entry in order to ensure that the binary layout of instances of the new type is identical to the `PyArrayObject`.

2. Fill in a new Python type-object structure with pointers to new functions that will over-ride the default behavior while leaving any function that should remain the same unfilled (or NULL). The `tp_name` element should be different.
3. Fill in the `tp_base` member of the new type-object structure with a pointer to the (main) parent type object. For multiple-inheritance, also fill in the `tp_bases` member with a tuple containing all of the parent objects in the order they should be used to define inheritance. Remember, all parent-types must have the same C-structure for multiple inheritance to work properly.
4. Call `PyType_Ready` (<pointer_to_new_type>). If this function returns a negative number, a failure occurred and the type is not initialized. Otherwise, the type is ready to be used. It is generally important to place a reference to the new type into the module dictionary so it can be accessed from Python.

More information on creating sub-types in C can be learned by reading PEP 253 (available at <https://www.python.org/dev/peps/pep-0253>).

Specific features of ndarray sub-typing

Some special methods and attributes are used by arrays in order to facilitate the interoperation of sub-types with the base `ndarray` type.

The `__array_finalize__` method

`ndarray.__array_finalize__`

Several array-creation functions of the `ndarray` allow specification of a particular sub-type to be created. This allows sub-types to be handled seamlessly in many routines. When a sub-type is created in such a fashion, however, neither the `__new__` method nor the `__init__` method gets called. Instead, the sub-type is allocated and the appropriate instance-structure members are filled in. Finally, the `__array_finalize__` attribute is looked-up in the object dictionary. If it is present and not None, then it can be either a CObject containing a pointer to a `PyArray_FinalizeFunc` or it can be a method taking a single argument (which could be None)

If the `__array_finalize__` attribute is a CObject, then the pointer must be a pointer to a function with the signature:

```
(int) (PyArrayObject *, PyObject *)
```

The first argument is the newly created sub-type. The second argument (if not NULL) is the “parent” array (if the array was created using slicing or some other operation where a clearly-distinguishable parent is present). This routine can do anything it wants to. It should return a -1 on error and 0 otherwise.

If the `__array_finalize__` attribute is not None nor a CObject, then it must be a Python method that takes the parent array as an argument (which could be None if there is no parent), and returns nothing. Errors in this method will be caught and handled.

The `__array_priority__` attribute

`ndarray.__array_priority__`

This attribute allows simple but flexible determination of which sub-type should be considered “primary” when an operation involving two or more sub-types arises. In operations where different sub-types are being used, the sub-type with the largest `__array_priority__` attribute will determine the sub-type of the output(s). If two sub-types have the same `__array_priority__` then the sub-type of the first argument determines the output. The default `__array_priority__` attribute returns a value of 0.0 for the base `ndarray` type and 1.0 for a sub-type. This attribute can also be defined by objects that are not sub-types of the `ndarray` and can be used to determine which `__array_wrap__` method should be called for the return output.

The `__array_wrap__` method

`ndarray.__array_wrap__`

Any class or type can define this method which should take an `ndarray` argument and return an instance of the type. It can be seen as the opposite of the `__array__` method. This method is used by the ufuncs (and other NumPy functions) to allow other objects to pass through. For Python >2.4, it can also be used to write a decorator that converts a function that works only with `ndarrays` to one that works with any type with `__array__` and `__array_wrap__` methods.

NUMPY HOW TOS

These documents are intended as recipes to common tasks using NumPy. For detailed reference documentation of the functions and classes contained in the package, see the API reference.

9.1 How to write a NumPy how-to

How-tos get straight to the point – they

- answer a focused question, or
- narrow a broad question into focused questions that the user can choose among.

9.1.1 A stranger has asked for directions...

“I need to refuel my car.”

9.1.2 Give a brief but explicit answer

- *“Three kilometers/miles, take a right at Hayseed Road, it’s on your left.”*

Add helpful details for newcomers (“Hayseed Road”, even though it’s the only turnoff at three km/mi). But not irrelevant ones:

- Don’t also give directions from Route 7.
- Don’t explain why the town has only one filling station.

If there’s related background (tutorial, explanation, reference, alternative approach), bring it to the user’s attention with a link (“Directions from Route 7,” “Why so few filling stations?”).

9.1.3 Delegate

- *“Three km/mi, take a right at Hayseed Road, follow the signs.”*

If the information is already documented and succinct enough for a how-to, just link to it, possibly after an introduction (“Three km/mi, take a right”).

9.1.4 If the question is broad, narrow and redirect it

“I want to see the sights.”

The *See the sights* how-to should link to a set of narrower how-tos:

- Find historic buildings
- Find scenic lookouts
- Find the town center

and these might in turn link to still narrower how-tos – so the town center page might link to

- Find the court house
- Find city hall

By organizing how-tos this way, you not only display the options for people who need to narrow their question, you also have provided answers for users who start with narrower questions (“I want to see historic buildings,” “Which way to city hall?”).

9.1.5 If there are many steps, break them up

If a how-to has many steps:

- Consider breaking a step out into an individual how-to and linking to it.
- Include subheadings. They help readers grasp what’s coming and return where they left off.

9.1.6 Why write how-tos when there’s Stack Overflow, Reddit, Gitter...?

- We have authoritative answers.
- How-tos make the site less forbidding to non-experts.
- How-tos bring people into the site and help them discover other information that’s here .
- Creating how-tos helps us see NumPy usability through new eyes.

9.1.7 Aren’t how-tos and tutorials the same thing?

People use the terms “how-to” and “tutorial” interchangeably, but we draw a distinction, following Daniele Procida’s [taxonomy of documentation](#).

Documentation needs to meet users where they are. *How-tos* offer get-it-done information; the user wants steps to copy and doesn’t necessarily want to understand NumPy. *Tutorials* are warm-fuzzy information; the user wants a feel for some aspect of NumPy (and again, may or may not care about deeper knowledge).

We distinguish both tutorials and how-tos from *Explanations*, which are deep dives intended to give understanding rather than immediate assistance, and *References*, which give complete, authoritative data on some concrete part of NumPy (like its API) but aren’t obligated to paint a broader picture.

For more on tutorials, see [Learn to write a NumPy tutorial](#)

9.1.8 Is this page an example of a how-to?

Yes – until the sections with question-mark headings; they explain rather than giving directions. In a how-to, those would be links.

9.2 Reading and writing files

This page tackles common applications; for the full collection of I/O routines, see [routines.io](#).

9.2.1 Reading text and CSV files

With no missing values

Use `numpy.loadtxt`.

With missing values

Use `numpy.genfromtxt`.

`numpy.genfromtxt` will either

- return a masked array **masking out missing values** (if `usemask=True`), or
- **fill in the missing value** with the value specified in `filling_values` (default is `np.nan` for float, `-1` for int).

With non-whitespace delimiters

```
>>> print(open("csv.txt").read())
1, 2, 3
4,, 6
7, 8, 9
```

Masked-array output

```
>>> np.genfromtxt("csv.txt", delimiter=",", usemask=True)
masked_array(
  data=[[1.0, 2.0, 3.0],
        [4.0, --, 6.0],
        [7.0, 8.0, 9.0]],
  mask=[[False, False, False],
        [False,  True, False],
        [False, False, False]],
  fill_value=1e+20)
```

Array output

```
>>> np.genfromtxt("csv.txt", delimiter=",")
array([[ 1.,  2.,  3.],
       [ 4., nan,  6.],
       [ 7.,  8.,  9.]])
```

Array output, specified fill-in value

```
>>> np.genfromtxt("csv.txt", delimiter=",", dtype=np.int8, filling_values=99)
array([[ 1,  2,  3],
       [ 4, 99,  6],
       [ 7,  8,  9]], dtype=int8)
```

Whitespace-delimited

`numpy.genfromtxt` can also parse whitespace-delimited data files that have missing values if

- **Each field has a fixed width:** Use the width as the *delimiter* argument.

```
# File with width=4. The data does not have to be justified (for example,
# the 2 in row 1), the last column can be less than width (for example, the 6
# in row 2), and no delimiting character is required (for instance 8888 and 9
# in row 3)

>>> f = open("fixedwidth.txt").read() # doctest: +SKIP
>>> print(f) # doctest: +SKIP
1   2   3
44      6
7  88889

# Showing spaces as ^
>>> print(f.replace(" ", "^")) # doctest: +SKIP
1^^2^^^^^3
44^^^^^6
7^^88889

>>> np.genfromtxt("fixedwidth.txt", delimiter=4) # doctest: +SKIP
array([[1.000e+00, 2.000e+00, 3.000e+00],
       [4.400e+01,          nan, 6.000e+00],
       [7.000e+00, 8.888e+03, 9.000e+00]])
```

- **A special value (e.g. “x”) indicates a missing field:** Use it as the *missing_values* argument.

```
>>> print(open("nan.txt").read())
1 2 3
44 x 6
7 8888 9

>>> np.genfromtxt("nan.txt", missing_values="x")
array([[1.000e+00, 2.000e+00, 3.000e+00],
       [4.400e+01,          nan, 6.000e+00],
       [7.000e+00, 8.888e+03, 9.000e+00]])
```

- **You want to skip the rows with missing values:** Set *invalid_raise=False*.

```
>>> print(open("skip.txt").read())
1 2   3
44    6
7 888 9

>>> np.genfromtxt("skip.txt", invalid_raise=False)
__main__:1: ConversionWarning: Some errors were detected !
      Line #2 (got 2 columns instead of 3)
array([[ 1.,   2.,   3.],
       [ 7., 888.,   9.]])
```

- **The delimiter whitespace character is different from the whitespace that indicates missing data.** For instance, if columns are delimited by `\t`, then missing data will be recognized if it consists of one or more spaces.

```
>>> f = open("tabs.txt").read()
>>> print(f)
1      2      3
44          6
7      888    9

# Tabs vs. spaces
>>> print(f.replace("\t", "^"))
1^2^3
44^  ^6
7^888^9

>>> np.genfromtxt("tabs.txt", delimiter="\t", missing_values=" ")
array([[ 1.,   2.,   3.],
       [44., nan,   6.],
       [ 7., 888.,   9.]])
```

9.2.2 Read a file in .npy or .npz format

Choices:

- Use `numpy.load`. It can read files generated by any of `numpy.save`, `numpy.savez`, or `numpy.savez_compressed`.
- Use memory mapping. See `numpy.lib.format.open_memmap`.

9.2.3 Write to a file to be read back by NumPy

Binary

Use `numpy.save`, or to store multiple arrays `numpy.savez` or `numpy.savez_compressed`.

For *security and portability*, set `allow_pickle=False` unless the dtype contains Python objects, which requires pickling.

Masked arrays can't currently be saved, nor can other arbitrary array subclasses.

Human-readable

`numpy.save` and `numpy.savez` create binary files. To **write a human-readable file**, use `numpy.savetxt`. The array can only be 1- or 2-dimensional, and there's no 'savetxtz' for multiple files.

Large arrays

See *Write or read large arrays*.

9.2.4 Read an arbitrarily formatted binary file (“binary blob”)

Use a *structured array*.

Example:

The `.wav` file header is a 44-byte block preceding `data_size` bytes of the actual sound data:

<code>chunk_id</code>	<code>"RIFF"</code>
<code>chunk_size</code>	4-byte unsigned little-endian integer
<code>format</code>	<code>"WAVE"</code>
<code>fmt_id</code>	<code>"fmt "</code>
<code>fmt_size</code>	4-byte unsigned little-endian integer
<code>audio_fmt</code>	2-byte unsigned little-endian integer
<code>num_channels</code>	2-byte unsigned little-endian integer
<code>sample_rate</code>	4-byte unsigned little-endian integer
<code>byte_rate</code>	4-byte unsigned little-endian integer
<code>block_align</code>	2-byte unsigned little-endian integer
<code>bits_per_sample</code>	2-byte unsigned little-endian integer
<code>data_id</code>	<code>"data"</code>
<code>data_size</code>	4-byte unsigned little-endian integer

The `.wav` file header as a NumPy structured dtype:

```
wav_header_dtype = np.dtype([
    ("chunk_id", (bytes, 4)), # flexible-sized scalar type, item size 4
    ("chunk_size", "<u4"),    # little-endian unsigned 32-bit integer
    ("format", "S4"),         # 4-byte string, alternate spelling of (bytes, 4)
    ("fmt_id", "S4"),
    ("fmt_size", "<u4"),
    ("audio_fmt", "<u2"),      #
    ("num_channels", "<u2"),   # .. more of the same ...
    ("sample_rate", "<u4"),    #
    ("byte_rate", "<u4"),
    ("block_align", "<u2"),
    ("bits_per_sample", "<u2"),
    ("data_id", "S4"),
    ("data_size", "<u4"),
    #
    # the sound data itself cannot be represented here:
    # it does not have a fixed size
])

header = np.fromfile(f, dtype=wav_header_dtype, count=1)[0]
```

This `.wav` example is for illustration; to read a `.wav` file in real life, use Python's built-in module `wave`.

(Adapted from Pauli Virtanen, [Advanced NumPy](#), licensed under CC BY 4.0.)

9.2.5 Write or read large arrays

Arrays too large to fit in memory can be treated like ordinary in-memory arrays using memory mapping.

- Raw array data written with `numpy.ndarray.tofile` or `numpy.ndarray.tobytes` can be read with `numpy.memmap`:

```
array = numpy.memmap("mydata/myarray.arr", mode="r", dtype=np.int16, shape=(1024, 1024))
```

- Files output by `numpy.save` (that is, using the numpy format) can be read using `numpy.load` with the `mmap_mode` keyword argument:

```
large_array[some_slice] = np.load("path/to/small_array", mmap_mode="r")
```

Memory mapping lacks features like data chunking and compression; more full-featured formats and libraries usable with NumPy include:

- **HDF5**: `h5py` or `PyTables`.
- **Zarr**: [here](#).
- **NetCDF**: `scipy.io.netcdf_file`.

For tradeoffs among memmap, Zarr, and HDF5, see python-speed.com.

9.2.6 Write files for reading by other (non-NumPy) tools

Formats for **exchanging data** with other tools include HDF5, Zarr, and NetCDF (see [Write or read large arrays](#)).

9.2.7 Write or read a JSON file

NumPy arrays are **not** directly JSON serializable.

9.2.8 Save/restore using a pickle file

Avoid when possible; `pickles` are not secure against erroneous or maliciously constructed data.

Use `numpy.save` and `numpy.load`. Set `allow_pickle=False`, unless the array dtype includes Python objects, in which case pickling is required.

9.2.9 Convert from a pandas DataFrame to a NumPy array

See `pandas.DataFrame.to_numpy`.

9.2.10 Save/restore using `tofile` and `fromfile`

In general, prefer `numpy.save` and `numpy.load`.

`numpy.ndarray.tofile` and `numpy.fromfile` lose information on endianness and precision and so are unsuitable for anything but scratch storage.

9.3 How to index `ndarrays`

See also:

Indexing on `ndarrays`

This page tackles common examples. For an in-depth look into indexing, refer to *Indexing on `ndarrays`*.

9.3.1 Access specific/arbitrary rows and columns

Use *Basic indexing* features like *Slicing and striding*, and *Dimensional indexing tools*.

```
>>> a = np.arange(30).reshape(2, 3, 5)
>>> a
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14]],

       [[15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]]])
>>> a[0, 2, :]
array([10, 11, 12, 13, 14])
>>> a[0, :, 3]
array([ 3,  8, 13])
```

Note that the output from indexing operations can have different shape from the original object. To preserve the original dimensions after indexing, you can use `newaxis`. To use other such tools, refer to *Dimensional indexing tools*.

```
>>> a[0, :, 3].shape
(3,)
>>> a[0, :, 3, np.newaxis].shape
(3, 1)
>>> a[0, :, 3, np.newaxis, np.newaxis].shape
(3, 1, 1)
```

Variables can also be used to index:

```
>>> y = 0
>>> a[y, :, y+3]
array([ 3,  8, 13])
```

Refer to *Dealing with variable numbers of indices within programs* to see how to use `slice` and `Ellipsis` in your index variables.

Index columns

To index columns, you have to index the last axis. Use *Dimensional indexing tools* to get the desired number of dimensions:

```
>>> a = np.arange(24).reshape(2, 3, 4)
>>> a
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]])
>>> a[:, :, 3]
array([[ 3,  7, 11],
       [15, 19, 23]])
```

To index specific elements in each column, make use of *Advanced indexing* as below:

```
>>> arr = np.arange(3*4).reshape(3, 4)
>>> arr
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> column_indices = [[1, 3], [0, 2], [2, 2]]
>>> np.arange(arr.shape[0])
array([0, 1, 2])
>>> row_indices = np.arange(arr.shape[0])[:, np.newaxis]
>>> row_indices
array([[0],
       [1],
       [2]])
```

Use the `row_indices` and `column_indices` for advanced indexing:

```
>>> arr[row_indices, column_indices]
array([[ 1,  3],
       [ 4,  6],
       [10, 10]])
```

Index along a specific axis

Use `take`. See also `take_along_axis` and `put_along_axis`.

```
>>> a = np.arange(30).reshape(2, 3, 5)
>>> a
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14]],

       [[15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]])
>>> np.take(a, [2, 3], axis=2)
array([[ 2,  3],
       [ 7,  8],
```

(continues on next page)

(continued from previous page)

```

        [12, 13]],
        [[17, 18],
         [22, 23],
         [27, 28]]])
>>> np.take(a, [2], axis=1)
array([[10, 11, 12, 13, 14]],

       [[25, 26, 27, 28, 29]]])

```

9.3.2 Create subsets of larger matrices

Use *Slicing and striding* to access chunks of a large array:

```

>>> a = np.arange(100).reshape(10, 10)
>>> a
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
       [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
       [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
       [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
       [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
>>> a[2:5, 2:5]
array([[22, 23, 24],
       [32, 33, 34],
       [42, 43, 44]])
>>> a[2:5, 1:3]
array([[21, 22],
       [31, 32],
       [41, 42]])
>>> a[:5, :5]
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])

```

The same thing can be done with advanced indexing in a slightly more complex way. Remember that *advanced indexing creates a copy*:

```

>>> a[np.arange(5)[: , None], np.arange(5)[None, :]]
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])

```

You can also use `mgrid` to generate indices:

```

>>> indices = np.mgrid[0:6:2]
>>> indices

```

(continues on next page)

(continued from previous page)

```
array([0, 2, 4])
>>> a[:, indices]
array([[ 0,  2,  4],
       [10, 12, 14],
       [20, 22, 24],
       [30, 32, 34],
       [40, 42, 44],
       [50, 52, 54],
       [60, 62, 64],
       [70, 72, 74],
       [80, 82, 84],
       [90, 92, 94]])
```

9.3.3 Filter values

Non-zero elements

Use `nonzero` to get a tuple of array indices of non-zero elements corresponding to every dimension:

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

Use `flatnonzero` to fetch indices of elements that are non-zero in the flattened version of the ndarray:

```
>>> np.flatnonzero(z)
array([0, 1, 2, 6, 7, 8, 9])
```

Arbitrary conditions

Use `where` to generate indices based on conditions and then use *Advanced indexing*.

```
>>> a = np.arange(30).reshape(2, 3, 5)
>>> indices = np.where(a % 2 == 0)
>>> indices
(array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1]),
 array([0, 0, 0, 1, 1, 2, 2, 2, 0, 0, 1, 1, 1, 2, 2]),
 array([0, 2, 4, 1, 3, 0, 2, 4, 1, 3, 0, 2, 4, 1, 3]))
>>> a[indices]
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28])
```

Or, use *Boolean array indexing*:

```
>>> a > 14
array([[False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False]],
      [[ True,  True,  True,  True,  True],
```

(continues on next page)

(continued from previous page)

```

        [ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True]])
>>> a[a > 14]
array([15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29])

```

Replace values after filtering

Use assignment with filtering to replace desired values:

```

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

       [[  0,   1,   2,   3,   4],
        [  5,   6,   7,   8,   9]])
>>> q = p < 0
>>> q
array([[[ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True]],

       [[False, False, False, False, False],
        [False, False, False, False, False]])
>>> p[q] = 0
>>> p
array([[[0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0]],

       [[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]])

```

9.3.4 Fetch indices of max/min values

Use `argmax` and `argmin`:

```

>>> a = np.arange(30).reshape(2, 3, 5)
>>> np.argmax(a)
29
>>> np.argmin(a)
0

```

Use the `axis` keyword to get the indices of maximum and minimum values along a specific axis:

```

>>> np.argmax(a, axis=0)
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])
>>> np.argmax(a, axis=1)
array([[2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2]])
>>> np.argmax(a, axis=2)
array([[4, 4, 4],
       [4, 4, 4]])

```

(continues on next page)

(continued from previous page)

```
>>> np.argmin(a, axis=1)
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
>>> np.argmin(a, axis=2)
array([[0, 0, 0],
       [0, 0, 0]])
```

Set `keepdims` to `True` to keep the axes which are reduced in the result as dimensions with size one:

```
>>> np.argmin(a, axis=2, keepdims=True)
array([[[0],
        [0],
        [0]],
       [[0],
        [0],
        [0]]])
>>> np.argmax(a, axis=1, keepdims=True)
array([[2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2]])
```

9.3.5 Index the same ndarray multiple times efficiently

It must be kept in mind that basic indexing produces *views* and advanced indexing produces *copies*, which are computationally less efficient. Hence, you should take care to use basic indexing wherever possible instead of advanced indexing.

9.3.6 Further reading

Nicolas Rougier's [100 NumPy exercises](#) provide a good insight into how indexing is combined with other operations. Exercises 6, 8, 10, 15, 16, 19, 20, 45, 59, 64, 65, 70, 71, 72, 76, 80, 81, 84, 87, 90, 93, 94 are specially focused on indexing.

FOR DOWNSTREAM PACKAGE AUTHORS

This document aims to explain some best practices for authoring a package that depends on NumPy.

10.1 Understanding NumPy’s versioning and API/ABI stability

NumPy uses a standard, [PEP 440](#) compliant, versioning scheme: `major.minor.bugfix`. A *major* release is highly unusual (NumPy is still at version `1.xx`) and if it happens it will likely indicate an ABI break. *Minor* versions are released regularly, typically every 6 months. Minor versions contain new features, deprecations, and removals of previously deprecated code. *Bugfix* releases are made even more frequently; they do not contain any new features or deprecations.

It is important to know that NumPy, like Python itself and most other well known scientific Python projects, does **not** use semantic versioning. Instead, backwards incompatible API changes require deprecation warnings for at least two releases. For more details, see [NEP 23 — Backwards compatibility and deprecation policy](#).

NumPy has both a Python API and a C API. The C API can be used directly or via Cython, f2py, or other such tools. If your package uses the C API, then ABI (application binary interface) stability of NumPy is important. NumPy’s ABI is forward but not backward compatible. This means: binaries compiled against a given version of NumPy will still run correctly with newer NumPy versions, but not with older versions.

10.2 Testing against the NumPy main branch or pre-releases

For large, actively maintained packages that depend on NumPy, we recommend testing against the development version of NumPy in CI. To make this easy, nightly builds are provided as wheels at <https://anaconda.org/scipy-wheels-nightly/>. This helps detect regressions in NumPy that need fixing before the next NumPy release. Furthermore, we recommend to raise errors on warnings in CI for this job, either all warnings or otherwise at least `DeprecationWarning` and `FutureWarning`. This gives you an early warning about changes in NumPy to adapt your code.

10.3 Adding a dependency on NumPy

10.3.1 Build-time dependency

If a package either uses the NumPy C API directly or it uses some other tool that depends on it like Cython or Pythran, NumPy is a *build-time* dependency of the package. Because the NumPy ABI is only forward compatible, you must build your own binaries (wheels or other package formats) against the lowest NumPy version that you support (or an even older version).

Picking the correct NumPy version to build against for each Python version and platform can get complicated. There are a couple of ways to do this. Build-time dependencies are specified in `pyproject.toml` (see PEP 517), which is the file used to build wheels by PEP 517 compliant tools (e.g., when using `pip wheel`).

You can specify everything manually in `pyproject.toml`, or you can instead rely on the [oldest-supported-numpy](#) metapackage. `oldest-supported-numpy` will specify the correct NumPy version at build time for wheels, taking into account Python version, Python implementation (CPython or PyPy), operating system and hardware platform. It will specify the oldest NumPy version that supports that combination of characteristics. Note: for platforms for which NumPy provides wheels on PyPI, it will be the first version with wheels (even if some older NumPy version happens to build).

For conda-forge it's a little less complicated: there's dedicated handling for NumPy in build-time and runtime dependencies, so typically this is enough (see [here](#) for docs):

```
host:
  - numpy
run:
  - {{ pin_compatible('numpy') }}
```

Note: `pip` has `--no-use-pep517` and `--no-build-isolation` flags that may ignore `pyproject.toml` or treat it differently - if users use those flags, they are responsible for installing the correct build dependencies themselves.

`conda` will always use `--no-build-isolation`; dependencies for conda builds are given in the conda recipe (`meta.yaml`), the ones in `pyproject.toml` have no effect.

Please do not use `setup_requires` (it is deprecated and may invoke `easy_install`).

Because for NumPy you have to care about ABI compatibility, you specify the version with `==` to the lowest supported version. For your other build dependencies you can probably be looser, however it's still important to set lower and upper bounds for each dependency. It's fine to specify either a range or a specific version for a dependency like `wheel` or `setuptools`. It's recommended to set the upper bound of the range to the latest already released version of `wheel` and `setuptools` - this prevents future releases from breaking your packages on PyPI.

10.3.2 Runtime dependency & version ranges

NumPy itself and many core scientific Python packages have agreed on a schedule for dropping support for old Python and NumPy versions: [NEP 29 — Recommend Python and NumPy version support as a community policy standard](#). We recommend all packages depending on NumPy to follow the recommendations in NEP 29.

For *run-time dependencies*, specify version bounds using `install_requires` in `setup.py` (assuming you use `numpy.distutils` or `setuptools` to build).

Most libraries that rely on NumPy will not need to set an upper version bound: NumPy is careful to preserve backward-compatibility.

That said, if you are (a) a project that is guaranteed to release frequently, (b) use a large part of NumPy's API surface, and (c) is worried that changes in NumPy may break your code, you can set an upper bound of `<MAJOR.MINOR + N` with `N` no less than 3, and `MAJOR.MINOR` being the current release of NumPy⁰. If you use the NumPy C API (directly or via Cython), you can also pin the current major version to prevent ABI breakage. Note that setting an upper bound on NumPy may [affect the ability of your library to be installed alongside other, newer packages](#).

⁰ The reason for setting `N=3` is that NumPy will, on the rare occasion where it makes breaking changes, raise warnings for at least two releases. (NumPy releases about once every six months, so this translates to a window of at least a year; hence the subsequent requirement that your project releases at least on that cadence.)

Note: SciPy has more documentation on how it builds wheels and deals with its build-time and runtime dependencies [here](#).

NumPy and SciPy wheel build CI may also be useful as a reference, it can be found [here for NumPy](#) and [here for SciPy](#).

F2PY USER GUIDE AND REFERENCE MANUAL

The purpose of the F2PY *–Fortran to Python interface generator–* utility is to provide a connection between Python and Fortran. F2PY is a part of NumPy (`numpy.f2py`) and also available as a standalone command line tool.

F2PY facilitates creating/building Python C/API extension modules that make it possible

- to call Fortran 77/90/95 external subroutines and Fortran 90/95 module subroutines as well as C functions;
- to access Fortran 77 COMMON blocks and Fortran 90/95 module data, including allocatable arrays

from Python.

F2PY can be used either as a command line tool `f2py` or as a Python module `numpy.f2py`. While we try to provide the command line tool as part of the numpy setup, some platforms like Windows make it difficult to reliably put the executables on the PATH. If the `f2py` command is not available in your system, you may have to run it as a module:

```
python -m numpy.f2py
```

If you run `f2py` with no arguments, and the line `numpy Version` at the end matches the NumPy version printed from `python -m numpy.f2py`, then you can use the shorter version. If not, or if you cannot run `f2py`, you should replace all calls to `f2py` mentioned in this guide with the longer version.

11.1 Three ways to wrap - getting started

Wrapping Fortran or C functions to Python using F2PY consists of the following steps:

- Creating the so-called *signature file* that contains descriptions of wrappers to Fortran or C functions, also called the signatures of the functions. For Fortran routines, F2PY can create an initial signature file by scanning Fortran source codes and tracking all relevant information needed to create wrapper functions.
 - Optionally, F2PY-created signature files can be edited to optimize wrapper functions, which can make them “smarter” and more “Pythonic”.
- F2PY reads a signature file and writes a Python C/API module containing Fortran/C/Python bindings.
- F2PY compiles all sources and builds an extension module containing the wrappers.
 - In building the extension modules, F2PY uses `numpy.distutils` which supports a number of Fortran 77/90/95 compilers, including Gnu, Intel, Sun Fortran, SGI MIPSpro, Absoft, NAG, Compaq etc. For different build systems, see *F2PY and Build Systems*.

Depending on the situation, these steps can be carried out in a single composite command or step-by-step; in which case some steps can be omitted or combined with others.

Below, we describe three typical approaches of using F2PY. These can be read in order of increasing effort, but also cater to different access levels depending on whether the Fortran code can be freely modified.

The following example Fortran 77 code will be used for illustration, save it as `fib1.f`:

```
C FILE: FIB1.F
  SUBROUTINE FIB(A,N)
C
C   CALCULATE FIRST N FIBONACCI NUMBERS
C
  INTEGER N
  REAL*8 A(N)
  DO I=1,N
    IF (I.EQ.1) THEN
      A(I) = 0.0D0
    ELSEIF (I.EQ.2) THEN
      A(I) = 1.0D0
    ELSE
      A(I) = A(I-1) + A(I-2)
    ENDIF
  ENDDO
  END
C END FILE FIB1.F
```

Note: F2PY parses Fortran/C signatures to build wrapper functions to be used with Python. However, it is not a compiler, and does not check for additional errors in source code, nor does it implement the entire language standards. Some errors may pass silently (or as warnings) and need to be verified by the user.

11.1.1 The quick way

The quickest way to wrap the Fortran subroutine FIB for use in Python is to run

```
python -m numpy.f2py -c fib1.f -m fib1
```

or, alternatively, if the `f2py` command-line tool is available,

```
f2py -c fib1.f -m fib1
```

Note: Because the `f2py` command might not be available in all system, notably on Windows, we will use the `python -m numpy.f2py` command throughout this guide.

This command compiles and wraps `fib1.f` (`-c`) to create the extension module `fib1.so` (`-m`) in the current directory. A list of command line options can be seen by executing `python -m numpy.f2py`. Now, in Python the Fortran subroutine FIB is accessible via `fib1.fib`:

```
>>> import numpy as np
>>> import fib1
>>> print(fib1.fib.__doc__)
fib(a, [n])

Wrapper for ``fib``.

Parameters
-----
a : input rank-1 array('d') with bounds (n)
```

(continues on next page)

(continued from previous page)

```

Other Parameters
-----
n : input int, optional
    Default: len(a)

>>> a = np.zeros(8, 'd')
>>> fib1.fib(a)
>>> print(a)
[ 0.  1.  1.  2.  3.  5.  8. 13.]

```

Note:

- Note that F2PY recognized that the second argument `n` is the dimension of the first array argument `a`. Since by default all arguments are input-only arguments, F2PY concludes that `n` can be optional with the default value `len(a)`.
- One can use different values for optional `n`:

```

>>> a1 = np.zeros(8, 'd')
>>> fib1.fib(a1, 6)
>>> print(a1)
[ 0.  1.  1.  2.  3.  5.  0.  0.]

```

but an exception is raised when it is incompatible with the input array `a`:

```

>>> fib1.fib(a, 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
fib.error: (len(a)>=n) failed for 1st keyword n: fib:n=10
>>>

```

F2PY implements basic compatibility checks between related arguments in order to avoid unexpected crashes.

- When a NumPy array that is *Fortran contiguous* and has a `dtype` corresponding to a presumed Fortran type is used as an input array argument, then its C pointer is directly passed to Fortran.

Otherwise, F2PY makes a contiguous copy (with the proper `dtype`) of the input array and passes a C pointer of the copy to the Fortran subroutine. As a result, any possible changes to the (copy of) input array have no effect on the original argument, as demonstrated below:

```

>>> a = np.ones(8, 'i')
>>> fib1.fib(a)
>>> print(a)
[1 1 1 1 1 1 1 1]

```

Clearly, this is unexpected, as Fortran typically passes by reference. That the above example worked with `dtype=float` is considered accidental.

F2PY provides an `intent(inplace)` attribute that modifies the attributes of an input array so that any changes made by the Fortran routine will be reflected in the input argument. For example, if one specifies the `intent(inplace)` a directive (see *Attributes* for details), then the example above would read:

```

>>> a = np.ones(8, 'i')
>>> fib1.fib(a)
>>> print(a)
[ 0.  1.  1.  2.  3.  5.  8. 13.]

```

However, the recommended way to have changes made by Fortran subroutine propagate to Python is to use the `intent(out)` attribute. That approach is more efficient and also cleaner.

- The usage of `fib1.fib` in Python is very similar to using `FIB` in Fortran. However, using *in situ* output arguments in Python is poor style, as there are no safety mechanisms in Python to protect against wrong argument types. When using Fortran or C, compilers discover any type mismatches during the compilation process, but in Python the types must be checked at runtime. Consequently, using *in situ* output arguments in Python may lead to difficult to find bugs, not to mention the fact that the codes will be less readable when all required type checks are implemented.

Though the approach to wrapping Fortran routines for Python discussed so far is very straightforward, it has several drawbacks (see the comments above). The drawbacks are due to the fact that there is no way for F2PY to determine the actual intention of the arguments; that is, there is ambiguity in distinguishing between input and output arguments. Consequently, F2PY assumes that all arguments are input arguments by default.

There are ways (see below) to remove this ambiguity by “teaching” F2PY about the true intentions of function arguments, and F2PY is then able to generate more explicit, easier to use, and less error prone wrappers for Fortran functions.

11.1.2 The smart way

If we want to have more control over how F2PY will treat the interface to our Fortran code, we can apply the wrapping steps one by one.

- First, we create a signature file from `fib1.f` by running:

```
python -m numpy.f2py fib1.f -m fib2 -h fib1.pyf
```

The signature file is saved to `fib1.pyf` (see the `-h` flag) and its contents are shown below.

```
!      -*- f90 -*-
python module fib2 ! in
  interface ! in :fib2
    subroutine fib(a,n) ! in :fib2:fib1.f
      real*8 dimension(n) :: a
      integer optional,check(len(a)>=n),depend(a) :: n=len(a)
    end subroutine fib
  end interface
end python module fib2

! This file was auto-generated with f2py (version:2.28.198-1366).
! See http://cens.ioc.ee/projects/f2py2e/
```

- Next, we’ll teach F2PY that the argument `n` is an input argument (using the `intent(in)` attribute) and that the result, i.e., the contents of `a` after calling the Fortran function `FIB`, should be returned to Python (using the `intent(out)` attribute). In addition, an array `a` should be created dynamically using the size determined by the input argument `n` (using the `depend(n)` attribute to indicate this dependence relation).

The contents of a suitably modified version of `fib1.pyf` (saved as `fib2.pyf`) are as follows:

```
!      -*- f90 -*-
python module fib2
  interface
    subroutine fib(a,n)
      real*8 dimension(n),intent(out),depend(n) :: a
      integer intent(in) :: n
    end subroutine fib
  end interface
end python module fib2
```

- Finally, we build the extension module with `numpy.f2py` by running:

```
python -m numpy.f2py -c fib2.pyf fib1.f
```

In Python:

```
>>> import fib2
>>> print(fib2.fib.__doc__)
a = fib(n)

Wrapper for ``fib``.

Parameters
-----
n : input int

Returns
-----
a : rank-1 array('d') with bounds (n)

>>> print(fib2.fib(8))
[ 0.  1.  1.  2.  3.  5.  8. 13.]
```

Note:

- The signature of `fib2.fib` now more closely corresponds to the intention of the Fortran subroutine `FIB`: given the number `n`, `fib2.fib` returns the first `n` Fibonacci numbers as a NumPy array. The new Python signature `fib2.fib` also rules out the unexpected behaviour in `fib1.fib`.
- Note that by default, using a single `intent(out)` also implies `intent(hide)`. Arguments that have the `intent(hide)` attribute specified will not be listed in the argument list of a wrapper function.

For more details, see [Signature file](#).

11.1.3 The quick and smart way

The “smart way” of wrapping Fortran functions, as explained above, is suitable for wrapping (e.g. third party) Fortran codes for which modifications to their source codes are not desirable nor even possible.

However, if editing Fortran codes is acceptable, then the generation of an intermediate signature file can be skipped in most cases. F2PY specific attributes can be inserted directly into Fortran source codes using F2PY directives. A F2PY directive consists of special comment lines (starting with `Cf2py` or `!f2py`, for example) which are ignored by Fortran compilers but interpreted by F2PY as normal lines.

Consider a modified version of the previous Fortran code with F2PY directives, saved as `fib3.f`:

```
C FILE: FIB3.F
C      SUBROUTINE FIB(A,N)
C
C      CALCULATE FIRST N FIBONACCI NUMBERS
C
C      INTEGER N
C      REAL*8 A(N)
Cf2py intent(in) n
Cf2py intent(out) a
```

(continues on next page)

(continued from previous page)

```
Cf2py depend(n) a
    DO I=1,N
        IF (I.EQ.1) THEN
            A(I) = 0.0D0
        ELSEIF (I.EQ.2) THEN
            A(I) = 1.0D0
        ELSE
            A(I) = A(I-1) + A(I-2)
        ENDIF
    ENDDO
END
C END FILE FIB3.F
```

Building the extension module can be now carried out in one command:

```
python -m numpy.f2py -c -m fib3 fib3.f
```

Notice that the resulting wrapper to FIB is as “smart” (unambiguous) as in the previous case:

```
>>> import fib3
>>> print(fib3.fib.__doc__)
a = fib(n)

Wrapper for ``fib``.

Parameters
-----
n : input int

Returns
-----
a : rank-1 array('d') with bounds (n)

>>> print(fib3.fib(8))
[ 0.  1.  1.  2.  3.  5.  8. 13.]
```

11.2 F2PY user guide

11.2.1 Using F2PY

This page contains a reference to all command-line options for the `f2py` command, as well as a reference to internal functions of the `numpy.f2py` module.

Using f2py as a command-line tool

When used as a command-line tool, f2py has three major modes, distinguished by the usage of `-c` and `-h` switches.

1. Signature file generation

To scan Fortran sources and generate a signature file, use

```
f2py -h <filename.pyf> <options> <fortran files> \
[[ only: <fortran functions> : ] \
[ skip: <fortran functions> : ]]... \
[<fortran files> ...]
```

Note: A Fortran source file can contain many routines, and it is often not necessary to allow all routines to be usable from Python. In such cases, either specify which routines should be wrapped (in the `only: .. :` part) or which routines F2PY should ignore (in the `skip: .. :` part).

If `<filename.pyf>` is specified as `stdout`, then signatures are written to standard output instead of a file.

Among other options (see below), the following can be used in this mode:

--overwrite-signature

Overwrites an existing signature file.

2. Extension module construction

To construct an extension module, use

```
f2py -m <modulename> <options> <fortran files> \
[[ only: <fortran functions> : ] \
[ skip: <fortran functions> : ]]... \
[<fortran files> ...]
```

The constructed extension module is saved as `<modulename>module.c` to the current directory.

Here `<fortran files>` may also contain signature files. Among other options (see below), the following options can be used in this mode:

--debug-capi

Adds debugging hooks to the extension module. When using this extension module, various diagnostic information about the wrapper is written to the standard output, for example, the values of variables, the steps taken, etc.

-include '<includefile>'

Add a CPP `#include` statement to the extension module source. `<includefile>` should be given in one of the following forms

```
"filename.ext"
<filename.ext>
```

The include statement is inserted just before the wrapper functions. This feature enables using arbitrary C functions (defined in `<includefile>`) in F2PY generated wrappers.

Note: This option is deprecated. Use `usercode` statement to specify C code snippets directly in signature files.

--[no-]wrap-functions

Create Fortran subroutine wrappers to Fortran functions. `--wrap-functions` is default because it ensures maximum portability and compiler independence.

--include-paths <path1>:<path2>:...

Search include files from given directories.

--help-link [<list of resources names>]

List system resources found by `numpy_distutils/system_info.py`. For example, try `f2py --help-link lapack_opt`.

3. Building a module

To build an extension module, use

```
f2py -c <options> <fortran files> \
[[ only: <fortran functions> : ] \
[ skip: <fortran functions> : ]]... \
[ <fortran/c source files> ] [ <.o, .a, .so files> ]
```

If `<fortran files>` contains a signature file, then the source for an extension module is constructed, all Fortran and C sources are compiled, and finally all object and library files are linked to the extension module `<modulename>.so` which is saved into the current directory.

If `<fortran files>` does not contain a signature file, then an extension module is constructed by scanning all Fortran source codes for routine signatures, before proceeding to build the extension module.

Among other options (see below) and options described for previous modes, the following options can be used in this mode:

--help-fcompiler

List the available Fortran compilers.

--help-compiler [deprecated]

List the available Fortran compilers.

--fcompiler=<Vendor>

Specify a Fortran compiler type by vendor.

--f77exec=<path>

Specify the path to a F77 compiler

--fcompiler-exec=<path> [deprecated]

Specify the path to a F77 compiler

--f90exec=<path>

Specify the path to a F90 compiler

--f90compiler-exec=<path> [deprecated]

Specify the path to a F90 compiler

--f77flags=<string>

Specify F77 compiler flags

--f90flags=<string>

Specify F90 compiler flags

--opt=<string>
Specify optimization flags

--arch=<string>
Specify architecture specific optimization flags

--noopt
Compile without optimization flags

--noarch
Compile without arch-dependent optimization flags

--debug
Compile with debugging information

-l<libname>
Use the library <libname> when linking.

-D<macro> [=<defn=1>]
Define macro <macro> as <defn>.

-U<macro>
Define macro <macro>

-I<dir>
Append directory <dir> to the list of directories searched for include files.

-L<dir>
Add directory <dir> to the list of directories to be searched for -l.

link-<resource>
Link the extension module with <resource> as defined by `numpy.distutils.system_info.py`. E.g. to link with optimized LAPACK libraries (vecLib on MacOSX, ATLAS elsewhere), use `--link-lapack_opt`. See also `--help-link` switch.

Note: The `f2py -c` option must be applied either to an existing `.pyf` file (plus the source/object/library files) or one must specify the `-m <module name>` option (plus the sources/object/library files). Use one of the following options:

```
f2py -c -m fib1 fib1.f
```

or

```
f2py -m fib1 fib1.f -h fib1.pyf
f2py -c fib1.pyf fib1.f
```

For more information, see the [Building C and C++ Extensions](#) Python documentation for details.

When building an extension module, a combination of the following macros may be required for non-gcc Fortran compilers:

```
-DPREPEND_FORTRAN
-DNO_APPEND_FORTRAN
-DUPPERCASE_FORTRAN
```

To test the performance of F2PY generated interfaces, use `-DF2PY_REPORT_ATEXIT`. Then a report of various timings is printed out at the exit of Python. This feature may not work on all platforms, and currently only Linux is supported.

To see whether F2PY generated interface performs copies of array arguments, use `-DF2PY_REPORT_ON_ARRAY_COPY=<int>`. When the size of an array argument is larger than `<int>`, a message about the copying is sent to `stderr`.

Other options

-m <modulename>

Name of an extension module. Default is `untitled`.

Warning: Don't use this option if a signature file (`*.pyf`) is used.

--[no-]lower

Do [not] lower the cases in `<fortran files>`. By default, `--lower` is assumed with `-h` switch, and `--no-lower` without the `-h` switch.

-include<header>

Writes additional headers in the C wrapper, can be passed multiple times, generates `#include <header>` each time. Note that this is meant to be passed in single quotes and without spaces, for example `'-include<stdbool.h>'`

--build-dir <dirname>

All F2PY generated files are created in `<dirname>`. Default is `tempfile.mkdtemp()`.

--quiet

Run quietly.

--verbose

Run with extra verbosity.

--skip-empty-wrappers

Do not generate wrapper files unless required by the inputs. This is a backwards compatibility flag to restore pre 1.22.4 behavior.

-v

Print the F2PY version and exit.

Execute `f2py` without any options to get an up-to-date list of available options.

Python module `numpy.f2py`

The `f2py` program is written in Python and can be run from inside your code to compile Fortran code at runtime, as follows:

```
from numpy import f2py
with open("add.f") as sourcefile:
    sourcecode = sourcefile.read()
f2py.compile(sourcecode, modulename='add')
import add
```

The source string can be any valid Fortran code. If you want to save the extension-module source code then a suitable file-name can be provided by the `source_fn` keyword to the `compile` function.

When using `numpy.f2py` as a module, the following functions can be invoked.

Warning: The current Python interface to the `f2py` module is not mature and may change in the future.

Fortran to Python Interface Generator.

```
numpy.f2py.compile(source, modulename='untitled', extra_args="", verbose=True, source_fn=None,
                  extension='.f', full_output=False)
```

Build extension module from a Fortran 77 source string with `f2py`.

Parameters

source

[str or bytes] Fortran source of module / subroutine to compile

Changed in version 1.16.0: Accept str as well as bytes

modulename

[str, optional] The name of the compiled python module

extra_args

[str or list, optional] Additional parameters passed to `f2py`

Changed in version 1.16.0: A list of args may also be provided.

verbose

[bool, optional] Print `f2py` output to screen

source_fn

[str, optional] Name of the file where the fortran source is written. The default is to use a temporary file with the extension provided by the `extension` parameter

extension

[{'f', 'f90'}, optional] Filename extension if `source_fn` is not provided. The extension tells which fortran standard is used. The default is `.f`, which implies F77 standard.

New in version 1.11.0.

full_output

[bool, optional] If True, return a `subprocess.CompletedProcess` containing the std-out and stderr of the compile process, instead of just the status code.

New in version 1.20.0.

Returns

result

[int or `subprocess.CompletedProcess`] 0 on success, or a `subprocess.CompletedProcess` if `full_output=True`

Examples

```
>>> import numpy.f2py
>>> fsource = '''
...     subroutine foo
...     print*, "Hello world!"
...     end
... '''
>>> numpy.f2py.compile(fsource, modulename='hello', verbose=0)
0
>>> import hello
>>> hello.foo()
Hello world!
```

`numpy.f2py.get_include()`

Return the directory that contains the `fortranobject.c` and `.h` files.

Note: This function is not needed when building an extension with `numpy.distutils` directly from `.f` and/or `.pyf` files in one go.

Python extension modules built with `f2py`-generated code need to use `fortranobject.c` as a source file, and include the `fortranobject.h` header. This function can be used to obtain the directory containing both of these files.

Returns

include_path

[str] Absolute path to the directory containing `fortranobject.c` and `fortranobject.h`.

See also:

`numpy.get_include`

function that returns the numpy include directory

Notes

New in version 1.21.1.

Unless the build system you are using has specific support for f2py, building a Python extension using a .pyf signature file is a two-step process. For a module `mymod`:

- Step 1: run `python -m numpy.f2py mymod.pyf --quiet`. This generates `_mymodmodule.c` and (if needed) `_fblas-f2pywrappers.f` files next to `mymod.pyf`.
- Step 2: build your Python extension module. This requires the following source files:
 - `_mymodmodule.c`
 - `_mymod-f2pywrappers.f` (if it was generated in Step 1)
 - `fortranobject.c`

`numpy.f2py.run_main(comline_list)`

Equivalent to running:

```
f2py <args>
```

where `<args>=string.join(<list>, ' ')`, but in Python. Unless `-h` is used, this function returns a dictionary containing information on generated modules and their dependencies on source files.

You cannot build extension modules with this function, that is, using `-c` is not allowed. Use the `compile` command instead.

Examples

The command `f2py -m scalar scalar.f` can be executed from Python as follows.

```
>>> import numpy.f2py
>>> r = numpy.f2py.run_main(['-m', 'scalar', 'doc/source/f2py/scalar.f'])
Reading fortran codes...
    Reading file 'doc/source/f2py/scalar.f' (format:fix,strict)
Post-processing...
    Block: scalar
                                Block: FOO
Building modules...
    Building module "scalar"...
    Wrote C/API module "scalar" to file "./scalarmodule.c"
>>> print(r)
{'scalar': {'h': ['/home/users/pearu/src_cvs/f2py/src/fortranobject.h'],
            'csrc': ['./scalarmodule.c',
                    '/home/users/pearu/src_cvs/f2py/src/fortranobject.c']}}
```

Automatic extension module generation

If you want to distribute your f2py extension module, then you only need to include the .pyf file and the Fortran code. The distutils extensions in NumPy allow you to define an extension module entirely in terms of this interface file. A valid setup.py file allowing distribution of the add.f module (as part of the package f2py_examples so that it would be loaded as f2py_examples.add) is:

```
def configuration(parent_package='', top_path=None)
    from numpy.distutils.misc_util import Configuration
    config = Configuration('f2py_examples', parent_package, top_path)
    config.add_extension('add', sources=['add.pyf', 'add.f'])
    return config

if __name__ == '__main__':
    from numpy.distutils.core import setup
    setup(**configuration(top_path='').todict())
```

Installation of the new package is easy using:

```
pip install .
```

assuming you have the proper permissions to write to the main site-packages directory for the version of Python you are using. For the resulting package to work, you need to create a file named __init__.py (in the same directory as add.pyf). Notice the extension module is defined entirely in terms of the add.pyf and add.f files. The conversion of the .pyf file to a .c file is handled by numpy.distutils.

11.2.2 F2PY examples

Below are some examples of F2PY usage. This list is not comprehensive, but can be used as a starting point when wrapping your own code.

F2PY walkthrough: a basic extension module

Creating source for a basic extension module

Consider the following subroutine, contained in a file named add.f

```
C
    SUBROUTINE ZADD (A,B,C,N)
C
    DOUBLE COMPLEX A(*)
    DOUBLE COMPLEX B(*)
    DOUBLE COMPLEX C(*)
    INTEGER N
    DO 20 J = 1, N
        C(J) = A(J) + B(J)
20    CONTINUE
    END
```

This routine simply adds the elements in two contiguous arrays and places the result in a third. The memory for all three arrays must be provided by the calling routine. A very basic interface to this routine can be automatically generated by f2py:

```
python -m numpy.f2py -m add add.f
```


This command will produce an extension module named `addmodule.c` in the current directory. This extension module can now be compiled and used from Python just like any other extension module.

Creating a compiled extension module

Note: This usage depends heavily on `numpy.distutils`, see *F2PY and Build Systems* for more details.

You can also get `f2py` to both compile `add.f` along with the produced extension module leaving only a shared-library extension file that can be imported from Python:

```
python -m numpy.f2py -c -m add add.f
```

This command produces a Python extension module compatible with your platform. This module may then be imported from Python. It will contain a method for each subroutine in `add`. The docstring of each method contains information about how the module method may be called:

```
>>> import add
>>> print(add.zadd.__doc__)
zadd(a,b,c,n)

Wrapper for ``zadd``.

Parameters
-----
a : input rank-1 array('D') with bounds (*)
b : input rank-1 array('D') with bounds (*)
c : input rank-1 array('D') with bounds (*)
n : input int
```

Improving the basic interface

The default interface is a very literal translation of the Fortran code into Python. The Fortran array arguments are converted to NumPy arrays and the integer argument should be mapped to a C integer. The interface will attempt to convert all arguments to their required types (and shapes) and issue an error if unsuccessful. However, because `f2py` knows nothing about the semantics of the arguments (such that `C` is an output and `n` should really match the array sizes), it is possible to abuse this function in ways that can cause Python to crash. For example:

```
>>> add.zadd([1, 2, 3], [1, 2], [3, 4], 1000)
```

will cause a program crash on most systems. Under the hood, the lists are being converted to arrays but then the underlying `add` function is told to cycle way beyond the borders of the allocated memory.

In order to improve the interface, `f2py` supports directives. This is accomplished by constructing a signature file. It is usually best to start from the interfaces that `f2py` produces in that file, which correspond to the default behavior. To get `f2py` to generate the interface file use the `-h` option:

```
python -m numpy.f2py -h add.pyf -m add add.f
```

This command creates the `add.pyf` file in the current directory. The section of this file corresponding to `zadd` is:

```
subroutine zadd(a,b,c,n) ! in :add:add.f
  double complex dimension(*) :: a
  double complex dimension(*) :: b
  double complex dimension(*) :: c
  integer :: n
end subroutine zadd
```

By placing intent directives and checking code, the interface can be cleaned up quite a bit so the Python module method is both easier to use and more robust to malformed inputs.

```
subroutine zadd(a,b,c,n) ! in :add:add.f
  double complex dimension(n) :: a
  double complex dimension(n) :: b
  double complex intent(out),dimension(n) :: c
  integer intent(hide),depend(a) :: n=len(a)
end subroutine zadd
```

The intent directive, `intent(out)` is used to tell f2py that `c` is an output variable and should be created by the interface before being passed to the underlying code. The `intent(hide)` directive tells f2py to not allow the user to specify the variable, `n`, but instead to get it from the size of `a`. The `depend(a)` directive is necessary to tell f2py that the value of `n` depends on the input `a` (so that it won't try to create the variable `n` until the variable `a` is created).

After modifying `add.pyf`, the new Python module file can be generated by compiling both `add.f` and `add.pyf`:

```
python -m numpy.f2py -c add.pyf add.f
```

The new interface's docstring is:

```
>>> import add
>>> print(add.zadd.__doc__)
c = zadd(a,b)

Wrapper for ``zadd``.

Parameters
-----
a : input rank-1 array('D') with bounds (n)
b : input rank-1 array('D') with bounds (n)

Returns
-----
c : rank-1 array('D') with bounds (n)
```

Now, the function can be called in a much more robust way:

```
>>> add.zadd([1, 2, 3], [4, 5, 6])
array([5.+0.j, 7.+0.j, 9.+0.j])
```

Notice the automatic conversion to the correct format that occurred.

Inserting directives in Fortran source

The robust interface of the previous section can also be generated automatically by placing the variable directives as special comments in the original Fortran code.

Note: For projects where the Fortran code is being actively developed, this may be preferred.

Thus, if the source code is modified to contain:

```
C
    SUBROUTINE ZADD(A,B,C,N)
C
CF2PY INTENT(OUT) :: C
CF2PY INTENT(HIDE) :: N
```

(continues on next page)

(continued from previous page)

```

CF2PY DOUBLE COMPLEX :: A(N)
CF2PY DOUBLE COMPLEX :: B(N)
CF2PY DOUBLE COMPLEX :: C(N)
      DOUBLE COMPLEX A(*)
      DOUBLE COMPLEX B(*)
      DOUBLE COMPLEX C(*)
      INTEGER N
      DO 20 J = 1, N
        C(J) = A(J) + B(J)
20    CONTINUE
      END

```

Then, one can compile the extension module using:

```
python -m numpy.f2py -c -m add add.f
```

The resulting signature for the function `add.zadd` is exactly the same one that was created previously. If the original source code had contained `A(N)` instead of `A(*)` and so forth with `B` and `C`, then nearly the same interface can be obtained by placing the `INTENT(OUT) :: C` comment line in the source code. The only difference is that `N` would be an optional input that would default to the length of `A`.

A filtering example

This example shows a function that filters a two-dimensional array of double precision floating-point numbers using a fixed averaging filter. The advantage of using Fortran to index into multi-dimensional arrays should be clear from this example.

```

C
      SUBROUTINE DFILTER2D(A,B,M,N)
C
      DOUBLE PRECISION A(M,N)
      DOUBLE PRECISION B(M,N)
      INTEGER N, M
CF2PY INTENT(OUT) :: B
CF2PY INTENT(HIDE) :: N
CF2PY INTENT(HIDE) :: M
      DO 20 I = 2,M-1
        DO 40 J = 2,N-1
          B(I,J) = A(I,J) +
&              (A(I-1,J)+A(I+1,J) +
&              A(I,J-1)+A(I,J+1) ) *0.5D0 +
&              (A(I-1,J-1) + A(I-1,J+1) +
&              A(I+1,J-1) + A(I+1,J+1) ) *0.25D0
40    CONTINUE
20    CONTINUE
      END

```

This code can be compiled and linked into an extension module named `filter` using:

```
python -m numpy.f2py -c -m filter filter.f
```

This will produce an extension module in the current directory with a method named `dfilter2d` that returns a filtered version of the input.

depends keyword example

Consider the following code, saved in the file `myroutine.f90`:

```
subroutine s(n, m, c, x)
  implicit none
  integer, intent(in) :: n, m
  real(kind=8), intent(out), dimension(n,m) :: x
  real(kind=8), intent(in) :: c(:)

  x = 0.0d0
  x(1, 1) = c(1)

end subroutine s
```

Wrapping this with `python -m numpy.f2py -c myroutine.f90 -m myroutine`, we can do the following in Python:

```
>>> import numpy as np
>>> import myroutine
>>> x = myroutine.s(2, 3, np.array([5, 6, 7]))
>>> x
array([[5., 0., 0.],
       [0., 0., 0.]])
```

Now, instead of generating the extension module directly, we will create a signature file for this subroutine first. This is a common pattern for multi-step extension module generation. In this case, after running

```
python -m numpy.f2py myroutine.f90 -h myroutine.pyf
```

the following signature file is generated:

```
!      -*- f90 -*-
! Note: the context of this file is case sensitive.

python module myroutine ! in
  interface ! in :myroutine
    subroutine s(n,m,c,x) ! in :myroutine:myroutine.f90
      integer intent(in) :: n
      integer intent(in) :: m
      real(kind=8) dimension(:), intent(in) :: c
      real(kind=8) dimension(n,m), intent(out), depend(m,n) :: x
    end subroutine s
  end interface
end python module myroutine

! This file was auto-generated with f2py (version:1.23.0.dev0+120.g4da01f42d).
! See:
! https://web.archive.org/web/20140822061353/http://cens.ioc.ee/projects/f2py2e
```

Now, if we run `python -m numpy.f2py -c myroutine.pyf myroutine.f90` we see an error; note that the signature file included a `depend(m,n)` statement for `x` which is not necessary. Indeed, editing the file above to read

```
!      -*- f90 -*-
! Note: the context of this file is case sensitive.

python module myroutine ! in
```

(continues on next page)

(continued from previous page)

```

interface ! in :myroutine
  subroutine s(n,m,c,x) ! in :myroutine:myroutine.f90
    integer intent(in) :: n
    integer intent(in) :: m
    real(kind=8) dimension(:), intent(in) :: c
    real(kind=8) dimension(n,m), intent(out) :: x
  end subroutine s
end interface
end python module myroutine

! This file was auto-generated with f2py (version:1.23.0.dev0+120.g4da01f42d).
! See:
! https://web.archive.org/web/20140822061353/http://cens.ioc.ee/projects/f2py2e

```

and running `f2py -c myroutine.pyf myroutine.f90` yields correct results.

Read more

- Wrapping C codes using f2py
- F2py section on the SciPy Cookbook
- F2py example: Interactive System for Ice sheet Simulation
- “Interfacing With Other Languages” section on the SciPy Cookbook.

11.3 F2PY reference manual

11.3.1 Signature file

The interface definition file (.pyf) is how you can fine-tune the interface between Python and Fortran. The syntax specification for signature files (.pyf files) is modeled on the Fortran 90/95 language specification. Almost all Fortran 90/95 standard constructs are understood, both in free and fixed format (recall that Fortran 77 is a subset of Fortran 90/95). F2PY introduces some extensions to the Fortran 90/95 language specification that help in the design of the Fortran to Python interface, making it more “Pythonic”.

Signature files may contain arbitrary Fortran code so that any Fortran 90/95 codes can be treated as signature files. F2PY silently ignores Fortran constructs that are irrelevant for creating the interface. However, this also means that syntax errors are not caught by F2PY and will only be caught when the library is built.

Note: Currently, F2PY may fail with valid Fortran constructs, such as intrinsic modules. If this happens, you can check the [NumPy GitHub issue tracker](#) for possible workarounds or work-in-progress ideas.

In general, the contents of the signature files are case-sensitive. When scanning Fortran codes to generate a signature file, F2PY lowers all cases automatically except in multi-line blocks or when the `--no-lower` option is used.

The syntax of signature files is presented below.

Signature files syntax

Python module block

A signature file may contain one (recommended) or more `python module` blocks. The `python module` block describes the contents of a Python/C extension module `<modulename>module.c` that F2PY generates.

Warning: Exception: if `<modulename>` contains a substring `__user__`, then the corresponding `python module` block describes the signatures of call-back functions (see [Call-back arguments](#)).

A `python module` block has the following structure:

```
python module <modulename>
  [<usercode statement>]...
  [
    interface
      <usercode statement>
      <Fortran block data signatures>
      <Fortran/C routine signatures>
    end [interface]
  ]...
  [
    interface
      module <F90 modulename>
        [<F90 module data type declarations>]
        [<F90 module routine signatures>]
      end [module [<F90 modulename>]]
    end [interface]
  ]...
end [python module [<modulename>]]
```

Here brackets `[]` indicate an optional section, dots `...` indicate one or more of a previous section. So, `[]...` is to be read as zero or more of a previous section.

Fortran/C routine signatures

The signature of a Fortran routine has the following structure:

```
[<typespec>] function | subroutine <routine name> \
      [ ( [<arguments>] ) ] [ result ( <entityname> ) ]
  [<argument/variable type declarations>]
  [<argument/variable attribute statements>]
  [<use statements>]
  [<common block statements>]
  [<other statements>]
end [ function | subroutine [<routine name>] ]
```

From a Fortran routine signature F2PY generates a Python/C extension function that has the following signature:

```
def <routine name>(<required arguments>[,<optional arguments>]):
    ...
    return <return variables>
```

The signature of a Fortran block data has the following structure:

```

block data [ <block data name> ]
  [<variable type declarations>]
  [<variable attribute statements>]
  [<use statements>]
  [<common block statements>]
  [<include statements>]
end [ block data [<block data name>] ]

```

Type declarations

The definition of the <argument/variable type declaration> part is

```
<typespec> [ [<attrspec>] :: ] <entitydecl>
```

where

```

<typespec> := byte | character [<charselector>]
           | complex [<kindselector>] | real [<kindselector>]
           | double complex | double precision
           | integer [<kindselector>] | logical [<kindselector>]

<charselector> := * <charlen>
               | ( [len=] <len> [ , [kind=] <kind> ] )
               | ( kind= <kind> [ , len= <len> ] )
<kindselector> := * <intlen> | ( [kind=] <kind> )

<entitydecl> := <name> [ [ * <charlen> ] [ ( <arrayspec> ) ]
                     | [ ( <arrayspec> ) ] * <charlen> ]
               | [ / <init_expr> / | = <init_expr> ] \
               [ , <entitydecl> ]

```

and

- <attrspec> is a comma separated list of *attributes*;
- <arrayspec> is a comma separated list of dimension bounds;
- <init_expr> is a *C expression*;
- <intlen> may be negative integer for integer type specifications. In such cases integer*<negintlen> represents unsigned C integers;

If an argument has no <argument type declaration>, its type is determined by applying implicit rules to its name.

Statements

Attribute statements

The <argument/variable attribute statement> is similar to the <argument/variable type declaration>, but without <typespec>.

An attribute statement cannot contain other attributes, and <entitydecl> can be only a list of names. See *Attributes* for more details on the attributes that can be used by F2PY.

Use statements

- The definition of the `<use statement>` part is

```
use <modulename> [ , <rename_list> | , ONLY : <only_list> ]
```

where

```
<rename_list> := <local_name> => <use_name> [ , <rename_list> ]
```

- Currently F2PY uses use statements only for linking call-back modules and external arguments (call-back functions). See [Call-back arguments](#).

Common block statements

- The definition of the `<common block statement>` part is

```
common / <common name> / <shortentitydecl>
```

where

```
<shortentitydecl> := <name> [ ( <arrayspec> ) ] [ , <shortentitydecl> ]
```

- If a python module block contains two or more common blocks with the same name, the variables from the additional declarations are appended. The types of variables in `<shortentitydecl>` are defined using `<argument type declarations>`. Note that the corresponding `<argument type declarations>` may contain array specifications; then these need not be specified in `<shortentitydecl>`.

Other statements

- The `<other statement>` part refers to any other Fortran language constructs that are not described above. F2PY ignores most of them except the following:

- call statements and function calls of external arguments (see [more details on external arguments](#));
- **include statements**

```
include '<filename>'  
include "<filename>"
```

If a file `<filename>` does not exist, the `include` statement is ignored. Otherwise, the file `<filename>` is included to a signature file. `include` statements can be used in any part of a signature file, also outside the Fortran/C routine signature blocks.

- **implicit statements**

```
implicit none  
implicit <list of implicit maps>
```

where

```
<implicit map> := <typespec> ( <list of letters or range of letters> )
```

Implicit rules are used to determine the type specification of a variable (from the first-letter of its name) if the variable is not defined using `<variable type declaration>`. Default implicit rules are given by:


```
implicit real (a-h,o-z,$_), integer (i-m)
```

– entry statements

```
entry <entry name> [[<arguments>]]
```

F2PY generates wrappers for all entry names using the signature of the routine block.

Note: The `entry` statement can be used to describe the signature of an arbitrary subroutine or function allowing F2PY to generate a number of wrappers from only one routine block signature. There are few restrictions while doing this: `fortranname` cannot be used, `callstatement` and `callprotoargument` can be used only if they are valid for all entry routines, etc.

F2PY statements

In addition, F2PY introduces the following statements:

threadsafe

Uses a `Py_BEGIN_ALLOW_THREADS .. Py_END_ALLOW_THREADS` block around the call to Fortran/C function.

callstatement <C-expr|multi-line block>

Replaces the F2PY generated call statement to Fortran/C function with `<C-expr|multi-line block>`. The wrapped Fortran/C function is available as `(*f2py_func)`.

To raise an exception, set `f2py_success = 0` in `<C-expr|multi-line block>`.

callprotoargument <C-typespecs>

When the `callstatement` statement is used, F2PY may not generate proper prototypes for Fortran/C functions (because `<C-expr>` may contain function calls, and F2PY has no way to determine what should be the proper prototype).

With this statement you can explicitly specify the arguments of the corresponding prototype:

```
extern <return type> FUNC_F(<routine name>, <ROUTINE NAME>) (<callprotoargument>);
```

fortranname [<actual Fortran/C routine name>]

F2PY allows for the use of an arbitrary `<routine name>` for a given Fortran/C function. Then this statement is used for the `<actual Fortran/C routine name>`.

If `fortranname` statement is used without `<actual Fortran/C routine name>` then a dummy wrapper is generated.

usercode <multi-line block>

When this is used inside a `python module` block, the given C code will be inserted to generated C/API source just before wrapper function definitions.

Here you can define arbitrary C functions to be used for the initialization of optional arguments.

For example, if `usercode` is used twice inside `python module` block then the second multi-line block is inserted after the definition of the external routines.

When used inside `<routine signature>`, then the given C code will be inserted into the corresponding wrapper function just after the declaration of variables but before any C statements. So, the `usercode` follow-up can contain both declarations and C statements.

When used inside the first `interface` block, then the given C code will be inserted at the end of the initialization function of the extension module. This is how the extension modules dictionary can be modified and has many use-cases; for example, to define additional variables.

pymethoddef <multiline block>

This is a multi-line block which will be inserted into the definition of a module methods `PyMethodDef`-array. It must be a comma-separated list of C arrays (see [Extending and Embedding Python](#) documentation for details). `pymethoddef` statement can be used only inside `python module` block.

Attributes

The following attributes can be used by F2PY.

optional

The corresponding argument is moved to the end of `<optional arguments>` list. A default value for an optional argument can be specified via `<init_expr>` (see the [entitydecl definition](#))

Note:

- The default value must be given as a valid C expression.
 - Whenever `<init_expr>` is used, the `optional` attribute is set automatically by F2PY.
 - For an optional array argument, all its dimensions must be bounded.
-

required

The corresponding argument with this attribute is considered mandatory. This is the default. `required` should only be specified if there is a need to disable the automatic `optional` setting when `<init_expr>` is used.

If a Python `None` object is used as a required argument, the argument is treated as optional. That is, in the case of array arguments, the memory is allocated. If `<init_expr>` is given, then the corresponding initialization is carried out.

dimension (<arrayspec>)

The corresponding variable is considered as an array with dimensions given in `<arrayspec>`.

intent (<intentspec>)

This specifies the “intention” of the corresponding argument. `<intentspec>` is a comma separated list of the following keys:

- **in**

The corresponding argument is considered to be input-only. This means that the value of the argument is passed to a Fortran/C function and that the function is expected to not change the value of this argument.

- **inout**

The corresponding argument is marked for input/output or as an *in situ* output argument. `intent(inout)` arguments can be only *contiguous* NumPy arrays (in either the Fortran or C sense) with proper type and size. The latter coincides with the default contiguous concept used in NumPy and is effective only if `intent(c)` is used. F2PY assumes Fortran contiguous arguments by default.

Note: Using `intent(inout)` is generally not recommended, as it can cause unexpected results. For example, scalar arguments using `intent(inout)` are assumed to be array objects in order to have *in situ* changes be effective. Use `intent(in, out)` instead.

See also the `intent(inplace)` attribute.

- **inplace**

The corresponding argument is considered to be an input/output or *in situ* output argument. `intent(inplace)` arguments must be NumPy arrays of a proper size. If the type of an array is not “proper” or the array is non-contiguous then the array will be modified in-place to fix the type and make it contiguous.

Note: Using `intent(inplace)` is generally not recommended either.

For example, when slices have been taken from an `intent(inplace)` argument then after in-place changes, the data pointers for the slices may point to an unallocated memory area.

- **out**

The corresponding argument is considered to be a return variable. It is appended to the <returned variables> list. Using `intent(out)` sets `intent(hide)` automatically, unless `intent(in)` or `intent(inout)` are specified as well.

By default, returned multidimensional arrays are Fortran-contiguous. If `intent(c)` attribute is used, then the returned multidimensional arrays are C-contiguous.

- **hide**

The corresponding argument is removed from the list of required or optional arguments. Typically `intent(hide)` is used with `intent(out)` or when `<init_expr>` completely determines the value of the argument like in the following example:

```
integer intent(hide),depend(a) :: n = len(a)
real intent(in),dimension(n) :: a
```

- **c**

The corresponding argument is treated as a C scalar or C array argument. For the case of a scalar argument, its value is passed to a C function as a C scalar argument (recall that Fortran scalar arguments are actually C pointer arguments). For array arguments, the wrapper function is assumed to treat multidimensional arrays as C-contiguous arrays.

There is no need to use `intent(c)` for one-dimensional arrays, irrespective of whether the wrapped function is in Fortran or C. This is because the concepts of Fortran- and C contiguity overlap in one-dimensional cases.

If `intent(c)` is used as a statement but without an entity declaration list, then F2PY adds the `intent(c)` attribute to all arguments.

Also, when wrapping C functions, one must use `intent(c)` attribute for <routine name> in order to disable Fortran specific `F_FUNC(...)` macros.

- **cache**

The corresponding argument is treated as junk memory. No Fortran nor C contiguity checks are carried out. Using `intent(cache)` makes sense only for array arguments, also in conjunction with `intent(hide)` or optional attributes.

- **copy**

Ensures that the original contents of `intent(in)` argument is preserved. Typically used with the `intent(in,out)` attribute. F2PY creates an optional argument `overwrite_<argument name>` with the default value 0.

- **overwrite**

This indicates that the original contents of the `intent(in)` argument may be altered by the Fortran/C function. F2PY creates an optional argument `overwrite_<argument name>` with the default value 1.

- **out=<new name>**

Replaces the returned name with `<new name>` in the `__doc__` string of the wrapper function.

- **callback**

Constructs an external function suitable for calling Python functions from Fortran. `intent(callback)` must be specified before the corresponding external statement. If the 'argument' is not in the argument list then it will be added to Python wrapper but only by initializing an external function.

Note: Use `intent(callback)` in situations where the Fortran/C code assumes that the user implemented a function with a given prototype and linked it to an executable. Don't use `intent(callback)` if the function appears in the argument list of a Fortran routine.

With `intent(hide)` or optional attributes specified and using a wrapper function without specifying the callback argument in the argument list; then the call-back function is assumed to be found in the namespace of the F2PY generated extension module where it can be set as a module attribute by a user.

- **aux**

Defines an auxiliary C variable in the F2PY generated wrapper function. Useful to save parameter values so that they can be accessed in initialization expressions for other variables.

Note: `intent(aux)` silently implies `intent(c)`.

The following rules apply:

- If none of `intent(in | inout | out | hide)` are specified, `intent(in)` is assumed.
 - `intent(in,inout)` is `intent(in)`;
 - `intent(in,hide)` or `intent(inout,hide)` is `intent(hide)`;
 - `intent(out)` is `intent(out,hide)` unless `intent(in)` or `intent(inout)` is specified.
- If `intent(copy)` or `intent(overwrite)` is used, then an additional optional argument is introduced with a name `overwrite_<argument name>` and a default value 0 or 1, respectively.
 - `intent(inout,inplace)` is `intent(inplace)`;
 - `intent(in,inplace)` is `intent(inplace)`;
 - `intent(hide)` disables optional and required.

`check([<C-booleanexpr>])`

Performs a consistency check on the arguments by evaluating `<C-booleanexpr>`; if `<C-booleanexpr>` returns 0, an exception is raised.

Note: If `check (. .)` is not used then F2PY automatically generates a few standard checks (e.g. in a case of an array argument, it checks for the proper shape and size). Use `check ()` to disable checks generated by F2PY.

depend ([<names>])

This declares that the corresponding argument depends on the values of variables in the `<names>` list. For example, `<init_expr>` may use the values of other arguments. Using information given by `depend (. .)` attributes, F2PY ensures that arguments are initialized in a proper order. If the `depend (. .)` attribute is not used then F2PY determines dependence relations automatically. Use `depend ()` to disable the dependence relations generated by F2PY.

When you edit dependence relations that were initially generated by F2PY, be careful not to break the dependence relations of other relevant variables. Another thing to watch out for is cyclic dependencies. F2PY is able to detect cyclic dependencies when constructing wrappers and it complains if any are found.

allocatable

The corresponding variable is a Fortran 90 allocatable array defined as Fortran 90 module data.

external

The corresponding argument is a function provided by user. The signature of this call-back function can be defined

- in `__user__` module block,
- or by demonstrative (or real, if the signature file is a real Fortran code) call in the `<other statements>` block.

For example, F2PY generates from:

```
external cb_sub, cb_fun
integer n
real a(n), r
call cb_sub(a, n)
r = cb_fun(4)
```

the following call-back signatures:

```
subroutine cb_sub(a, n)
  real dimension(n) :: a
  integer optional, check(len(a) >= n), depend(a) :: n=len(a)
end subroutine cb_sub
function cb_fun(e_4_e) result (r)
  integer :: e_4_e
  real :: r
end function cb_fun
```

The corresponding user-provided Python function are then:

```
def cb_sub(a, [n]):
    ...
    return
def cb_fun(e_4_e):
    ...
    return r
```

See also the `intent (callback)` attribute.

parameter

This indicates that the corresponding variable is a parameter and it must have a fixed value. F2PY replaces all parameter occurrences by their corresponding values.

Extensions**F2PY directives**

The F2PY directives allow using F2PY signature file constructs in Fortran 77/90 source codes. With this feature one can (almost) completely skip the intermediate signature file generation and apply F2PY directly to Fortran source codes.

F2PY directives have the following form:

```
<comment char>f2py ...
```

where allowed comment characters for fixed and free format Fortran codes are `cC*!#` and `!`, respectively. Everything that follows `<comment char>f2py` is ignored by a compiler but read by F2PY as a normal non-comment Fortran line:

Note: When F2PY finds a line with F2PY directive, the directive is first replaced by 5 spaces and then the line is reread.

For fixed format Fortran codes, `<comment char>` must be at the first column of a file, of course. For free format Fortran codes, the F2PY directives can appear anywhere in a file.

C expressions

C expressions are used in the following parts of signature files:

- `<init_expr>` for variable initialization;
- `<C-booleanexpr>` of the `check` attribute;
- `<arrayspec>` of the `dimension` attribute;
- `callstatement` statement, here also a C multi-line block can be used.

A C expression may contain:

- standard C constructs;
- functions from `math.h` and `Python.h`;
- variables from the argument list, presumably initialized before according to given dependence relations;
- the following CPP macros:
 - `rank(<name>)` Returns the rank of an array `<name>`.
 - `shape(<name>, <n>)` Returns the `<n>`-th dimension of an array `<name>`.
 - `len(<name>)` Returns the length of an array `<name>`.
 - `size(<name>)` Returns the size of an array `<name>`.
 - `slen(<name>)` Returns the length of a string `<name>`.

For initializing an array `<array name>`, F2PY generates a loop over all indices and dimensions that executes the following pseudo-statement:

```
<array name>(_i[0],_i[1],...) = <init_expr>;
```

where `_i[<i>]` refers to the `<i>`-th index value and that runs from 0 to `shape(<array name>, <i>)-1`.

For example, a function `myrange(n)` generated from the following signature

```
subroutine myrange(a,n)
  fortranname      ! myrange is a dummy wrapper
  integer intent(in) :: n
  real*8 intent(c,out),dimension(n),depend(n) :: a = _i[0]
end subroutine myrange
```

is equivalent to `numpy.arange(n, dtype=float)`.

Warning: F2PY may lower cases also in C expressions when scanning Fortran codes (see `--[no]-lower` option).

Multi-line blocks

A multi-line block starts with `'''` (triple single-quotes) and ends with `'''` in some *strictly* subsequent line. Multi-line blocks can be used only within `.pyf` files. The contents of a multi-line block can be arbitrary (except that it cannot contain `'''`) and no transformations (e.g. lowering cases) are applied to it.

Currently, multi-line blocks can be used in the following constructs:

- as a C expression of the `callstatement` statement;
- as a C type specification of the `callprotoargument` statement;
- as a C code block of the `usercode` statement;
- as a list of C arrays of the `pymethoddef` statement;
- as a documentation string.

11.3.2 Using F2PY bindings in Python

In this page, you can find a full description and a few examples of common usage patterns for F2PY with Python and different argument types. For more examples and use cases, see [F2PY examples](#).

Fortran type objects

All wrappers for Fortran/C routines, common blocks, or for Fortran 90 module data generated by F2PY are exposed to Python as `fortran` type objects. Routine wrappers are callable `fortran` type objects while wrappers to Fortran data have attributes referring to data objects.

All `fortran` type objects have an attribute `_cpointer` that contains a `CObject` referring to the C pointer of the corresponding Fortran/C function or variable at the C level. Such `CObjects` can be used as callback arguments for F2PY generated functions to bypass the Python C/API layer for calling Python functions from Fortran or C. This can be useful when the computational aspects of such functions are implemented in C or Fortran and wrapped with F2PY (or any other tool capable of providing the `CObject` of a function).

Consider a Fortran 77 file ``ftype.f``:

```

C FILE: FTYPE.F
  SUBROUTINE FOO(N)
    INTEGER N
Cf2py integer optional,intent(in) :: n = 13
    REAL A,X
    COMMON /DATA/ A,X(3)
    PRINT*, "IN FOO: N=",N, " A=",A, " X=[",X(1),X(2),X(3), "]"
    END
C END OF FTYPE.F

```

and a wrapper built using `f2py -c ftype.f -m ftype`.

In Python, you can observe the types of `foo` and `data`, and how to access individual objects of the wrapped Fortran code.

```

>>> import ftype
>>> print(ftype.__doc__)
This module 'ftype' is auto-generated with f2py (version:2).
Functions:
  foo(n=13)
COMMON blocks:
  /data/ a,x(3)
.
>>> type(ftype.foo), type(ftype.data)
(<class 'fortran'>, <class 'fortran'>)
>>> ftype.foo()
IN FOO: N= 13 A=  0. X=[  0.  0.  0.]
>>> ftype.data.a = 3
>>> ftype.data.x = [1,2,3]
>>> ftype.foo()
IN FOO: N= 13 A=  3. X=[  1.  2.  3.]
>>> ftype.data.x[1] = 45
>>> ftype.foo(24)
IN FOO: N= 24 A=  3. X=[  1. 45.  3.]
>>> ftype.data.x
array([  1., 45.,  3.], dtype=float32)

```

Scalar arguments

In general, a scalar argument for a F2PY generated wrapper function can be an ordinary Python scalar (integer, float, complex number) as well as an arbitrary sequence object (list, tuple, array, string) of scalars. In the latter case, the first element of the sequence object is passed to the Fortran routine as a scalar argument.

Note:

- When type-casting is required and there is possible loss of information via narrowing e.g. when type-casting float to integer or complex to float, F2PY *does not* raise an exception.
 - For complex to real type-casting only the real part of a complex number is used.
 - `intent(inout)` scalar arguments are assumed to be array objects in order to have *in situ* changes be effective. It is recommended to use arrays with proper type but also other types work. [Read more about the intent attribute.](#)
-

Consider the following Fortran 77 code:


```

C FILE: SCALAR.F
  SUBROUTINE FOO(A,B)
    REAL*8 A, B
Cf2py intent(in) a
Cf2py intent(inout) b
    PRINT*, "      A=",A, " B=",B
    PRINT*, "INCREMENT A AND B"
    A = A + 1D0
    B = B + 1D0
    PRINT*, "NEW A=",A, " B=",B
    END
C END OF FILE SCALAR.F

```

and wrap it using `f2py -c -m scalar scalar.f`.

In Python:

```

>>> import scalar
>>> print(scalar.foo.__doc__)
foo(a,b)

Wrapper for ``foo``.

Parameters
-----
a : input float
b : in/output rank-0 array(float,'d')

>>> scalar.foo(2, 3)
      A=  2. B=  3.
      INCREMENT A AND B
      NEW A=  3. B=  4.
>>> import numpy
>>> a = numpy.array(2)      # these are integer rank-0 arrays
>>> b = numpy.array(3)
>>> scalar.foo(a, b)
      A=  2. B=  3.
      INCREMENT A AND B
      NEW A=  3. B=  4.
>>> print(a, b)           # note that only b is changed in situ
2 4

```

String arguments

F2PY generated wrapper functions accept almost any Python object as a string argument, since `str` is applied for non-string objects. Exceptions are NumPy arrays that must have type code `'S1'` or `'b'` (corresponding to the outdated `'c'` or `'1'` typecodes, respectively) when used as string arguments. See `arrays.scalars` for more information on these typecodes.

A string can have an arbitrary length when used as a string argument for an F2PY generated wrapper function. If the length is greater than expected, the string is truncated silently. If the length is smaller than expected, additional memory is allocated and filled with `\0`.

Because Python strings are immutable, an `intent(inout)` argument expects an array version of a string in order to have *in situ* changes be effective.

Consider the following Fortran 77 code:

```

C FILE: STRING.F
      SUBROUTINE FOO(A,B,C,D)
      CHARACTER*5 A, B
      CHARACTER*(*) C,D
Cf2py intent(in) a,c
Cf2py intent(inout) b,d
      PRINT*, "A=",A
      PRINT*, "B=",B
      PRINT*, "C=",C
      PRINT*, "D=",D
      PRINT*, "CHANGE A,B,C,D"
      A(1:1) = 'A'
      B(1:1) = 'B'
      C(1:1) = 'C'
      D(1:1) = 'D'
      PRINT*, "A=",A
      PRINT*, "B=",B
      PRINT*, "C=",C
      PRINT*, "D=",D
      END
C END OF FILE STRING.F

```

and wrap it using `f2py -c -m mystring string.f`.

Python session:

```

>>> import mystring
>>> print(mystring.foo.__doc__)
foo(a,b,c,d)

Wrapper for ``foo``.

Parameters
-----
a : input string(len=5)
b : in/output rank-0 array(string(len=5),'c')
c : input string(len=-1)
d : in/output rank-0 array(string(len=-1),'c')

>>> from numpy import array
>>> a = array(b'123\0\0')
>>> b = array(b'123\0\0')
>>> c = array(b'123')
>>> d = array(b'123')
>>> mystring.foo(a, b, c, d)
A=123
B=123
C=123
D=123
CHANGE A,B,C,D
A=A23
B=B23
C=C23
D=D23
>>> a[()], b[()], c[()], d[()]
(b'123', b'B23', b'123', b'D2')

```

Array arguments

In general, array arguments for F2PY generated wrapper functions accept arbitrary sequences that can be transformed to NumPy array objects. There are two notable exceptions:

- `intent(inout)` array arguments must always be *proper-contiguous* and have a compatible `dtype`, otherwise an exception is raised.
- `intent(inplace)` array arguments will be changed *in situ* if the argument has a different type than expected (see the `intent(inplace)` *attribute* for more information).

In general, if a NumPy array is *proper-contiguous* and has a proper type then it is directly passed to the wrapped Fortran/C function. Otherwise, an element-wise copy of the input array is made and the copy, being proper-contiguous and with proper type, is used as the array argument.

Usually there is no need to worry about how the arrays are stored in memory and whether the wrapped functions, being either Fortran or C functions, assume one or another storage order. F2PY automatically ensures that wrapped functions get arguments with the proper storage order; the underlying algorithm is designed to make copies of arrays only when absolutely necessary. However, when dealing with very large multidimensional input arrays with sizes close to the size of the physical memory in your computer, then care must be taken to ensure the usage of proper-contiguous and proper type arguments.

To transform input arrays to column major storage order before passing them to Fortran routines, use the function `numpy.asfortranarray`.

Consider the following Fortran 77 code:

```
C FILE: ARRAY.F
  SUBROUTINE FOO(A,N,M)
C
C   INCREMENT THE FIRST ROW AND DECREMENT THE FIRST COLUMN OF A
C
  INTEGER N,M,I,J
  REAL*8 A(N,M)
Cf2py intent(in,out,copy) a
Cf2py integer intent(hide),depend(a) :: n=shape(a,0), m=shape(a,1)
  DO J=1,M
    A(1,J) = A(1,J) + 1D0
  ENDDO
  DO I=1,N
    A(I,1) = A(I,1) - 1D0
  ENDDO
  END
C END OF FILE ARRAY.F
```

and wrap it using `f2py -c -m arr array.f -DF2PY_REPORT_ON_ARRAY_COPY=1`.

In Python:

```
>>> import arr
>>> from numpy import asfortranarray
>>> print(arr.foo.__doc__)
a = foo(a,[overwrite_a])

Wrapper for ``foo``.

Parameters
-----
a : input rank-2 array('d') with bounds (n,m)
```

(continues on next page)

(continued from previous page)

```

Other Parameters
-----
overwrite_a : input int, optional
    Default: 0

Returns
-----
a : rank-2 array('d') with bounds (n,m)

>>> a = arr.foo([[1, 2, 3],
...             [4, 5, 6]])
created an array from object
>>> print(a)
[[ 1.  3.  4.]
 [ 3.  5.  6.]]
>>> a.flags.c_contiguous
False
>>> a.flags.f_contiguous
True
# even if a is proper-contiguous and has proper type,
# a copy is made forced by intent(copy) attribute
# to preserve its original contents
>>> b = arr.foo(a)
copied an array: size=6, elsize=8
>>> print(a)
[[ 1.  3.  4.]
 [ 3.  5.  6.]]
>>> print(b)
[[ 1.  4.  5.]
 [ 2.  5.  6.]]
>>> b = arr.foo(a, overwrite_a = 1) # a is passed directly to Fortran
...                               # routine and its contents is discarded
...
>>> print(a)
[[ 1.  4.  5.]
 [ 2.  5.  6.]]
>>> print(b)
[[ 1.  4.  5.]
 [ 2.  5.  6.]]
>>> a is b                               # a and b are actually the same objects
True
>>> print(arr.foo([1, 2, 3]))           # different rank arrays are allowed
created an array from object
[ 1.  1.  2.]
>>> print(arr.foo([[[1], [2], [3]]]))
created an array from object
[[[ 1.]
   [ 1.]
   [ 2.]]]
>>>
>>> # Creating arrays with column major data storage order:
...
>>> s = asfortranarray([[1, 2, 3], [4, 5, 6]])
>>> s.flags.f_contiguous
True
>>> print(s)

```

(continues on next page)

(continued from previous page)

```

[[1 2 3]
 [4 5 6]]
>>> print(arr.foo(s))
>>> s2 = asfortranarray(s)
>>> s2 is s      # an array with column major storage order
                  # is returned immediately
True
>>> # Note that arr.foo returns a column major data storage order array:
...
>>> s3 = ascontiguousarray(s)
>>> s3.flags.f_contiguous
False
>>> s3.flags.c_contiguous
True
>>> s3 = arr.foo(s3)
copied an array: size=6, elsize=8
>>> s3.flags.f_contiguous
True
>>> s3.flags.c_contiguous
False

```

Call-back arguments

F2PY supports calling Python functions from Fortran or C codes.

Consider the following Fortran 77 code:

```

C FILE: CALLBACK.F
      SUBROUTINE FOO(FUN,R)
      EXTERNAL FUN
      INTEGER I
      REAL*8 R, FUN
Cf2py intent(out) r
      R = 0D0
      DO I=-5,5
        R = R + FUN(I)
      ENDDO
      END
C END OF FILE CALLBACK.F

```

and wrap it using `f2py -c -m callback callback.f`.

In Python:

```

>>> import callback
>>> print(callback.foo.__doc__)
r = foo(fun,[fun_extra_args])

Wrapper for ``foo``.

Parameters
-----
fun : call-back function

Other Parameters
-----

```

(continues on next page)

(continued from previous page)

```

fun_extra_args : input tuple, optional
    Default: ()

Returns
-----
r : float

Notes
-----
Call-back functions::

    def fun(i): return r
    Required arguments:
        i : input int
    Return objects:
        r : float

>>> def f(i): return i*i
...
>>> print(callback.foo(f))
110.0
>>> print(callback.foo(lambda i:1))
11.0

```

In the above example F2PY was able to guess accurately the signature of the call-back function. However, sometimes F2PY cannot establish the appropriate signature; in these cases the signature of the call-back function must be explicitly defined in the signature file.

To facilitate this, signature files may contain special modules (the names of these modules contain the special `__user__` sub-string) that define the various signatures for call-back functions. Callback arguments in routine signatures have the external attribute (see also the `intent(callback)` *attribute*). To relate a callback argument with its signature in a `__user__` module block, a `use` statement can be utilized as illustrated below. The same signature for a callback argument can be referred to in different routine signatures.

We use the same Fortran 77 code as in the previous example but now we will pretend that F2PY was not able to guess the signatures of call-back arguments correctly. First, we create an initial signature file `callback2.pyf` using F2PY:

```
f2py -m callback2 -h callback2.pyf callback.f
```

Then modify it as follows

```

!      -*- f90 -*-
python module __user__routines
  interface
    function fun(i) result (r)
      integer :: i
      real*8 :: r
    end function fun
  end interface
end python module __user__routines

python module callback2
  interface
    subroutine foo(f,r)
      use __user__routines, f=>fun
      external f
      real*8 intent(out) :: r
    end subroutine foo
  end interface
end python module callback2

```

(continues on next page)

(continued from previous page)

```

        end subroutine foo
    end interface
end python module callback2

```

Finally, we build the extension module using `f2py -c callback2.pyf callback.f`.

An example Python session for this snippet would be identical to the previous example except that the argument names would differ.

Sometimes a Fortran package may require that users provide routines that the package will use. F2PY can construct an interface to such routines so that Python functions can be called from Fortran.

Consider the following Fortran 77 subroutine that takes an array as its input and applies a function `func` to its elements.

```

    subroutine calculate(x,n)
cf2py intent(callback) func
    external func
c      The following lines define the signature of func for F2PY:
cf2py real*8 y
cf2py y = func(y)
c
cf2py intent(in,out,copy) x
    integer n,i
    real*8 x(n), func
    do i=1,n
        x(i) = func(x(i))
    end do
end

```

The Fortran code expects that the function `func` has been defined externally. In order to use a Python function for `func`, it must have an attribute `intent(callback)` and it must be specified before the `external` statement.

Finally, build an extension module using `f2py -c -m foo calculate.f`

In Python:

```

>>> import foo
>>> foo.calculate(range(5), lambda x: x*x)
array([ 0.,  1.,  4.,  9., 16.])
>>> import math
>>> foo.calculate(range(5), math.exp)
array([ 1.          ,  2.71828183,  7.3890561, 20.08553692, 54.59815003])

```

The function is included as an argument to the python function call to the Fortran subroutine even though it was *not* in the Fortran subroutine argument list. The “external” keyword refers to the C function generated by f2py, not the Python function itself. The python function is essentially being supplied to the C function.

The callback function may also be explicitly set in the module. Then it is not necessary to pass the function in the argument list to the Fortran function. This may be desired if the Fortran function calling the Python callback function is itself called by another Fortran function.

Consider the following Fortran 77 subroutine:

```

    subroutine f1()
    print *, "in f1, calling f2 twice.."
    call f2()
    call f2()
    return

```

(continues on next page)

(continued from previous page)

```

        end

        subroutine f2()
cf2py      intent(callback, hide) fpy
           external fpy
           print *, "in f2, calling f2py.."
           call fpy()
           return
        end

```

and wrap it using `f2py -c -m pfromf extcallback.f`.

In Python:

```

>>> import pfromf
>>> pfromf.f2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pfromf.error: Callback fpy not defined (as an argument or module pfromf attribute).

>>> def f(): print("python f")
...
>>> pfromf.fpy = f
>>> pfromf.f2()
  in f2, calling f2py..
python f
>>> pfromf.f1()
  in f1, calling f2 twice..
  in f2, calling f2py..
python f
  in f2, calling f2py..
python f
>>>

```

Resolving arguments to call-back functions

F2PY generated interfaces are very flexible with respect to call-back arguments. For each call-back argument an additional optional argument `<name>_extra_args` is introduced by F2PY. This argument can be used to pass extra arguments to user provided call-back functions.

If a F2PY generated wrapper function expects the following call-back argument:

```

def fun(a_1, ..., a_n):
    ...
    return x_1, ..., x_k

```

but the following Python function

```

def gun(b_1, ..., b_m):
    ...
    return y_1, ..., y_l

```

is provided by a user, and in addition,

```
fun_extra_args = (e_1, ..., e_p)
```

is used, then the following rules are applied when a Fortran or C function evaluates the call-back argument `gun`:

- If $p == 0$ then `gun(a_1, ..., a_q)` is called, here $q = \min(m, n)$.
- If $n + p \leq m$ then `gun(a_1, ..., a_n, e_1, ..., e_p)` is called.
- If $p \leq m < n + p$ then `gun(a_1, ..., a_q, e_1, ..., e_p)` is called, and here $q = m - p$.
- If $p > m$ then `gun(e_1, ..., e_m)` is called.
- If $n + p$ is less than the number of required arguments to `gun` then an exception is raised.

If the function `gun` may return any number of objects as a tuple; then the following rules are applied:

- If $k < 1$, then `y_{k+1}, ..., y_l` are ignored.
- If $k > 1$, then only `x_1, ..., x_l` are set.

Common blocks

F2PY generates wrappers to `common` blocks defined in a routine signature block. Common blocks are visible to all Fortran codes linked to the current extension module, but not to other extension modules (this restriction is due to the way Python imports shared libraries). In Python, the F2PY wrappers to `common` blocks are `fortran` type objects that have (dynamic) attributes related to the data members of the common blocks. When accessed, these attributes return as NumPy array objects (multidimensional arrays are Fortran-contiguous) which directly link to data members in common blocks. Data members can be changed by direct assignment or by in-place changes to the corresponding array objects.

Consider the following Fortran 77 code:

```
C FILE: COMMON.F
      SUBROUTINE FOO
      INTEGER I,X
      REAL A
      COMMON /DATA/ I,X(4),A(2,3)
      PRINT*, "I=",I
      PRINT*, "X=[",X,"]"
      PRINT*, "A=[ "
      PRINT*, "[",A(1,1),",",A(1,2),",",A(1,3),"]"
      PRINT*, "[",A(2,1),",",A(2,2),",",A(2,3),"]"
      PRINT*, "]"
      END
C END OF COMMON.F
```

and wrap it using `f2py -c -m common common.f`.

In Python:

```
>>> import common
>>> print(common.data.__doc__)
i : 'i'-scalar
x : 'i'-array(4)
a : 'f'-array(2,3)

>>> common.data.i = 5
>>> common.data.x[1] = 2
>>> common.data.a = [[1,2,3],[4,5,6]]
>>> common.foo()
>>> common.foo()
I=
5
X=[
0          2          0          0 ]
A=[
[ 1.00000000 , 2.00000000 , 3.00000000 ]
```

(continues on next page)

(continued from previous page)

```

[ 4.00000000 , 5.00000000 , 6.00000000 ]
]
>>> common.data.a[1] = 45
>>> common.foo()
I=      5
X=[      0      2      0      0 ]
A=[
[ 1.00000000 , 2.00000000 , 3.00000000 ]
[ 45.0000000 , 45.0000000 , 45.0000000 ]
]
>>> common.data.a          # a is Fortran-contiguous
array([[ 1.,  2.,  3.],
       [ 45., 45., 45.]], dtype=float32)
>>> common.data.a.flags.f_contiguous
True

```

Fortran 90 module data

The F2PY interface to Fortran 90 module data is similar to the handling of Fortran 77 common blocks.

Consider the following Fortran 90 code:

```

module mod
  integer i
  integer :: x(4)
  real, dimension(2,3) :: a
  real, allocatable, dimension(:, :) :: b
contains
  subroutine foo
    integer k
    print*, "i=", i
    print*, "x=[", x, "]"
    print*, "a=[
    print*, "[", a(1,1), ",", a(1,2), ",", a(1,3), "]"
    print*, "[", a(2,1), ",", a(2,2), ",", a(2,3), "]"
    print*, "]"
    print*, "Setting a(1,2)=a(1,2)+3"
    a(1,2) = a(1,2)+3
  end subroutine foo
end module mod

```

and wrap it using `f2py -c -m moddata moddata.f90`.

In Python:

```

>>> import moddata
>>> print(moddata.mod.__doc__)
i : 'i'-scalar
x : 'i'-array(4)
a : 'f'-array(2,3)
b : 'f'-array(-1,-1), not allocated
foo()

Wrapper for ``foo``.

```

(continues on next page)

(continued from previous page)

```

>>> moddata.mod.i = 5
>>> moddata.mod.x[:2] = [1,2]
>>> moddata.mod.a = [[1,2,3],[4,5,6]]
>>> moddata.mod.foo()
i=
      5
x=[
      1      2      0      0 ]
a=[
 [ 1.000000 , 2.000000 , 3.000000 ]
 [ 4.000000 , 5.000000 , 6.000000 ]
]
Setting a(1,2)=a(1,2)+3
>>> moddata.mod.a          # a is Fortran-contiguous
array([[ 1.,  5.,  3.],
       [ 4.,  5.,  6.]], dtype=float32)
>>> moddata.mod.a.flags.f_contiguous
True

```

Allocatable arrays

F2PY has basic support for Fortran 90 module allocatable arrays.

Consider the following Fortran 90 code:

```

module mod
  real, allocatable, dimension(:,:) :: b
contains
  subroutine foo
    integer k
    if (allocated(b)) then
      print*, "b=["
      do k = 1,size(b,1)
        print*, b(k,1:size(b,2))
      enddo
      print*, "]"
    else
      print*, "b is not allocated"
    endif
  end subroutine foo
end module mod

```

and wrap it using `f2py -c -m allocarr allocarr.f90`.

In Python:

```

>>> import allocarr
>>> print(allocarr.mod.__doc__)
b : 'f'-array(-1,-1), not allocated
foo()

Wrapper for ``foo``.

>>> allocarr.mod.foo()
b is not allocated

```

(continues on next page)

(continued from previous page)

```
>>> allocarr.mod.b = [[1, 2, 3], [4, 5, 6]]           # allocate/initialize b
>>> allocarr.mod.foo()
b=[
  1.000000      2.000000      3.000000
  4.000000      5.000000      6.000000
]
>>> allocarr.mod.b                                     # b is Fortran-contiguous
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]], dtype=float32)
>>> allocarr.mod.b.flags.f_contiguous
True
>>> allocarr.mod.b = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # reallocate/initialize b
>>> allocarr.mod.foo()
b=[
  1.000000      2.000000      3.000000
  4.000000      5.000000      6.000000
  7.000000      8.000000      9.000000
]
>>> allocarr.mod.b = None                             # deallocate array
>>> allocarr.mod.foo()
b is not allocated
```

11.3.3 F2PY and Build Systems

In this section we will cover the various popular build systems and their usage with `f2py`.

Note: As of November 2021

The default build system for F2PY has traditionally been the through the enhanced `numpy.distutils` module. This module is based on `distutils` which will be removed in Python 3.12.0 in **October 2023**; `setuptools` does not have support for Fortran or F2PY and it is unclear if it will be supported in the future. Alternative methods are thus increasingly more important.

Basic Concepts

Building an extension module which includes Python and Fortran consists of:

- Fortran source(s)
- One or more generated files from `f2py`
 - A C wrapper file is always created
 - Code with modules require an additional `.f90` wrapper
 - Code with functions generate an additional `.f` wrapper
- `fortranobject.{c,h}`
 - Distributed with `numpy`
 - Can be queried via `python -c "import numpy.f2py; print(numpy.f2py.get_include())"`
- NumPy headers

- Can be queried via `python -c "import numpy; print(numpy.get_include())"`
- Python libraries and development headers

Broadly speaking there are three cases which arise when considering the outputs of `f2py`:

Fortran 77 programs

- Input file `blah.f`
- Generates
 - `blahmodule.c`
 - `blah-f2pywrappers.f`

When no `COMMON` blocks are present only a C wrapper file is generated. Wrappers are also generated to rewrite assumed shape arrays as automatic arrays.

Fortran 90 programs

- Input file `blah.f90`
- Generates:
 - `blahmodule.c`
 - `blah-f2pywrappers.f`
 - `blah-f2pywrappers2.f90`

The `f90` wrapper is used to handle code which is subdivided into modules. The `f` wrapper makes subroutines for functions. It rewrites assumed shape arrays as automatic arrays.

Signature files

- Input file `blah.pyf`
- Generates:
 - `blahmodule.c`
 - `blah-f2pywrappers2.f90` (occasionally)
 - `blah-f2pywrappers.f` (occasionally)

Signature files `.pyf` do not signal their language standard via the file extension, they may generate the `F90` and `F77` specific wrappers depending on their contents; which shifts the burden of checking for generated files onto the build system.

Note: From NumPy 1.22.4 onwards, `f2py` will deterministically generate wrapper files based on the input file Fortran standard (`F77` or greater). `--skip-empty-wrappers` can be passed to `f2py` to restore the previous behaviour of only generating wrappers when needed by the input .

In theory keeping the above requirements in hand, any build system can be adapted to generate `f2py` extension modules. Here we will cover a subset of the more popular systems.

Note: `make` has no place in a modern multi-language setup, and so is not discussed further.

Build Systems

Using via `numpy.distutils`

`numpy.distutils` is part of NumPy, and extends the standard Python `distutils` module to deal with Fortran sources and F2PY signature files, e.g. compile Fortran sources, call F2PY to construct extension modules, etc.

Example

Consider the following `setup_file.py` for the `fib` and `scalar` examples from *Three ways to wrap - getting started* section:

```
from numpy.distutils.core import Extension

ext1 = Extension(name = 'scalar',
                  sources = ['scalar.f'])
ext2 = Extension(name = 'fib2',
                  sources = ['fib2.pyf', 'fib1.f'])

if __name__ == "__main__":
    from numpy.distutils.core import setup
    setup(name = 'f2py_example',
          description = "F2PY Users Guide examples",
          author = "Pearu Peterson",
          author_email = "pearu@cens.ioc.ee",
          ext_modules = [ext1, ext2]
    )
# End of setup_example.py
```

Running

```
python setup_example.py build
```

will build two extension modules `scalar` and `fib2` to the build directory.

Extensions to `distutils`

`numpy.distutils` extends `distutils` with the following features:

- Extension class argument `sources` may contain Fortran source files. In addition, the list `sources` may contain at most one F2PY signature file, and in this case, the name of an Extension module must match with the `<modulename>` used in signature file. It is assumed that an F2PY signature file contains exactly one python module block.

If `sources` do not contain a signature file, then F2PY is used to scan Fortran source files to construct wrappers to the Fortran codes.

Additional options to the F2PY executable can be given using the Extension class argument `f2py_options`.

- The following new `distutils` commands are defined:

build_src

to construct Fortran wrapper extension modules, among many other things.

config_fc

to change Fortran compiler options.

Additionally, the `build_ext` and `build_clib` commands are also enhanced to support Fortran sources.

Run

```
python <setup.py file> config_fc build_src build_ext --help
```

to see available options for these commands.

- When building Python packages containing Fortran sources, one can choose different Fortran compilers by using the `build_ext` command option `--fcompiler=<Vendor>`. Here `<Vendor>` can be one of the following names (on linux systems):

```
absoft compaq fujitsu g95 gnu gnu95 intel intele intelem lahey nag nagfor nv_
↳pathf95 pg vast
```

See `numpy.distutils/fcompiler.py` for an up-to-date list of supported compilers for different platforms, or run

```
python -m numpy.f2py -c --help-fcompiler
```

Using via meson

The key advantage gained by leveraging `meson` over the techniques described in *Using via numpy.distutils* is that this feeds into existing systems and larger projects with ease. `meson` has a rather pythonic syntax which makes it more comfortable and amenable to extension for python users.

Note: Meson needs to be at-least 0.46.0 in order to resolve the python include directories.

Fibonacci Walkthrough (F77)

We will need the generated C wrapper before we can use a general purpose build system like `meson`. We will acquire this by:

```
python -n numpy.f2py fib1.f -m fib2
```

Now, consider the following `meson.build` file for the `fib` and `scalar` examples from *Three ways to wrap - getting started* section:

```
project('f2py_examples', 'c',
  version : '0.1',
  default_options : ['warning_level=2'])

add_languages('fortran')

py_mod = import('python')
py3 = py_mod.find_installation('python3')
py3_dep = py3.dependency()
message(py3.path())
message(py3.get_install_dir())

incdir_numpy = run_command(py3,
  ['-c', 'import os; os.chdir(".."); import numpy; print(numpy.get_include())'],
  check : true
).stdout().strip()
```

(continues on next page)

(continued from previous page)

```

incdir_f2py = run_command(py3,
    ['-c', 'import os; os.chdir(".."); import numpy.f2py; print(numpy.f2py.get_
    ↪include())'],
    check : true
).stdout().strip()

fibby_source = custom_target('fibbymodule.c',
    input : ['fib1.f'], # .f so no F90 wrappers
    output : ['fibbymodule.c', 'fibby-f2pywrappers.f'],
    command : [ py3, '-m', 'numpy.f2py', '@INPUT@',
                '-m', 'fibby', '--lower'
    ]
)

inc_np = include_directories(incdir_numpy, incdir_f2py)

py3.extension_module('fibby',
    'fib1.f',
    fibby_source,
    incdir_f2py+'//fortranobject.c',
    include_directories: inc_np,
    dependencies : py3_dep,
    install : true)

```

At this point the build will complete, but the import will fail:

```

meson setup builddir
meson compile -C builddir
cd builddir
python -c 'import fib2'
Traceback (most recent call last):
File "<string>", line 1, in <module>
ImportError: fib2.cpython-39-x86_64-linux-gnu.so: undefined symbol: FIB_
# Check this isn't a false positive
nm -A fib2.cpython-39-x86_64-linux-gnu.so | grep FIB_
fib2.cpython-39-x86_64-linux-gnu.so: U FIB_

```

Recall that the original example, as reproduced below, was in SCREAMCASE:

```

C FILE: FIB1.F
    SUBROUTINE FIB(A,N)
C
C    CALCULATE FIRST N FIBONACCI NUMBERS
C
    INTEGER N
    REAL*8 A(N)
    DO I=1,N
        IF (I.EQ.1) THEN
            A(I) = 0.0D0
        ELSEIF (I.EQ.2) THEN
            A(I) = 1.0D0
        ELSE
            A(I) = A(I-1) + A(I-2)
        ENDIF
    ENDDO
    END
C END FILE FIB1.F

```


With the standard approach, the subroutine exposed to python is `fib` and not `FIB`. This means we have a few options. One approach (where possible) is to lowercase the original Fortran file with say:

```
tr "[:upper:]" "[:lower:]" < fib1.f > fib1.f
python -n numpy.f2py fib1.f -m fib2
meson --wipe builddir
meson compile -C builddir
cd builddir
python -c 'import fib2'
```

However this requires the ability to modify the source which is not always possible. The easiest way to solve this is to let `f2py` deal with it:

```
python -n numpy.f2py fib1.f -m fib2 --lower
meson --wipe builddir
meson compile -C builddir
cd builddir
python -c 'import fib2'
```

Automating wrapper generation

A major pain point in the workflow defined above, is the manual tracking of inputs. Although it would require more effort to figure out the actual outputs for reasons discussed in *F2PY and Build Systems*.

Note: From NumPy 1.22.4 onwards, `f2py` will deterministically generate wrapper files based on the input file Fortran standard (F77 or greater). `--skip-empty-wrappers` can be passed to `f2py` to restore the previous behaviour of only generating wrappers when needed by the input.

However, we can augment our workflow in a straightforward to take into account files for which the outputs are known when the build system is set up.

```
project('f2py_examples', 'c',
        version : '0.1',
        default_options : ['warning_level=2'])

add_languages('fortran')

py_mod = import('python')
py3 = py_mod.find_installation('python3')
py3_dep = py3.dependency()
message(py3.path())
message(py3.get_install_dir())

incdir_numpy = run_command(py3,
    ['-c', 'import os; os.chdir(".."); import numpy; print(numpy.get_include())'],
    check : true
).stdout().strip()

incdir_f2py = run_command(py3,
    ['-c', 'import os; os.chdir(".."); import numpy.f2py; print(numpy.f2py.get_
↪include())'],
    check : true
).stdout().strip()
```

(continues on next page)

(continued from previous page)

```
fibby_source = custom_target('fibbymodule.c',
                             input : ['fib1.f'], # .f so no F90 wrappers
                             output : ['fibbymodule.c', 'fibby-f2pywrappers.f'],
                             command : [ py3, '-m', 'numpy.f2py', '@INPUT@',
                                           '-m', 'fibby', '--lower'])

inc_np = include_directories(incdir_numpy, incdir_f2py)

py3.extension_module('fibby',
                     'fib1.f',
                     fibby_source,
                     incdir_f2py+'/fortranobject.c',
                     include_directories: inc_np,
                     dependencies : py3_dep,
                     install : true)
```

This can be compiled and run as before.

```
rm -rf builddir
meson setup builddir
meson compile -C builddir
cd builddir
python -c "import numpy as np; import fibby; a = np.zeros(9); fibby.fib(a); print (a)"
# [ 0.  1.  1.  2.  3.  5.  8. 13. 21.]
```

Salient points

It is worth keeping in mind the following:

- meson will default to passing `-fimplicit-none` under gfortran by default, which differs from that of the standard `np.distutils` behaviour
- It is not possible to use SCREAMCASE in this context, so either the contents of the `.f` file or the generated wrapper `.c` needs to be lowered to regular letters; which can be facilitated by the `--lower` option of F2PY

Using via cmake

In terms of complexity, cmake falls between make and meson. The learning curve is steeper since CMake syntax is not pythonic and is closer to make with environment variables.

However, the trade-off is enhanced flexibility and support for most architectures and compilers. An introduction to the syntax is out of scope for this document, but this [extensive CMake collection](#) of resources is great.

Note: cmake is very popular for mixed-language systems, however support for `f2py` is not particularly native or pleasant; and a more natural approach is to consider *Using via scikit-build*

Fibonacci Walkthrough (F77)

Returning to the `fib` example from *Three ways to wrap - getting started* section.

```
C FILE: FIB1.F
      SUBROUTINE FIB(A,N)
C
C      CALCULATE FIRST N FIBONACCI NUMBERS
C
      INTEGER N
      REAL*8 A(N)
      DO I=1,N
        IF (I.EQ.1) THEN
          A(I) = 0.0D0
        ELSEIF (I.EQ.2) THEN
          A(I) = 1.0D0
        ELSE
          A(I) = A(I-1) + A(I-2)
        ENDIF
      ENDDO
      END
C END FILE FIB1.F
```

We do not need to explicitly generate the `python -m numpy.f2py fib1.f` output, which is `fib1module.c`, which is beneficial. With this; we can now initialize a `CMakeLists.txt` file as follows:

```
cmake_minimum_required(VERSION 3.18) # Needed to avoid requiring embedded Python libs_
↳ too

project(fibby
  VERSION 1.0
  DESCRIPTION "FIB module"
  LANGUAGES C Fortran
)

# Safety net
if(PROJECT_SOURCE_DIR STREQUAL PROJECT_BINARY_DIR)
  message(
    FATAL_ERROR
    "In-source builds not allowed. Please make a new directory (called a build_
↳ directory) and run CMake from there.\n"
  )
endif()

# Grab Python, 3.8 or newer
find_package(Python 3.8 REQUIRED
  COMPONENTS Interpreter Development.Module NumPy)

# Grab the variables from a local Python installation
# F2PY headers
execute_process(
  COMMAND "${Python_EXECUTABLE}"
  -c "import numpy.f2py; print(numpy.f2py.get_include())"
  OUTPUT_VARIABLE F2PY_INCLUDE_DIR
  OUTPUT_STRIP_TRAILING_WHITESPACE
)
```

(continues on next page)

(continued from previous page)

```

# Print out the discovered paths
include(CMakePrintHelpers)
cmake_print_variables(Python_INCLUDE_DIRS)
cmake_print_variables(F2PY_INCLUDE_DIR)
cmake_print_variables(Python_NumPy_INCLUDE_DIRS)

# Common variables
set(f2py_module_name "fibby")
set(fortran_src_file "${CMAKE_SOURCE_DIR}/fib1.f")
set(f2py_module_c "${f2py_module_name}.c")

# Generate sources
add_custom_target(
    genpyf
    DEPENDS "${CMAKE_CURRENT_BINARY_DIR}/${f2py_module_c}"
)
add_custom_command(
    OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/${f2py_module_c}"
    COMMAND ${Python_EXECUTABLE} -m "numpy.f2py"
            "${fortran_src_file}"
            -m "fibby"
            --lower # Important
    DEPENDS fib1.f # Fortran source
)

# Set up target
Python_add_library(${CMAKE_PROJECT_NAME} MODULE WITH_SOABI
    "${CMAKE_CURRENT_BINARY_DIR}/${f2py_module_c}" # Generated
    "${F2PY_INCLUDE_DIR}/fortranobject.c" # From NumPy
    "${fortran_src_file}" # Fortran source(s)
)

# Depend on sources
target_link_libraries(${CMAKE_PROJECT_NAME} PRIVATE Python::NumPy)
add_dependencies(${CMAKE_PROJECT_NAME} genpyf)
target_include_directories(${CMAKE_PROJECT_NAME} PRIVATE "${F2PY_INCLUDE_DIR}")

```

A key element of the `CMakeLists.txt` file defined above is that the `add_custom_command` is used to generate the wrapper C files and then added as a dependency of the actual shared library target via a `add_custom_target` directive which prevents the command from running every time. Additionally, the method used for obtaining the `fortranobject.c` file can also be used to grab the numpy headers on older cmake versions.

This then works in the same manner as the other modules, although the naming conventions are different and the output library is not automatically prefixed with the `cython` information.

```

ls .
# CMakeLists.txt fib1.f
cmake -S . -B build
cmake --build build
cd build
python -c "import numpy as np; import fibby; a = np.zeros(9); fibby.fib(a); print (a)"
# [ 0.  1.  1.  2.  3.  5.  8. 13. 21.]

```

This is particularly useful where an existing toolchain already exists and `scikit-build` or other additional python dependencies are discouraged.

Using via scikit-build

scikit-build provides two separate concepts geared towards the users of Python extension modules.

1. A `setuptools` replacement (legacy behaviour)
2. A series of `cmake` modules with definitions which help building Python extensions

Note: It is possible to use `scikit-build`'s `cmake` modules to [bypass the `cmake` setup mechanism](#) completely, and to write targets which call `f2py -c`. This usage is **not recommended** since the point of these build system documents are to move away from the internal `numpy.distutils` methods.

For situations where no `setuptools` replacements are required or wanted (i.e. if `wheels` are not needed), it is recommended to instead use the vanilla `cmake` setup described in [Using via `cmake`](#).

Fibonacci Walkthrough (F77)

We will consider the `fib` example from *Three ways to wrap - getting started* section.

```
C FILE: FIB1.F
      SUBROUTINE FIB (A,N)
C
C      CALCULATE FIRST N FIBONACCI NUMBERS
C
      INTEGER N
      REAL*8 A(N)
      DO I=1,N
        IF (I.EQ.1) THEN
          A(I) = 0.0D0
        ELSEIF (I.EQ.2) THEN
          A(I) = 1.0D0
        ELSE
          A(I) = A(I-1) + A(I-2)
        ENDIF
      ENDDO
      END
C END FILE FIB1.F
```

CMake modules only

Consider using the following `CMakeLists.txt`.

```
### setup project ###
cmake_minimum_required(VERSION 3.9)

project(fibby
  VERSION 1.0
  DESCRIPTION "FIB module"
  LANGUAGES C Fortran
)

# Safety net
if(PROJECT_SOURCE_DIR STREQUAL PROJECT_BINARY_DIR)
  message(
```

(continues on next page)

(continued from previous page)

```

        FATAL_ERROR
        "In-source builds not allowed. Please make a new directory (called a build_
↳directory) and run CMake from there.\n"
    )
endif()

# Ensure scikit-build modules
if (NOT SKBUILD)
    find_package(PythonInterp 3.8 REQUIRED)
    # Kanged --> https://github.com/Kitware/torch_liberator/blob/master/CMakeLists.txt
    # If skbuild is not the driver; include its utilities in CMAKE_MODULE_PATH
    execute_process(
        COMMAND "${PYTHON_EXECUTABLE}"
        -c "import os, skbuild; print(os.path.dirname(skbld.__file__))"
        OUTPUT_VARIABLE SKBLD_DIR
        OUTPUT_STRIP_TRAILING_WHITESPACE
    )
    list(APPEND CMAKE_MODULE_PATH "${SKBLD_DIR}/resources/cmake")
    message(STATUS "Looking in ${SKBLD_DIR}/resources/cmake for CMake modules")
endif()

# scikit-build style includes
find_package(PythonExtensions REQUIRED) # for ${PYTHON_EXTENSION_MODULE_SUFFIX}

# Grab the variables from a local Python installation
# NumPy headers
execute_process(
    COMMAND "${PYTHON_EXECUTABLE}"
    -c "import numpy; print(numpy.get_include())"
    OUTPUT_VARIABLE NumPy_INCLUDE_DIRS
    OUTPUT_STRIP_TRAILING_WHITESPACE
)
# F2PY headers
execute_process(
    COMMAND "${PYTHON_EXECUTABLE}"
    -c "import numpy.f2py; print(numpy.f2py.get_include())"
    OUTPUT_VARIABLE F2PY_INCLUDE_DIR
    OUTPUT_STRIP_TRAILING_WHITESPACE
)

# Prepping the module
set(f2py_module_name "fibby")
set(fortran_src_file "${CMAKE_SOURCE_DIR}/fib1.f")
set(f2py_module_c "${f2py_module_name}.module.c")

# Target for enforcing dependencies
add_custom_target(genpyf
    DEPENDS "${fortran_src_file}"
)
add_custom_command(
    OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/${f2py_module_c}"
    COMMAND ${PYTHON_EXECUTABLE} -m "numpy.f2py"
        "${fortran_src_file}"
        -m "fibby"
        --lower # Important
    DEPENDS fib1.f # Fortran source
)

```

(continues on next page)

(continued from previous page)

```

add_library(${CMAKE_PROJECT_NAME} MODULE
    "${f2py_module_name}module.c"
    "${F2PY_INCLUDE_DIR}/fortranobject.c"
    "${fortran_src_file}")

target_include_directories(${CMAKE_PROJECT_NAME} PUBLIC
    ${F2PY_INCLUDE_DIR}
    ${NumPy_INCLUDE_DIRS}
    ${PYTHON_INCLUDE_DIRS})
set_target_properties(${CMAKE_PROJECT_NAME} PROPERTIES SUFFIX "${PYTHON_EXTENSION_
↪MODULE_SUFFIX}")
set_target_properties(${CMAKE_PROJECT_NAME} PROPERTIES PREFIX "")

# Linker fixes
if (UNIX)
    if (APPLE)
        set_target_properties(${CMAKE_PROJECT_NAME} PROPERTIES
            LINK_FLAGS '-Wl,-dylib,-undefined,dynamic_lookup')
    else()
        set_target_properties(${CMAKE_PROJECT_NAME} PROPERTIES
            LINK_FLAGS '-Wl,--allow-shlib-undefined')
    endif()
endif()

add_dependencies(${CMAKE_PROJECT_NAME} genpyf)

install(TARGETS ${CMAKE_PROJECT_NAME} DESTINATION fibby)

```

Much of the logic is the same as in *Using via cmake*, however notably here the appropriate module suffix is generated via `sysconfig.get_config_var("SO")`. The resulting extension can be built and loaded in the standard workflow.

```

ls .
# CMakeLists.txt fib1.f
cmake -S . -B build
cmake --build build
cd build
python -c "import numpy as np; import fibby; a = np.zeros(9); fibby.fib(a); print (a)"
# [ 0.  1.  1.  2.  3.  5.  8. 13. 21.]

```

setuptools replacement

Note: As of November 2021

The behavior described here of driving the `cmake` build of a module is considered to be legacy behaviour and should not be depended on.

The utility of `scikit-build` lies in being able to drive the generation of more than extension modules, in particular a common usage pattern is the generation of Python distributables (for example for PyPI).

The workflow with `scikit-build` straightforwardly supports such packaging requirements. Consider augmenting the project with a `setup.py` as defined:

```

from skbuild import setup

setup(
    name="fibby",
    version="0.0.1",
    description="a minimal example package (fortran version)",
    license="MIT",
    packages=['fibby'],
    python_requires=">=3.7",
)

```

Along with a commensurate `pyproject.toml`

```

[build-system]
requires = ["setuptools>=42", "wheel", "scikit-build", "cmake>=3.9", "numpy>=1.21"]
build-backend = "setuptools.build_meta"

```

Together these can build the extension using `cmake` in tandem with other standard `setuptools` outputs. Running `cmake` through `setup.py` is mostly used when it is necessary to integrate with extension modules not built with `cmake`.

```

ls .
# CMakeLists.txt fib1.f pyproject.toml setup.py
python setup.py build_ext --inplace
python -c "import numpy as np; import fibby.fibby; a = np.zeros(9); fibby.fibby.
↪fib(a); print (a)"
# [ 0.  1.  1.  2.  3.  5.  8. 13. 21.]

```

Where we have modified the path to the module as `--inplace` places the extension module in a subfolder.

11.3.4 Advanced F2PY use cases

Adding user-defined functions to F2PY generated modules

User-defined Python C/API functions can be defined inside signature files using `usercode` and `pymethoddef` statements (they must be used inside the `python module` block). For example, the following signature file `spam.pyf`

```

!      -*- f90 -*-
python module spam
    usercode '''
    static char doc_spam_system[] = "Execute a shell command.";
    static PyObject *spam_system(PyObject *self, PyObject *args)
    {
        char *command;
        int sts;

        if (!PyArg_ParseTuple(args, "s", &command))
            return NULL;
        sts = system(command);
        return Py_BuildValue("i", sts);
    }
    '''
    pymethoddef '''
    {"system", spam_system, METH_VARARGS, doc_spam_system},
    '''
end python module spam

```


wraps the C library function `system()`:

```
f2py -c spam.pyf
```

In Python this can then be used as:

```
>>> import spam
>>> status = spam.system('whoami')
pearu
>>> status = spam.system('blah')
sh: line 1: blah: command not found
```

Adding user-defined variables

The following example illustrates how to add user-defined variables to a F2PY generated extension module by modifying the dictionary of a F2PY generated module. Consider the following signature file (compiled with `f2py -c var.pyf`):

```
!      -*- f90 -*-
python module var
  usercode '''
    int BAR = 5;
    '''
  interface
    usercode '''
      PyDict_SetItemString(d, "BAR", PyInt_FromLong(BAR));
    '''
  end interface
end python module
```

Notice that the second `usercode` statement must be defined inside an `interface` block and the module dictionary is available through the variable `d` (see `varmodule.c` generated by `f2py var.pyf` for additional details).

Usage in Python:

```
>>> import var
>>> var.BAR
5
```

Dealing with KIND specifiers

Currently, F2PY can handle only `<type spec>(kind=<kindselector>)` declarations where `<kindselector>` is a numeric integer (e.g. 1, 2, 4,...), but not a function call `KIND(..)` or any other expression. F2PY needs to know what would be the corresponding C type and a general solution for that would be too complicated to implement.

However, F2PY provides a hook to overcome this difficulty, namely, users can define their own `<Fortran type>` to `<C type>` maps. For example, if Fortran 90 code contains:

```
REAL(kind=KIND(0.0D0)) ...
```

then create a mapping file containing a Python dictionary:

```
{ 'real': { 'KIND(0.0D0)': 'double' } }
```

for instance.

Use the `--f2cmap` command-line option to pass the file name to F2PY. By default, F2PY assumes file name is `f2py_f2cmap` in the current working directory.

More generally, the `f2cmap` file must contain a dictionary with items:

```
<Fortran typespec> : {<selector_expr>:<C type>}
```

that defines mapping between Fortran type:

```
<Fortran typespec> ([kind=]<selector_expr>)
```

and the corresponding `<C type>`. The `<C type>` can be one of the following:

```
double
float
long_double
char
signed_char
unsigned_char
short
unsigned_short
int
long
long_long
unsigned
complex_float
complex_double
complex_long_double
string
```

For more information, see the F2PY source code `numpy/f2py/capi_maps.py`.

11.3.5 F2PY test suite

F2PY's test suite is present in the directory `numpy/f2py/tests`. Its aim is to ensure that Fortran language features are correctly translated to Python. For example, the user can specify starting and ending indices of arrays in Fortran. This behaviour is translated to the generated CPython library where the arrays strictly start from 0 index.

The directory of the test suite looks like the following:

```
./tests/
├── __init__.py
├── src
│   ├── abstract_interface
│   ├── array_from_pyobj
│   ├── // ... several test folders
│   └── string
├── test_abstract_interface.py
├── test_array_from_pyobj.py
├── // ... several test files
├── test_symbolic.py
└── util.py
```

Files starting with `test_` contain tests for various aspects of `f2py` from parsing Fortran files to checking modules' documentation. `src` directory contains the Fortran source files upon which we do the testing. `util.py` contains utility functions for building and importing Fortran modules during test time using a temporary location.

Adding a test

F2PY's current test suite predates `pytest` and therefore does not use fixtures. Instead, the test files contain test classes that inherit from `F2PyTest` class present in `util.py`.

```

1 class F2PyTest:
2     code = None
3     sources = None
4     options = []
5     skip = []
6     only = []
7     suffix = ".f"
8     module = None
9     module_name = None
10

```

This class many helper functions for parsing and compiling test source files. Its child classes can override its `sources` data member to provide their own source files. This superclass will then compile the added source files upon object creation and their functions will be appended to `self.module` data member. Thus, the child classes will be able to access the fortran functions specified in source file by calling `self.module.[fortran_function_name]`.

Example

Consider the following subroutines, contained in a file named `add-test.f`

```

subroutine addb(k)
    real(8), intent(inout) :: k(:)
    k=k+1
endsubroutine

subroutine addc(w,k)
    real(8), intent(in) :: w(:)
    real(8), intent(out) :: k(size(w))
    k=w+1
endsubroutine

```

The first routine *addb* simply takes an array and increases its elements by 1. The second subroutine *addc* assigns a new array *k* with elements greater than the elements of the input array *w* by 1.

A test can be implemented as follows:

```

class TestAdd(util.F2PyTest):
    sources = [util.getpath("add-test.f")]

    def test_module(self):
        k = np.array([1, 2, 3], dtype=np.float64)
        w = np.array([1, 2, 3], dtype=np.float64)
        self.module.subb(k)
        assert np.allclose(k, w + 1)
        self.module.subc([w, k])
        assert np.allclose(k, w + 1)

```

We override the `sources` data member to provide the source file. The source files are compiled and subroutines are attached to module data member when the class object is created. The `test_module` function calls the subroutines and tests their results.

11.4 F2PY and Windows

Warning: F2PY support for Windows is not at par with Linux support, and OS specific flags can be seen via `python -m numpy.f2py`

Broadly speaking, there are two issues working with F2PY on Windows:

- the lack of actively developed FOSS Fortran compilers, and,
- the linking issues related to the C runtime library for building Python-C extensions.

The focus of this section is to establish a guideline for developing and extending Fortran modules for Python natively, via F2PY on Windows.

11.4.1 Overview

From a user perspective, the most UNIX compatible Windows development environment is through emulation, either via the Windows Subsystem on Linux, or facilitated by Docker. In a similar vein, traditional virtualization methods like VirtualBox are also reasonable methods to develop UNIX tools on Windows.

Native Windows support is typically stunted beyond the usage of commercial compilers. However, as of 2022, most commercial compilers have free plans which are sufficient for general use. Additionally, the Fortran language features supported by `f2py` (partial coverage of Fortran 2003), means that newer toolchains are often not required. Briefly, then, for an end user, in order of use:

Classic Intel Compilers (commercial)

These are maintained actively, though licensing restrictions may apply as further detailed in *F2PY and Windows Intel Fortran*.

Suitable for general use for those building native Windows programs by building off of MSVC.

MSYS2 (FOSS)

In conjunction with the `mingw-w64` project, `gfortran` and `gcc` toolchains can be used to natively build Windows programs.

Windows Subsystem for Linux

Assuming the usage of `gfortran`, this can be used for cross-compiling Windows applications, but is significantly more complicated.

Conda

Windows support for compilers in `conda` is facilitated by pulling MSYS2 binaries, however these are *outdated*, and therefore not recommended (as of 30-01-2022).

PGI Compilers (commercial)

Unmaintained but sufficient if an existing license is present. Works natively, but has been superseded by the Nvidia HPC SDK, with no *native Windows support*.

Cygwin (FOSS)

Can also be used for `gfortran`. However, the POSIX API compatibility layer provided by Cygwin is meant to compile UNIX software on Windows, instead of building native Windows programs. This means cross compilation is required.

The compilation suites described so far are compatible with the [now deprecated](#) `np.distutils` build backend which is exposed by the F2PY CLI. Additional build system usage (`meson`, `cmake`) as described in [F2PY and Build Systems](#) allows for a more flexible set of compiler backends including:

Intel oneAPI

The newer Intel compilers (`ifx`, `icx`) are based on LLVM and can be used for native compilation. Licensing requirements can be onerous.

Classic Flang (FOSS)

The backbone of the PGI compilers were cannibalized to form the “classic” or [legacy version of Flang](#). This may be compiled from source and used natively. [LLVM Flang](#) does not support Windows yet (30-01-2022).

LFortran (FOSS)

One of two LLVM based compilers. Not all of F2PY supported Fortran can be compiled yet (30-01-2022) but uses MSVC for native linking.

11.4.2 Baseline

For this document we will assume the following basic tools:

- The IDE being considered is the community supported [Microsoft Visual Studio Code](#)
- The terminal being used is the [Windows Terminal](#)
- The shell environment is assumed to be [Powershell 7.x](#)
- **Python 3.10 from the [Microsoft Store](#) and this can be tested with**

```
Get-Command python.exe resolving to C:\Users\%USERNAME%\AppData\Local\
Microsoft\WindowsApps\python.exe
```

- The Microsoft Visual C++ (MSVC) toolset

With this baseline configuration, we will further consider a configuration matrix as follows:

Table 1: Support matrix, exe implies a Windows installer

Fortran Compiler	C/C++ Compiler	Source
Intel Fortran	MSVC / ICC	exe
GFortran	MSVC	MSYS2/exe
GFortran	GCC	WSL
Classic Flang	MSVC	Source / Conda
Anaconda GFortran	Anaconda GCC	exe

For an understanding of the key issues motivating the need for such a matrix [Pauli Virtanen’s in-depth post on wheels with Fortran for Windows](#) is an excellent resource. An entertaining explanation of an application binary interface (ABI) can be found in this post by [JeanHeyd Meneide](#).

11.4.3 Powershell and MSVC

MSVC is installed either via the Visual Studio Bundle or the lighter (preferred) [Build Tools for Visual Studio](#) with the Desktop development with C++ setting.

Note: This can take a significant amount of time as it includes a download of around 2GB and requires a restart.

It is possible to use the resulting environment from a [standard command prompt](#). However, it is more pleasant to use a [developer powershell](#), with a [profile in Windows Terminal](#). This can be achieved by adding the following block to the profiles->list section of the JSON file used to configure Windows Terminal (see Settings->Open JSON file):

```
{
  "name": "Developer PowerShell for VS 2019",
  "commandline": "powershell.exe -noe -c \"$vsPath = (Join-Path ${env:ProgramFiles(x86)}
  ↪ -ChildPath 'Microsoft Visual Studio\\2019\\BuildTools'); Import-Module (Join-Path
  ↪ $vsPath 'Common7\\Tools\\Microsoft.VisualStudio.DevShell.dll'); Enter-VsDevShell -
  ↪ VsInstallPath $vsPath -SkipAutomaticLocation\"",
  "icon": "ms-appx:///ProfileIcons/{61c54bbd-c2c6-5271-96e7-009a87ff44bf}.png"
}
```

Now, testing the compiler toolchain could look like:

```
# New Windows Developer Powershell instance / tab
# or
$vsPath = (Join-Path ${env:ProgramFiles(x86)} -ChildPath 'Microsoft Visual Studio\\
↪ 2019\\BuildTools');
Import-Module (Join-Path $vsPath 'Common7\\Tools\\Microsoft.VisualStudio.DevShell.dll
↪ ');
Enter-VsDevShell -VsInstallPath $vsPath -SkipAutomaticLocation
*****
** Visual Studio 2019 Developer PowerShell v16.11.9
** Copyright (c) 2021 Microsoft Corporation
*****
cd $HOME
echo "#include<stdio.h>" > blah.cpp; echo 'int main(){printf("Hi");return 1;}' >>_
↪ blah.cpp
cl blah.cpp
.\blah.exe
# Hi
rm blah.cpp
```

It is also possible to check that the environment has been updated correctly with `$ENV:PATH`.

11.4.4 Windows Store Python Paths

The MS Windows version of Python discussed here installs to a non-deterministic path using a hash. This needs to be added to the `PATH` variable.

```
$Env:Path += ";$env:LOCALAPPDATA\packages\pythonsoftwarefoundation.python.3.10_
↪ qbz5n2kf8ap0\localcache\local-packages\python310\scripts"
```

F2PY and Windows Intel Fortran

As of NumPy 1.23, only the classic Intel compilers (`ifort`) are supported.

Note: The licensing restrictions for beta software [have been relaxed](#) during the transition to the LLVM backed `ifx/icc` family of compilers. However this document does not endorse the usage of Intel in downstream projects due to the issues pertaining to [disassembly of components and liability](#).

Neither the Python Intel installation nor the *Classic Intel C/C++ Compiler* are required.

- The [Intel Fortran Compilers](#) come in a combined installer providing both Classic and Beta versions; these also take around a gigabyte and a half or so.

We will consider the classic example of the generation of Fibonacci numbers, `fib1.f`, given by:

```
C FILE: FIB1.F
  SUBROUTINE FIB(A,N)
C
C   CALCULATE FIRST N FIBONACCI NUMBERS
C
  INTEGER N
  REAL*8 A(N)
  DO I=1,N
    IF (I.EQ.1) THEN
      A(I) = 0.0D0
    ELSEIF (I.EQ.2) THEN
      A(I) = 1.0D0
    ELSE
      A(I) = A(I-1) + A(I-2)
    ENDIF
  ENDDO
  END
C END FILE FIB1.F
```

For `cmd.exe` fans, using the Intel oneAPI command prompt is the easiest approach, as it loads the required environment for both `ifort` and `msvc`. Helper batch scripts are also provided.

```
# cmd.exe
"C:\Program Files (x86)\Intel\oneAPI\setvars.bat"
python -m numpy.f2py -c fib1.f -m fib1
python -c "import fib1; import numpy as np; a=np.zeros(8); fib1.fib(a); print(a)"
```

Powershell usage is a little less pleasant, and this configuration now works with MSVC as:

```
# Powershell
python -m numpy.f2py -c fib1.f -m fib1 --f77exec='C:\Program Files (x86)\Intel\oneAPI\
→compiler\latest\windows\bin\intel64\ifort.exe' --f90exec='C:\Program Files (x86)\
→Intel\oneAPI\compiler\latest\windows\bin\intel64\ifort.exe' -L'C:\Program Files.
→(x86)\Intel\oneAPI\compiler\latest\windows\compiler\lib\ia32'
python -c "import fib1; import numpy as np; a=np.zeros(8); fib1.fib(a); print(a)"
# Alternatively, set environment and reload Powershell in one line
cmd.exe /k "C:\Program Files (x86)\Intel\oneAPI\setvars.bat" && powershell'
python -m numpy.f2py -c fib1.f -m fib1
python -c "import fib1; import numpy as np; a=np.zeros(8); fib1.fib(a); print(a)"
```

Note that the actual path to your local installation of *ifort* may vary, and the command above will need to be updated accordingly.

F2PY and Windows with MSYS2

Follow the standard [installation instructions](#). Then, to grab the requisite Fortran compiler with MVSC:

```
# Assuming a fresh install
pacman -Syu # Restart the terminal
pacman -Su  # Update packages
# Get the toolchains
pacman -S --needed base-devel gcc-fortran
pacman -S mingw-w64-x86_64-toolchain
```

F2PY and Conda on Windows

As a convenience measure, we will additionally assume the existence of `scoop`, which can be used to install tools without administrative access.

```
Invoke-Expression (New-Object System.Net.WebClient).DownloadString('https://get.scoop.
↪sh')
```

Now we will setup a conda environment.

```
scoop install miniconda3
# For conda activate / deactivate in powershell
conda install -n root -c pscondaenvs pscondaenvs
Powershell -c Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
conda init powershell
# Open a new shell for the rest
```

conda pulls packages from `msys2`, however, the UX is sufficiently different enough to warrant a separate discussion.

Warning: As of 30-01-2022, the `MSYS2` binaries shipped with conda are **outdated** and this approach is **not preferred**.

F2PY and PGI Fortran on Windows

A variant of these are part of the so called “classic” Flang, however, as classic Flang requires a custom LLVM and compilation from sources.

Warning: Since the proprietary compilers are no longer available for usage they are not recommended and will not be ported to the new `f2py` CLI.

Note: As of November 2021

As of 29-01-2022, `PGI compiler toolchains` have been superseded by the Nvidia HPC SDK, with no `native Windows` support.

However,

GLOSSARY

(n,)

A parenthesized number followed by a comma denotes a tuple with one element. The trailing comma distinguishes a one-element tuple from a parenthesized `n`.

-1

- **In a dimension entry**, instructs NumPy to choose the length that will keep the total number of array elements the same.

```
>>> np.arange(12).reshape(4, -1).shape
(4, 3)
```

- **In an index**, any negative value denotes indexing from the right.

...

An **Ellipsis**.

- **When indexing an array**, shorthand that the missing axes, if they exist, are full slices.

```
>>> a = np.arange(24).reshape(2, 3, 4)
```

```
>>> a[...].shape
(2, 3, 4)
```

```
>>> a[:, :, 0].shape
(2, 3)
```

```
>>> a[0, ...].shape
(3, 4)
```

```
>>> a[0, ..., 0].shape
(3, )
```

It can be used at most once; `a[:, :, 0, ...]` raises an `IndexError`.

- **In printouts**, NumPy substitutes `...` for the middle elements of large arrays. To see the entire array, use `numpy.printoptions`

:

The Python **slice** operator. In ndarrays, slicing can be applied to every axis:

```
>>> a = np.arange(24).reshape(2,3,4)
>>> a
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

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

>>> a[1:,-2:,:-1]
array([[[16, 17, 18],
        [20, 21, 22]]])
```

Trailing slices can be omitted:

```
>>> a[1] == a[1,:,:)
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

In contrast to Python, where slicing creates a copy, in NumPy slicing creates a *view*.

For details, see [Combining advanced and basic indexing](#).

<

In a dtype declaration, indicates that the data is *little-endian* (the bracket is big on the right).

```
>>> dt = np.dtype('<f') # little-endian single-precision float
```

>

In a dtype declaration, indicates that the data is *big-endian* (the bracket is big on the left).

```
>>> dt = np.dtype('>H') # big-endian unsigned short
```

advanced indexing

Rather than using a scalar or slice as an index, an axis can be indexed with an array, providing fine-grained selection. This is known as *advanced indexing* or “fancy indexing”.

along an axis

An operation *along axis n* of array a behaves as if its argument were an array of slices of a where each slice has a successive index of axis n .

For example, if a is a $3 \times N$ array, an operation along axis 0 behaves as if its argument were an array containing slices of each row:

```
>>> np.array((a[0,:], a[1,:], a[2,:]))
```

To make it concrete, we can pick the operation to be the array-reversal function `numpy.flip`, which accepts an `axis` argument. We construct a 3×4 array a :

```
>>> a = np.arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Reversing along axis 0 (the row axis) yields

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

Recalling the definition of *along an axis*, flip along axis 0 is treating its argument as if it were

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

and the result of `np.flip(a,axis=0)` is to reverse the slices:

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

array

Used synonymously in the NumPy docs with *ndarray*.

array_like

Any scalar or *sequence* that can be interpreted as an ndarray. In addition to ndarrays and scalars this category includes lists (possibly nested and with different element types) and tuples. Any argument accepted by `numpy.array` is *array_like*.

```
>>> a = np.array([[1, 2.0], [0, 0], (1+1j, 3.)])

>>> a
array([[1.+0.j, 2.+0.j],
       [0.+0.j, 0.+0.j],
       [1.+1.j, 3.+0.j]])
```

array scalar

An array scalar is an instance of the types/classes `float32`, `float64`, etc.. For uniformity in handling operands, NumPy treats a scalar as an array of zero dimension. In contrast, a 0-dimensional array is an ndarray instance containing precisely one value.

axis

Another term for an array dimension. Axes are numbered left to right; axis 0 is the first element in the shape tuple.

In a two-dimensional vector, the elements of axis 0 are rows and the elements of axis 1 are columns.

In higher dimensions, the picture changes. NumPy prints higher-dimensional vectors as replications of row-by-column building blocks, as in this three-dimensional vector:

```
>>> a = np.arange(12).reshape(2,2,3)
>>> a
array([[[ 0,  1,  2],
       [ 3,  4,  5]],
       [[ 6,  7,  8],
       [ 9, 10, 11]]])
```

`a` is depicted as a two-element array whose elements are 2x3 vectors. From this point of view, rows and columns are the final two axes, respectively, in any shape.

This rule helps you anticipate how a vector will be printed, and conversely how to find the index of any of the printed elements. For instance, in the example, the last two values of 8's index must be 0 and 2. Since 8 appears in the second of the two 2x3's, the first index must be 1:

```
>>> a[1,0,2]
8
```

A convenient way to count dimensions in a printed vector is to count `[` symbols after the open-parenthesis. This is useful in distinguishing, say, a (1,2,3) shape from a (2,3) shape:

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

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

.base

If an array does not own its memory, then its `base` attribute returns the object whose memory the array is referencing. That object may be referencing the memory from still another object, so the owning object may be `a.base.base.base...`. Some writers erroneously claim that testing `base` determines if arrays are *views*. For the correct way, see `numpy.shares_memory`.

big-endian

See [Endianness](#).

BLAS

[Basic Linear Algebra Subprograms](#)

broadcast

broadcasting is NumPy's ability to process ndarrays of different sizes as if all were the same size.

It permits an elegant do-what-I-mean behavior where, for instance, adding a scalar to a vector adds the scalar value to every element.

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

```
>>> a + [3, 3, 3]
array([3, 4, 5])
```

```
>>> a + 3
array([3, 4, 5])
```

Ordinarily, vector operands must all be the same size, because NumPy works element by element – for instance, `c = a * b` is

```
c[0,0,0] = a[0,0,0] * b[0,0,0]
c[0,0,1] = a[0,0,1] * b[0,0,1]
...
```

But in certain useful cases, NumPy can duplicate data along “missing” axes or “too-short” dimensions so shapes will match. The duplication costs no memory or time. For details, see [Broadcasting](#).

C order

Same as [row-major](#).

column-major

See [Row- and column-major order](#).

contiguous

An array is contiguous if:

- it occupies an unbroken block of memory, and
- array elements with higher indexes occupy higher addresses (that is, no [stride](#) is negative).

There are two types of proper-contiguous NumPy arrays:

- Fortran-contiguous arrays refer to data that is stored column-wise, i.e. the indexing of data as stored in memory starts from the lowest dimension;
- C-contiguous, or simply contiguous arrays, refer to data that is stored row-wise, i.e. the indexing of data as stored in memory starts from the highest dimension.

For one-dimensional arrays these notions coincide.

For example, a 2x2 array `A` is Fortran-contiguous if its elements are stored in memory in the following order:

```
A[0,0] A[1,0] A[0,1] A[1,1]
```

and C-contiguous if the order is as follows:

```
A[0,0] A[0,1] A[1,0] A[1,1]
```

To test whether an array is C-contiguous, use the `.flags.c_contiguous` attribute of NumPy arrays. To test for Fortran contiguity, use the `.flags.f_contiguous` attribute.

copy

See [view](#).

dimension

See [axis](#).

dtype

The datatype describing the (identically typed) elements in an ndarray. It can be changed to reinterpret the array contents. For details, see [Data type objects \(dtype\)](#).

fancy indexing

Another term for [advanced indexing](#).

field

In a *structured data type*, each subtype is called a *field*. The *field* has a name (a string), a type (any valid dtype), and an optional *title*. See `arrays.dtypes`.

Fortran order

Same as *column-major*.

flattened

See *ravel*.

homogeneous

All elements of a homogeneous array have the same type. `ndarrays`, in contrast to Python lists, are homogeneous. The type can be complicated, as in a *structured array*, but all elements have that type.

NumPy *object arrays*, which contain references to Python objects, fill the role of heterogeneous arrays.

itemsize

The size of the dtype element in bytes.

little-endian

See *Endianness*.

mask

A boolean array used to select only certain elements for an operation:

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

```
>>> mask = (x > 2)
>>> mask
array([False, False, False,  True,  True])
```

```
>>> x[mask] = -1
>>> x
array([ 0,  1,  2, -1, -1])
```

masked array

Bad or missing data can be cleanly ignored by putting it in a masked array, which has an internal boolean array indicating invalid entries. Operations with masked arrays ignore these entries.

```
>>> a = np.ma.masked_array([np.nan, 2, np.nan], [True, False, True])
>>> a
masked_array(data=[--, 2.0, --],
             mask=[ True, False,  True],
             fill_value=1e+20)

>>> a + [1, 2, 3]
masked_array(data=[--, 4.0, --],
             mask=[ True, False,  True],
             fill_value=1e+20)
```

For details, see *Masked arrays*.

matrix

NumPy's two-dimensional matrix class should no longer be used; use regular ndarrays.

ndarray

NumPy's basic structure.

object array

An array whose dtype is `object`; that is, it contains references to Python objects. Indexing the array dereferences the Python objects, so unlike other ndarrays, an object array has the ability to hold heterogeneous objects.

ravel

`numpy.ravel` and `numpy.flatten` both flatten an ndarray. `ravel` will return a view if possible; `flatten` always returns a copy.

Flattening collapses a multidimensional array to a single dimension; details of how this is done (for instance, whether `a[n+1]` should be the next row or next column) are parameters.

record array

A *structured array* with allowing access in an attribute style (`a.field`) in addition to `a['field']`. For details, see `numpy.recarray`.

row-major

See [Row- and column-major order](#). NumPy creates arrays in row-major order by default.

scalar

In NumPy, usually a synonym for *array scalar*.

shape

A tuple showing the length of each dimension of an ndarray. The length of the tuple itself is the number of dimensions (`numpy.ndim`). The product of the tuple elements is the number of elements in the array. For details, see `numpy.ndarray.shape`.

stride

Physical memory is one-dimensional; strides provide a mechanism to map a given index to an address in memory. For an N-dimensional array, its `strides` attribute is an N-element tuple; advancing from index `i` to index `i+1` on axis `n` means adding `a.strides[n]` bytes to the address.

Strides are computed automatically from an array's dtype and shape, but can be directly specified using `as_strided`.

For details, see `numpy.ndarray.strides`.

To see how striding underlies the power of NumPy views, see [The NumPy array: a structure for efficient numerical computation](#).

structured array

Array whose dtype is a *structured data type*.

structured data type

Users can create arbitrarily complex dtypes that can include other arrays and dtypes. These composite dtypes are called *structured data types*.

subarray

An array nested in a *structured data type*, as `b` is here:

```
>>> dt = np.dtype([('a', np.int32), ('b', np.float32, (3,))])
>>> np.zeros(3, dtype=dt)
array([(0, [0., 0., 0.]), (0, [0., 0., 0.]), (0, [0., 0., 0.])],
      dtype=[('a', '<i4'), ('b', '<f4', (3,))])
```

subarray data type

An element of a structured datatype that behaves like an ndarray.

title

An alias for a field name in a structured datatype.

type

In NumPy, usually a synonym for *dtype*. For the more general Python meaning, [see here](#).

ufunc

NumPy's fast element-by-element computation (*vectorization*) gives a choice which function gets applied. The general term for the function is *ufunc*, short for *universal function*. NumPy routines have built-in ufuncs, but users can also write their own.

vectorization

NumPy hands off array processing to C, where looping and computation are much faster than in Python. To exploit this, programmers using NumPy eliminate Python loops in favor of array-to-array operations. *vectorization* can refer both to the C offloading and to structuring NumPy code to leverage it.

view

Without touching underlying data, NumPy can make one array appear to change its datatype and shape.

An array created this way is a *view*, and NumPy often exploits the performance gain of using a view versus making a new array.

A potential drawback is that writing to a view can alter the original as well. If this is a problem, NumPy instead needs to create a physically distinct array – a *copy*.

Some NumPy routines always return views, some always return copies, some may return one or the other, and for some the choice can be specified. Responsibility for managing views and copies falls to the programmer. `numpy.shares_memory` will check whether `b` is a view of `a`, but an exact answer isn't always feasible, as the documentation page explains.

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

```
>>> y = x[::2]
>>> y
array([0, 2, 4])
```

```
>>> x[0] = 3 # changing x changes y as well, since y is a view on x
>>> y
array([3, 2, 4])
```


UNDER-THE-HOOD DOCUMENTATION FOR DEVELOPERS

These documents are intended as a low-level look into NumPy; focused towards developers.

13.1 Internal organization of NumPy arrays

It helps to understand a bit about how NumPy arrays are handled under the covers to help understand NumPy better. This section will not go into great detail. Those wishing to understand the full details are requested to refer to Travis Oliphant's book [Guide to NumPy](#).

NumPy arrays consist of two major components: the raw array data (from now on, referred to as the data buffer), and the information about the raw array data. The data buffer is typically what people think of as arrays in C or Fortran, a *contiguous* (and fixed) block of memory containing fixed-sized data items. NumPy also contains a significant set of data that describes how to interpret the data in the data buffer. This extra information contains (among other things):

- 1) The basic data element's size in bytes.
- 2) The start of the data within the data buffer (an offset relative to the beginning of the data buffer).
- 3) The number of *dimensions* and the size of each dimension.
- 4) The separation between elements for each dimension (the *stride*). This does not have to be a multiple of the element size.
- 5) The byte order of the data (which may not be the native byte order).
- 6) Whether the buffer is read-only.
- 7) Information (via the `dtype` object) about the interpretation of the basic data element. The basic data element may be as simple as an int or a float, or it may be a compound object (e.g., *struct-like*), a fixed character field, or Python object pointers.
- 8) Whether the array is to be interpreted as *C-order* or *Fortran-order*.

This arrangement allows for the very flexible use of arrays. One thing that it allows is simple changes to the metadata to change the interpretation of the array buffer. Changing the byteorder of the array is a simple change involving no rearrangement of the data. The *shape* of the array can be changed very easily without changing anything in the data buffer or any data copying at all.

Among other things that are made possible is one can create a new array metadata object that uses the same data buffer to create a new *view* of that data buffer that has a different interpretation of the buffer (e.g., different shape, offset, byte order, strides, etc) but shares the same data bytes. Many operations in NumPy do just this such as *slicing*. Other operations, such as transpose, don't move data elements around in the array, but rather change the information about the shape and strides so that the indexing of the array changes, but the data in the doesn't move.

Typically these new versions of the array metadata but the same data buffer are new views into the data buffer. There is a different `ndarray` object, but it uses the same data buffer. This is why it is necessary to force copies through the use of the `copy` method if one really wants to make a new and independent copy of the data buffer.

New views into arrays mean the object reference counts for the data buffer increase. Simply doing away with the original array object will not remove the data buffer if other views of it still exist.

13.1.1 Multidimensional array indexing order issues

See also:

Indexing on ndarrays

What is the right way to index multi-dimensional arrays? Before you jump to conclusions about the one and true way to index multi-dimensional arrays, it pays to understand why this is a confusing issue. This section will try to explain in detail how NumPy indexing works and why we adopt the convention we do for images, and when it may be appropriate to adopt other conventions.

The first thing to understand is that there are two conflicting conventions for indexing 2-dimensional arrays. Matrix notation uses the first index to indicate which row is being selected and the second index to indicate which column is selected. This is opposite the geometrically oriented-convention for images where people generally think the first index represents x position (i.e., column) and the second represents y position (i.e., row). This alone is the source of much confusion; matrix-oriented users and image-oriented users expect two different things with regard to indexing.

The second issue to understand is how indices correspond to the order in which the array is stored in memory. In Fortran, the first index is the most rapidly varying index when moving through the elements of a two-dimensional array as it is stored in memory. If you adopt the matrix convention for indexing, then this means the matrix is stored one column at a time (since the first index moves to the next row as it changes). Thus Fortran is considered a Column-major language. C has just the opposite convention. In C, the last index changes most rapidly as one moves through the array as stored in memory. Thus C is a Row-major language. The matrix is stored by rows. Note that in both cases it presumes that the matrix convention for indexing is being used, i.e., for both Fortran and C, the first index is the row. Note this convention implies that the indexing convention is invariant and that the data order changes to keep that so.

But that's not the only way to look at it. Suppose one has large two-dimensional arrays (images or matrices) stored in data files. Suppose the data are stored by rows rather than by columns. If we are to preserve our index convention (whether matrix or image) that means that depending on the language we use, we may be forced to reorder the data if it is read into memory to preserve our indexing convention. For example, if we read row-ordered data into memory without reordering, it will match the matrix indexing convention for C, but not for Fortran. Conversely, it will match the image indexing convention for Fortran, but not for C. For C, if one is using data stored in row order, and one wants to preserve the image index convention, the data must be reordered when reading into memory.

In the end, what you do for Fortran or C depends on which is more important, not reordering data or preserving the indexing convention. For large images, reordering data is potentially expensive, and often the indexing convention is inverted to avoid that.

The situation with NumPy makes this issue yet more complicated. The internal machinery of NumPy arrays is flexible enough to accept any ordering of indices. One can simply reorder indices by manipulating the internal *stride* information for arrays without reordering the data at all. NumPy will know how to map the new index order to the data without moving the data.

So if this is true, why not choose the index order that matches what you most expect? In particular, why not define row-ordered images to use the image convention? (This is sometimes referred to as the Fortran convention vs the C convention, thus the 'C' and 'FORTRAN' order options for array ordering in NumPy.) The drawback of doing this is potential performance penalties. It's common to access the data sequentially, either implicitly in array operations or explicitly by looping over rows of an image. When that is done, then the data will be accessed in non-optimal order. As the first index is incremented, what is actually happening is that elements spaced far apart in memory are being sequentially accessed, with usually poor memory access speeds. For example, for a two-dimensional image `im` defined so that `im[0,`

`10]` represents the value at $x = 0, y = 10$. To be consistent with usual Python behavior then `im[0]` would represent a column at $x = 0$. Yet that data would be spread over the whole array since the data are stored in row order. Despite the flexibility of NumPy's indexing, it can't really paper over the fact basic operations are rendered inefficient because of data order or that getting contiguous subarrays is still awkward (e.g., `im[:, 0]` for the first row, vs `im[0]`). Thus one can't use an idiom such as `for row in im; for col in im` does work, but doesn't yield contiguous column data.

As it turns out, NumPy is smart enough when dealing with *ufuncs* to determine which index is the most rapidly varying one in memory and uses that for the innermost loop. Thus for *ufuncs*, there is no large intrinsic advantage to either approach in most cases. On the other hand, use of `ndarray.flat` with a FORTRAN ordered array will lead to non-optimal memory access as adjacent elements in the flattened array (iterator, actually) are not contiguous in memory.

Indeed, the fact is that Python indexing on lists and other sequences naturally leads to an outside-to-inside ordering (the first index gets the largest grouping, the next largest, and the last gets the smallest element). Since image data are normally stored in rows, this corresponds to the position within rows being the last item indexed.

If you do want to use Fortran ordering realize that there are two approaches to consider: 1) accept that the first index is just not the most rapidly changing in memory and have all your I/O routines reorder your data when going from memory to disk or visa versa, or use NumPy's mechanism for mapping the first index to the most rapidly varying data. We recommend the former if possible. The disadvantage of the latter is that many of NumPy's functions will yield arrays without Fortran ordering unless you are careful to use the `order` keyword. Doing this would be highly inconvenient.

Otherwise, we recommend simply learning to reverse the usual order of indices when accessing elements of an array. Granted, it goes against the grain, but it is more in line with Python semantics and the natural order of the data.

13.2 NumPy C code explanations

Fanaticism consists of redoubling your efforts when you have forgotten your aim. — *George Santayana*

An authority is a person who can tell you more about something than you really care to know. — *Unknown*

This page attempts to explain the logic behind some of the new pieces of code. The purpose behind these explanations is to enable somebody to be able to understand the ideas behind the implementation somewhat more easily than just staring at the code. Perhaps in this way, the algorithms can be improved on, borrowed from, and/or optimized by more people.

13.2.1 Memory model

One fundamental aspect of the `ndarray` is that an array is seen as a “chunk” of memory starting at some location. The interpretation of this memory depends on the *stride* information. For each dimension in an N -dimensional array, an integer (*stride*) dictates how many bytes must be skipped to get to the next element in that dimension. Unless you have a single-segment array, this *stride* information must be consulted when traversing through an array. It is not difficult to write code that accepts strides, you just have to use `char*` pointers because strides are in units of bytes. Keep in mind also that strides do not have to be unit-multiples of the element size. Also, remember that if the number of dimensions of the array is 0 (sometimes called a *rank-0* array), then the *strides* and *dimensions* variables are `NULL`.

Besides the structural information contained in the *strides* and *dimensions* members of the `PyArrayObject`, the flags contain important information about how the data may be accessed. In particular, the `NPY_ARRAY_ALIGNED` flag is set when the memory is on a suitable boundary according to the datatype array. Even if you have a *contiguous* chunk of memory, you cannot just assume it is safe to dereference a datatype-specific pointer to an element. Only if the `NPY_ARRAY_ALIGNED` flag is set, this is a safe operation. On some platforms it will work but on others, like Solaris, it will cause a bus error. The `NPY_ARRAY_WRITEABLE` should also be ensured if you plan on writing to the memory area of the array. It is also possible to obtain a pointer to an unwritable memory area. Sometimes, writing to the memory area when the `NPY_ARRAY_WRITEABLE` flag is not set will just be rude. Other times it can cause program crashes (e.g. a data-area that is a read-only memory-mapped file).

13.2.2 Data-type encapsulation

See also:

`arrays.dtypes`

The datatype is an important abstraction of the `ndarray`. Operations will look to the datatype to provide the key functionality that is needed to operate on the array. This functionality is provided in the list of function pointers pointed to by the `f` member of the `PyArray_Descr` structure. In this way, the number of datatypes can be extended simply by providing a `PyArray_Descr` structure with suitable function pointers in the `f` member. For built-in types, there are some optimizations that bypass this mechanism, but the point of the datatype abstraction is to allow new datatypes to be added.

One of the built-in datatypes, the `void` datatype allows for arbitrary *structured types* containing 1 or more fields as elements of the array. A *field* is simply another datatype object along with an offset into the current structured type. In order to support arbitrarily nested fields, several recursive implementations of datatype access are implemented for the void type. A common idiom is to cycle through the elements of the dictionary and perform a specific operation based on the datatype object stored at the given offset. These offsets can be arbitrary numbers. Therefore, the possibility of encountering misaligned data must be recognized and taken into account if necessary.

13.2.3 N-D Iterators

See also:

`arrays.nditer`

A very common operation in much of NumPy code is the need to iterate over all the elements of a general, strided, N-dimensional array. This operation of a general-purpose N-dimensional loop is abstracted in the notion of an iterator object. To write an N-dimensional loop, you only have to create an iterator object from an `ndarray`, work with the `dataptr` member of the iterator object structure and call the macro `PyArray_ITER_NEXT` on the iterator object to move to the next element. The `next` element is always in C-contiguous order. The macro works by first special-casing the C-contiguous, 1-D, and 2-D cases which work very simply.

For the general case, the iteration works by keeping track of a list of coordinate counters in the iterator object. At each iteration, the last coordinate counter is increased (starting from 0). If this counter is smaller than one less than the size of the array in that dimension (a pre-computed and stored value), then the counter is increased and the `dataptr` member is increased by the strides in that dimension and the macro ends. If the end of a dimension is reached, the counter for the last dimension is reset to zero and the `dataptr` is moved back to the beginning of that dimension by subtracting the strides value times one less than the number of elements in that dimension (this is also pre-computed and stored in the `backstrides` member of the iterator object). In this case, the macro does not end, but a local dimension counter is decremented so that the next-to-last dimension replaces the role that the last dimension played and the previously-described tests are executed again on the next-to-last dimension. In this way, the `dataptr` is adjusted appropriately for arbitrary striding.

The `coordinates` member of the `PyArrayIterObject` structure maintains the current N-d counter unless the underlying array is C-contiguous in which case the coordinate counting is bypassed. The `index` member of the `PyArrayIterObject` keeps track of the current flat index of the iterator. It is updated by the `PyArray_ITER_NEXT` macro.

13.2.4 Broadcasting

See also:

Broadcasting

In Numeric, the ancestor of NumPy, broadcasting was implemented in several lines of code buried deep in `ufuncobject.c`. In NumPy, the notion of broadcasting has been abstracted so that it can be performed in multiple places. Broadcasting is handled by the function `PyArray_Broadcast`. This function requires a `PyArrayMultiIterObject` (or something that is a binary equivalent) to be passed in. The `PyArrayMultiIterObject` keeps track of the broadcast number of dimensions and size in each dimension along with the total size of the broadcast result. It also keeps track of the number of arrays being broadcast and a pointer to an iterator for each of the arrays being broadcast.

The `PyArray_Broadcast` function takes the iterators that have already been defined and uses them to determine the broadcast shape in each dimension (to create the iterators at the same time that broadcasting occurs then use the `PyArray_MultiIterNew` function). Then, the iterators are adjusted so that each iterator thinks it is iterating over an array with the broadcast size. This is done by adjusting the iterators number of dimensions, and the *shape* in each dimension. This works because the iterator strides are also adjusted. Broadcasting only adjusts (or adds) length-1 dimensions. For these dimensions, the strides variable is simply set to 0 so that the data-pointer for the iterator over that array doesn't move as the broadcasting operation operates over the extended dimension.

Broadcasting was always implemented in Numeric using 0-valued strides for the extended dimensions. It is done in exactly the same way in NumPy. The big difference is that now the array of strides is kept track of in a `PyArrayIterObject`, the iterators involved in a broadcast result are kept track of in a `PyArrayMultiIterObject`, and the `PyArray_Broadcast` call implements the *General Broadcasting Rules*.

13.2.5 Array Scalars

See also:

`arrays.scalars`

The array scalars offer a hierarchy of Python types that allow a one-to-one correspondence between the datatype stored in an array and the Python-type that is returned when an element is extracted from the array. An exception to this rule was made with object arrays. Object arrays are heterogeneous collections of arbitrary Python objects. When you select an item from an object array, you get back the original Python object (and not an object array scalar which does exist but is rarely used for practical purposes).

The array scalars also offer the same methods and attributes as arrays with the intent that the same code can be used to support arbitrary dimensions (including 0-dimensions). The array scalars are read-only (immutable) with the exception of the void scalar which can also be written to so that structured array field setting works more naturally (`a[0]['f1'] = value`).

13.2.6 Indexing

See also:

Indexing on ndarrays, `arrays.indexing`

All Python indexing operations `arr[index]` are organized by first preparing the index and finding the index type. The supported index types are:

- `integer`
- `newaxis`
- `slice`

- `Ellipsis`
- integer arrays/array-likes (advanced)
- boolean (single boolean array); if there is more than one boolean array as the index or the shape does not match exactly, the boolean array will be converted to an integer array instead.
- 0-d boolean (and also integer); 0-d boolean arrays are a special case that has to be handled in the advanced indexing code. They signal that a 0-d boolean array had to be interpreted as an integer array.

As well as the scalar array special case signaling that an integer array was interpreted as an integer index, which is important because an integer array index forces a copy but is ignored if a scalar is returned (full integer index). The prepared index is guaranteed to be valid with the exception of out of bound values and broadcasting errors for advanced indexing. This includes that an `Ellipsis` is added for incomplete indices for example when a two-dimensional array is indexed with a single integer.

The next step depends on the type of index which was found. If all dimensions are indexed with an integer a scalar is returned or set. A single boolean indexing array will call specialized boolean functions. Indices containing an `Ellipsis` or `slice` but no advanced indexing will always create a view into the old array by calculating the new strides and memory offset. This view can then either be returned or, for assignments, filled using `PyArray_CopyObject`. Note that `PyArray_CopyObject` may also be called on temporary arrays in other branches to support complicated assignments when the array is of object dtype.

Advanced indexing

By far the most complex case is advanced indexing, which may or may not be combined with typical view-based indexing. Here integer indices are interpreted as view-based. Before trying to understand this, you may want to make yourself familiar with its subtleties. The advanced indexing code has three different branches and one special case:

- There is one indexing array and it, as well as the assignment array, can be iterated trivially. For example, they may be contiguous. Also, the indexing array must be of `intp` type and the value array in assignments should be of the correct type. This is purely a fast path.
- There are only integer array indices so that no subarray exists.
- View-based and advanced indexing is mixed. In this case, the view-based indexing defines a collection of subarrays that are combined by the advanced indexing. For example, `arr[[1, 2, 3], :]` is created by vertically stacking the subarrays `arr[1, :]`, `arr[2, :]`, and `arr[3, :]`.
- There is a subarray but it has exactly one element. This case can be handled as if there is no subarray but needs some care during setup.

Deciding what case applies, checking broadcasting, and determining the kind of transposition needed are all done in `PyArray_MapIterNew`. After setting up, there are two cases. If there is no subarray or it only has one element, no subarray iteration is necessary and an iterator is prepared which iterates all indexing arrays *as well as* the result or value array. If there is a subarray, there are three iterators prepared. One for the indexing arrays, one for the result or value array (minus its subarray), and one for the subarrays of the original and the result/assignment array. The first two iterators give (or allow calculation) of the pointers into the start of the subarray, which then allows restarting the subarray iteration.

When advanced indices are next to each other transposing may be necessary. All necessary transposing is handled by `PyArray_MapIterSwapAxes` and has to be handled by the caller unless `PyArray_MapIterNew` is asked to allocate the result.

After preparation, getting and setting are relatively straightforward, although the different modes of iteration need to be considered. Unless there is only a single indexing array during item getting, the validity of the indices is checked beforehand. Otherwise, it is handled in the inner loop itself for optimization.

13.2.7 Universal functions

See also:

ufuncs, *Universal functions (ufunc) basics*

Universal functions are callable objects that take N inputs and produce M outputs by wrapping basic 1-D loops that work element-by-element into full easy-to-use functions that seamlessly implement *broadcasting*, *type-checking*, *buffered coercion*, and *output-argument handling*. New universal functions are normally created in C, although there is a mechanism for creating ufuncs from Python functions (`frompyfunc`). The user must supply a 1-D loop that implements the basic function taking the input scalar values and placing the resulting scalars into the appropriate output slots as explained in implementation.

Setup

Every `ufunc` calculation involves some overhead related to setting up the calculation. The practical significance of this overhead is that even though the actual calculation of the `ufunc` is very fast, you will be able to write array and type-specific code that will work faster for small arrays than the `ufunc`. In particular, using ufuncs to perform many calculations on 0-D arrays will be slower than other Python-based solutions (the silently-imported `scalarmath` module exists precisely to give array scalars the look-and-feel of `ufunc` based calculations with significantly reduced overhead).

When a `ufunc` is called, many things must be done. The information collected from these setup operations is stored in a loop object. This loop object is a C-structure (that could become a Python object but is not initialized as such because it is only used internally). This loop object has the layout needed to be used with `PyArray_Broadcast` so that the broadcasting can be handled in the same way as it is handled in other sections of code.

The first thing done is to look up in the thread-specific global dictionary the current values for the buffer-size, the error mask, and the associated error object. The state of the error mask controls what happens when an error condition is found. It should be noted that checking of the hardware error flags is only performed after each 1-D loop is executed. This means that if the input and output arrays are contiguous and of the correct type so that a single 1-D loop is performed, then the flags may not be checked until all elements of the array have been calculated. Looking up these values in a thread-specific dictionary takes time which is easily ignored for all but very small arrays.

After checking, the thread-specific global variables, the inputs are evaluated to determine how the `ufunc` should proceed and the input and output arrays are constructed if necessary. Any inputs which are not arrays are converted to arrays (using context if necessary). Which of the inputs are scalars (and therefore converted to 0-D arrays) is noted.

Next, an appropriate 1-D loop is selected from the 1-D loops available to the `ufunc` based on the input array types. This 1-D loop is selected by trying to match the signature of the datatypes of the inputs against the available signatures. The signatures corresponding to built-in types are stored in the `ufunc.types` member of the `ufunc` structure. The signatures corresponding to user-defined types are stored in a linked list of function information with the head element stored as a `CObject` in the `userloops` dictionary keyed by the datatype number (the first user-defined type in the argument list is used as the key). The signatures are searched until a signature is found to which the input arrays can all be cast safely (ignoring any scalar arguments which are not allowed to determine the type of the result). The implication of this search procedure is that “lesser types” should be placed below “larger types” when the signatures are stored. If no 1-D loop is found, then an error is reported. Otherwise, the `argument_list` is updated with the stored signature — in case casting is necessary and to fix the output types assumed by the 1-D loop.

If the `ufunc` has 2 inputs and 1 output and the second input is an `Object` array then a special-case check is performed so that `NotImplemented` is returned if the second input is not an `ndarray`, has the `__array_priority__` attribute, and has an `__r{op}__` special method. In this way, Python is signaled to give the other object a chance to complete the operation instead of using generic object-array calculations. This allows (for example) sparse matrices to override the multiplication operator 1-D loop.

For input arrays that are smaller than the specified buffer size, copies are made of all non-contiguous, misaligned, or out-of-byteorder arrays to ensure that for small arrays, a single loop is used. Then, array iterators are created for all the input arrays and the resulting collection of iterators is broadcast to a single shape.

The output arguments (if any) are then processed and any missing return arrays are constructed. If any provided output array doesn't have the correct type (or is misaligned) and is smaller than the buffer size, then a new output array is constructed with the special `NPY_ARRAY_WRITEBACKIFCOPY` flag set. At the end of the function, `PyArray_ResolveWritebackIfCopy` is called so that its contents will be copied back into the output array. Iterators for the output arguments are then processed.

Finally, the decision is made about how to execute the looping mechanism to ensure that all elements of the input arrays are combined to produce the output arrays of the correct type. The options for loop execution are one-loop (for 'contiguous', aligned, and correct data type), strided-loop (for non-contiguous but still aligned and correct data type), and a buffered loop (for misaligned or incorrect data type situations). Depending on which execution method is called for, the loop is then set up and computed.

Function call

This section describes how the basic universal function computation loop is set up and executed for each of the three different kinds of execution. If `NPY_ALLOW_THREADS` is defined during compilation, then as long as no object arrays are involved, the Python Global Interpreter Lock (GIL) is released prior to calling the loops. It is re-acquired if necessary to handle error conditions. The hardware error flags are checked only after the 1-D loop is completed.

One loop

This is the simplest case of all. The ufunc is executed by calling the underlying 1-D loop exactly once. This is possible only when we have aligned data of the correct type (including byteorder) for both input and output and all arrays have uniform strides (either *contiguous*, 0-D, or 1-D). In this case, the 1-D computational loop is called once to compute the calculation for the entire array. Note that the hardware error flags are only checked after the entire calculation is complete.

Strided loop

When the input and output arrays are aligned and of the correct type, but the striding is not uniform (non-contiguous and 2-D or larger), then a second looping structure is employed for the calculation. This approach converts all of the iterators for the input and output arguments to iterate over all but the largest dimension. The inner loop is then handled by the underlying 1-D computational loop. The outer loop is a standard iterator loop on the converted iterators. The hardware error flags are checked after each 1-D loop is completed.

Buffered loop

This is the code that handles the situation whenever the input and/or output arrays are either misaligned or of the wrong datatype (including being byteswapped) from what the underlying 1-D loop expects. The arrays are also assumed to be non-contiguous. The code works very much like the strided-loop except for the inner 1-D loop is modified so that pre-processing is performed on the inputs and post-processing is performed on the outputs in `bufsize` chunks (where `bufsize` is a user-settable parameter). The underlying 1-D computational loop is called on data that is copied over (if it needs to be). The setup code and the loop code is considerably more complicated in this case because it has to handle:

- memory allocation of the temporary buffers
- deciding whether or not to use buffers on the input and output data (misaligned and/or wrong datatype)
- copying and possibly casting data for any inputs or outputs for which buffers are necessary.
- special-casing `Object` arrays so that reference counts are properly handled when copies and/or casts are necessary.
- breaking up the inner 1-D loop into `bufsize` chunks (with a possible remainder).

Again, the hardware error flags are checked at the end of each 1-D loop.

Final output manipulation

Ufuncs allow other array-like classes to be passed seamlessly through the interface in that inputs of a particular class will induce the outputs to be of that same class. The mechanism by which this works is the following. If any of the inputs are not ndarrays and define the `__array_wrap__` method, then the class with the largest `__array_priority__` attribute determines the type of all the outputs (with the exception of any output arrays passed in). The `__array_wrap__` method of the input array will be called with the ndarray being returned from the ufunc as its input. There are two calling styles of the `__array_wrap__` function supported. The first takes the ndarray as the first argument and a tuple of “context” as the second argument. The context is (ufunc, arguments, output argument number). This is the first call tried. If a `TypeError` occurs, then the function is called with just the ndarray as the first argument.

Methods

There are three methods of ufuncs that require calculation similar to the general-purpose ufuncs. These are `ufunc.reduce`, `ufunc.accumulate`, and `ufunc.reduceat`. Each of these methods requires a setup command followed by a loop. There are four loop styles possible for the methods corresponding to no-elements, one-element, strided-loop, and buffered-loop. These are the same basic loop styles as implemented for the general-purpose function call except for the no-element and one-element cases which are special-cases occurring when the input array objects have 0 and 1 elements respectively.

Setup

The setup function for all three methods is `construct_reduce`. This function creates a reducing loop object and fills it with the parameters needed to complete the loop. All of the methods only work on ufuncs that take 2-inputs and return 1 output. Therefore, the underlying 1-D loop is selected assuming a signature of `[otype, otype, otype]` where `otype` is the requested reduction datatype. The buffer size and error handling are then retrieved from (per-thread) global storage. For small arrays that are misaligned or have incorrect datatype, a copy is made so that the un-buffered section of code is used. Then, the looping strategy is selected. If there is 1 element or 0 elements in the array, then a simple looping method is selected. If the array is not misaligned and has the correct datatype, then strided looping is selected. Otherwise, buffered looping must be performed. Looping parameters are then established, and the return array is constructed. The output array is of a different *shape* depending on whether the method is `reduce`, `accumulate`, or `reduceat`. If an output array is already provided, then its shape is checked. If the output array is not C-contiguous, aligned, and of the correct data type, then a temporary copy is made with the `NPY_ARRAY_WRITEBACKIFCOPY` flag set. In this way, the methods will be able to work with a well-behaved output array but the result will be copied back into the true output array when `PyArray_ResolveWritebackIfCopy` is called at function completion. Finally, iterators are set up to loop over the correct *axis* (depending on the value of `axis` provided to the method) and the setup routine returns to the actual computation routine.

Reduce

All of the ufunc methods use the same underlying 1-D computational loops with input and output arguments adjusted so that the appropriate reduction takes place. For example, the key to the functioning of `reduce` is that the 1-D loop is called with the output and the second input pointing to the same position in memory and both having a step-size of 0. The first input is pointing to the input array with a step-size given by the appropriate stride for the selected axis. In this way, the operation performed is

$$\begin{aligned} o &= i[0] \\ o &= i[k] \langle op \rangle o \quad k = 1 \dots N \end{aligned}$$

where $N + 1$ is the number of elements in the input, i , o is the output, and $i[k]$ is the k^{th} element of i along the selected axis. This basic operation is repeated for arrays with greater than 1 dimension so that the reduction takes place for every 1-D sub-array along the selected axis. An iterator with the selected dimension removed handles this looping.

For buffered loops, care must be taken to copy and cast data before the loop function is called because the underlying loop expects aligned data of the correct datatype (including byteorder). The buffered loop must handle this copying and casting prior to calling the loop function on chunks no greater than the user-specified `bufsize`.

Accumulate

The `accumulate` method is very similar to the `reduce` method in that the output and the second input both point to the output. The difference is that the second input points to memory one stride behind the current output pointer. Thus, the operation performed is

$$\begin{aligned} o[0] &= i[0] \\ o[k] &= i[k] + o[k-1] \quad k = 1 \dots N. \end{aligned}$$

The output has the same shape as the input and each 1-D loop operates over N elements when the shape in the selected axis is $N + 1$. Again, buffered loops take care to copy and cast the data before calling the underlying 1-D computational loop.

Reduceat

The `reduceat` function is a generalization of both the `reduce` and `accumulate` functions. It implements a `reduce` over ranges of the input array specified by indices. The extra indices argument is checked to be sure that every input is not too large for the input array along the selected dimension before the loop calculations take place. The loop implementation is handled using code that is very similar to the `reduce` code repeated as many times as there are elements in the indices input. In particular: the first input pointer passed to the underlying 1-D computational loop points to the input array at the correct location indicated by the index array. In addition, the output pointer and the second input pointer passed to the underlying 1-D loop point to the same position in memory. The size of the 1-D computational loop is fixed to be the difference between the current index and the next index (when the current index is the last index, then the next index is assumed to be the length of the array along the selected dimension). In this way, the 1-D loop will implement a `reduce` over the specified indices.

Misaligned or a loop datatype that does not match the input and/or output datatype is handled using buffered code wherein data is copied to a temporary buffer and cast to the correct datatype if necessary prior to calling the underlying 1-D function. The temporary buffers are created in (element) sizes no bigger than the user settable buffer-size value. Thus, the loop must be flexible enough to call the underlying 1-D computational loop enough times to complete the total calculation in chunks no bigger than the buffer-size.

13.3 Memory Alignment

13.3.1 NumPy alignment goals

There are three use-cases related to memory alignment in NumPy (as of 1.14):

1. Creating *structured datatypes* with *fields* aligned like in a C-struct.
2. Speeding up copy operations by using `uint` assignment in instead of `memcpy`.
3. Guaranteeing safe aligned access for `ufuncs/setitem/casting` code.

NumPy uses two different forms of alignment to achieve these goals: “True alignment” and “Uint alignment”.

“True” alignment refers to the architecture-dependent alignment of an equivalent C-type in C. For example, in x64 systems `float64` is equivalent to `double` in C. On most systems, this has either an alignment of 4 or 8 bytes (and this can be controlled in GCC by the option `align-double`). A variable is aligned in memory if its memory offset is a multiple of its alignment. On some systems (eg. `sparc`) memory alignment is required; on others, it gives a speedup.

“Uint” alignment depends on the size of a datatype. It is defined to be the “True alignment” of the `uint` used by NumPy’s copy-code to copy the datatype, or undefined/unaligned if there is no equivalent `uint`. Currently, NumPy uses `uint8`, `uint16`, `uint32`, `uint64`, and `uint64` to copy data of size 1, 2, 4, 8, 16 bytes respectively, and all other sized datatypes cannot be `uint`-aligned.

For example, on a (typical Linux x64 GCC) system, the NumPy `complex64` datatype is implemented as `struct { float real, imag; }`. This has “true” alignment of 4 and “uint” alignment of 8 (equal to the true alignment of `uint64`).

Some cases where uint and true alignment are different (default GCC Linux):

arch	type	true-aln	uint-aln
x86_64	complex64	4	8
x86_64	float128	16	8
x86	float96	4	-

13.3.2 Variables in NumPy which control and describe alignment

There are 4 relevant uses of the word `align` used in NumPy:

- The `dtype.alignment` attribute (`descr->alignment` in C). This is meant to reflect the “true alignment” of the type. It has arch-dependent default values for all datatypes, except for the structured types created with `align=True` as described below.
- The `ALIGNED` flag of an ndarray, computed in `IsAligned` and checked by `PyArray_ISALIGNED`. This is computed from `dtype.alignment`. It is set to `True` if every item in the array is at a memory location consistent with `dtype.alignment`, which is the case if the `data_ptr` and all strides of the array are multiples of that alignment.
- The `align` keyword of the dtype constructor, which only affects *Structured arrays*. If the structure’s field offsets are not manually provided, NumPy determines offsets automatically. In that case, `align=True` pads the structure so that each field is “true” aligned in memory and sets `dtype.alignment` to be the largest of the field “true” alignments. This is like what C-structs usually do. Otherwise if offsets or itemsize were manually provided `align=True` simply checks that all the fields are “true” aligned and that the total itemsize is a multiple of the largest field alignment. In either case `dtype.isalignedstruct` is also set to `True`.
- `IsUintAligned` is used to determine if an ndarray is “uint aligned” in an analogous way to how `IsAligned` checks for true alignment.

13.3.3 Consequences of alignment

Here is how the variables above are used:

1. Creating aligned structs: To know how to offset a field when `align=True`, NumPy looks up `field.dtype.alignment`. This includes fields that are nested structured arrays.
2. Ufuncs: If the `ALIGNED` flag of an array is `False`, ufuncs will buffer/cast the array before evaluation. This is needed since ufunc inner loops access raw elements directly, which might fail on some archs if the elements are not true-aligned.
3. `Getitem/setitem/copyswap` function: Similar to ufuncs, these functions generally have two code paths. If `ALIGNED` is `False` they will use a code path that buffers the arguments so they are true-aligned.
4. Strided copy code: Here, “uint alignment” is used instead. If the itemsize of an array is equal to 1, 2, 4, 8 or 16 bytes and the array is uint aligned then instead NumPy will do `*(uintN*)dst) = *(uintN*)src)` for appropriate N. Otherwise, NumPy copies by doing `memcpy(dst, src, N)`.
5. `Nditer` code: Since this often calls the strided copy code, it must check for “uint alignment”.
6. Cast code: This checks for “true” alignment, as it does `*dst = CASTFUNC(*src)` if aligned. Otherwise, it does `memcpy(dst, src, N)` where `dstval/srcval` are aligned.

Note that the strided-copy and strided-cast code are deeply intertwined and so any arrays being processed by them must be both uint and true aligned, even though the copy-code only needs uint alignment and the cast code only true alignment. If there is ever a big rewrite of this code it would be good to allow them to use different alignments.

REPORTING BUGS

File bug reports or feature requests, and make contributions (e.g. code patches), by opening a “new issue” on GitHub:

- NumPy Issues: <https://github.com/numpy/numpy/issues>

Please give as much information as you can in the ticket. It is extremely useful if you can supply a small self-contained code snippet that reproduces the problem. Also specify the component, the version you are referring to and the milestone.

Report bugs to the appropriate GitHub project (there is one for NumPy and a different one for SciPy).

More information can be found on the <https://www.scipy.org/scipylib/dev-zone.html> website.

RELEASE NOTES

15.1 NumPy 1.23.0 Release Notes

The NumPy 1.23.0 release continues the ongoing work to improve the handling and promotion of dtypes, increase the execution speed, clarify the documentation, and expire old deprecations. The highlights are:

- Implementation of `loadtxt` in C, greatly improving its performance.
- Exposing DLPack at the Python level for easy data exchange.
- Changes to the promotion and comparisons of structured dtypes.
- Improvements to `f2py`.

See below for the details,

15.1.1 New functions

- A masked array specialization of `ndenumerate` is now available as `numpy.ma.ndenumerate`. It provides an alternative to `numpy.ndenumerate` and skips masked values by default.
([gh-20020](#))
- `numpy.from_dlpack` has been added to allow easy exchange of data using the DLPack protocol. It accepts Python objects that implement the `__dlpack__` and `__dlpack_device__` methods and returns a `ndarray` object which is generally the view of the data of the input object.
([gh-21145](#))

15.1.2 Deprecations

- Setting `__array_finalize__` to `None` is deprecated. It must now be a method and may wish to call `super().__array_finalize__(obj)` after checking for `None` or if the NumPy version is sufficiently new.
([gh-20766](#))
- Using `axis=32` (`axis=np.MAXDIMS`) in many cases had the same meaning as `axis=None`. This is deprecated and `axis=None` must be used instead.
([gh-20920](#))
- The hook function `PyDataMem_SetEventHook` has been deprecated and the demonstration of its use in `tool/allocation_tracking` has been removed. The ability to track allocations is now built-in to python via `tracemalloc`.

(gh-20394)

- `numpy.distutils` has been deprecated, as a result of `distutils` itself being deprecated. It will not be present in NumPy for Python ≥ 3.12 , and will be removed completely 2 years after the release of Python 3.12. For more details, see [distutils-status-migration](#).

(gh-20875)

- `numpy.loadtxt` will now give a `DeprecationWarning` when an integer `dtype` is requested but the value is formatted as a floating point number.

(gh-21663)

15.1.3 Expired deprecations

- The `NpzFile.iteritems()` and `NpzFile.iterkeys()` methods have been removed as part of the continued removal of Python 2 compatibility. This concludes the deprecation from 1.15.

(gh-16830)

- The `alen` and `asscalar` functions have been removed.

(gh-20414)

- The `UPDATEIFCOPY` array flag has been removed together with the enum `NPY_ARRAY_UPDATEIFCOPY`. The associated (and deprecated) `PyArray_XDECREf_ERR` was also removed. These were all deprecated in 1.14. They are replaced by `WRITEBACKIFCOPY`, that requires calling `PyArray_ResoveWritebackIfCopy` before the array is deallocated.

(gh-20589)

- Exceptions will be raised during array-like creation. When an object raised an exception during access of the special attributes `__array__` or `__array_interface__`, this exception was usually ignored. This behaviour was deprecated in 1.21, and the exception will now be raised.

(gh-20835)

- Multidimensional indexing with non-tuple values is not allowed. Previously, code such as `arr[ind]` where `ind = [[0, 1], [0, 1]]` produced a `FutureWarning` and was interpreted as a multidimensional index (i.e., `arr[tuple(ind)]`). Now this example is treated like an array index over a single dimension (`arr[array(ind)]`). Multidimensional indexing with anything but a tuple was deprecated in NumPy 1.15.

(gh-21029)

- Changing to a `dtype` of different size in F-contiguous arrays is no longer permitted. Deprecated since Numpy 1.11.0. See below for an extended explanation of the effects of this change.

(gh-20722)

15.1.4 New Features

crackfortran has support for operator and assignment overloading

`crackfortran` parser now understands operator and assignment definitions in a module. They are added in the `body` list of the module which contains a new key `implementedby` listing the names of the subroutines or functions implementing the operator or assignment.

(gh-15006)

f2py supports reading access type attributes from derived type statements

As a result, one does not need to use `public` or `private` statements to specify derived type access properties.

(gh-15844)

New parameter `ndmin` added to `genfromtxt`

This parameter behaves the same as `ndmin` from `numpy.loadtxt`.

(gh-20500)

`np.loadtxt` now supports quote character and single converter function

`numpy.loadtxt` now supports an additional `quotechar` keyword argument which is not set by default. Using `quotechar='\"'` will read quoted fields as used by the Excel CSV dialect.

Further, it is now possible to pass a single callable rather than a dictionary for the `converters` argument.

(gh-20580)

Changing to `dtype` of a different size now requires contiguity of only the last axis

Previously, viewing an array with a `dtype` of a different item size required that the entire array be C-contiguous. This limitation would unnecessarily force the user to make contiguous copies of non-contiguous arrays before being able to change the `dtype`.

This change affects not only `ndarray.view`, but other construction mechanisms, including the discouraged direct assignment to `ndarray.dtype`.

This change expires the deprecation regarding the viewing of F-contiguous arrays, described elsewhere in the release notes.

(gh-20722)

Deterministic output files for F2PY

For F77 inputs, `f2py` will generate `modname-f2pywrappers.f` unconditionally, though these may be empty. For free-form inputs, `modname-f2pywrappers.f`, `modname-f2pywrappers2.f90` will both be generated unconditionally, and may be empty. This allows writing generic output rules in `cmake` or `meson` and other build systems. Older behavior can be restored by passing `--skip-empty-wrappers` to `f2py`. *Using via meson* details usage.

(gh-21187)

`keepdims` parameter for `average`

The parameter `keepdims` was added to the functions `numpy.average` and `numpy.ma.average`. The parameter has the same meaning as it does in reduction functions such as `numpy.sum` or `numpy.mean`.

(gh-21485)

New parameter `equal_nan` added to `np.unique`

`np.unique` was changed in 1.21 to treat all NaN values as equal and return a single NaN. Setting `equal_nan=False` will restore pre-1.21 behavior to treat NaNs as unique. Defaults to `True`.

(gh-21623)

15.1.5 Compatibility notes

1D `np.linalg.norm` preserves float input types, even for scalar results

Previously, this would promote to `float64` when the `ord` argument was not one of the explicitly listed values, e.g. `ord=3`:

```
>>> f32 = np.float32([1, 2])
>>> np.linalg.norm(f32, 2).dtype
dtype('float32')
>>> np.linalg.norm(f32, 3)
dtype('float64') # numpy 1.22
dtype('float32') # numpy 1.23
```

This change affects only `float32` and `float16` vectors with `ord` other than `-Inf`, `0`, `1`, `2`, and `Inf`.

(gh-17709)

Changes to structured (void) dtype promotion and comparisons

In general, NumPy now defines correct, but slightly limited, promotion for structured dtypes by promoting the subtypes of each field instead of raising an exception:

```
>>> np.result_type(np.dtype("i,i"), np.dtype("i,d"))
dtype([('f0', '<i4'), ('f1', '<f8')])
```

For promotion matching field names, order, and titles are enforced, however padding is ignored. Promotion involving structured dtypes now always ensures native byte-order for all fields (which may change the result of `np.concatenate`) and ensures that the result will be “packed”, i.e. all fields are ordered contiguously and padding is removed. See [Structure Comparison and Promotion](#) for further details.

The `repr` of aligned structures will now never print the long form including `offsets` and `itemsize` unless the structure includes padding not guaranteed by `align=True`.

In alignment with the above changes to the promotion logic, the casting safety has been updated:

- “`equiv`” enforces matching names and titles. The `itemsize` is allowed to differ due to padding.
- “`safe`” allows mismatching field names and titles
- The cast safety is limited by the cast safety of each included field.
- The order of fields is used to decide cast safety of each individual field. Previously, the field names were used and only unsafe casts were possible when names mismatched.

The main important change here is that name mismatches are now considered “safe” casts.

(gh-19226)

NPY_RELAXED_STRIDES_CHECKING has been removed

NumPy cannot be compiled with `NPY_RELAXED_STRIDES_CHECKING=0` anymore. Relaxed strides have been the default for many years and the option was initially introduced to allow a smoother transition.

(gh-20220)

`np.loadtxt` has recieved several changes

The row counting of `numpy.loadtxt` was fixed. `loadtxt` ignores fully empty lines in the file, but counted them towards `max_rows`. When `max_rows` is used and the file contains empty lines, these will now not be counted. Previously, it was possible that the result contained fewer than `max_rows` rows even though more data was available to be read. If the old behaviour is required, `itertools.islice` may be used:

```
import itertools
lines = itertools.islice(open("file"), 0, max_rows)
result = np.loadtxt(lines, ...)
```

While generally much faster and improved, `numpy.loadtxt` may now fail to converter certain strings to numbers that were previously successfully read. The most important cases for this are:

- Parsing floating point values such as `1.0` into integers is now deprecated.
- Parsing hexadecimal floats such as `0x3p3` will fail
- An `_` was previously accepted as a thousands delimiter `100_000`. This will now result in an error.

If you experience these limitations, they can all be worked around by passing appropriate `converters=`. NumPy now supports passing a single converter to be used for all columns to make this more convenient. For example, `converters=float.fromhex` can read hexadecimal float numbers and `converters=int` will be able to read `100_000`.

Further, the error messages have been generally improved. However, this means that error types may differ. In particularly, a `ValueError` is now always raised when parsing of a single entry fails.

(gh-20580)

15.1.6 Improvements

`ndarray.__array_finalize__` is now callable

This means subclasses can now use `super().__array_finalize__(obj)` without worrying whether `ndarray` is their superclass or not. The actual call remains a no-op.

(gh-20766)

Add support for VSX4/Power10

With VSX4/Power10 enablement, the new instructions available in Power ISA 3.1 can be used to accelerate some NumPy operations, e.g., `floor_divide`, `modulo`, etc.

(gh-20821)

`np.fromiter` now accepts objects and subarrays

The `numpy.fromiter` function now supports object and subarray dtypes. Please see the function documentation for examples.

(gh-20993)

Math C library feature detection now uses correct signatures

Compiling is preceded by a detection phase to determine whether the underlying libc supports certain math operations. Previously this code did not respect the proper signatures. Fixing this enables compilation for the `wasm-lld` backend (compilation for web assembly) and reduces the number of warnings.

(gh-21154)

`np.kron` now maintains subclass information

`np.kron` maintains subclass information now such as masked arrays while computing the Kronecker product of the inputs

```
>>> x = ma.array([[1, 2], [3, 4]], mask=[[0, 1], [1, 0]])
>>> np.kron(x, x)
masked_array(
  data=[[1, --, --, --],
        [--, 4, --, --],
        [--, --, 4, --],
        [--, --, --, 16]],
  mask=[[False,  True,  True,  True],
        [ True, False,  True,  True],
        [ True,  True, False,  True],
        [ True,  True,  True, False]],
  fill_value=999999)
```

Warning: `np.kron` output now follows `ufunc` ordering (`multiply`) to determine the output class type

```
>>> class myarr(np.ndarray):
>>>     __array_priority__ = -1
>>> a = np.ones([2, 2])
>>> ma = myarray(a.shape, a.dtype, a.data)
>>> type(np.kron(a, ma)) == np.ndarray
False # Before it was True
>>> type(np.kron(a, ma)) == myarr
True
```

(gh-21262)

15.1.7 Performance improvements and changes

Faster `np.loadtxt`

`numpy.loadtxt` is now generally much faster than previously as most of it is now implemented in C.

(gh-20580)

Faster reduction operators

Reduction operations like `numpy.sum`, `numpy.prod`, `numpy.add.reduce`, `numpy.logical_and.reduce` on contiguous integer-based arrays are now much faster.

(gh-21001)

Faster `np.where`

`numpy.where` is now much faster than previously on unpredictable/random input data.

(gh-21130)

Faster operations on NumPy scalars

Many operations on NumPy scalars are now significantly faster, although rare operations (e.g. with 0-D arrays rather than scalars) may be slower in some cases. However, even with these improvements users who want the best performance for their scalars, may want to convert a known NumPy scalar into a Python one using `scalar.item()`.

(gh-21188)

Faster `np.kron`

`numpy.kron` is about 80% faster as the product is now computed using broadcasting.

(gh-21354)

15.2 NumPy 1.22.4 Release Notes

NumPy 1.22.4 is a maintenance release that fixes bugs discovered after the 1.22.3 release. In addition, the wheels for this release are built using the recently released Cython 0.29.30, which should fix the reported problems with [debugging](#).

The Python versions supported for this release are 3.8-3.10. Note that the Mac wheels are based on OS X 10.15 rather than 10.9 that was used in previous NumPy release cycles.

15.2.1 Contributors

A total of 12 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Alexander Shadchin
- Bas van Beek
- Charles Harris
- Hood Chatham
- Jarrod Millman
- John-Mark Gurney +
- Junyan Ou +
- Mariusz Felisiak +
- Ross Barnowski
- Sebastian Berg
- Serge Guelton
- Stefan van der Walt

15.2.2 Pull requests merged

A total of 22 pull requests were merged for this release.

- [#21191](#): TYP, BUG: Fix `np.lib.stride_tricks` re-exported under the...
- [#21192](#): TST: Bump mypy from 0.931 to 0.940
- [#21243](#): MAINT: Explicitly re-export the types in `numpy._typing`
- [#21245](#): MAINT: Specify sphinx, numpydoc versions for CI doc builds
- [#21275](#): BUG: Fix typos
- [#21277](#): ENH, BLD: Fix math feature detection for wasm
- [#21350](#): MAINT: Fix failing simd and cygwin tests.
- [#21438](#): MAINT: Fix failing Python 3.8 32-bit Windows test.
- [#21444](#): BUG: add linux guard per [#21386](#)
- [#21445](#): BUG: Allow legacy dtypes to cast to datetime again
- [#21446](#): BUG: Make mmap handling safer in frombuffer
- [#21447](#): BUG: Stop using `PyBytesObject.ob_shash` deprecated in Python 3.11.
- [#21448](#): ENH: Introduce `numpy.core.setup_common.NPY_CXX_FLAGS`
- [#21472](#): BUG: Ensure compile errors are raised correctly
- [#21473](#): BUG: Fix segmentation fault
- [#21474](#): MAINT: Update doc requirements
- [#21475](#): MAINT: Mark `npy_memchr` with `no_sanitize("alignment")` on clang
- [#21512](#): DOC: Proposal - make the doc landing page cards more similar...

- [#21525](#): MAINT: Update Cython version to 0.29.30.
- [#21536](#): BUG: Fix GCC error during build configuration
- [#21541](#): REL: Prepare for the NumPy 1.22.4 release.
- [#21547](#): MAINT: Skip tests that fail on PyPy.

15.3 NumPy 1.22.3 Release Notes

NumPy 1.22.3 is a maintenance release that fixes bugs discovered after the 1.22.2 release. The most noticeable fixes may be those for DLPack. One that may cause some problems is disallowing strings as inputs to logical ufuncs. It is still undecided how strings should be treated in those functions and it was thought best to simply disallow them until a decision was reached. That should not cause problems with older code.

The Python versions supported for this release are 3.8-3.10. Note that the Mac wheels are now based on OS X 10.14 rather than 10.9 that was used in previous NumPy release cycles. 10.14 is the oldest release supported by Apple.

15.3.1 Contributors

A total of 9 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- @GalaxySnail +
- Alexandre de Siqueira
- Bas van Beek
- Charles Harris
- Melissa Weber Mendonça
- Ross Barnowski
- Sebastian Berg
- Tirth Patel
- Matthieu Darbois

15.3.2 Pull requests merged

A total of 10 pull requests were merged for this release.

- [#21048](#): MAINT: Use “3.10” instead of “3.10-dev” on travis.
- [#21106](#): TYP,MAINT: Explicitly allow sequences of array-likes in `np.concatenate`
- [#21137](#): BLD,DOC: skip broken ipython 8.1.0
- [#21138](#): BUG, ENH: `np._from_dlpack`: export correct device information
- [#21139](#): BUG: Fix numba DUFuncs added loops getting picked up
- [#21140](#): BUG: Fix unpickling an empty ndarray with a non-zero dimension...
- [#21141](#): BUG: use `ThreadPoolExecutor` instead of `ThreadPool`
- [#21142](#): API: Disallow strings in logical ufuncs
- [#21143](#): MAINT, DOC: Fix SciPy intersphinx link

- [#21148](#): BUG,ENH: `np._from_dlpack`: export arrays with any strided size-1...

15.4 NumPy 1.22.2 Release Notes

The NumPy 1.22.2 is maintenance release that fixes bugs discovered after the 1.22.1 release. Notable fixes are:

- Several build related fixes for downstream projects and other platforms.
- Various Annotation fixes/additions.
- Numpy wheels for Windows will use the 1.41 tool chain, fixing downstream link problems for projects using NumPy provided libraries on Windows.
- Deal with CVE-2021-41495 complaint.

The Python versions supported for this release are 3.8-3.10.

15.4.1 Contributors

A total of 14 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Andrew J. Hesford +
- Bas van Beek
- Brénainn Woodsend +
- Charles Harris
- Hood Chatham
- Janus Heide +
- Leo Singer
- Matti Pícus
- Mukulika Pahari
- Niyas Sait
- Pearu Peterson
- Ralf Gommers
- Sebastian Berg
- Serge Guelton

15.4.2 Pull requests merged

A total of 21 pull requests were merged for this release.

- [#20842](#): BLD: Add `NPY_DISABLE_SVML` env var to opt out of SVML
- [#20843](#): BUG: Fix build of third party extensions with `Py_LIMITED_API`
- [#20844](#): TYP: Fix pyright being unable to infer the `real` and `imag`...
- [#20845](#): BUG: Fix comparator function signatures
- [#20906](#): BUG: Avoid importing `numpy.distutils` on `import numpy.testing`

- #20907: MAINT: remove outdated mingw32 fseek support
- #20908: TYP: Relax the return type of `np.vectorize`
- #20909: BUG: fix f2py's define for threading when building with Mingw
- #20910: BUG: distutils: fix building mixed C/Fortran extensions
- #20912: DOC,TST: Fix Pandas code example as per new release
- #20935: TYP, MAINT: Add annotations for `flatiter.__setitem__`
- #20936: MAINT, TYP: Added missing where typehints in `fromnumeric.pyi`
- #20937: BUG: Fix build_ext interaction with non numpy extensions
- #20938: BUG: Fix missing intrinsics for windows/arm64 target
- #20945: REL: Prepare for the NumPy 1.22.2 release.
- #20982: MAINT: f2py: don't generate code that triggers `-Wsometimes-uninitialized`.
- #20983: BUG: Fix incorrect return type in reduce without initial value
- #20984: ENH: review return values for `PyArray_DescrNew`
- #20985: MAINT: be more tolerant of `setuptools >= 60`
- #20986: BUG: Fix misplaced return.
- #20992: MAINT: Further small return value validation fixes

15.5 NumPy 1.22.1 Release Notes

The NumPy 1.22.1 is a maintenance release that fixes bugs discovered after the 1.22.0 release. Notable fixes are:

- Fix f2PY docstring problems (SciPy)
- Fix reduction type problems (AstroPy)
- Fix various typing bugs.

The Python versions supported for this release are 3.8-3.10.

15.5.1 Contributors

A total of 14 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Arryan Singh
- Bas van Beek
- Charles Harris
- Denis Laxalde
- Isuru Fernando
- Kevin Sheppard
- Matthew Barber
- Matti Pícus
- Melissa Weber Mendonça

- Mukulika Pahari
- Omid Rajaei +
- Pearu Peterson
- Ralf Gommers
- Sebastian Berg

15.5.2 Pull requests merged

A total of 20 pull requests were merged for this release.

- [#20702](#): MAINT, DOC: Post 1.22.0 release fixes.
- [#20703](#): DOC, BUG: Use pngs instead of svgs.
- [#20704](#): DOC: Fixed the link on user-guide landing page
- [#20714](#): BUG: Restore vc141 support
- [#20724](#): BUG: Fix array dimensions solver for multidimensional arguments...
- [#20725](#): TYP: change type annotation for `__array_namespace__` to `ModuleType`
- [#20726](#): TYP, MAINT: Allow `ndindex` to accept integer tuples
- [#20757](#): BUG: Relax dtype identity check in reductions
- [#20763](#): TYP: Allow time manipulation functions to accept `date` and `timedelta`...
- [#20768](#): TYP: Relax the type of `ndarray.__array_finalize__`
- [#20795](#): MAINT: Raise `RuntimeError` if `setuptools` version is too recent.
- [#20796](#): BUG, DOC: Fixes SciPy docs build warnings
- [#20797](#): DOC: fix OpenBLAS version in release note
- [#20798](#): PERF: Optimize array check for bounded 0,1 values
- [#20805](#): BUG: Fix that `reduce`-likes honor out always (and live in the...
- [#20806](#): BUG: `array_api.argsort(descending=True)` respects relative...
- [#20807](#): BUG: Allow integer inputs for pow-related functions in `array_api`
- [#20814](#): DOC: Refer to NumPy, not pandas, in main page
- [#20815](#): DOC: Update Copyright to 2022 [License]
- [#20819](#): BUG: Return correctly shaped inverse indices in `array_api.set...`

15.6 NumPy 1.22.0 Release Notes

NumPy 1.22.0 is a big release featuring the work of 153 contributors spread over 609 pull requests. There have been many improvements, highlights are:

- Annotations of the main namespace are essentially complete. Upstream is a moving target, so there will likely be further improvements, but the major work is done. This is probably the most user visible enhancement in this release.

- A preliminary version of the proposed Array-API is provided. This is a step in creating a standard collection of functions that can be used across applications such as CuPy and JAX.
- NumPy now has a DLPack backend. DLPack provides a common interchange format for array (tensor) data.
- New methods for `quantile`, `percentile`, and related functions. The new methods provide a complete set of the methods commonly found in the literature.
- The universal functions have been refactored to implement most of [NEP 43](#). This also unlocks the ability to experiment with the future DType API.
- A new configurable allocator for use by downstream projects.

These are in addition to the ongoing work to provide SIMD support for commonly used functions, improvements to F2PY, and better documentation.

The Python versions supported in this release are 3.8-3.10, Python 3.7 has been dropped. Note that the Mac wheels are now based on OS X 10.14 rather than 10.9 that was used in previous NumPy release cycles. 10.14 is the oldest release supported by Apple. Also note that 32 bit wheels are only provided for Python 3.8 and 3.9 on Windows, all other wheels are 64 bits on account of Ubuntu, Fedora, and other Linux distributions dropping 32 bit support. All 64 bit wheels are also linked with 64 bit integer OpenBLAS, which should fix the occasional problems encountered by folks using truly huge arrays.

15.6.1 Expired deprecations

Deprecated numeric style dtype strings have been removed

Using the strings `"Bytes0"`, `"Datetime64"`, `"Str0"`, `"Uint32"`, and `"Uint64"` as a dtype will now raise a `TypeError`.

([gh-19539](#))

Expired deprecations for `loads`, `ndfromtxt`, and `mafromtxt` in `numpyio`

`numpy.loads` was deprecated in v1.15, with the recommendation that users use `pickle.loads` instead. `ndfromtxt` and `mafromtxt` were both deprecated in v1.17 - users should use `numpy.genfromtxt` instead with the appropriate value for the `usemask` parameter.

([gh-19615](#))

15.6.2 Deprecations

Use `delimiter` rather than `delimitor` as `kwargs` in `mrecords`

The misspelled keyword argument `delimitor` of `numpy.ma.mrecords.fromtextfile()` has been changed to `delimiter`, using it will emit a deprecation warning.

([gh-19921](#))

Passing boolean `kth` values to `(arg-)partition` has been deprecated

`numpy.partition` and `numpy.argpartition` would previously accept boolean values for the `kth` parameter, which would subsequently be converted into integers. This behavior has now been deprecated.

(gh-20000)

The `np.MachAr` class has been deprecated

The `numpy.MachAr` class and `numpy.finfo.machar` (`numpy.finfo` attribute) have been deprecated. Users are encouraged to access the property if interest directly from the corresponding `numpy.finfo` attribute.

(gh-20201)

15.6.3 Compatibility notes

Distutils forces strict floating point model on clang

NumPy now sets the `-ftrapping-math` option on clang to enforce correct floating point error handling for universal functions. Clang defaults to non-IEEE and C99 conform behaviour otherwise. This change (using the equivalent but newer `-ffp-exception-behavior=strict`) was attempted in NumPy 1.21, but was effectively never used.

(gh-19479)

Removed floor division support for complex types

Floor division of complex types will now result in a `TypeError`

```
>>> a = np.arange(10) + 1j* np.arange(10)
>>> a // 1
TypeError: ufunc 'floor_divide' not supported for the input types...
```

(gh-19135)

`numpy.vectorize` functions now produce the same output class as the base function

When a function that respects `numpy.ndarray` subclasses is vectorized using `numpy.vectorize`, the vectorized function will now be subclass-safe also for cases that a signature is given (i.e., when creating a `gufunc`): the output class will be the same as that returned by the first call to the underlying function.

(gh-19356)

Python 3.7 is no longer supported

Python support has been dropped. This is rather strict, there are changes that require Python `>= 3.8`.

(gh-19665)

str/repr of complex dtypes now include space after punctuation

The `repr` of `np.dtype({"names": ["a"], "formats": [int], "offsets": [2]})` is now `dtype({'names': ['a'], 'formats': ['<i8'], 'offsets': [2], 'itemsize': 10})`, whereas spaces were previously omitted after colons and between fields.

The old behavior can be restored via `np.set_printoptions(legacy="1.21")`.

(gh-19687)

Corrected advance in PCG64DSXM and PCG64

Fixed a bug in the `advance` method of `PCG64DSXM` and `PCG64`. The bug only affects results when the step was larger than 2^{64} on platforms that do not support 128-bit integers (e.g., Windows and 32-bit Linux).

(gh-20049)

Change in generation of random 32 bit floating point variates

There was a bug in the generation of 32 bit floating point values from the uniform distribution that would result in the least significant bit of the random variate always being 0. This has been fixed.

This change affects the variates produced by the `random.Generator` methods `random`, `standard_normal`, `standard_exponential`, and `standard_gamma`, but only when the dtype is specified as `numpy.float32`.

(gh-20314)

15.6.4 C API changes**Masked inner-loops cannot be customized anymore**

The masked inner-loop selector is now never used. A warning will be given in the unlikely event that it was customized.

We do not expect that any code uses this. If you do use it, you must unset the selector on newer NumPy version. Please also contact the NumPy developers, we do anticipate providing a new, more specific, mechanism.

The customization was part of a never-implemented feature to allow for faster masked operations.

(gh-19259)

Experimental exposure of future DType and UFunc API

The new header `experimental_public_dtype_api.h` allows to experiment with future API for improved universal function and especially user DType support. At this time it is advisable to experiment using the development version of NumPy since some changes are expected and new features will be unlocked.

(gh-19919)

15.6.5 New Features

NEP 49 configurable allocators

As detailed in [NEP 49](#), the function used for allocation of the data segment of a ndarray can be changed. The policy can be set globally or in a context. For more information see the [NEP](#) and the [data_memory](#) reference docs. Also add a `NUMPY_WARN_IF_NO_MEM_POLICY` override to warn on dangerous use of transferring ownership by setting `NPY_ARRAY_OWNDATA`.

(gh-17582)

Implementation of the NEP 47 (adopting the array API standard)

An initial implementation of [NEP 47](#) (adoption the array API standard) has been added as `numpy.array_api`. The implementation is experimental and will issue a `UserWarning` on import, as the [array API standard](#) is still in draft state. `numpy.array_api` is a conforming implementation of the array API standard, which is also minimal, meaning that only those functions and behaviors that are required by the standard are implemented (see the [NEP](#) for more info). Libraries wishing to make use of the array API standard are encouraged to use `numpy.array_api` to check that they are only using functionality that is guaranteed to be present in standard conforming implementations.

(gh-18585)

Generate C/C++ API reference documentation from comments blocks is now possible

This feature depends on [Doxygen](#) in the generation process and on [Breathe](#) to integrate it with Sphinx.

(gh-18884)

Assign the platform-specific `c_intp` precision via a mypy plugin

The [mypy](#) plugin, introduced in [numpy/numpy#17843](#), has again been expanded: the plugin now is now responsible for setting the platform-specific precision of `numpy.ctypeslib.c_intp`, the latter being used as data type for various `numpy.ndarray.ctypes` attributes.

Without the plugin, aforementioned type will default to `ctypes.c_int64`.

To enable the plugin, one must add it to their mypy [configuration file](#):

```
[mypy]
plugins = numpy.typing.mypy_plugin
```

(gh-19062)

Add NEP 47-compatible dlpack support

Add a `ndarray.__dlpack__()` method which returns a `dlpack C` structure wrapped in a `PyCapsule`. Also add a `np._from_dlpack(obj)` function, where `obj` supports `__dlpack__()`, and returns an `ndarray`.

(gh-19083)

keepdims optional argument added to `numpy.argmax`, `numpy.argmin`

`keepdims` argument is added to `numpy.argmax`, `numpy.argmin`. If set to `True`, the axes which are reduced are left in the result as dimensions with size one. The resulting array has the same number of dimensions and will broadcast with the input array.

(gh-19211)

`bit_count` to compute the number of 1-bits in an integer

Computes the number of 1-bits in the absolute value of the input. This works on all the numpy integer types. Analogous to the builtin `int.bit_count` or `popcount` in C++.

```
>>> np.uint32(1023).bit_count()
10
>>> np.int32(-127).bit_count()
7
```

(gh-19355)

The `ndim` and `axis` attributes have been added to `numpy.AxisError`

The `ndim` and `axis` parameters are now also stored as attributes within each `numpy.AxisError` instance.

(gh-19459)

Preliminary support for windows/arm64 target

numpy added support for windows/arm64 target. Please note OpenBLAS support is not yet available for windows/arm64 target.

(gh-19513)

Added support for LoongArch

LoongArch is a new instruction set, numpy compilation failure on LoongArch architecture, so add the commit.

(gh-19527)

A `.clang-format` file has been added

Clang-format is a C/C++ code formatter, together with the added `.clang-format` file, it produces code close enough to the NumPy `C_STYLE_GUIDE` for general use. Clang-format version 12+ is required due to the use of several new features, it is available in Fedora 34 and Ubuntu Focal among other distributions.

(gh-19754)

`is_integer` is now available to `numpy.floating` and `numpy.integer`

Based on its counterpart in Python `float` and `int`, the numpy floating point and integer types now support `float.is_integer`. Returns `True` if the number is finite with integral value, and `False` otherwise.

```
>>> np.float32(-2.0).is_integer()
True
>>> np.float64(3.2).is_integer()
False
>>> np.int32(-2).is_integer()
True
```

(gh-19803)

Symbolic parser for Fortran dimension specifications

A new symbolic parser has been added to `f2py` in order to correctly parse dimension specifications. The parser is the basis for future improvements and provides compatibility with Draft Fortran 202x.

(gh-19805)

`ndarray`, `dtype` and `number` are now runtime-subscriptable

Mimicking [PEP 585](#), the `numpy.ndarray`, `numpy.dtype` and `numpy.number` classes are now subscriptable for python 3.9 and later. Consequently, expressions that were previously only allowed in `.pyi` stub files or with the help of `from __future__ import annotations` are now also legal during runtime.

```
>>> import numpy as np
>>> from typing import Any

>>> np.ndarray[Any, np.dtype[np.float64]]
numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]
```

(gh-19879)

15.6.6 Improvements

`ctypeslib.load_library` can now take any path-like object

All parameters in the can now take any [path-like object](#). This includes the likes of strings, bytes and objects implementing the `__fspath__` protocol.

(gh-17530)

Add `smallest_normal` and `smallest_subnormal` attributes to `finfo`

The attributes `smallest_normal` and `smallest_subnormal` are available as an extension of `finfo` class for any floating-point data type. To use these new attributes, write `np.finfo(np.float64).smallest_normal` or `np.finfo(np.float64).smallest_subnormal`.

(gh-18536)

`numpy.linalg.qr` accepts stacked matrices as inputs

`numpy.linalg.qr` is able to produce results for stacked matrices as inputs. Moreover, the implementation of QR decomposition has been shifted to C from Python.

(gh-19151)

`numpy.fromregex` now accepts `os.PathLike` implementations

`numpy.fromregex` now accepts objects implementing the `__fspath__`<`os.PathLike`> protocol, e.g. `pathlib.Path`.

(gh-19680)

Add new methods for `quantile` and `percentile`

`quantile` and `percentile` now have a `method=` keyword argument supporting 13 different methods. This replaces the `interpolation=` keyword argument.

The methods are now aligned with nine methods which can be found in scientific literature and the R language. The remaining methods are the previous discontinuous variations of the default “linear” one.

Please see the documentation of `numpy.percentile` for more information.

(gh-19857)

Missing parameters have been added to the `nan<x>` functions

A number of the `nan<x>` functions previously lacked parameters that were present in their `<x>`-based counterpart, e.g. the `where` parameter was present in `numpy.mean` but absent from `numpy.nanmean`.

The following parameters have now been added to the `nan<x>` functions:

- `nanmin`: `initial` & `where`
- `nanmax`: `initial` & `where`
- `nanargmin`: `keepdims` & `out`
- `nanargmax`: `keepdims` & `out`
- `nansum`: `initial` & `where`
- `nanprod`: `initial` & `where`
- `nanmean`: `where`
- `nanvar`: `where`
- `nanstd`: `where`

(gh-20027)

Annotating the main Numpy namespace

Starting from the 1.20 release, PEP 484 type annotations have been included for parts of the NumPy library; annotating the remaining functions being a work in progress. With the release of 1.22 this process has been completed for the main NumPy namespace, which is now fully annotated.

Besides the main namespace, a limited number of sub-packages contain annotations as well. This includes, among others, `numpy.testing`, `numpy.linalg` and `numpy.random` (available since 1.21).

(gh-20217)

Vectorize umath module using AVX-512

By leveraging Intel Short Vector Math Library (SVML), 18 umath functions (`exp2`, `log2`, `log10`, `expm1`, `log1p`, `cbrt`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `sinh`, `cosh`, `tanh`, `arcsinh`, `arccosh`, `arctanh`) are vectorized using AVX-512 instruction set for both single and double precision implementations. This change is currently enabled only for Linux users and on processors with AVX-512 instruction set. It provides an average speed up of 32x and 14x for single and double precision functions respectively.

(gh-19478)

OpenBLAS v0.3.18

Update the OpenBLAS used in testing and in wheels to v0.3.18

(gh-20058)

15.7 NumPy 1.21.6 Release Notes

NumPy 1.21.6 is a very small release that achieves two things:

- Backs out the mistaken backport of C++ code into 1.21.5.
- Provides a 32 bit Windows wheel for Python 3.10.

The provision of the 32 bit wheel is intended to make life easier for oldest-supported-numpy.

15.8 NumPy 1.21.5 Release Notes

NumPy 1.21.5 is a maintenance release that fixes a few bugs discovered after the 1.21.4 release and does some maintenance to extend the 1.21.x lifetime. The Python versions supported in this release are 3.7-3.10. If you want to compile your own version using gcc-11, you will need to use gcc-11.2+ to avoid problems.

15.8.1 Contributors

A total of 7 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Bas van Beek
- Charles Harris
- Matti Pícus
- Rohit Goswami
- Ross Barnowski
- Sayed Adel
- Sebastian Berg

15.8.2 Pull requests merged

A total of 11 pull requests were merged for this release.

- [#20357](#): MAINT: Do not forward `__ (deep) copy__` calls of `_GenericAlias...`
- [#20462](#): BUG: Fix float16 einsum fastpaths using wrong tempvar
- [#20463](#): BUG, DIST: Print os error message when the executable not exist
- [#20464](#): BLD: Verify the ability to compile C++ sources before initiating...
- [#20465](#): BUG: Force `npymath` ` to respect ``numpy_longdouble`
- [#20466](#): BUG: Fix failure to create aligned, empty structured dtype
- [#20467](#): ENH: provide a convenience function to replace `numpy_load_module`
- [#20495](#): MAINT: update wheel to version that supports python3.10
- [#20497](#): BUG: Clear errors correctly in F2PY conversions
- [#20613](#): DEV: add a warningfilter to fix pytest workflow.
- [#20618](#): MAINT: Help boost::python libraries at least not crash

15.9 NumPy 1.21.4 Release Notes

The NumPy 1.21.4 is a maintenance release that fixes a few bugs discovered after 1.21.3. The most important fix here is a fix for the NumPy header files to make them work for both x86_64 and M1 hardware when included in the Mac universal2 wheels. Previously, the header files only worked for M1 and this caused problems for folks building x86_64 extensions. This problem was not seen before Python 3.10 because there were thin wheels for x86_64 that had precedence. This release also provides thin x86_64 Mac wheels for Python 3.10.

The Python versions supported in this release are 3.7-3.10. If you want to compile your own version using gcc-11, you will need to use gcc-11.2+ to avoid problems.

15.9.1 Contributors

A total of 7 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Bas van Beek
- Charles Harris
- Isuru Fernando
- Matthew Brett
- Sayed Adel
- Sebastian Berg
- Chris Fu⁺ +

15.9.2 Pull requests merged

A total of 9 pull requests were merged for this release.

- [#20278](#): BUG: Fix shadowed reference of `dtype` in type stub
- [#20293](#): BUG: Fix headers for universal2 builds
- [#20294](#): BUG: `VOID_nonzero` could sometimes mutate alignment flag
- [#20295](#): BUG: Do not use nonzero fastpath on unaligned arrays
- [#20296](#): BUG: Distutils patch to allow for 2 as a minor version (!)
- [#20297](#): BUG, SIMD: Fix 64-bit/8-bit integer division by a scalar
- [#20298](#): BUG, SIMD: Workaround broadcasting SIMD 64-bit integers on MSVC...
- [#20300](#): REL: Prepare for the NumPy 1.21.4 release.
- [#20302](#): TST: Fix a `Arrayterator` typing test failure

15.10 NumPy 1.21.3 Release Notes

NumPy 1.21.3 is a maintenance release that fixes a few bugs discovered after 1.21.2. It also provides 64 bit Python 3.10.0 wheels. Note a few oddities about Python 3.10:

- There are no 32 bit wheels for Windows, Mac, or Linux.
- The Mac Intel builds are only available in universal2 wheels.

The Python versions supported in this release are 3.7-3.10. If you want to compile your own version using gcc-11, you will need to use gcc-11.2+ to avoid problems.

15.10.1 Contributors

A total of 7 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Aaron Meurer
- Bas van Beek
- Charles Harris
- Developer-Ecosystem-Engineering +
- Kevin Sheppard
- Sebastian Berg
- Warren Weckesser

15.10.2 Pull requests merged

A total of 8 pull requests were merged for this release.

- [#19745](#): ENH: Add dtype-support to 3 `generic/ndarray` methods
- [#19955](#): BUG: Resolve Divide by Zero on Apple silicon + test failures...
- [#19958](#): MAINT: Mark type-check-only ufunc subclasses as ufunc aliases...
- [#19994](#): BUG: `np.tan(np.inf)` test failure
- [#20080](#): BUG: Correct incorrect advance in PCG with emulated int128
- [#20081](#): BUG: Fix NaT handling in the `PyArray_CompareFunc` for datetime...
- [#20082](#): DOC: Ensure that we add documentation also as to the dict for...
- [#20106](#): BUG: core: `result_type(0, np.timedelta64(4))` would seg. fault.

15.11 NumPy 1.21.2 Release Notes

The NumPy 1.21.2 is a maintenance release that fixes bugs discovered after 1.21.1. It also provides 64 bit manylinux Python 3.10.0rc1 wheels for downstream testing. Note that Python 3.10 is not yet final. It also has preliminary support for Windows on ARM64, but there is no OpenBLAS for that platform and no wheels are available.

The Python versions supported for this release are 3.7-3.9. The 1.21.x series is compatible with Python 3.10.0rc1 and Python 3.10 will be officially supported after it is released. The previous problems with gcc-11.1 have been fixed by gcc-11.2, check your version if you are using gcc-11.

15.11.1 Contributors

A total of 10 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Bas van Beek
- Carl Johnsen +
- Charles Harris
- Gwyn Ciesla +
- Matthieu Dartiailh

- Matti Pícus
- Niyas Sait +
- Ralf Gommers
- Sayed Adel
- Sebastian Berg

15.11.2 Pull requests merged

A total of 18 pull requests were merged for this release.

- [#19497](#): MAINT: set Python version for 1.21.x to <3.11
- [#19533](#): BUG: Fix an issue wherein importing `numpy.typing` could raise
- [#19646](#): MAINT: Update Cython version for Python 3.10.
- [#19648](#): TST: Bump the python 3.10 test version from beta4 to rc1
- [#19651](#): TST: avoid `distutils.sysconfig` in `runtests.py`
- [#19652](#): MAINT: add missing dunder method to `nditer` type hints
- [#19656](#): BLD, SIMD: Fix testing extra checks when `-Werror` isn't applicable...
- [#19657](#): BUG: Remove logical object ufuncs with bool output
- [#19658](#): MAINT: Include `.coveragerc` in source distributions to support...
- [#19659](#): BUG: Fix bad write in masked iterator output copy paths
- [#19660](#): ENH: Add support for windows on arm targets
- [#19661](#): BUG: add base to templated arguments for `platlib`
- [#19662](#): BUG,DEP: Non-default UFunc signature/dtype usage should be deprecated
- [#19666](#): MAINT: Add Python 3.10 to supported versions.
- [#19668](#): TST,BUG: Sanitize path-separators when running `runtest.py`
- [#19671](#): BLD: load extra flags when checking for `libflame`
- [#19676](#): BLD: update circleCI docker image
- [#19677](#): REL: Prepare for 1.21.2 release.

15.12 NumPy 1.21.1 Release Notes

The NumPy 1.21.1 is maintenance release that fixes bugs discovered after the 1.21.0 release and updates OpenBLAS to v0.3.17 to deal with problems on arm64.

The Python versions supported for this release are 3.7-3.9. The 1.21.x series is compatible with development Python 3.10. Python 3.10 will be officially supported after it is released.

Warning: There are unresolved problems compiling NumPy 1.20.0 with gcc-11.1.

- Optimization level `-O3` results in many incorrect warnings when running the tests.

- On some hardware NumPY will hang in an infinite loop.

15.12.1 Contributors

A total of 11 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Bas van Beek
- Charles Harris
- Ganesh Kathiresan
- Gregory R. Lee
- Hugo Defois +
- Kevin Sheppard
- Matti Pícus
- Ralf Gommers
- Sayed Adel
- Sebastian Berg
- Thomas J. Fan

15.12.2 Pull requests merged

A total of 26 pull requests were merged for this release.

- [#19311](#): REV,BUG: Replace `NotImplemented` with `typing.Any`
- [#19324](#): MAINT: Fixed the return-dtype of `ndarray.real` and `imag`
- [#19330](#): MAINT: Replace `"dtype[Any]"` with `dtype` in the definiton of...
- [#19342](#): DOC: Fix some docstrings that crash pdf generation.
- [#19343](#): MAINT: bump `scipy-mathjax`
- [#19347](#): BUG: Fix `arr.flat.index` for large arrays and big-endian machines
- [#19348](#): ENH: add `numpy.f2py.get_include` function
- [#19349](#): BUG: Fix reference count leak in `ufunc dtype` handling
- [#19350](#): MAINT: Annotate missing attributes of `np.number` subclasses
- [#19351](#): BUG: Fix cast safety and comparisons for zero sized voids
- [#19352](#): BUG: Correct Cython declaration in `random`
- [#19353](#): BUG: protect against accessing base attribute of a `NULL` subarray
- [#19365](#): BUG, SIMD: Fix detecting `AVX512` features on Darwin
- [#19366](#): MAINT: remove `print()`’s in `distutils` template handling
- [#19390](#): ENH: SIMD architectures to `show_config`
- [#19391](#): BUG: Do not raise deprecation warning for all nans in `unique...`
- [#19392](#): BUG: Fix `NULL` special case in object-to-any cast code

- [#19430](#): MAINT: Use arm64-graviton2 for testing on travis
- [#19495](#): BUILD: update OpenBLAS to v0.3.17
- [#19496](#): MAINT: Avoid unicode characters in division SIMD code comments
- [#19499](#): BUG, SIMD: Fix infinite loop during count non-zero on GCC-11
- [#19500](#): BUG: fix a numpy.npyiter leak in npyiter_multi_index_set
- [#19501](#): TST: Fix a `GenericAlias` test failure for python 3.9.0
- [#19502](#): MAINT: Start testing with Python 3.10.0b3.
- [#19503](#): MAINT: Add missing dtype overloads for object- and ctypes-based...
- [#19510](#): REL: Prepare for NumPy 1.21.1 release.

15.13 NumPy 1.21.0 Release Notes

The NumPy 1.21.0 release highlights are

- continued SIMD work covering more functions and platforms,
- initial work on the new dtype infrastructure and casting,
- universal2 wheels for Python 3.8 and Python 3.9 on Mac,
- improved documentation,
- improved annotations,
- new PCG64DXSM bitgenerator for random numbers.

In addition there are the usual large number of bug fixes and other improvements.

The Python versions supported for this release are 3.7-3.9. Official support for Python 3.10 will be added when it is released.

Warning: There are unresolved problems compiling NumPy 1.20.0 with gcc-11.1.

- Optimization level `-O3` results in many incorrect warnings when running the tests.
- On some hardware NumPY will hang in an infinite loop.

15.13.1 New functions

Add PCG64DXSM BitGenerator

Uses of the PCG64 BitGenerator in a massively-parallel context have been shown to have statistical weaknesses that were not apparent at the first release in numpy 1.17. Most users will never observe this weakness and are safe to continue to use PCG64. We have introduced a new PCG64DXSM BitGenerator that will eventually become the new default BitGenerator implementation used by `default_rng` in future releases. PCG64DXSM solves the statistical weakness while preserving the performance and the features of PCG64.

See upgrading-pcg64 for more details.

([gh-18906](#))

15.13.2 Expired deprecations

- The `shape` argument `unravel_index` cannot be passed as `dims` keyword argument anymore. (Was deprecated in NumPy 1.16.)
([gh-17900](#))
- The function `PyUFunc_GenericFunction` has been disabled. It was deprecated in NumPy 1.19. Users should call the `ufunc` directly using the Python API.
([gh-18697](#))
- The function `PyUFunc_SetUsesArraysAsData` has been disabled. It was deprecated in NumPy 1.19.
([gh-18697](#))
- The class `PolyBase` has been removed (deprecated in numpy 1.9.0). Please use the abstract `ABCPolyBase` class instead.
([gh-18963](#))
- The unused `PolyError` and `PolyDomainError` exceptions are removed.
([gh-18963](#))

15.13.3 Deprecations

The `.dtype` attribute must return a `dtype`

A `DeprecationWarning` is now given if the `.dtype` attribute of an object passed into `np.dtype` or as a `dtype=obj` argument is not a `dtype`. NumPy will stop attempting to recursively coerce the result of `.dtype`.

([gh-13578](#))

Inexact matches for `numpy.convolve` and `numpy.correlate` are deprecated

`convolve` and `correlate` now emit a warning when there are case insensitive and/or inexact matches found for `mode` argument in the functions. Pass full "same", "valid", "full" strings instead of "s", "v", "f" for the `mode` argument.

([gh-17492](#))

`np.typeDict` has been formally deprecated

`np.typeDict` is a deprecated alias for `np.sctypeDict` and has been so for over 14 years ([6689502](#)). A deprecation warning will now be issued whenever getting `np.typeDict`.

([gh-17586](#))

Exceptions will be raised during array-like creation

When an object raised an exception during access of the special attributes `__array__` or `__array_interface__`, this exception was usually ignored. A warning is now given when the exception is anything but `AttributeError`. To silence the warning, the type raising the exception has to be adapted to raise an `AttributeError`.

(gh-19001)

Four `ndarray.ctypes` methods have been deprecated

Four methods of the `ndarray.ctypes` object have been deprecated, as they are (undocumented) implementation artifacts of their respective properties.

The methods in question are:

- `_ctypes.get_data` (use `_ctypes.data` instead)
- `_ctypes.get_shape` (use `_ctypes.shape` instead)
- `_ctypes.get_strides` (use `_ctypes.strides` instead)
- `_ctypes.get_as_parameter` (use `_ctypes._as_parameter_` instead)

(gh-19031)

15.13.4 Expired deprecations

- The shape argument `numpy.unravel_index` cannot be passed as `dims` keyword argument anymore. (Was deprecated in NumPy 1.16.)

(gh-17900)

- The function `PyUFunc_GenericFunction` has been disabled. It was deprecated in NumPy 1.19. Users should call the ufunc directly using the Python API.

(gh-18697)

- The function `PyUFunc_SetUsesArraysAsData` has been disabled. It was deprecated in NumPy 1.19.

(gh-18697)

Remove deprecated `PolyBase` and unused `PolyError` and `PolyDomainError`

The class `PolyBase` has been removed (deprecated in numpy 1.9.0). Please use the abstract `ABCPolyBase` class instead.

Furthermore, the unused `PolyError` and `PolyDomainError` exceptions are removed from the `numpy.polynomial`.

(gh-18963)

15.13.5 Compatibility notes

Error type changes in universal functions

The universal functions may now raise different errors on invalid input in some cases. The main changes should be that a `RuntimeError` was replaced with a more fitting `TypeError`. When multiple errors were present in the same call, NumPy may now raise a different one.

(gh-15271)

`__array_ufunc__` argument validation

NumPy will now partially validate arguments before calling `__array_ufunc__`. Previously, it was possible to pass on invalid arguments (such as a non-existing keyword argument) when dispatch was known to occur.

(gh-15271)

`__array_ufunc__` and additional positional arguments

Previously, all positionally passed arguments were checked for `__array_ufunc__` support. In the case of `reduce`, `accumulate`, and `reduceat` all arguments may be passed by position. This means that when they were passed by position, they could previously have been asked to handle the ufunc call via `__array_ufunc__`. Since this depended on the way the arguments were passed (by position or by keyword), NumPy will now only dispatch on the input and output array. For example, NumPy will never dispatch on the `where` array in a reduction such as `np.add.reduce`.

(gh-15271)

Validate input values in `Generator.uniform`

Checked that `high - low >= 0` in `np.random.Generator.uniform`. Raises `ValueError` if `low > high`. Previously out-of-order inputs were accepted and silently swapped, so that if `low > high`, the value generated was `high + (low - high) * random()`.

(gh-17921)

`/usr/include` removed from default include paths

The default include paths when building a package with `numpy.distutils` no longer include `/usr/include`. This path is normally added by the compiler, and hardcoding it can be problematic. In case this causes a problem, please open an issue. A workaround is documented in PR 18658.

(gh-18658)

Changes to comparisons with `dtype=...`

When the `dtype=` (or `signature`) arguments to comparison ufuncs (`equal`, `less`, etc.) is used, this will denote the desired output dtype in the future. This means that:

```
np.equal(2, 3, dtype=object)
```

will give a `FutureWarning` that it will return an object array in the future, which currently happens for:

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

due to the fact that `np.array(None)` is already an object array. (This also happens for some other dtypes.)

Since comparisons normally only return boolean arrays, providing any other dtype will always raise an error in the future and give a `DeprecationWarning` now.

(gh-18718)

Changes to `dtype` and `signature` arguments in ufuncs

The universal function arguments `dtype` and `signature` which are also valid for reduction such as `np.add.reduce` (which is the implementation for `np.sum`) will now issue a warning when the `dtype` provided is not a “basic” dtype.

NumPy almost always ignored metadata, byteorder or time units on these inputs. NumPy will now always ignore it and raise an error if byteorder or time unit changed. The following are the most important examples of changes which will give the error. In some cases previously the information stored was not ignored, in all of these an error is now raised:

```
# Previously ignored the byte-order (affect if non-native)
np.add(3, 5, dtype=>i32")

# The biggest impact is for timedelta or datetimes:
arr = np.arange(10, dtype="m8[s]")
# The examples always ignored the time unit "ns":
np.add(arr, arr, dtype="m8[ns]")
np.maximum.reduce(arr, dtype="m8[ns]")

# The following previously did use "ns" (as opposed to `arr.dtype`)
np.add(3, 5, dtype="m8[ns]") # Now return generic time units
np.maximum(arr, arr, dtype="m8[ns]") # Now returns "s" (from `arr`)
```

The same applies for functions like `np.sum` which use these internally. This change is necessary to achieve consistent handling within NumPy.

If you run into these, in most cases pass for example `dtype=np.timedelta64` which clearly denotes a general `timedelta64` without any unit or byte-order defined. If you need to specify the output dtype precisely, you may do so by either casting the inputs or providing an output array using `out=`.

NumPy may choose to allow providing an exact output dtype here in the future, which would be preceded by a `FutureWarning`.

(gh-18718)

Ufunc signature=... and dtype= generalization and casting

The behaviour for `np.ufunc(1.0, 1.0, signature=...)` or `np.ufunc(1.0, 1.0, dtype=...)` can now yield different loops in 1.21 compared to 1.20 because of changes in promotion. When `signature` was previously used, the casting check on inputs was relaxed, which could lead to downcasting inputs unsafely especially if combined with `casting="unsafe"`.

Casting is now guaranteed to be safe. If a signature is only partially provided, for example using `signature=("float64", None, None)`, this could lead to no loop being found (an error). In that case, it is necessary to provide the complete signature to enforce casting the inputs. If `dtype="float64"` is used or only outputs are set (e.g. `signature=(None, None, "float64")`) the is unchanged. We expect that very few users are affected by this change.

Further, the meaning of `dtype="float64"` has been slightly modified and now strictly enforces only the correct output (and not input) DTypes. This means it is now always equivalent to:

```
signature=(None, None, "float64")
```

(If the ufunc has two inputs and one output). Since this could lead to no loop being found in some cases, NumPy will normally also search for the loop:

```
signature=("float64", "float64", "float64")
```

if the first search failed. In the future, this behaviour may be customized to achieve the expected results for more complex ufuncs. (For some universal functions such as `np.ldexp` inputs can have different DTypes.)

(gh-18880)

Distutils forces strict floating point model on clang

NumPy distutils will now always add the `-ffp-exception-behavior=strict` compiler flag when compiling with clang. Clang defaults to a non-strict version, which allows the compiler to generate code that does not set floating point warnings/errors correctly.

(gh-19049)

15.13.6 C API changes

Use of `ufunc->type_resolver` and “type tuple”

NumPy now normalizes the “type tuple” argument to the type resolver functions before calling it. Note that in the use of this type resolver is legacy behaviour and NumPy will not do so when possible. Calling `ufunc->type_resolver` or `PyUFunc_DefaultTypeResolver` is strongly discouraged and will now enforce a normalized type tuple if done. Note that this does not affect providing a type resolver, which is expected to keep working in most circumstances. If you have an unexpected use-case for calling the type resolver, please inform the NumPy developers so that a solution can be found.

(gh-18718)

15.13.7 New Features

Added a mypy plugin for handling platform-specific `numpy.number` precisions

A `mypy` plugin is now available for automatically assigning the (platform-dependent) precisions of certain `number` subclasses, including the likes of `int_`, `intp` and `longlong`. See the documentation on scalar types for a comprehensive overview of the affected classes.

Note that while usage of the plugin is completely optional, without it the precision of above-mentioned classes will be inferred as `Any`.

To enable the plugin, one must add it to their `mypy` configuration file:

```
[mypy]
plugins = numpy.typing.mypy_plugin
```

(gh-17843)

Let the mypy plugin manage extended-precision `numpy.number` subclasses

The `mypy` plugin, introduced in [numpy/numpy#17843](#), has been expanded: the plugin now removes annotations for platform-specific extended-precision types that are not available to the platform in question. For example, it will remove `float128` when not available.

Without the plugin *all* extended-precision types will, as far as `mypy` is concerned, be available on all platforms.

To enable the plugin, one must add it to their `mypy` configuration file:

```
[mypy]
plugins = numpy.typing.mypy_plugin
```

(gh-18322)

New `min_digits` argument for printing float values

A new `min_digits` argument has been added to the `dragon4` float printing functions `format_float_positional` and `format_float_scientific`. This kwarg guarantees that at least the given number of digits will be printed when printing in `unique=True` mode, even if the extra digits are unnecessary to uniquely specify the value. It is the counterpart to the `precision` argument which sets the maximum number of digits to be printed. When `unique=False` in fixed precision mode, it has no effect and the `precision` argument fixes the number of digits.

(gh-18629)

`f2py` now recognizes Fortran abstract interface blocks

`f2py` can now parse abstract interface blocks.

(gh-18695)

BLAS and LAPACK configuration via environment variables

Autodetection of installed BLAS and LAPACK libraries can be bypassed by using the `NPY_BLAS_LIBS` and `NPY_LAPACK_LIBS` environment variables. Instead, the link flags in these environment variables will be used directly, and the language is assumed to be F77. This is especially useful in automated builds where the BLAS and LAPACK that are installed are known exactly. A use case is replacing the actual implementation at runtime via stub library links.

If `NPY_CBLAS_LIBS` is set (optional in addition to `NPY_BLAS_LIBS`), this will be used as well, by defining `HAVE_CBLAS` and appending the environment variable content to the link flags.

(gh-18737)

A runtime-subscriptable alias has been added for `ndarray`

`numpy.typing.NDArray` has been added, a runtime-subscriptable alias for `np.ndarray[Any, np.dtype[~Scalar]]`. The new type alias can be used for annotating arrays with a given dtype and unspecified shape.

¹ NumPy does not support the annotating of array shapes as of 1.21, this is expected to change in the future though (see [PEP 646](#)).

Examples

```
>>> import numpy as np
>>> import numpy.typing as npt

>>> print(npt.NDArray)
numpy.ndarray[typing.Any, numpy.dtype[~ScalarType]]

>>> print(npt.NDArray[np.float64])
numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]

>>> NDArrayInt = npt.NDArray[np.int_]
>>> a: NDArrayInt = np.arange(10)

>>> def func(a: npt.ArrayLike) -> npt.NDArray[Any]:
...     return np.array(a)
```

(gh-18935)

15.13.8 Improvements

Arbitrary period option for `numpy.unwrap`

The size of the interval over which phases are unwrapped is no longer restricted to $2 * \pi$. This is especially useful for unwrapping degrees, but can also be used for other intervals.

```
>>> phase_deg = np.mod(np.linspace(0, 720, 19), 360) - 180
>>> phase_deg
array([-180., -140., -100.,  -60.,  -20.,   20.,   60.,  100.,  140.,
        -180., -140., -100.,  -60.,  -20.,   20.,   60.,  100.,  140.,
        -180.])

>>> unwrap(phase_deg, period=360)
array([-180., -140., -100.,  -60.,  -20.,   20.,   60.,  100.,  140.,
```

(continues on next page)

(continued from previous page)

```
180., 220., 260., 300., 340., 380., 420., 460., 500.,  
540.]
```

[\(gh-16987\)](#)

`np.unique` now returns single NaN

When `np.unique` operated on an array with multiple NaN entries, its return included a NaN for each entry that was NaN in the original array. This is now improved such that the returned array contains just one NaN as the last element.

Also for complex arrays all NaN values are considered equivalent (no matter whether the NaN is in the real or imaginary part). As the representant for the returned array the smallest one in the lexicographical order is chosen - see `np.sort` for how the lexicographical order is defined for complex arrays.

[\(gh-18070\)](#)

`Generator.rayleigh` and `Generator.geometric` performance improved

The performance of Rayleigh and geometric random variate generation in `Generator` has improved. These are both transformation of exponential random variables and the slow log-based inverse cdf transformation has been replaced with the Ziggurat-based exponential variate generator.

This change breaks the stream of variates generated when variates from either of these distributions are produced.

[\(gh-18666\)](#)

Placeholder annotations have been improved

All placeholder annotations, that were previously annotated as `typing.Any`, have been improved. Where appropriate they have been replaced with explicit function definitions, classes or other miscellaneous objects.

[\(gh-18934\)](#)

15.13.9 Performance improvements

Improved performance in integer division of NumPy arrays

Integer division of NumPy arrays now uses `libdivide` when the divisor is a constant. With the usage of `libdivide` and other minor optimizations, there is a large speedup. The `//` operator and `np.floor_divide` makes use of the new changes.

[\(gh-17727\)](#)

Improve performance of `np.save` and `np.load` for small arrays

`np.save` is now a lot faster for small arrays.

`np.load` is also faster for small arrays, but only when serializing with a version $\geq (3, 0)$.

Both are done by removing checks that are only relevant for Python 2, while still maintaining compatibility with arrays which might have been created by Python 2.

(gh-18657)

15.13.10 Changes

`numpy.piecewise` output class now matches the input class

When `ndarray` subclasses are used on input to `piecewise`, they are passed on to the functions. The output will now be of the same subclass as well.

(gh-18110)

Enable Accelerate Framework

With the release of macOS 11.3, several different issues that numpy was encountering when using Accelerate Framework's implementation of BLAS and LAPACK should be resolved. This change enables the Accelerate Framework as an option on macOS. If additional issues are found, please file a bug report against Accelerate using the developer feedback assistant tool (<https://developer.apple.com/bug-reporting/>). We intend to address issues promptly and plan to continue supporting and updating our BLAS and LAPACK libraries.

(gh-18874)

15.14 NumPy 1.20.3 Release Notes

NumPy 1.20.3 is a bugfix release containing several fixes merged to the main branch after the NumPy 1.20.2 release.

15.14.1 Contributors

A total of 7 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Anne Archibald
- Bas van Beek
- Charles Harris
- Dong Keun Oh +
- Kamil Choudhury +
- Sayed Adel
- Sebastian Berg

15.14.2 Pull requests merged

A total of 15 pull requests were merged for this release.

- [#18763](#): BUG: Correct `datetime64` missing type overload for `datetime.date...`
- [#18764](#): MAINT: Remove `__all__` in favor of explicit re-exports
- [#18768](#): BLD: Strip extra newline when dumping gfortran version on MacOS
- [#18769](#): BUG: fix segfault in object/longdouble operations
- [#18794](#): MAINT: Use `towncrier` build explicitly
- [#18887](#): MAINT: Relax certain integer-type constraints
- [#18915](#): MAINT: Remove unsafe unions and ABCs from return-annotations
- [#18921](#): MAINT: Allow more recursion depth for scalar tests.
- [#18922](#): BUG: Initialize the full `nditer` buffer in case of error
- [#18923](#): BLD: remove unnecessary flag `-faltivec` on macOS
- [#18924](#): MAINT, CI: treats `_SIMD` module build warnings as errors through...
- [#18925](#): BUG: for MINGW, `threads.h` existence test requires `GLIBC > 2.12`
- [#18941](#): BUG: Make changelog recognize `gh-` as a PR number prefix.
- [#18948](#): REL, DOC: Prepare for the NumPy 1.20.3 release.
- [#18953](#): BUG: Fix failing `mypy` test in 1.20.x.

15.15 NumPy 1.20.2 Release Notes

NumPy 1.20.2 is a bugfix release containing several fixes merged to the main branch after the NumPy 1.20.1 release.

15.15.1 Contributors

A total of 7 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Allan Haldane
- Bas van Beek
- Charles Harris
- Christoph Gohlke
- Mateusz Sokół +
- Michael Lamparski
- Sebastian Berg

15.15.2 Pull requests merged

A total of 20 pull requests were merged for this release.

- [#18382](#): MAINT: Update f2py from master.
- [#18459](#): BUG: `diagflat` could overflow on windows or 32-bit platforms
- [#18460](#): BUG: Fix refcount leak in `f2py complex_double_from_pyobj`.
- [#18461](#): BUG: Fix tiny memory leaks when `like=` overrides are used
- [#18462](#): BUG: Remove temporary change of `descr/flags` in VOID functions
- [#18469](#): BUG: Segfault in `nditer` buffer dealloc for Object arrays
- [#18485](#): BUG: Remove suspicious type casting
- [#18486](#): BUG: remove nonsensical comparison of `pointer < 0`
- [#18487](#): BUG: verify pointer against NULL before using it
- [#18488](#): BUG: check if `PyArray_malloc` succeeded
- [#18546](#): BUG: incorrect error fallthrough in `nditer`
- [#18559](#): CI: Backport CI fixes from main.
- [#18599](#): MAINT: Add annotations for `dtype.__getitem__`, `__mul__` and...
- [#18611](#): BUG: `NameError` in `numpy.distutils.fcompiler.compaq`
- [#18612](#): BUG: Fixed `where` keyword for `np.mean` & `np.var` methods
- [#18617](#): CI: Update apt package list before Python install
- [#18636](#): MAINT: Ensure that re-exported sub-modules are properly annotated
- [#18638](#): BUG: Fix `ma` coercion list-of-`ma`-arrays if they do not cast to...
- [#18661](#): BUG: Fix small valgrind-found issues
- [#18671](#): BUG: Fix small issues found with `pytest-leaks`

15.16 NumPy 1.20.1 Release Notes

NumPy 1.20.1 is a rapid bugfix release fixing several bugs and regressions reported after the 1.20.0 release.

15.16.1 Highlights

- The `distutils` bug that caused problems with downstream projects is fixed.
- The `random.shuffle` regression is fixed.

15.16.2 Contributors

A total of 8 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Bas van Beek
- Charles Harris
- Nicholas McKibben +
- Pearu Peterson
- Ralf Gommers
- Sebastian Berg
- Tyler Reddy
- @Aerysv +

15.16.3 Pull requests merged

A total of 15 pull requests were merged for this release.

- [#18306](#): MAINT: Add missing placeholder annotations
- [#18310](#): BUG: Fix typo in `numpy.__init__.py`
- [#18326](#): BUG: don't mutate list of fake libraries while iterating over...
- [#18327](#): MAINT: gracefully shuffle memoryviews
- [#18328](#): BUG: Use C linkage for random distributions
- [#18336](#): CI: fix when GitHub Actions builds trigger, and allow ci skips
- [#18337](#): BUG: Allow unmodified use of `isclose`, `allclose`, etc. with `timedelta`
- [#18345](#): BUG: Allow pickling all relevant DType types/classes
- [#18351](#): BUG: Fix missing `signed_char` dependency. Closes [#18335](#).
- [#18352](#): DOC: Change license date 2020 -> 2021
- [#18353](#): CI: CircleCI seems to occasionally time out, increase the limit
- [#18354](#): BUG: Fix `f2py` bugs when wrapping F90 subroutines.
- [#18356](#): MAINT: crackfortran regex simplify
- [#18357](#): BUG: `threads.h` existence test requires `GLIBC > 2.12`.
- [#18359](#): REL: Prepare for the NumPy 1.20.1 release.

15.17 NumPy 1.20.0 Release Notes

This NumPy release is the largest so made to date, some 684 PRs contributed by 184 people have been merged. See the list of highlights below for more details. The Python versions supported for this release are 3.7-3.9, support for Python 3.6 has been dropped. Highlights are

- Annotations for NumPy functions. This work is ongoing and improvements can be expected pending feedback from users.
- Wider use of SIMD to increase execution speed of ufuncs. Much work has been done in introducing universal functions that will ease use of modern features across different hardware platforms. This work is ongoing.
- Preliminary work in changing the dtype and casting implementations in order to provide an easier path to extending dtypes. This work is ongoing but enough has been done to allow experimentation and feedback.
- Extensive documentation improvements comprising some 185 PR merges. This work is ongoing and part of the larger project to improve NumPy's online presence and usefulness to new users.
- Further cleanups related to removing Python 2.7. This improves code readability and removes technical debt.
- Preliminary support for the upcoming Cython 3.0.

15.17.1 New functions

The `random.Generator` class has a new `permuted` function.

The new function differs from `shuffle` and `permutation` in that the subarrays indexed by an axis are permuted rather than the axis being treated as a separate 1-D array for every combination of the other indexes. For example, it is now possible to permute the rows or columns of a 2-D array.

(gh-15121)

`sliding_window_view` provides a sliding window view for numpy arrays

`numpy.lib.stride_tricks.sliding_window_view` constructs views on numpy arrays that offer a sliding or moving window access to the array. This allows for the simple implementation of certain algorithms, such as running means.

(gh-17394)

`numpy.broadcast_shapes` is a new user-facing function

`broadcast_shapes` gets the resulting shape from broadcasting the given shape tuples against each other.

```
>>> np.broadcast_shapes((1, 2), (3, 1))
(3, 2)

>>> np.broadcast_shapes(2, (3, 1))
(3, 2)

>>> np.broadcast_shapes((6, 7), (5, 6, 1), (7,), (5, 1, 7))
(5, 6, 7)
```

(gh-17535)

15.17.2 Deprecations

Using the aliases of builtin types like `np.int` is deprecated

For a long time, `np.int` has been an alias of the builtin `int`. This is repeatedly a cause of confusion for newcomers, and existed mainly for historic reasons.

These aliases have been deprecated. The table below shows the full list of deprecated aliases, along with their exact meaning. Replacing uses of items in the first column with the contents of the second column will work identically and silence the deprecation warning.

The third column lists alternative NumPy names which may occasionally be preferential. See also [Data types](#) for additional details.

Deprecated name	Identical to	NumPy scalar type names
<code>numpy.bool</code>	<code>bool</code>	<code>numpy.bool_</code>
<code>numpy.int</code>	<code>int</code>	<code>numpy.int_</code> (default), <code>numpy.int64</code> , or <code>numpy.int32</code>
<code>numpy.float</code>	<code>float</code>	<code>numpy.float64</code> , <code>numpy.float_</code> , <code>numpy.double</code> (equivalent)
<code>numpy.complex</code>	<code>complex</code>	<code>numpy.complex128</code> , <code>numpy.complex_</code> , <code>numpy.cdouble</code> (equivalent)
<code>numpy.object</code>	<code>object</code>	<code>numpy.object_</code>
<code>numpy.str</code>	<code>str</code>	<code>numpy.str_</code>
<code>numpy.long</code>	<code>int</code>	<code>numpy.int_</code> (C long), <code>numpy.longlong</code> (largest integer type)
<code>numpy.unicode</code>	<code>str</code>	<code>numpy.unicode_</code>

To give a clear guideline for the vast majority of cases, for the types `bool`, `object`, `str` (and `unicode`) using the plain version is shorter and clear, and generally a good replacement. For `float` and `complex` you can use `float64` and `complex128` if you wish to be more explicit about the precision.

For `np.int` a direct replacement with `np.int_` or `int` is also good and will not change behavior, but the precision will continue to depend on the computer and operating system. If you want to be more explicit and review the current use, you have the following alternatives:

- `np.int64` or `np.int32` to specify the precision exactly. This ensures that results cannot depend on the computer or operating system.
- `np.int_` or `int` (the default), but be aware that it depends on the computer and operating system.
- The C types: `np.cint` (`int`), `np.int_` (`long`), `np.longlong`.
- `np.intp` which is 32bit on 32bit machines 64bit on 64bit machines. This can be the best type to use for indexing.

When used with `np.dtype(...)` or `dtype=...` changing it to the NumPy name as mentioned above will have no effect on the output. If used as a scalar with:

```
np.float(123)
```

changing it can subtly change the result. In this case, the Python version `float(123)` or `int(12.)` is normally preferable, although the NumPy version may be useful for consistency with NumPy arrays (for example, NumPy behaves differently for things like division by zero).

(gh-14882)

Passing `shape=None` to functions with a non-optional shape argument is deprecated

Previously, this was an alias for passing `shape=()`. This deprecation is emitted by *PyArray_IntpConverter* in the C API. If your API is intended to support passing `None`, then you should check for `None` prior to invoking the converter, so as to be able to distinguish `None` and `()`.

(gh-15886)

Indexing errors will be reported even when index result is empty

In the future, NumPy will raise an `IndexError` when an integer array index contains out of bound values even if a non-indexed dimension is of length 0. This will now emit a `DeprecationWarning`. This can happen when the array is previously empty, or an empty slice is involved:

```
arr1 = np.zeros((5, 0))
arr1[[20]]
arr2 = np.zeros((5, 5))
arr2[[20], :0]
```

Previously the non-empty index `[20]` was not checked for correctness. It will now be checked causing a deprecation warning which will be turned into an error. This also applies to assignments.

(gh-15900)

Inexact matches for `mode` and `searchside` are deprecated

Inexact and case insensitive matches for `mode` and `searchside` were valid inputs earlier and will give a `DeprecationWarning` now. For example, below are some example usages which are now deprecated and will give a `DeprecationWarning`:

```
import numpy as np
arr = np.array([[3, 6, 6], [4, 5, 1]])
# mode: inexact match
np.ravel_multi_index(arr, (7, 6), mode="clap") # should be "clip"
# searchside: inexact match
np.searchsorted(arr[0], 4, side='random') # should be "right"
```

(gh-16056)

Deprecation of `numpy.dual`

The module `numpy.dual` is deprecated. Instead of importing functions from `numpy.dual`, the functions should be imported directly from NumPy or SciPy.

(gh-16156)

outer and ufunc.outer deprecated for matrix

`np.matrix` use with `outer` or generic `ufunc` `outer` calls such as `numpy.add.outer`. Previously, `matrix` was converted to an array here. This will not be done in the future requiring a manual conversion to arrays.

(gh-16232)

Further Numeric Style types Deprecated

The remaining numeric-style type codes `Bytes0`, `Str0`, `Uint32`, `Uint64`, and `Datetime64` have been deprecated. The lower-case variants should be used instead. For bytes and string `"S"` and `"U"` are further alternatives.

(gh-16554)

The `ndincr` method of `ndindex` is deprecated

The documentation has warned against using this function since NumPy 1.8. Use `next(it)` instead of `it.ndincr()`.

(gh-17233)

ArrayLike objects which do not define `__len__` and `__getitem__`

Objects which define one of the protocols `__array__`, `__array_interface__`, or `__array_struct__` but are not sequences (usually defined by having a `__len__` and `__getitem__`) will behave differently during array-coercion in the future.

When nested inside sequences, such as `np.array([array_like])`, these were handled as a single Python object rather than an array. In the future they will behave identically to:

```
np.array([np.array(array_like)])
```

This change should only have an effect if `np.array(array_like)` is not 0-D. The solution to this warning may depend on the object:

- Some array-likes may expect the new behaviour, and users can ignore the warning. The object can choose to expose the sequence protocol to opt-in to the new behaviour.
- For example, `shapely` will allow conversion to an array-like using `line.coords` rather than `np.asarray(line)`. Users may work around the warning, or use the new convention when it becomes available.

Unfortunately, using the new behaviour can only be achieved by calling `np.array(array_like)`.

If you wish to ensure that the old behaviour remains unchanged, please create an object array and then fill it explicitly, for example:

```
arr = np.empty(3, dtype=object)
arr[:] = [array_like1, array_like2, array_like3]
```

This will ensure NumPy knows to not enter the array-like and use it as a object instead.

(gh-17973)

15.17.3 Future Changes

Arrays cannot be using subarray dtypes

Array creation and casting using `np.array(arr, dtype)` and `arr.astype(dtype)` will use different logic when `dtype` is a subarray dtype such as `np.dtype("(2)i,")`.

For such a `dtype` the following behaviour is true:

```
res = np.array(arr, dtype)

res.dtype is not dtype
res.dtype is dtype.base
res.shape == arr.shape + dtype.shape
```

But `res` is filled using the logic:

```
res = np.empty(arr.shape + dtype.shape, dtype=dtype.base)
res[...] = arr
```

which uses incorrect broadcasting (and often leads to an error). In the future, this will instead cast each element individually, leading to the same result as:

```
res = np.array(arr, dtype=np.dtype(["f", dtype]))["f"]
```

Which can normally be used to opt-in to the new behaviour.

This change does not affect `np.array(list, dtype="(2)i,")` unless the `list` itself includes at least one array. In particular, the behaviour is unchanged for a list of tuples.

(gh-17596)

15.17.4 Expired deprecations

- The deprecation of numeric style type-codes `np.dtype("Complex64")` (with upper case spelling), is expired. "Complex64" corresponded to "complex128" and "Complex32" corresponded to "complex64".
- The deprecation of `np.sctypeNA` and `np.typeNA` is expired. Both have been removed from the public API. Use `np.typeDict` instead.

(gh-16554)

- The 14-year deprecation of `np.ctypeslib.ctypes_load_library` is expired. Use `load_library` instead, which is identical.

(gh-17116)

Financial functions removed

In accordance with NEP 32, the financial functions are removed from NumPy 1.20. The functions that have been removed are `fv`, `ipmt`, `irr`, `mirr`, `nper`, `npv`, `pmt`, `ppmt`, `pvt`, and `rate`. These functions are available in the `numpy_financial` library.

(gh-17067)

15.17.5 Compatibility notes

`isinstance(dtype, np.dtype)` and `not type(dtype) is np.dtype`

NumPy dtypes are not direct instances of `np.dtype` anymore. Code that may have used `type(dtype) is np.dtype` will always return `False` and must be updated to use the correct version `isinstance(dtype, np.dtype)`.

This change also affects the C-side macro `PyArray_DescrCheck` if compiled against a NumPy older than 1.16.6. If code uses this macro and wishes to compile against an older version of NumPy, it must replace the macro (see also [C API changes](#) section).

Same kind casting in concatenate with `axis=None`

When `concatenate` is called with `axis=None`, the flattened arrays were cast with `unsafe`. Any other axis choice uses “same kind”. That different default has been deprecated and “same kind” casting will be used instead. The new casting keyword argument can be used to retain the old behaviour.

(gh-16134)

NumPy Scalars are cast when assigned to arrays

When creating or assigning to arrays, in all relevant cases NumPy scalars will now be cast identically to NumPy arrays. In particular this changes the behaviour in some cases which previously raised an error:

```
np.array([np.float64(np.nan)], dtype=np.int64)
```

will succeed and return an undefined result (usually the smallest possible integer). This also affects assignments:

```
arr[0] = np.float64(np.nan)
```

At this time, NumPy retains the behaviour for:

```
np.array(np.float64(np.nan), dtype=np.int64)
```

The above changes do not affect Python scalars:

```
np.array([float("NaN")], dtype=np.int64)
```

remains unaffected (`np.nan` is a Python `float`, not a NumPy one). Unlike signed integers, unsigned integers do not retain this special case, since they always behaved more like casting. The following code stops raising an error:

```
np.array([np.float64(np.nan)], dtype=np.uint64)
```

To avoid backward compatibility issues, at this time assignment from `datetime64` scalar to strings of too short length remains supported. This means that `np.asarray(np.datetime64("2020-10-10"), dtype="S5")` succeeds now, when it failed before. In the long term this may be deprecated or the unsafe cast may be allowed generally to make assignment of arrays and scalars behave consistently.

Array coercion changes when Strings and other types are mixed

When strings and other types are mixed, such as:

```
np.array(["string", np.float64(3.)], dtype="S")
```

The results will change, which may lead to string dtypes with longer strings in some cases. In particular, if `dtype="S"` is not provided any numerical value will lead to a string results long enough to hold all possible numerical values. (e.g. "S32" for floats). Note that you should always provide `dtype="S"` when converting non-strings to strings.

If `dtype="S"` is provided the results will be largely identical to before, but NumPy scalars (not a Python float like `1.0`), will still enforce a uniform string length:

```
np.array([np.float64(3.)], dtype="S") # gives "S32"
np.array([3.0], dtype="S") # gives "S3"
```

Previously the first version gave the same result as the second.

Array coercion restructure

Array coercion has been restructured. In general, this should not affect users. In extremely rare corner cases where array-likes are nested:

```
np.array([array_like1])
```

Things will now be more consistent with:

```
np.array([np.array(array_like1)])
```

This can subtly change output for some badly defined array-likes. One example for this are array-like objects which are not also sequences of matching shape. In NumPy 1.20, a warning will be given when an array-like is not also a sequence (but behaviour remains identical, see deprecations). If an array like is also a sequence (defines `__getitem__` and `__len__`) NumPy will now only use the result given by `__array__`, `__array_interface__`, or `__array_struct__`. This will result in differences when the (nested) sequence describes a different shape.

(gh-16200)

Writing to the result of `numpy.broadcast_arrays` will export readonly buffers

In NumPy 1.17 `numpy.broadcast_arrays` started warning when the resulting array was written to. This warning was skipped when the array was used through the buffer interface (e.g. `memoryview(arr)`). The same thing will now occur for the two protocols `__array_interface__`, and `__array_struct__` returning read-only buffers instead of giving a warning.

(gh-16350)

Numeric-style type names have been removed from type dictionaries

To stay in sync with the deprecation for `np.dtype("Complex64")` and other numeric-style (capital case) types. These were removed from `np.sctypeDict` and `np.typeDict`. You should use the lower case versions instead. Note that "Complex64" corresponds to "complex128" and "Complex32" corresponds to "complex64". The numpy style (new) versions, denote the full size and not the size of the real/imaginary part.

(gh-16554)

The `operator.concat` function now raises `TypeError` for array arguments

The previous behavior was to fall back to addition and add the two arrays, which was thought to be unexpected behavior for a concatenation function.

(gh-16570)

`nickname` attribute removed from `ABCPolyBase`

An abstract property `nickname` has been removed from `ABCPolyBase` as it was no longer used in the derived convenience classes. This may affect users who have derived classes from `ABCPolyBase` and overridden the methods for representation and display, e.g. `__str__`, `__repr__`, `_repr_latex`, etc.

(gh-16589)

`float->timedelta` and `uint64->timedelta` promotion will raise a `TypeError`

Float and timedelta promotion consistently raises a `TypeError`. `np.promote_types("float32", "m8")` aligns with `np.promote_types("m8", "float32")` now and both raise a `TypeError`. Previously, `np.promote_types("float32", "m8")` returned "m8" which was considered a bug.

Uint64 and timedelta promotion consistently raises a `TypeError`. `np.promote_types("uint64", "m8")` aligns with `np.promote_types("m8", "uint64")` now and both raise a `TypeError`. Previously, `np.promote_types("uint64", "m8")` returned "m8" which was considered a bug.

(gh-16592)

`numpy.genfromtxt` now correctly unpacks structured arrays

Previously, `numpy.genfromtxt` failed to unpack if it was called with `unpack=True` and a structured datatype was passed to the `dtype` argument (or `dtype=None` was passed and a structured datatype was inferred). For example:

```
>>> data = StringIO("21 58.0\n35 72.0")
>>> np.genfromtxt(data, dtype=None, unpack=True)
array([(21, 58.), (35, 72.)], dtype=[('f0', '<i8'), ('f1', '<f8')])
```

Structured arrays will now correctly unpack into a list of arrays, one for each column:

```
>>> np.genfromtxt(data, dtype=None, unpack=True)
[array([21, 35]), array([58., 72.])]
```

(gh-16650)

mgrid, r_, etc. consistently return correct outputs for non-default precision input

Previously, `np.mgrid[np.float32(0.1):np.float32(0.35):np.float32(0.1),]` and `np.r_[0:10:np.complex64(3j)]` failed to return meaningful output. This bug potentially affects `mgrid`, `ogrid`, `r_`, and `c_` when an input with dtype other than the default `float64` and `complex128` and equivalent Python types were used. The methods have been fixed to handle varying precision correctly.

(gh-16815)

Boolean array indices with mismatching shapes now properly give IndexError

Previously, if a boolean array index matched the size of the indexed array but not the shape, it was incorrectly allowed in some cases. In other cases, it gave an error, but the error was incorrectly a `ValueError` with a message about broadcasting instead of the correct `IndexError`.

For example, the following used to incorrectly give `ValueError: operands could not be broadcast together with shapes (2,2) (1,4)`:

```
np.empty((2, 2))[np.array([[True, False, False, False]])]
```

And the following used to incorrectly return `array([], dtype=float64)`:

```
np.empty((2, 2))[np.array([[False, False, False, False]])]
```

Both now correctly give `IndexError: boolean index did not match indexed array along dimension 0; dimension is 2 but corresponding boolean dimension is 1`.

(gh-17010)

Casting errors interrupt iteration

When iterating while casting values, an error may stop the iteration earlier than before. In any case, a failed casting operation always returned undefined, partial results. Those may now be even more undefined and partial. For users of the `NpyIter` C-API such cast errors will now cause the `iternext()` function to return 0 and thus abort iteration. Currently, there is no API to detect such an error directly. It is necessary to check `PyErr_Occurred()`, which may be problematic in combination with `NpyIter_Reset`. These issues always existed, but new API could be added if required by users.

(gh-17029)

f2py generated code may return unicode instead of byte strings

Some byte strings previously returned by `f2py` generated code may now be unicode strings. This results from the ongoing Python2 -> Python3 cleanup.

(gh-17068)

The first element of the `__array_interface__["data"]` tuple must be an integer

This has been the documented interface for many years, but there was still code that would accept a byte string representation of the pointer address. That code has been removed, passing the address as a byte string will now raise an error.

(gh-17241)

`poly1d` respects the dtype of all-zero argument

Previously, constructing an instance of `poly1d` with all-zero coefficients would cast the coefficients to `np.float64`. This affected the output dtype of methods which construct `poly1d` instances internally, such as `np.polymul`.

(gh-17577)

The `numpy.i` file for swig is Python 3 only.

Uses of Python 2.7 C-API functions have been updated to Python 3 only. Users who need the old version should take it from an older version of NumPy.

(gh-17580)

Void dtype discovery in `np.array`

In calls using `np.array(..., dtype="V")`, `arr.astype("V")`, and similar a `TypeError` will now be correctly raised unless all elements have the identical void length. An example for this is:

```
np.array([b"1", b"12"], dtype="V")
```

Which previously returned an array with dtype `"V2"` which cannot represent `b"1"` faithfully.

(gh-17706)

15.17.6 C API changes

The `PyArray_DescrCheck` macro is modified

The `PyArray_DescrCheck` macro has been updated since NumPy 1.16.6 to be:

```
#define PyArray_DescrCheck(op) PyObject_TypeCheck(op, &PyArrayDescr_Type)
```

Starting with NumPy 1.20 code that is compiled against an earlier version will be API incompatible with NumPy 1.20. The fix is to either compile against 1.16.6 (if the NumPy 1.16 release is the oldest release you wish to support), or manually inline the macro by replacing it with the new definition:

```
PyObject_TypeCheck(op, &PyArrayDescr_Type)
```

which is compatible with all NumPy versions.

Size of `np.ndarray` and `np.void_` changed

The size of the `PyArrayObject` and `PyVoidScalarObject` structures have changed. The following header definition has been removed:

```
#define NPY_SIZEOF_PYARRAYOBJECT (sizeof(PyArrayObject_fields))
```

since the size must not be considered a compile time constant: it will change for different runtime versions of NumPy.

The most likely relevant use are potential subclasses written in C which will have to be recompiled and should be updated. Please see the documentation for `PyArrayObject` for more details and contact the NumPy developers if you are affected by this change.

NumPy will attempt to give a graceful error but a program expecting a fixed structure size may have undefined behaviour and likely crash.

(gh-16938)

15.17.7 New Features

where keyword argument for `numpy.all` and `numpy.any` functions

The keyword argument `where` is added and allows to only consider specified elements or subaxes from an array in the Boolean evaluation of `all` and `any`. This new keyword is available to the functions `all` and `any` both via `numpy` directly or in the methods of `numpy.ndarray`.

Any broadcastable Boolean array or a scalar can be set as `where`. It defaults to `True` to evaluate the functions for all elements in an array if `where` is not set by the user. Examples are given in the documentation of the functions.

where keyword argument for `numpy` functions `mean`, `std`, `var`

The keyword argument `where` is added and allows to limit the scope in the calculation of `mean`, `std` and `var` to only a subset of elements. It is available both via `numpy` directly or in the methods of `numpy.ndarray`.

Any broadcastable Boolean array or a scalar can be set as `where`. It defaults to `True` to evaluate the functions for all elements in an array if `where` is not set by the user. Examples are given in the documentation of the functions.

(gh-15852)

norm=backward, forward keyword options for `numpy.fft` functions

The keyword argument option `norm=backward` is added as an alias for `None` and acts as the default option; using it has the direct transforms unscaled and the inverse transforms scaled by $1/n$.

Using the new keyword argument option `norm=forward` has the direct transforms scaled by $1/n$ and the inverse transforms unscaled (i.e. exactly opposite to the default option `norm=backward`).

(gh-16476)

NumPy is now typed

Type annotations have been added for large parts of NumPy. There is also a new `numpy.typing` module that contains useful types for end-users. The currently available types are

- `ArrayLike`: for objects that can be coerced to an array
- `DtypeLike`: for objects that can be coerced to a dtype

(gh-16515)

`numpy.typing` is accessible at runtime

The types in `numpy.typing` can now be imported at runtime. Code like the following will now work:

```
from numpy.typing import ArrayLike
x: ArrayLike = [1, 2, 3, 4]
```

(gh-16558)

New `__f2py_numpy_version__` attribute for f2py generated modules.

Because f2py is released together with NumPy, `__f2py_numpy_version__` provides a way to track the version of f2py used to generate the module.

(gh-16594)

mypy tests can be run via `runtests.py`

Currently running mypy with the NumPy stubs configured requires either:

- Installing NumPy
- Adding the source directory to `MYPYPATH` and linking to the `mypy.ini`

Both options are somewhat inconvenient, so add a `--mypy` option to `runtests` that handles setting things up for you. This will also be useful in the future for any typing codegen since it will ensure the project is built before type checking.

(gh-17123)

Negation of user defined BLAS/LAPACK detection order

`distutils` allows negation of libraries when determining BLAS/LAPACK libraries. This may be used to remove an item from the library resolution phase, i.e. to disallow NetLIB libraries one could do:

```
NPY_BLAS_ORDER='^blas' NPY_LAPACK_ORDER='^lapack' python setup.py build
```

That will use any of the accelerated libraries instead.

(gh-17219)

Allow passing optimizations arguments to asv build

It is now possible to pass `-j`, `--cpu-baseline`, `--cpu-dispatch` and `--disable-optimization` flags to ASV build when the `--bench-compare` argument is used.

(gh-17284)

The NVIDIA HPC SDK nvfortran compiler is now supported

Support for the nvfortran compiler, a version of pgfortran, has been added.

(gh-17344)

dtype option for cov and corrcoef

The `dtype` option is now available for `numpy.cov` and `numpy.corrcoef`. It specifies which data-type the returned result should have. By default the functions still return a `numpy.float64` result.

(gh-17456)

15.17.8 Improvements

Improved string representation for polynomials (`__str__`)

The string representation (`__str__`) of all six polynomial types in `numpy.polynomial` has been updated to give the polynomial as a mathematical expression instead of an array of coefficients. Two package-wide formats for the polynomial expressions are available - one using Unicode characters for superscripts and subscripts, and another using only ASCII characters.

(gh-15666)

Remove the Accelerate library as a candidate LAPACK library

Apple no longer supports Accelerate. Remove it.

(gh-15759)

Object arrays containing multi-line objects have a more readable repr

If elements of an object array have a `repr` containing new lines, then the wrapped lines will be aligned by column. Notably, this improves the `repr` of nested arrays:

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

(gh-15997)

Concatenate supports providing an output dtype

Support was added to `concatenate` to provide an output `dtype` and `casting` using keyword arguments. The `dtype` argument cannot be provided in conjunction with the `out` one.

(gh-16134)

Thread safe f2py callback functions

Callback functions in `f2py` are now thread safe.

(gh-16519)

`numpy.core.records.fromfile` now supports file-like objects

`numpy.rec.fromfile` can now use file-like objects, for instance `io.BytesIO`

(gh-16675)

RPATH support on AIX added to `distutils`

This allows SciPy to be built on AIX.

(gh-16710)

Use `f90` compiler specified by the command line args

The compiler command selection for Fortran Portland Group Compiler is changed in `numpy.distutils.fcompiler`. This only affects the linking command. This forces the use of the executable provided by the command line option (if provided) instead of the `pgfortran` executable. If no executable is provided to the command line option it defaults to the `pgf90` executable, which is an alias for `pgfortran` according to the PGI documentation.

(gh-16730)

Add NumPy declarations for Cython 3.0 and later

The `pxd` declarations for Cython 3.0 were improved to avoid using deprecated NumPy C-API features. Extension modules built with Cython 3.0+ that use NumPy can now set the C macro `NPY_NO_DEPRECATED_API=NPY_1_7_API_VERSION` to avoid C compiler warnings about deprecated API usage.

(gh-16986)

Make the window functions exactly symmetric

Make sure the window functions provided by NumPy are symmetric. There were previously small deviations from symmetry due to numerical precision that are now avoided by better arrangement of the computation.

(gh-17195)

15.17.9 Performance improvements and changes

Enable multi-platform SIMD compiler optimizations

A series of improvements for NumPy infrastructure to pave the way to **NEP-38**, that can be summarized as follow:

- **New Build Arguments**

- `--cpu-baseline` to specify the minimal set of required optimizations, default value is `min` which provides the minimum CPU features that can safely run on a wide range of users platforms.
- `--cpu-dispatch` to specify the dispatched set of additional optimizations, default value is `max -xop -fma4` which enables all CPU features, except for AMD legacy features.
- `--disable-optimization` to explicitly disable the whole new improvements, It also adds a new C compiler #definition called `NPY_DISABLE_OPTIMIZATION` which it can be used as guard for any SIMD code.

- **Advanced CPU dispatcher**

A flexible cross-architecture CPU dispatcher built on the top of Python/NumPy distutils, support all common compilers with a wide range of CPU features.

The new dispatcher requires a special file extension `*.dispatch.c` to mark the dispatch-able C sources. These sources have the ability to be compiled multiple times so that each compilation process represents certain CPU features and provides different #definitions and flags that affect the code paths.

- **New auto-generated C header “core/src/common/_cpu_dispatch.h”**

This header is generated by the distutils module `ccompiler_opt`, and contains all the #definitions and headers of instruction sets, that had been configured through command arguments ‘`-cpu-baseline`’ and ‘`-cpu-dispatch`’.

- **New C header “core/src/common/npv_cpu_dispatch.h”**

This header contains all utilities that required for the whole CPU dispatching process, it also can be considered as a bridge linking the new infrastructure work with NumPy CPU runtime detection.

- **Add new attributes to NumPy umath module(Python level)**

- `__cpu_baseline__` a list contains the minimal set of required optimizations that supported by the compiler and platform according to the specified values to command argument ‘`-cpu-baseline`’.
- `__cpu_dispatch__` a list contains the dispatched set of additional optimizations that supported by the compiler and platform according to the specified values to command argument ‘`-cpu-dispatch`’.

- **Print the supported CPU features during the run of PytestTester**

(gh-13516)

15.17.10 Changes

Changed behavior of `divmod(1., 0.)` and related functions

The changes also assure that different compiler versions have the same behavior for nan or inf usages in these operations. This was previously compiler dependent, we now force the invalid and divide by zero flags, making the results the same across compilers. For example, gcc-5, gcc-8, or gcc-9 now result in the same behavior. The changes are tabulated below:

Table 1: Summary of New Behavior

Operator	Old Warning	New Warning	Old Result	New Result	Works on MacOS
<code>np.divmod(1.0, 0.0)</code>	Invalid	Invalid and Divide-byzero	nan, nan	inf, nan	Yes
<code>np.fmod(1.0, 0.0)</code>	Invalid	Invalid	nan	nan	No? Yes
<code>np.floor_divide(1.0, 0.0)</code>	Invalid	Dividebyzero	nan	inf	Yes
<code>np.remainder(1.0, 0.0)</code>	Invalid	Invalid	nan	nan	Yes

(gh-16161)

`np.linspace` on integers now uses floor

When using a `int` dtype in `numpy.linspace`, previously float values would be rounded towards zero. Now `numpy.floor` is used instead, which rounds toward $-\infty$. This changes the results for negative values. For example, the following would previously give:

```
>>> np.linspace(-3, 1, 8, dtype=int)
array([-3, -2, -1, -1,  0,  0,  0,  1])
```

and now results in:

```
>>> np.linspace(-3, 1, 8, dtype=int)
array([-3, -3, -2, -2, -1, -1,  0,  1])
```

The former result can still be obtained with:

```
>>> np.linspace(-3, 1, 8).astype(int)
array([-3, -2, -1, -1,  0,  0,  0,  1])
```

(gh-16841)

15.18 NumPy 1.19.5 Release Notes

NumPy 1.19.5 is a short bugfix release. Apart from fixing several bugs, the main improvement is the update to OpenBLAS 0.3.13 that works around the windows 2004 bug while not breaking execution on other platforms. This release supports Python 3.6-3.9 and is planned to be the last release in the 1.19.x cycle.

15.18.1 Contributors

A total of 8 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Christoph Gohlke
- Matti Pícus
- Raghuveer Devulapalli
- Sebastian Berg
- Simon Graham +
- Veniamin Petrenko +
- Bernie Gray +

15.18.2 Pull requests merged

A total of 11 pull requests were merged for this release.

- [#17756](#): BUG: Fix segfault due to out of bound pointer in floatstatus...
- [#17774](#): BUG: fix np.timedelta64('nat').__format__ throwing an exception
- [#17775](#): BUG: Fixed file handle leak in array_tofile.
- [#17786](#): BUG: Raise recursion error during dimension discovery
- [#17917](#): BUG: Fix subarray dtype used with too large count in fromfile
- [#17918](#): BUG: 'bool' object has no attribute 'ndim'
- [#17919](#): BUG: ensure _UFuncNoLoopError can be pickled
- [#17924](#): BLD: use BUFFERSIZE=20 in OpenBLAS
- [#18026](#): BLD: update to OpenBLAS 0.3.13
- [#18036](#): BUG: make a variable volatile to work around clang compiler bug
- [#18114](#): REL: Prepare for the NumPy 1.19.5 release.

15.19 NumPy 1.19.4 Release Notes

NumPy 1.19.4 is a quick release to revert the OpenBLAS library version. It was hoped that the 0.3.12 OpenBLAS version used in 1.19.3 would work around the Microsoft fmod bug, but problems in some docker environments turned up. Instead, 1.19.4 will use the older library and run a sanity check on import, raising an error if the problem is detected. Microsoft is aware of the problem and has promised a fix, users should upgrade when it becomes available.

This release supports Python 3.6-3.9

15.19.1 Contributors

A total of 1 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris

15.19.2 Pull requests merged

A total of 2 pull requests were merged for this release.

- [#17679](#): MAINT: Add check for Windows 10 version 2004 bug.
- [#17680](#): REV: Revert OpenBLAS to 1.19.2 version for 1.19.4

15.20 NumPy 1.19.3 Release Notes

NumPy 1.19.3 is a small maintenance release with two major improvements:

- Python 3.9 binary wheels on all supported platforms.
- OpenBLAS fixes for Windows 10 version 2004 fmod bug.

This release supports Python 3.6-3.9 and is linked with OpenBLAS 0.3.12 to avoid some of the fmod problems on Windows 10 version 2004. Microsoft is aware of the problem and users should upgrade when the fix becomes available, the fix here is limited in scope.

15.20.1 Contributors

A total of 8 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Chris Brown +
- Daniel Vanzo +
- E. Madison Bray +
- Hugo van Kemenade +
- Ralf Gommers
- Sebastian Berg
- @danbeibei +

15.20.2 Pull requests merged

A total of 10 pull requests were merged for this release.

- [#17298](#): BLD: set upper versions for build dependencies
- [#17336](#): BUG: Set deprecated fields to null in PyArray_InitArrFuncs
- [#17446](#): ENH: Warn on unsupported Python 3.10+
- [#17450](#): MAINT: Update test_requirements.txt.
- [#17522](#): ENH: Support for the NVIDIA HPC SDK nvfortran compiler

- [#17568](#): BUG: Cygwin Workaround for [#14787](#) on affected platforms
- [#17647](#): BUG: Fix memory leak of buffer-info cache due to relaxed strides
- [#17652](#): MAINT: Backport openblas_support from master.
- [#17653](#): TST: Add Python 3.9 to the CI testing on Windows, Mac.
- [#17660](#): TST: Simplify source path names in test_extending.

15.21 NumPy 1.19.2 Release Notes

NumPy 1.19.2 fixes several bugs, prepares for the upcoming Cython 3.x release. and pins setuptools to keep distutils working while upstream modifications are ongoing. The aarch64 wheels are built with the latest manylinux2014 release that fixes the problem of differing page sizes used by different linux distros.

This release supports Python 3.6-3.8. Cython $\geq 0.29.21$ needs to be used when building with Python 3.9 for testing purposes.

There is a known problem with Windows 10 version=2004 and OpenBLAS svd that we are trying to debug. If you are running that Windows version you should use a NumPy version that links to the MKL library, earlier Windows versions are fine.

15.21.1 Improvements

Add NumPy declarations for Cython 3.0 and later

The pxd declarations for Cython 3.0 were improved to avoid using deprecated NumPy C-API features. Extension modules built with Cython 3.0+ that use NumPy can now set the C macro `NPY_NO_DEPRECATED_API=NPY_1_7_API_VERSION` to avoid C compiler warnings about deprecated API usage.

15.21.2 Contributors

A total of 8 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Matti Picus
- Pauli Virtanen
- Philippe Ombredanne +
- Sebastian Berg
- Stefan Behnel +
- Stephan Loyd +
- Zac Hatfield-Dodds

15.21.3 Pull requests merged

A total of 9 pull requests were merged for this release.

- [#16959](#): TST: Change aarch64 to arm64 in travis.yml.
- [#16998](#): MAINT: Configure hypothesis in `np.test()` for determinism,...
- [#17000](#): BLD: pin setuptools < 49.2.0
- [#17015](#): ENH: Add NumPy declarations to be used by Cython 3.0+
- [#17125](#): BUG: Remove non-threadsafe sigint handling from fft calculation
- [#17243](#): BUG: core: fix ilp64 blas dot/vdot/... for strides > int32 max
- [#17244](#): DOC: Use SPDX license expressions with correct license
- [#17245](#): DOC: Fix the link to the quick-start in the old API functions
- [#17272](#): BUG: fix pickling of arrays larger than 2GiB

15.22 NumPy 1.19.1 Release Notes

NumPy 1.19.1 fixes several bugs found in the 1.19.0 release, replaces several functions deprecated in the upcoming Python-3.9 release, has improved support for AIX, and has a number of development related updates to keep CI working with recent upstream changes.

This release supports Python 3.6-3.8. Cython `>= 0.29.21` needs to be used when building with Python 3.9 for testing purposes.

15.22.1 Contributors

A total of 15 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Abhinav Reddy +
- Anirudh Subramanian
- Antonio Larrosa +
- Charles Harris
- Chunlin Fang
- Eric Wieser
- Etienne Guesnet +
- Kevin Sheppard
- Matti Pícus
- Raghuveer Devulapalli
- Roman Yurchak
- Ross Barnowski
- Sayed Adel
- Sebastian Berg
- Tyler Reddy

15.22.2 Pull requests merged

A total of 25 pull requests were merged for this release.

- [#16649](#): MAINT, CI: disable Shippable cache
- [#16652](#): MAINT: Replace `PyUString_GET_SIZE` with `PyUnicode_GetLength`.
- [#16654](#): REL: Fix outdated docs link
- [#16656](#): BUG: raise IEEE exception on AIX
- [#16672](#): BUG: Fix bug in AVX complex absolute while processing array of...
- [#16693](#): TST: Add extra debugging information to CPU features detection
- [#16703](#): BLD: Add CPU entry for Emscripten / WebAssembly
- [#16705](#): TST: Disable Python 3.9-dev testing.
- [#16714](#): MAINT: Disable use_hugepages in case of ValueError
- [#16724](#): BUG: Fix `PyArray_SearchSorted` signature.
- [#16768](#): MAINT: Fixes for deprecated functions in `scalartypes.c.src`
- [#16772](#): MAINT: Remove unneeded call to `PyUnicode_READY`
- [#16776](#): MAINT: Fix deprecated functions in `scalarmapi.c`
- [#16779](#): BLD, ENH: Add RPATH support for AIX
- [#16780](#): BUG: Fix default fallback in `genfromtxt`
- [#16784](#): BUG: Added missing return after raising error in `methods.c`
- [#16795](#): BLD: update cython to 0.29.21
- [#16832](#): MAINT: setuptools 49.2.0 emits a warning, avoid it
- [#16872](#): BUG: Validate output size in `bin-` and `multinomial`
- [#16875](#): BLD, MAINT: Pin setuptools
- [#16904](#): DOC: Reconstruct Testing Guideline.
- [#16905](#): TST, BUG: Re-raise `MemoryError` exception in `test_large_zip`'s...
- [#16906](#): BUG,DOC: Fix bad MPL kwarg.
- [#16916](#): BUG: Fix string/bytes to complex assignment
- [#16922](#): REL: Prepare for NumPy 1.19.1 release

15.23 NumPy 1.19.0 Release Notes

This NumPy release is marked by the removal of much technical debt: support for Python 2 has been removed, many deprecations have been expired, and documentation has been improved. The polishing of the random module continues apace with bug fixes and better usability from Cython.

The Python versions supported for this release are 3.6-3.8. Downstream developers should use Cython $\geq 0.29.16$ for Python 3.8 support and OpenBLAS ≥ 3.7 to avoid problems on the Skylake architecture.

15.23.1 Highlights

- Code compatibility with Python versions < 3.6 (including Python 2) was dropped from both the python and C code. The shims in `numpy.compat` will remain to support third-party packages, but they may be deprecated in a future release. Note that 1.19.x will *not* compile with earlier versions of Python due to the use of f-strings.

(gh-15233)

15.23.2 Expired deprecations

`numpy.insert` and `numpy.delete` can no longer be passed an axis on 0d arrays

This concludes a deprecation from 1.9, where when an `axis` argument was passed to a call to `~numpy.insert` and `~numpy.delete` on a 0d array, the `axis` and `obj` argument and indices would be completely ignored. In these cases, `insert(arr, "nonsense", 42, axis=0)` would actually overwrite the entire array, while `delete(arr, "nonsense", axis=0)` would be `arr.copy()`

Now passing `axis` on a 0d array raises `~numpy.AxisError`.

(gh-15802)

`numpy.delete` no longer ignores out-of-bounds indices

This concludes deprecations from 1.8 and 1.9, where `np.delete` would ignore both negative and out-of-bounds items in a sequence of indices. This was at odds with its behavior when passed a single index.

Now out-of-bounds items throw `IndexError`, and negative items index from the end.

(gh-15804)

`numpy.insert` and `numpy.delete` no longer accept non-integral indices

This concludes a deprecation from 1.9, where sequences of non-integers indices were allowed and cast to integers. Now passing sequences of non-integral indices raises `IndexError`, just like it does when passing a single non-integral scalar.

(gh-15805)

`numpy.delete` no longer casts boolean indices to integers

This concludes a deprecation from 1.8, where `np.delete` would cast boolean arrays and scalars passed as an index argument into integer indices. The behavior now is to treat boolean arrays as a mask, and to raise an error on boolean scalars.

(gh-15815)

15.23.3 Compatibility notes

Changed random variate stream from `numpy.random.Generator.dirichlet`

A bug in the generation of random variates for the Dirichlet distribution with small ‘alpha’ values was fixed by using a different algorithm when `max(alpha) < 0.1`. Because of the change, the stream of variates generated by `dirichlet` in this case will be different from previous releases.

(gh-14924)

Scalar promotion in `PyArray_ConvertToCommonType`

The promotion of mixed scalars and arrays in `PyArray_ConvertToCommonType` has been changed to adhere to those used by `np.result_type`. This means that input such as `(1000, np.array([1], dtype=np.uint8))` will now return `uint16` dtypes. In most cases the behaviour is unchanged. Note that the use of this C-API function is generally discouraged. This also fixes `np.choose` to behave the same way as the rest of NumPy in this respect.

(gh-14933)

Fasttake and fastputmask slots are deprecated and NULL’ed

The `fasttake` and `fastputmask` slots are now never used and must always be set to `NULL`. This will result in no change in behaviour. However, if a user dtype should set one of these a `DeprecationWarning` will be given.

(gh-14942)

`np.ediff1d` casting behaviour with `to_end` and `to_begin`

`np.ediff1d` now uses the “`same_kind`” casting rule for its additional `to_end` and `to_begin` arguments. This ensures type safety except when the input array has a smaller integer type than `to_begin` or `to_end`. In rare cases, the behaviour will be more strict than it was previously in 1.16 and 1.17. This is necessary to solve issues with floating point NaN.

(gh-14981)

Converting of empty array-like objects to NumPy arrays

Objects with `len(obj) == 0` which implement an “array-like” interface, meaning an object implementing `obj.__array__()`, `obj.__array_interface__`, `obj.__array_struct__`, or the python buffer interface and which are also sequences (i.e. Pandas objects) will now always retain their shape correctly when converted to an array. If such an object has a shape of `(0, 1)` previously, it could be converted into an array of shape `(0,)` (losing all dimensions after the first 0).

(gh-14995)

Removed `multiarray.int_asbuffer`

As part of the continued removal of Python 2 compatibility, `multiarray.int_asbuffer` was removed. On Python 3, it threw a `NotImplementedError` and was unused internally. It is expected that there are no downstream use cases for this method with Python 3.

(gh-15229)

`numpy.distutils.compat` has been removed

This module contained only the function `get_exception()`, which was used as:

```
try:
    ...
except Exception:
    e = get_exception()
```

Its purpose was to handle the change in syntax introduced in Python 2.6, from `except Exception, e:` to `except Exception as e:`, meaning it was only necessary for codebases supporting Python 2.5 and older.

(gh-15255)

`issubdtype` no longer interprets `float` as `np.floating`

`numpy.issubdtype` had a `FutureWarning` since NumPy 1.14 which has expired now. This means that certain input where the second argument was neither a datatype nor a NumPy scalar type (such as a string or a python type like `int` or `float`) will now be consistent with passing in `np.dtype(arg2).type`. This makes the result consistent with expectations and leads to a false result in some cases which previously returned `true`.

(gh-15773)

Change output of `round` on scalars to be consistent with Python

Output of the `__round__` dunder method and consequently the Python built-in `round` has been changed to be a Python `int` to be consistent with calling it on Python `float` objects when called with no arguments. Previously, it would return a scalar of the `np.dtype` that was passed in.

(gh-15840)

The `numpy.ndarray` constructor no longer interprets `strides=()` as `strides=None`

The former has changed to have the expected meaning of setting `numpy.ndarray.strides` to `()`, while the latter continues to result in `strides` being chosen automatically.

(gh-15882)

C-Level string to datetime casts changed

The C-level casts from strings were simplified. This changed also fixes string to datetime and timedelta casts to behave correctly (i.e. like Python casts using `string_arr.astype("M8")`) while previously the cast would behave like `string_arr.astype(np.int_).astype("M8")`. This only affects code using low-level C-API to do manual casts (not full array casts) of single scalar values or using e.g. `PyArray_GetCastFunc`, and should thus not affect the vast majority of users.

(gh-16068)

SeedSequence with small seeds no longer conflicts with spawning

Small seeds (less than 2^{96}) were previously implicitly 0-padded out to 128 bits, the size of the internal entropy pool. When spawned, the spawn key was concatenated before the 0-padding. Since the first spawn key is $(0,)$, small seeds before the spawn created the same states as the first spawned `SeedSequence`. Now, the seed is explicitly 0-padded out to the internal pool size before concatenating the spawn key. Spawned `SeedSequences` will produce different results than in the previous release. Unspawned `SeedSequences` will still produce the same results.

(gh-16551)

15.23.4 Deprecations

Deprecate automatic dtype=object for ragged input

Calling `np.array([[1, [1, 2, 3]])` will issue a `DeprecationWarning` as per [NEP 34](#). Users should explicitly use `dtype=object` to avoid the warning.

(gh-15119)

Passing shape=0 to factory functions in `numpy.rec` is deprecated

0 is treated as a special case and is aliased to `None` in the functions:

- `numpy.core.records.fromarrays`
- `numpy.core.records.fromrecords`
- `numpy.core.records.fromstring`
- `numpy.core.records.fromfile`

In future, 0 will not be special cased, and will be treated as an array length like any other integer.

(gh-15217)

Deprecation of probably unused C-API functions

The following C-API functions are probably unused and have been deprecated:

- `PyArray_GetArrayParamsFromObject`
- `PyUFunc_GenericFunction`
- `PyUFunc_SetUsesArraysAsData`

In most cases `PyArray_GetArrayParamsFromObject` should be replaced by converting to an array, while `PyUFunc_GenericFunction` can be replaced with `PyObject_Call` (see documentation for details).

(gh-15427)

Converting certain types to dtypes is Deprecated

The super classes of scalar types, such as `np.integer`, `np.generic`, or `np.inexact` will now give a deprecation warning when converted to a dtype (or used in a dtype keyword argument). The reason for this is that `np.integer` is converted to `np.int_`, while it would be expected to represent *any* integer (e.g. also `int8`, `int16`, etc. For example, `dtype=np.floating` is currently identical to `dtype=np.float64`, even though also `np.float32` is a subclass of `np.floating`.

(gh-15534)

Deprecation of `round` for `np.complexfloating` scalars

Output of the `__round__` dunder method and consequently the Python built-in `round` has been deprecated on complex scalars. This does not affect `np.round`.

(gh-15840)

`numpy.ndarray.tostring()` is deprecated in favor of `tobytes()`

`~numpy.ndarray.tobytes` has existed since the 1.9 release, but until this release `~numpy.ndarray.tostring` emitted no warning. The change to emit a warning brings NumPy in line with the builtin `array.array` methods of the same name.

(gh-15867)

15.23.5 C API changes

Better support for `const` dimensions in API functions

The following functions now accept a constant array of `npy_intp`:

- `PyArray_BroadcastToShape`
- `PyArray_IntTupleFromIntp`
- `PyArray_OverflowMultiplyList`

Previously the caller would have to cast away the `const`-ness to call these functions.

(gh-15251)

Const qualify UFunc inner loops

UFuncGenericFunction now expects pointers to const dimension and strides as arguments. This means inner loops may no longer modify either dimension or strides. This change leads to an incompatible-pointer-types warning forcing users to either ignore the compiler warnings or to const qualify their own loop signatures.

(gh-15355)

15.23.6 New Features

`numpy.frompyfunc` now accepts an identity argument

This allows the `numpy.ufunc.identity` attribute to be set on the resulting ufunc, meaning it can be used for empty and multi-dimensional calls to `numpy.ufunc.reduce`.

(gh-8255)

`np.str_` scalars now support the buffer protocol

`np.str_` arrays are always stored as UCS4, so the corresponding scalars now expose this through the buffer interface, meaning `memoryview(np.str_('test'))` now works.

(gh-15385)

`subok` option for `numpy.copy`

A new kwarg, `subok`, was added to `numpy.copy` to allow users to toggle the behavior of `numpy.copy` with respect to array subclasses. The default value is `False` which is consistent with the behavior of `numpy.copy` for previous numpy versions. To create a copy that preserves an array subclass with `numpy.copy`, call `np.copy(arr, subok=True)`. This addition better documents that the default behavior of `numpy.copy` differs from the `numpy.ndarray.copy` method which respects array subclasses by default.

(gh-15685)

`numpy.linalg.multi_dot` now accepts an `out` argument

`out` can be used to avoid creating unnecessary copies of the final product computed by `numpy.linalg.multidot`.

(gh-15715)

`keepdims` parameter for `numpy.count_nonzero`

The parameter `keepdims` was added to `numpy.count_nonzero`. The parameter has the same meaning as it does in reduction functions such as `numpy.sum` or `numpy.mean`.

(gh-15870)

equal_nan parameter for `numpy.array_equal`

The keyword argument `equal_nan` was added to `numpy.array_equal`. `equal_nan` is a boolean value that toggles whether or not `nan` values are considered equal in comparison (default is `False`). This matches API used in related functions such as `numpy.isclose` and `numpy.allclose`.

(gh-16128)

15.23.7 Improvements

15.23.8 Improve detection of CPU features

Replace `npy_cpu_supports` which was a gcc specific mechanism to test support of AVX with more general functions `npy_cpu_init` and `npy_cpu_have`, and expose the results via a `NPY_CPU_HAVE` c-macro as well as a python-level `__cpu_features__` dictionary.

(gh-13421)

Use 64-bit integer size on 64-bit platforms in fallback `lapack_lite`

Use 64-bit integer size on 64-bit platforms in the fallback LAPACK library, which is used when the system has no LAPACK installed, allowing it to deal with linear algebra for large arrays.

(gh-15218)

Use AVX512 intrinsic to implement `np.exp` when input is `np.float64`

Use AVX512 intrinsic to implement `np.exp` when input is `np.float64`, which can improve the performance of `np.exp` with `np.float64` input 5-7x faster than before. The `_multiarray_umath.so` module has grown about 63 KB on linux64.

(gh-15648)

Ability to disable `madvise` hugepages

On Linux NumPy has previously added support for `madvise` hugepages which can improve performance for very large arrays. Unfortunately, on older Kernel versions this led to performance regressions, thus by default the support has been disabled on kernels before version 4.6. To override the default, you can use the environment variable:

`NUMPY_MADVISE_HUGEPAGE=0`

or set it to 1 to force enabling support. Note that this only makes a difference if the operating system is set up to use `madvise` transparent hugepage.

(gh-15769)

numpy.einsum accepts NumPy int64 type in subscript list

There is no longer a type error thrown when `numpy.einsum` is passed a NumPy `int64` array as its subscript list.
(gh-16080)

np.logaddexp2.identity changed to -inf

The ufunc `~numpy.logaddexp2` now has an identity of `-inf`, allowing it to be called on empty sequences. This matches the identity of `~numpy.logaddexp`.
(gh-16102)

15.23.9 Changes

Remove handling of extra argument to `__array__`

A code path and test have been in the code since NumPy 0.4 for a two-argument variant of `__array__(dtype=None, context=None)`. It was activated when calling `ufunc(op)` or `ufunc.reduce(op)` if `op.__array__` existed. However that variant is not documented, and it is not clear what the intention was for its use. It has been removed.
(gh-15118)

numpy.random._bit_generator moved to numpy.random.bit_generator

In order to expose `numpy.random.BitGenerator` and `numpy.random.SeedSequence` to Cython, the `_bitgenerator` module is now public as `numpy.random.bit_generator`

Cython access to the random distributions is provided via a pxd file

`c_distributions.pxd` provides access to the c functions behind many of the random distributions from Cython, making it convenient to use and extend them.
(gh-15463)

Fixed `eigh` and `cholesky` methods in `numpy.random.multivariate_normal`

Previously, when passing `method='eigh'` or `method='cholesky'`, `numpy.random.multivariate_normal` produced samples from the wrong distribution. This is now fixed.
(gh-15872)

Fixed the jumping implementation in `MT19937.jumped`

This fix changes the stream produced from jumped MT19937 generators. It does not affect the stream produced using `RandomState` or `MT19937` that are directly seeded.

The translation of the jumping code for the MT19937 contained a reversed loop ordering. `MT19937.jumped` matches the Makoto Matsumoto's original implementation of the Horner and Sliding Window jump methods.

(gh-16153)

15.24 NumPy 1.18.5 Release Notes

This is a short release to allow pickle `protocol=5` to be used in Python3.5. It is motivated by the recent backport of pickle5 to Python3.5.

The Python versions supported in this release are 3.5-3.8. Downstream developers should use Cython `>= 0.29.15` for Python 3.8 support and OpenBLAS `>= 3.7` to avoid errors on the Skylake architecture.

15.24.1 Contributors

A total of 3 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Matti Picus
- Siyuan Zhuang +

15.24.2 Pull requests merged

A total of 2 pull requests were merged for this release.

- [#16439](#): ENH: enable pickle protocol 5 support for python3.5
- [#16441](#): BUG: relpath fails for different drives on windows

15.25 NumPy 1.18.4 Release Notes

This is the last planned release in the 1.18.x series. It reverts the `bool ("0")` behavior introduced in 1.18.3 and fixes a bug in `Generator.integers`. There is also a link to a new troubleshooting section in the documentation included in the error message emitted when `numpy` import fails.

The Python versions supported in this release are 3.5-3.8. Downstream developers should use Cython `>= 0.29.15` for Python 3.8 support and OpenBLAS `>= 3.7` to avoid errors on the Skylake architecture.

15.25.1 Contributors

A total of 4 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Matti Pícus
- Sebastian Berg
- Warren Weckesser

15.25.2 Pull requests merged

A total of 6 pull requests were merged for this release.

- [#16055](#): BLD: add i686 for 1.18 builds
- [#16090](#): BUG: `random.integers(2**32)` always returned 0.
- [#16091](#): BLD: fix path to libgfortran on macOS
- [#16109](#): REV: Reverts side-effect changes to casting
- [#16114](#): BLD: put openblas library in local directory on windows
- [#16132](#): DOC: Change import error “howto” to link to new troubleshooting...

15.26 NumPy 1.18.3 Release Notes

This release contains various bug/regression fixes.

The Python versions supported in this release are 3.5-3.8. Downstream developers should use Cython \geq 0.29.15 for Python 3.8 support and OpenBLAS \geq 3.7 to avoid errors on the Skylake architecture.

15.26.1 Highlights

- Fix for the `method='eigh'` and `method='cholesky'` methods in `numpy.random.multivariate_normal`. Those were producing samples from the wrong distribution.

15.26.2 Contributors

A total of 6 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Max Balandat +
- @Mibu287 +
- Pan Jan +
- Sebastian Berg
- @panpiort8 +

15.26.3 Pull requests merged

A total of 5 pull requests were merged for this release.

- [#15916](#): BUG: Fix eig and cholesky methods of `numpy.random.multivariate_normal`
- [#15929](#): BUG,MAINT: Remove incorrect special case in string to number...
- [#15930](#): BUG: Guarantee array is in valid state after memory error occurs...
- [#15954](#): BUG: Check that *pvals* is 1D in `_generator.multinomial`.
- [#16017](#): BUG: Alpha parameter must be 1D in `generator.dirichlet`

15.27 NumPy 1.18.2 Release Notes

This small release contains a fix for a performance regression in `numpy/random` and several bug/maintenance updates.

The Python versions supported in this release are 3.5-3.8. Downstream developers should use Cython $\geq 0.29.15$ for Python 3.8 support and OpenBLAS ≥ 3.7 to avoid errors on the Skylake architecture.

15.27.1 Contributors

A total of 5 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Ganesh Kathiresan +
- Matti Picus
- Sebastian Berg
- przemb +

15.27.2 Pull requests merged

A total of 7 pull requests were merged for this release.

- [#15675](#): TST: move `_no_tracing` to `testing._private`
- [#15676](#): MAINT: Large overhead in some random functions
- [#15677](#): TST: Do not create gfortran link in azure Mac testing.
- [#15679](#): BUG: Added missing error check in `ndarray.__contains__`
- [#15722](#): MAINT: use list-based APIs to call subprocesses
- [#15729](#): REL: Prepare for 1.18.2 release.
- [#15734](#): BUG: fix logic error when `nm` fails on 32-bit

15.28 NumPy 1.18.1 Release Notes

This release contains fixes for bugs reported against NumPy 1.18.0. Two bugs in particular that caused widespread problems downstream were:

- The cython random extension test was not using a temporary directory for building, resulting in a permission violation. Fixed.
- Numpy distutils was appending `-std=c99` to all C compiler runs, leading to changed behavior and compile problems downstream. That flag is now only applied when building numpy C code.

The Python versions supported in this release are 3.5-3.8. Downstream developers should use Cython $\geq 0.29.14$ for Python 3.8 support and OpenBLAS ≥ 3.7 to avoid errors on the Skylake architecture.

15.28.1 Contributors

A total of 7 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Matti Pícus
- Maxwell Aladago
- Pauli Virtanen
- Ralf Gommers
- Tyler Reddy
- Warren Weckesser

15.28.2 Pull requests merged

A total of 13 pull requests were merged for this release.

- [#15158](#): MAINT: Update pavement.py for towncrier.
- [#15159](#): DOC: add moved modules to 1.18 release note
- [#15161](#): MAINT, DOC: Minor backports and updates for 1.18.x
- [#15176](#): TST: Add `assert_array_equal` test for big integer arrays
- [#15184](#): BUG: use tmp dir and check version for cython test ([#15170](#))
- [#15220](#): BUG: distutils: fix msvc+gfortran openblas handling corner case
- [#15221](#): BUG: remove `-std=c99` for c++ compilation ([#15194](#))
- [#15222](#): MAINT: unskip test on win32
- [#15223](#): TST: add BLAS ILP64 run in Travis & Azure
- [#15245](#): MAINT: only add `-std=c99` where needed
- [#15246](#): BUG: lib: Fix handling of integer arrays by gradient.
- [#15247](#): MAINT: Do not use private Python function in testing
- [#15250](#): REL: Prepare for the NumPy 1.18.1 release.

15.29 NumPy 1.18.0 Release Notes

In addition to the usual bug fixes, this NumPy release cleans up and documents the new random C-API, expires a large number of old deprecations, and improves the appearance of the documentation. The Python versions supported are 3.5-3.8. This is the last NumPy release series that will support Python 3.5.

Downstream developers should use Cython $\geq 0.29.14$ for Python 3.8 support and OpenBLAS ≥ 3.7 to avoid problems on the Skylake architecture.

15.29.1 Highlights

- The C-API for `numpy.random` has been defined and documented.
- Basic infrastructure for linking with 64 bit BLAS and LAPACK libraries.
- Many documentation improvements.

15.29.2 New functions

Multivariate hypergeometric distribution added to `numpy.random`

The method `multivariate_hypergeometric` has been added to the class `numpy.random.Generator`. This method generates random variates from the multivariate hypergeometric probability distribution. ([gh-13794](#))

15.29.3 Deprecations

`np.fromfile` and `np.fromstring` will error on bad data

In future numpy releases, the functions `np.fromfile` and `np.fromstring` will throw an error when parsing bad data. This will now give a `DeprecationWarning` where previously partial or even invalid data was silently returned. This deprecation also affects the C defined functions `PyArray_FromString` and `PyArray_FromFile` ([gh-13605](#))

Deprecate non-scalar arrays as fill values in `ma.fill_value`

Setting a `MaskedArray.fill_value` to a non-scalar array is deprecated since the logic to broadcast the fill value to the array is fragile, especially when slicing. ([gh-13698](#))

Deprecate `PyArray_As1D`, `PyArray_As2D`

`PyArray_As1D`, `PyArray_As2D` are deprecated, use `PyArray_AsCArray` instead ([gh-14036](#))

Deprecate `np.alen`

`np.alen` was deprecated. Use `len` instead. ([gh-14181](#))

Deprecate the financial functions

In accordance with [NEP-32](#), the financial functions `fv`, `ipmt`, `irr`, `mirr`, `nper`, `npv`, `pmt`, `ppmt`, `pv` and `rate` are deprecated, and will be removed from NumPy 1.20. The replacement for these functions is the Python package `numpy-financial`. ([gh-14720](#))

The `axis` argument to `numpy.ma.mask_cols` and `numpy.ma.mask_row` is deprecated

This argument was always ignored. ([gh-14996](#))

15.29.4 Expired deprecations

- `PyArray_As1D` and `PyArray_As2D` have been removed in favor of `PyArray_AsCArray` ([gh-14036](#))
- `np.rank` has been removed. This was deprecated in NumPy 1.10 and has been replaced by `np.ndim`. ([gh-14039](#))
- The deprecation of `expand_dims` out-of-range axes in 1.13.0 has expired. ([gh-14051](#))
- `PyArray_FromDimsAndDataAndDescr` and `PyArray_FromDims` have been removed (they will always raise an error). Use `PyArray_NewFromDescr` and `PyArray_SimpleNew` instead. ([gh-14100](#))
- `numeric.loads`, `numeric.load`, `np.ma.dump`, `np.ma.dumps`, `np.ma.load`, `np.ma.loads` are removed, use pickle methods instead ([gh-14256](#))
- `arrayprint.FloatFormat`, `arrayprint.LongFloatFormat` has been removed, use `FloatingFormat` instead
- `arrayprint.ComplexFormat`, `arrayprint.LongComplexFormat` has been removed, use `ComplexFloatingFormat` instead
- `arrayprint.StructureFormat` has been removed, use `StructureVoidFormat` instead ([gh-14259](#))
- `np.testing.rand` has been removed. This was deprecated in NumPy 1.11 and has been replaced by `np.random.rand`. ([gh-14325](#))
- Class `SafeEval` in `numpy/lib/Utils.py` has been removed. This was deprecated in NumPy 1.10. Use `np.safe_eval` instead. ([gh-14335](#))
- Remove deprecated support for boolean and empty condition lists in `np.select` ([gh-14583](#))
- Array order only accepts 'C', 'F', 'A', and 'K'. More permissive options were deprecated in NumPy 1.11. ([gh-14596](#))
- `np.linspace` parameter `num` must be an integer. Deprecated in NumPy 1.12. ([gh-14620](#))
- UFuncs with multiple outputs must use a tuple for the `out` kwarg. This finishes a deprecation started in NumPy 1.10. ([gh-14682](#))

The files `numpy/testing/decorators.py`, `numpy/testing/noseclasses.py` and `numpy/testing/nosetester.py` have been removed. They were never meant to be public (all relevant objects are present in the `numpy.testing` namespace), and importing them has given a deprecation warning since NumPy 1.15.0 ([gh-14567](#))

15.29.5 Compatibility notes

`numpy.lib.recfunctions.drop_fields` can no longer return `None`

If `drop_fields` is used to drop all fields, previously the array would be completely discarded and `None` returned. Now it returns an array of the same shape as the input, but with no fields. The old behavior can be retained with:

```
dropped_arr = drop_fields(arr, ['a', 'b'])
if dropped_arr.dtype.names == ():
    dropped_arr = None
```

converting the empty recarray to `None` ([gh-14510](#))

`numpy.argmax/min/max` returns `NaT` if it exists in array

`numpy.argmax`, `numpy.min`, and `numpy.max` will return `NaT` if it exists in the array. ([gh-14717](#))

`np.can_cast(np.uint64, np.timedelta64, casting='safe')` is now `False`

Previously this was `True` - however, this was inconsistent with `uint64` not being safely castable to `int64`, and resulting in strange type resolution.

If this impacts your code, cast `uint64` to `int64` first. ([gh-14718](#))

Changed random variate stream from `numpy.random.Generator.integers`

There was a bug in `numpy.random.Generator.integers` that caused biased sampling of 8 and 16 bit integer types. Fixing that bug has changed the output stream from what it was in previous releases. ([gh-14777](#))

Add more ufunc loops for `datetime64`, `timedelta64`

`np.datetime('NaT')` should behave more like `float('Nan')`. Add needed infrastructure so `np.isinf(a)` and `np.isnan(a)` will run on `datetime64` and `timedelta64` dtypes. Also added specific loops for `numpy.fmin` and `numpy.fmax` that mask `NaT`. This may require adjustment to user-facing code. Specifically, code that either disallowed the calls to `numpy.isinf` or `numpy.isnan` or checked that they raised an exception will require adaptation, and code that mistakenly called `numpy.fmax` and `numpy.fmin` instead of `numpy.maximum` or `numpy.minimum` respectively will require adjustment. This also affects `numpy.nanmax` and `numpy.nanmin`. ([gh-14841](#))

Moved modules in `numpy.random`

As part of the API cleanup, the submodules in `numpy.random` `bit_generator`, `philox`, `pcg64`, `sfc64`, `common`, `generator`, and `bounded_integers` were moved to `_bit_generator`, `_philox`, `_pcg64`, `_sfc64`, `__common`, `_generator`, and `_bounded_integers` respectively to indicate that they are not part of the public interface. ([gh-14608](#))

15.29.6 C API changes

PyDataType_ISUNSIZED (descr) now returns False for structured datatypes

Previously this returned True for any datatype of itemsize 0, but now this returns false for the non-flexible datatype with itemsize 0, `np.dtype([])`. ([gh-14393](#))

15.29.7 New Features

Add our own *.pxd cython import file

Added a `numpy/__init__.pxd` file. It will be used for `cimport numpy` ([gh-12284](#))

A tuple of axes can now be input to `expand_dims`

The `numpy.expand_dims` `axis` keyword can now accept a tuple of axes. Previously, `axis` was required to be an integer. ([gh-14051](#))

Support for 64-bit OpenBLAS

Added support for 64-bit (ILP64) OpenBLAS. See `site.cfg.example` for details. ([gh-15012](#))

Add `--f2cmmap` option to F2PY

Allow specifying a file to load Fortran-to-C type map customizations from. ([gh-15113](#))

15.29.8 Improvements

Different C numeric types of the same size have unique names

On any given platform, two of `np.intc`, `np.int_`, and `np.longlong` would previously appear indistinguishable through their `repr`, despite their corresponding `dtype` having different properties. A similar problem existed for the unsigned counterparts to these types, and on some platforms for `np.double` and `np.longdouble`

These types now always print with a unique `__name__`. ([gh-10151](#))

`argwhere` now produces a consistent result on 0d arrays

On N-d arrays, `numpy.argwhere` now always produces an array of shape `(n_non_zero, arr.ndim)`, even when `arr.ndim == 0`. Previously, the last axis would have a dimension of 1 in this case. ([gh-13610](#))

Add `axis` argument for `random.permutation` and `random.shuffle`

Previously the `random.permutation` and `random.shuffle` functions can only shuffle an array along the first axis; they now have a new argument `axis` which allows shuffle along a specified axis. ([gh-13829](#))

method keyword argument for `np.random.multivariate_normal`

A method keyword argument is now available for `np.random.multivariate_normal` with possible values `{'svd', 'eigh', 'cholesky'}`. To use it, write `np.random.multivariate_normal(..., method=<method>)`. ([gh-14197](#))

Add complex number support for `numpy.fromstring`

Now `numpy.fromstring` can read complex numbers. ([gh-14227](#))

`numpy.unique` has consistent axes order when `axis` is not `None`

Using `moveaxis` instead of `swapaxes` in `numpy.unique`, so that the ordering of axes except the axis in arguments will not be broken. ([gh-14255](#))

`numpy.matmul` with boolean output now converts to boolean values

Calling `numpy.matmul` where the output is a boolean array would fill the array with `uint8` equivalents of the result, rather than 0/1. Now it forces the output to 0 or 1 (`NPY_TRUE` or `NPY_FALSE`). ([gh-14464](#))

`numpy.random.randint` produced incorrect value when the range was `232`**

The implementation introduced in 1.17.0 had an incorrect check when determining whether to use the 32-bit path or the full 64-bit path that incorrectly redirected random integer generation with a high - low range of `2**32` to the 64-bit generator. ([gh-14501](#))

Add complex number support for `numpy.fromfile`

Now `numpy.fromfile` can read complex numbers. ([gh-14730](#))

`std=c99` added if compiler is named `gcc`

GCC before version 5 requires the `-std=c99` command line argument. Newer compilers automatically turn on C99 mode. The compiler setup code will automatically add the code if the compiler name has `gcc` in it. ([gh-14771](#))

15.29.9 Changes

NaT now sorts to the end of arrays

NaT is now effectively treated as the largest integer for sorting purposes, so that it sorts to the end of arrays. This change is for consistency with NaN sorting behavior. ([gh-12658](#)) ([gh-15068](#))

Incorrect threshold in `np.set_printoptions` raises `TypeError` or `ValueError`

Previously an incorrect threshold raised `ValueError`; it now raises `TypeError` for non-numeric types and `ValueError` for nan values. ([gh-13899](#))

Warn when saving a dtype with metadata

A `UserWarning` will be emitted when saving an array via `numpy.save` with metadata. Saving such an array may not preserve metadata, and if metadata is preserved, loading it will cause a `ValueError`. This shortcoming in save and load will be addressed in a future release. ([gh-14142](#))

`numpy.distutils` append behavior changed for `LDFLAGS` and similar

`numpy.distutils` has always overridden rather than appended to `LDFLAGS` and other similar such environment variables for compiling Fortran extensions. Now the default behavior has changed to appending - which is the expected behavior in most situations. To preserve the old (overwriting) behavior, set the `NPY_DISTUTILS_APPEND_FLAGS` environment variable to 0. This applies to: `LDFLAGS`, `F77FLAGS`, `F90FLAGS`, `FREEFLAGS`, `FOPT`, `FDEBUG`, and `FFLAGS`. NumPy 1.16 and 1.17 gave build warnings in situations where this change in behavior would have affected the compile flags used. ([gh-14248](#))

Remove `numpy.random.entropy` without a deprecation

`numpy.random.entropy` was added to the `numpy.random` namespace in 1.17.0. It was meant to be a private c-extension module, but was exposed as public. It has been replaced by `numpy.random.SeedSequence` so the module was completely removed. ([gh-14498](#))

Add options to quiet build configuration and build with `-Werror`

Added two new configuration options. During the `build_src` subcommand, as part of configuring NumPy, the files `_numpyconfig.h` and `config.h` are created by probing support for various runtime functions and routines. Previously, the very verbose compiler output during this stage clouded more important information. By default the output is silenced. Running `runtests.py --debug-info` will add `--verbose-cfg` to the `build_src` subcommand, which will restore the previous behaviour.

Adding `CFLAGS=-Werror` to turn warnings into errors would trigger errors during the configuration. Now `runtests.py --warn-error` will add `--warn-error` to the `build` subcommand, which will percolate to the `build_ext` and `build_lib` subcommands. This will add the compiler flag to those stages and turn compiler warnings into errors while actually building NumPy itself, avoiding the `build_src` subcommand compiler calls.

([gh-14527](#)) ([gh-14518](#))

15.30 NumPy 1.17.5 Release Notes

This release contains fixes for bugs reported against NumPy 1.17.4 along with some build improvements. The Python versions supported in this release are 3.5-3.8.

Downstream developers should use Cython $\geq 0.29.14$ for Python 3.8 support and OpenBLAS ≥ 3.7 to avoid errors on the Skylake architecture.

It is recommended that developers interested in the new random bit generators upgrade to the NumPy 1.18.x series, as it has updated documentation and many small improvements.

15.30.1 Contributors

A total of 6 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Eric Wieser
- Ilhan Polat
- Matti Pícus
- Michael Hudson-Doyle
- Ralf Gommers

15.30.2 Pull requests merged

A total of 8 pull requests were merged for this release.

- [#14593](#): MAINT: backport Cython API cleanup to 1.17.x, remove docs
- [#14937](#): BUG: fix integer size confusion in handling array’s ndmin argument
- [#14939](#): BUILD: remove SSE2 flag from numpy.random builds
- [#14993](#): MAINT: Added Python3.8 branch to dll lib discovery
- [#15038](#): BUG: Fix refcounting in ufunc object loops
- [#15067](#): BUG: Exceptions tracebacks are dropped
- [#15175](#): ENH: Backport improvements to testing functions.
- [#15213](#): REL: Prepare for the NumPy 1.17.5 release.

15.31 NumPy 1.17.4 Release Notes

This release contains fixes for bugs reported against NumPy 1.17.3 along with some build improvements. The Python versions supported in this release are 3.5-3.8.

Downstream developers should use Cython $\geq 0.29.13$ for Python 3.8 support and OpenBLAS ≥ 3.7 to avoid errors on the Skylake architecture.

15.31.1 Highlights

- Fixed `random.random_integers` biased generation of 8 and 16 bit integers.
- Fixed `np.einsum` regression on Power9 and z/Linux.
- Fixed histogram problem with signed integer arrays.

15.31.2 Contributors

A total of 5 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Chris Burr +
- Matti Picus
- Qiming Sun +
- Warren Weckesser

15.31.3 Pull requests merged

A total of 8 pull requests were merged for this release.

- [#14758](#): BLD: declare support for python 3.8
- [#14781](#): BUG: random: biased samples from integers() with 8 or 16 bit...
- [#14851](#): BUG: Fix `_ctypes` class circular reference. ([#13808](#))
- [#14852](#): BLD: add ‘apt update’ to shippable
- [#14855](#): BUG: Fix `np.einsum` errors on Power9 Linux and z/Linux
- [#14857](#): BUG: lib: Fix histogram problem with signed integer arrays.
- [#14858](#): BLD: Prevent -flt to from optimising long double representation...
- [#14866](#): MAINT: move `buffer.h` -> `numpy_buffer.h` to avoid conflicts

15.32 NumPy 1.17.3 Release Notes

This release contains fixes for bugs reported against NumPy 1.17.2 along with a some documentation improvements. The Python versions supported in this release are 3.5-3.8.

Downstream developers should use Cython `>= 0.29.13` for Python 3.8 support and OpenBLAS `>= 3.7` to avoid errors on the Skylake architecture.

15.32.1 Highlights

- Wheels for Python 3.8
- Boolean `matmul` fixed to use booleans instead of integers.

15.32.2 Compatibility notes

- The seldom used `PyArray_DescrCheck` macro has been changed/fixed.

15.32.3 Contributors

A total of 7 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Allan Haldane
- Charles Harris
- Kevin Sheppard
- Matti Picus
- Ralf Gommers
- Sebastian Berg
- Warren Weckesser

15.32.4 Pull requests merged

A total of 12 pull requests were merged for this release.

- [#14456](#): MAINT: clean up pocketfft modules inside `numpy.fft` namespace.
- [#14463](#): BUG: `random.hypergeometric` assumes `numpy.long` is `numpy.int64`, hung...
- [#14502](#): BUG: `random`: Revert `gh-14458` and refix `gh-14557`.
- [#14504](#): BUG: add a specialized loop for boolean `matmul`.
- [#14506](#): MAINT: Update `pytest` version for Python 3.8
- [#14512](#): DOC: `random`: fix doc linking, was referencing private submodules.
- [#14513](#): BUG,MAINT: Some fixes and minor cleanup based on clang analysis
- [#14515](#): BUG: Fix `randint` when range is `2**32`
- [#14519](#): MAINT: remove the `entropy` c-extension module
- [#14563](#): DOC: remove note about `Pocketfft` license file (non-existing here).
- [#14578](#): BUG: `random`: Create a legacy implementation of `random.binomial`.
- [#14687](#): BUG: properly define `PyArray_DescrCheck`

15.33 NumPy 1.17.2 Release Notes

This release contains fixes for bugs reported against NumPy 1.17.1 along with a some documentation improvements. The most important fix is for lexsort when the keys are of type (u)int8 or (u)int16. If you are currently using 1.17 you should upgrade.

The Python versions supported in this release are 3.5-3.7, Python 2.7 has been dropped. Python 3.8b4 should work with the released source packages, but there are no future guarantees.

Downstream developers should use Cython $\geq 0.29.13$ for Python 3.8 support and OpenBLAS ≥ 3.7 to avoid errors on the Skylake architecture. The NumPy wheels on PyPI are built from the OpenBLAS development branch in order to avoid those errors.

15.33.1 Contributors

A total of 7 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- CakeWithSteak +
- Charles Harris
- Dan Allan
- Hameer Abbasi
- Lars Grueter
- Matti Picus
- Sebastian Berg

15.33.2 Pull requests merged

A total of 8 pull requests were merged for this release.

- [#14418](#): BUG: Fix aradixsort indirect indexing.
- [#14420](#): DOC: Fix a minor typo in dispatch documentation.
- [#14421](#): BUG: test, fix regression in converting to ctypes
- [#14430](#): BUG: Do not show Override module in private error classes.
- [#14432](#): BUG: Fixed maximum relative error reporting in assert_allclose.
- [#14433](#): BUG: Fix uint-overflow if padding with linear_ramp and negative...
- [#14436](#): BUG: Update 1.17.x with 1.18.0-dev pocketfft.py.
- [#14446](#): REL: Prepare for NumPy 1.17.2 release.

15.34 NumPy 1.17.1 Release Notes

This release contains a number of fixes for bugs reported against NumPy 1.17.0 along with a few documentation and build improvements. The Python versions supported are 3.5-3.7, note that Python 2.7 has been dropped. Python 3.8b3 should work with the released source packages, but there are no future guarantees.

Downstream developers should use Cython $\geq 0.29.13$ for Python 3.8 support and OpenBLAS ≥ 3.7 to avoid problems on the Skylake architecture. The NumPy wheels on PyPI are built from the OpenBLAS development branch in order to avoid those problems.

15.34.1 Contributors

A total of 17 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Alexander Jung +
- Allan Haldane
- Charles Harris
- Eric Wieser
- Giuseppe Cuccu +
- Hiroyuki V. Yamazaki
- Jérémie du Boisberranger
- Kmol Yuan +
- Matti Pícus
- Max Bolingbroke +
- Maxwell Aladago +
- Oleksandr Pavlyk
- Peter Andreas Entschéev
- Sergei Lebedev
- Seth Troisi +
- Vladimir Pershin +
- Warren Weckesser

15.34.2 Pull requests merged

A total of 24 pull requests were merged for this release.

- [#14156](#): TST: Allow fuss in testing strided/non-strided exp/log loops
- [#14157](#): BUG: avx2_scalef_ps must be static
- [#14158](#): BUG: Remove stray print that causes a SystemError on python 3.7.
- [#14159](#): BUG: Fix DeprecationWarning in python 3.8.
- [#14160](#): BLD: Add missing gcd/lcm definitions to npy_math.h
- [#14161](#): DOC, BUILD: cleanups and fix (again) ‘build dist’

- [#14166](#): TST: Add 3.8-dev to travisCI testing.
- [#14194](#): BUG: Remove the broken clip wrapper (Backport)
- [#14198](#): DOC: Fix hermitian argument docs in `svd`.
- [#14199](#): MAINT: Workaround for Intel compiler bug leading to failing test
- [#14200](#): TST: Clean up of `test_pocketfft.py`
- [#14201](#): BUG: Make advanced indexing result on read-only subclass writeable...
- [#14236](#): BUG: Fixed default BitGenerator name
- [#14237](#): ENH: add c-imported modules for freeze analysis in `np.random`
- [#14296](#): TST: Pin pytest version to 5.0.1
- [#14301](#): BUG: Fix leak in the f2py-generated module init and `PyMem_Del...`
- [#14302](#): BUG: Fix formatting error in exception message
- [#14307](#): MAINT: random: Match type of `SeedSequence.pool_size` to `DEFAULT_POOL_SIZE`.
- [#14308](#): BUG: Fix `numpy.random` bug in platform detection
- [#14309](#): ENH: Enable huge pages in all Linux builds
- [#14330](#): BUG: Fix segfault in `random.permutation(x)` when `x` is a string.
- [#14338](#): BUG: don't fail when lexsorting some empty arrays ([#14228](#))
- [#14339](#): BUG: Fix misuse of `.names` and `.fields` in various places (backport...)
- [#14345](#): BUG: fix behavior of `structured_to_unstructured` on non-trivial...
- [#14350](#): REL: Prepare 1.17.1 release

15.35 NumPy 1.17.0 Release Notes

This NumPy release contains a number of new features that should substantially improve its performance and usefulness, see Highlights below for a summary. The Python versions supported are 3.5-3.7, note that Python 2.7 has been dropped. Python 3.8b2 should work with the released source packages, but there are no future guarantees.

Downstream developers should use Cython `>= 0.29.11` for Python 3.8 support and OpenBLAS `>= 3.7` (not currently out) to avoid problems on the Skylake architecture. The NumPy wheels on PyPI are built from the OpenBLAS development branch in order to avoid those problems.

15.35.1 Highlights

- A new extensible `random` module along with four selectable *random number generators* and improved seeding designed for use in parallel processes has been added. The currently available bit generators are *MT19937*, *PCG64*, *Philox*, and *SFC64*. See below under New Features.
- NumPy's FFT implementation was changed from `fftpack` to `pocketfft`, resulting in faster, more accurate transforms and better handling of datasets of prime length. See below under Improvements.
- New radix sort and timsort sorting methods. It is currently not possible to choose which will be used. They are hardwired to the datatype and used when either `stable` or `mergesort` is passed as the method. See below under Improvements.
- Overriding `numpy` functions is now possible by default, see `__array_function__` below.

15.35.2 New functions

- `numpy.errstate` is now also a function decorator

15.35.3 Deprecations

`numpy.polynomial` functions warn when passed float in place of int

Previously functions in this module would accept `float` values provided they were integral (`1.0`, `2.0`, etc). For consistency with the rest of `numpy`, doing so is now deprecated, and in future will raise a `TypeError`.

Similarly, passing a float like `0.5` in place of an integer will now raise a `TypeError` instead of the previous `ValueError`.

Deprecate `numpy.distutils.exec_command` and `temp_file_name`

The internal use of these functions has been refactored and there are better alternatives. Replace `exec_command` with `subprocess.Popen` and `temp_file_name` with `tempfile.mkstemp`.

Writeable flag of C-API wrapped arrays

When an array is created from the C-API to wrap a pointer to data, the only indication we have of the read-write nature of the data is the `writeable` flag set during creation. It is dangerous to force the flag to writeable. In the future it will not be possible to switch the writeable flag to `True` from python. This deprecation should not affect many users since arrays created in such a manner are very rare in practice and only available through the NumPy C-API.

`numpy.nonzero` should no longer be called on 0d arrays

The behavior of `numpy.nonzero` on 0d arrays was surprising, making uses of it almost always incorrect. If the old behavior was intended, it can be preserved without a warning by using `nonzero(atleast_1d(arr))` instead of `nonzero(arr)`. In a future release, it is most likely this will raise a `ValueError`.

Writing to the result of `numpy.broadcast_arrays` will warn

Commonly `numpy.broadcast_arrays` returns a writeable array with internal overlap, making it unsafe to write to. A future version will set the `writeable` flag to `False`, and require users to manually set it to `True` if they are sure that is what they want to do. Now writing to it will emit a deprecation warning with instructions to set the `writeable` flag `True`. Note that if one were to inspect the flag before setting it, one would find it would already be `True`. Explicitly setting it, though, as one will need to do in future versions, clears an internal flag that is used to produce the deprecation warning. To help alleviate confusion, an additional *FutureWarning* will be emitted when accessing the `writeable` flag state to clarify the contradiction.

Note that for the C-side buffer protocol such an array will return a readonly buffer immediately unless a writable buffer is requested. If a writeable buffer is requested a warning will be given. When using `cython`, the `const` qualifier should be used with such arrays to avoid the warning (e.g. `cdef const double[:,1] view`).

15.35.4 Future Changes

Shape-1 fields in dtypes won't be collapsed to scalars in a future version

Currently, a field specified as `[(name, dtype, 1)]` or `"1type"` is interpreted as a scalar field (i.e., the same as `[(name, dtype)]` or `[(name, dtype, ())]`). This now raises a `FutureWarning`; in a future version, it will be interpreted as a shape-(1,) field, i.e. the same as `[(name, dtype, (1,))]` or `"(1,)type"` (consistently with `[(name, dtype, n)] / "ntype"` with $n > 1$, which is already equivalent to `[(name, dtype, (n,)) / "(n,)type"`).

15.35.5 Compatibility notes

float16 subnormal rounding

Casting from a different floating point precision to `float16` used incorrect rounding in some edge cases. This means in rare cases, subnormal results will now be rounded up instead of down, changing the last bit (ULP) of the result.

Signed zero when using divmod

Starting in version *1.12.0*, numpy incorrectly returned a negatively signed zero when using the `divmod` and `floor_divide` functions when the result was zero. For example:

```
>>> np.zeros(10)//1
array([-0., -0., -0., -0., -0., -0., -0., -0., -0., -0.])
```

With this release, the result is correctly returned as a positively signed zero:

```
>>> np.zeros(10)//1
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

MaskedArray.mask now returns a view of the mask, not the mask itself

Returning the mask itself was unsafe, as it could be reshaped in place which would violate expectations of the masked array code. The behavior of `mask` is now consistent with `data`, which also returns a view.

The underlying mask can still be accessed with `._mask` if it is needed. Tests that contain `assert x.mask is not y.mask` or similar will need to be updated.

Do not lookup `__buffer__` attribute in `numpy.frombuffer`

Looking up `__buffer__` attribute in `numpy.frombuffer` was undocumented and non-functional. This code was removed. If needed, use `frombuffer(memoryview(obj), ...)` instead.

out is buffered for memory overlaps in take, choose, put

If the out argument to these functions is provided and has memory overlap with the other arguments, it is now buffered to avoid order-dependent behavior.

Unpickling while loading requires explicit opt-in

The functions `load`, and `lib.format.read_array` take an `allow_pickle` keyword which now defaults to `False` in response to [CVE-2019-6446](#).

Potential changes to the random stream in old random module

Due to bugs in the application of `log` to random floating point numbers, the stream may change when sampling from `beta`, `binomial`, `laplace`, `logistic`, `logseries` or `multinomial` if a 0 is generated in the underlying MT19937 random stream. There is a 1 in 10^{53} chance of this occurring, so the probability that the stream changes for any given seed is extremely small. If a 0 is encountered in the underlying generator, then the incorrect value produced (either `numpy.inf` or `numpy.nan`) is now dropped.

i0 now always returns a result with the same shape as the input

Previously, the output was squeezed, such that, e.g., input with just a single element would lead to an array scalar being returned, and inputs with shapes such as `(10, 1)` would yield results that would not broadcast against the input.

Note that we generally recommend the SciPy implementation over the numpy one: it is a proper ufunc written in C, and more than an order of magnitude faster.

can_cast no longer assumes all unsafe casting is allowed

Previously, `can_cast` returned `True` for almost all inputs for `casting='unsafe'`, even for cases where casting was not possible, such as from a structured dtype to a regular one. This has been fixed, making it more consistent with actual casting using, e.g., the `.astype` method.

ndarray.flags.writeable can be switched to true slightly more often

In rare cases, it was not possible to switch an array from not writeable to writeable, although a base array is writeable. This can happen if an intermediate `ndarray.base` object is writeable. Previously, only the deepest base object was considered for this decision. However, in rare cases this object does not have the necessary information. In that case switching to writeable was never allowed. This has now been fixed.

15.35.6 C API changes

dimension or stride input arguments are now passed by `numpy_intp const*`

Previously these function arguments were declared as the more strict `numpy_intp*`, which prevented the caller passing constant data. This change is backwards compatible, but now allows code like:

```
numpy_intp const fixed_dims[] = {1, 2, 3};
// no longer complains that the const-qualifier is discarded
numpy_intp size = PyArray_MultiplyList(fixed_dims, 3);
```

15.35.7 New Features

New extensible `numpy.random` module with selectable random number generators

A new extensible `numpy.random` module along with four selectable random number generators and improved seeding designed for use in parallel processes has been added. The currently available *Bit Generators* are *MT19937*, *PCG64*, *Philox*, and *SFC64*. *PCG64* is the new default while *MT19937* is retained for backwards compatibility. Note that the legacy random module is unchanged and is now frozen, your current results will not change. More information is available in the [API change description](#) and in the [top-level view](#) documentation.

libFLAME

Support for building NumPy with the libFLAME linear algebra package as the LAPACK, implementation, see [libFLAME](#) for details.

User-defined BLAS detection order

`distutils` now uses an environment variable, comma-separated and case insensitive, to determine the detection order for BLAS libraries. By default `NPY_BLAS_ORDER=mkl,blis,openblas,atlas,accelerate,blas`. However, to force the use of OpenBLAS simply do:

```
NPY_BLAS_ORDER=openblas python setup.py build
```

which forces the use of OpenBLAS. This may be helpful for users which have a MKL installation but wishes to try out different implementations.

User-defined LAPACK detection order

`numpy.distutils` now uses an environment variable, comma-separated and case insensitive, to determine the detection order for LAPACK libraries. By default `NPY_LAPACK_ORDER=mkl,openblas,flame,atlas,accelerate,lapack`. However, to force the use of OpenBLAS simply do:

```
NPY_LAPACK_ORDER=openblas python setup.py build
```

which forces the use of OpenBLAS. This may be helpful for users which have a MKL installation but wishes to try out different implementations.

`ufunc.reduce` and related functions now accept a `where` mask

`ufunc.reduce`, `sum`, `prod`, `min`, `max` all now accept a `where` keyword argument, which can be used to tell which elements to include in the reduction. For reductions that do not have an identity, it is necessary to also pass in an initial value (e.g., `initial=np.inf` for `min`). For instance, the equivalent of `nansum` would be `np.sum(a, where=~np.isnan(a))`.

Timsort and radix sort have replaced mergesort for stable sorting

Both radix sort and timsort have been implemented and are now used in place of mergesort. Due to the need to maintain backward compatibility, the sorting `kind` options `"stable"` and `"mergesort"` have been made aliases of each other with the actual sort implementation depending on the array type. Radix sort is used for small integer types of 16 bits or less and timsort for the remaining types. Timsort features improved performance on data containing already or nearly sorted data and performs like mergesort on random data and requires $O(n/2)$ working space. Details of the timsort algorithm can be found at [CPython listsort.txt](#).

`packbits` and `unpackbits` accept an `order` keyword

The `order` keyword defaults to `big`, and will order the **bits** accordingly. For `'order=big'` 3 will become `[0, 0, 0, 0, 0, 0, 1, 1]`, and `[1, 1, 0, 0, 0, 0, 0, 0]` for `order=little`

`unpackbits` now accepts a `count` parameter

`count` allows subsetting the number of bits that will be unpacked up-front, rather than reshaping and subsetting later, making the `packbits` operation invertible, and the unpacking less wasteful. Counts larger than the number of available bits add zero padding. Negative counts trim bits off the end instead of counting from the beginning. None counts implement the existing behavior of unpacking everything.

`linalg.svd` and `linalg.pinv` can be faster on hermitian inputs

These functions now accept a `hermitian` argument, matching the one added to `linalg.matrix_rank` in 1.14.0.

`divmod` operation is now supported for two `timedelta64` operands

The `divmod` operator now handles two `timedelta64` operands, with type signature `mm->qm`.

`fromfile` now takes an `offset` argument

This function now takes an `offset` keyword argument for binary files, which specifies the offset (in bytes) from the file's current position. Defaults to 0.

New mode “empty” for `pad`

This mode pads an array to a desired shape without initializing the new entries.

`empty_like` and related functions now accept a `shape` argument

`empty_like`, `full_like`, `ones_like` and `zeros_like` now accept a `shape` keyword argument, which can be used to create a new array as the prototype, overriding its shape as well. This is particularly useful when combined with the `__array_function__` protocol, allowing the creation of new arbitrary-shape arrays from NumPy-like libraries when such an array is used as the prototype.

Floating point scalars implement `as_integer_ratio` to match the builtin float

This returns a (numerator, denominator) pair, which can be used to construct a `fractions.Fraction`.

Structured dtype objects can be indexed with multiple fields names

`arr.dtype[['a', 'b']]` now returns a dtype that is equivalent to `arr[['a', 'b']].dtype`, for consistency with `arr.dtype['a'] == arr['a'].dtype`.

Like the dtype of structured arrays indexed with a list of fields, this dtype has the same `itemsize` as the original, but only keeps a subset of the fields.

This means that `arr[['a', 'b']]` and `arr.view(arr.dtype[['a', 'b']])` are equivalent.

`.npy` files support unicode field names

A new format version of 3.0 has been introduced, which enables structured types with non-latin1 field names. This is used automatically when needed.

15.35.8 Improvements

Array comparison assertions include maximum differences

Error messages from array comparison tests such as `testing.assert_allclose` now include “max absolute difference” and “max relative difference,” in addition to the previous “mismatch” percentage. This information makes it easier to update absolute and relative error tolerances.

Replacement of the `fftpack` based `fft` module by the `pocketfft` library

Both implementations have the same ancestor (Fortran77 FFTPACK by Paul N. Swarztrauber), but `pocketfft` contains additional modifications which improve both accuracy and performance in some circumstances. For FFT lengths containing large prime factors, `pocketfft` uses Bluestein’s algorithm, which maintains $O(N \log N)$ run time complexity instead of deteriorating towards $O(N * N)$ for prime lengths. Also, accuracy for real valued FFTs with near prime lengths has improved and is on par with complex valued FFTs.

Further improvements to `ctypes` support in `numpy.ctypeslib`

A new `numpy.ctypeslib.as_ctypes_type` function has been added, which can be used to convert a dtype into a best-guess `ctypes` type. Thanks to this new function, `numpy.ctypeslib.as_ctypes` now supports a much wider range of array types, including structures, booleans, and integers of non-native endianness.

`numpy.errstate` is now also a function decorator

Currently, if you have a function like:

```
def foo():  
    pass
```

and you want to wrap the whole thing in `errstate`, you have to rewrite it like so:

```
def foo():  
    with np.errstate(...):  
        pass
```

but with this change, you can do:

```
@np.errstate(...)  
def foo():  
    pass
```

thereby saving a level of indentation

`numpy.exp` and `numpy.log` speed up for float32 implementation

float32 implementation of `exp` and `log` now benefit from AVX2/AVX512 instruction set which are detected during runtime. `exp` has a max ulp error of 2.52 and `log` has a max ulp error of 3.83.

Improve performance of `numpy.pad`

The performance of the function has been improved for most cases by filling in a preallocated array with the desired padded shape instead of using concatenation.

`numpy.interp` handles infinities more robustly

In some cases where `interp` would previously return `nan`, it now returns an appropriate infinity.

Pathlib support for `fromfile`, `tofile` and `ndarray.dump`

`fromfile`, `ndarray.ndarray.tofile` and `ndarray.dump` now support the `pathlib.Path` type for the `file/fid` parameter.

Specialized `isnan`, `isinf`, and `isfinite` ufuncs for bool and int types

The boolean and integer types are incapable of storing `nan` and `inf` values, which allows us to provide specialized ufuncs that are up to 250x faster than the previous approach.

isfinite supports datetime64 and timedelta64 types

Previously, `isfinite` used to raise a `TypeError` on being used on these two types.

New keywords added to `nan_to_num`

`nan_to_num` now accepts keywords `nan`, `posinf` and `neginf` allowing the user to define the value to replace the `nan`, positive and negative `np.inf` values respectively.

MemoryErrors caused by allocated overly large arrays are more descriptive

Often the cause of a `MemoryError` is incorrect broadcasting, which results in a very large and incorrect shape. The message of the error now includes this shape to help diagnose the cause of failure.

`floor`, `ceil`, and `trunc` now respect builtin magic methods

These ufuncs now call the `__floor__`, `__ceil__`, and `__trunc__` methods when called on object arrays, making them compatible with `decimal.Decimal` and `fractions.Fraction` objects.

`quantile` now works on `fraction.Fraction` and `decimal.Decimal` objects

In general, this handles object arrays more gracefully, and avoids floating- point operations if exact arithmetic types are used.

Support of object arrays in `matmul`

It is now possible to use `matmul` (or the `@` operator) with object arrays. For instance, it is now possible to do:

```
from fractions import Fraction
a = np.array([[Fraction(1, 2), Fraction(1, 3)], [Fraction(1, 3), Fraction(1, 2)]])
b = a @ a
```

15.35.9 Changes

`median` and `percentile` family of functions no longer warn about `nan`

`numpy.median`, `numpy.percentile`, and `numpy.quantile` used to emit a `RuntimeWarning` when encountering an `nan`. Since they return the `nan` value, the warning is redundant and has been removed.

`timedelta64 % 0` behavior adjusted to return `NaT`

The modulus operation with two `np.timedelta64` operands now returns `NaT` in the case of division by zero, rather than returning zero

NumPy functions now always support overrides with `__array_function__`

NumPy now always checks the `__array_function__` method to implement overrides of NumPy functions on non-NumPy arrays, as described in [NEP 18](#). The feature was available for testing with NumPy 1.16 if appropriate environment variables are set, but is now always enabled.

`lib.recfunctions.structured_to_unstructured` does not squeeze single-field views

Previously `structured_to_unstructured(arr[['a']])` would produce a squeezed result inconsistent with `structured_to_unstructured(arr[['a', b]])`. This was accidental. The old behavior can be retained with `structured_to_unstructured(arr[['a']]).squeeze(axis=-1)` or far more simply, `arr['a']`.

`clip` now uses a ufunc under the hood

This means that registering clip functions for custom dtypes in C via `descr->f->fastclip` is deprecated - they should use the ufunc registration mechanism instead, attaching to the `np.core.umath.clip` ufunc.

It also means that `clip` accepts `where` and `casting` arguments, and can be override with `__array_ufunc__`.

A consequence of this change is that some behaviors of the old `clip` have been deprecated:

- Passing `nan` to mean “do not clip” as one or both bounds. This didn’t work in all cases anyway, and can be better handled by passing infinities of the appropriate sign.
- Using “unsafe” casting by default when an `out` argument is passed. Using `casting="unsafe"` explicitly will silence this warning.

Additionally, there are some corner cases with behavior changes:

- Padding `max < min` has changed to be more consistent across dtypes, but should not be relied upon.
- Scalar `min` and `max` take part in promotion rules like they do in all other ufuncs.

`__array_interface__` `offset` now works as documented

The interface may use an `offset` value that was mistakenly ignored.

Pickle protocol in `savez` set to 3 for force `zip64` flag

`savez` was not using the `force_zip64` flag, which limited the size of the archive to 2GB. But using the flag requires us to use pickle protocol 3 to write object arrays. The protocol used was bumped to 3, meaning the archive will be unreadable by Python2.

Structured arrays indexed with non-existent fields raise `KeyError` not `ValueError`

`arr['bad_field']` on a structured type raises `KeyError`, for consistency with `dict['bad_field']`.

15.36 NumPy 1.16.6 Release Notes

The NumPy 1.16.6 release fixes bugs reported against the 1.16.5 release, and also backports several enhancements from master that seem appropriate for a release series that is the last to support Python 2.7. The wheels on PyPI are linked with OpenBLAS v0.3.7, which should fix errors on Skylake series cpus.

Downstream developers building this release should use Cython `>= 0.29.2` and, if using OpenBLAS, OpenBLAS `>= v0.3.7`. The supported Python versions are 2.7 and 3.5-3.7.

15.36.1 Highlights

- The `np.testing.utils` functions have been updated from 1.19.0-dev0. This improves the function documentation and error messages as well extending the `assert_array_compare` function to additional types.

15.36.2 New functions

Allow `matmul` (`@` operator) to work with object arrays.

This is an enhancement that was added in NumPy 1.17 and seems reasonable to include in the LTS 1.16 release series.

15.36.3 Compatibility notes

Fix regression in `matmul` (`@` operator) for boolean types

Booleans were being treated as integers rather than booleans, which was a regression from previous behavior.

15.36.4 Improvements

Array comparison assertions include maximum differences

Error messages from array comparison tests such as `testing.assert_allclose` now include “max absolute difference” and “max relative difference,” in addition to the previous “mismatch” percentage. This information makes it easier to update absolute and relative error tolerances.

15.36.5 Contributors

A total of 10 people contributed to this release.

- [CakeWithSteak](#)
- [Charles Harris](#)
- [Chris Burr](#)
- [Eric Wieser](#)
- [Fernando Saravia](#)
- [Lars Grueter](#)
- [Matti Picus](#)
- [Maxwell Aladago](#)
- [Qiming Sun](#)
- [Warren Weckesser](#)

15.36.6 Pull requests merged

A total of 14 pull requests were merged for this release.

- [#14211](#): BUG: Fix uint-overflow if padding with linear_ramp and negative...
- [#14275](#): BUG: fixing to allow unpickling of PY3 pickles from PY2
- [#14340](#): BUG: Fix misuse of .names and .fields in various places (backport...
- [#14423](#): BUG: test, fix regression in converting to ctypes.
- [#14434](#): BUG: Fixed maximum relative error reporting in assert_allclose
- [#14509](#): BUG: Fix regression in boolean matmul.
- [#14686](#): BUG: properly define PyArray_DescrCheck
- [#14853](#): BLD: add 'apt update' to shippable
- [#14854](#): BUG: Fix _ctypes class circular reference. (#13808)
- [#14856](#): BUG: Fix *np.einsum* errors on Power9 Linux and z/Linux
- [#14863](#): BLD: Prevent -flto from optimising long double representation...
- [#14864](#): BUG: lib: Fix histogram problem with signed integer arrays.
- [#15172](#): ENH: Backport improvements to testing functions.
- [#15191](#): REL: Prepare for 1.16.6 release.

15.37 NumPy 1.16.5 Release Notes

The NumPy 1.16.5 release fixes bugs reported against the 1.16.4 release, and also backports several enhancements from master that seem appropriate for a release series that is the last to support Python 2.7. The wheels on PyPI are linked with OpenBLAS v0.3.7-dev, which should fix errors on Skylake series cpus.

Downstream developers building this release should use Cython $\geq 0.29.2$ and, if using OpenBLAS, OpenBLAS $\geq v0.3.7$. The supported Python versions are 2.7 and 3.5-3.7.

15.37.1 Contributors

A total of 18 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Alexander Shadchin
- Allan Haldane
- Bruce Merry +
- Charles Harris
- Colin Snyder +
- Dan Allan +
- Emile +
- Eric Wieser
- Grey Baker +
- Maksim Shabunin +
- Marten van Kerkwijk
- Matti Pícus
- Peter Andreas Entsché +
- Ralf Gommers
- Richard Harris +
- Sebastian Berg
- Sergei Lebedev +
- Stephan Hoyer

15.37.2 Pull requests merged

A total of 23 pull requests were merged for this release.

- [#13742](#): ENH: Add project URLs to setup.py
- [#13823](#): TEST, ENH: fix tests and ctypes code for PyPy
- [#13845](#): BUG: use npy_intp instead of int for indexing array
- [#13867](#): TST: Ignore DeprecationWarning during nose imports
- [#13905](#): BUG: Fix use-after-free in boolean indexing
- [#13933](#): MAINT/BUG/DOC: Fix errors in _add_newdocs

- [#13984](#): BUG: fix byte order reversal for datetime64[ns]
- [#13994](#): MAINT,BUG: Use nbytes to also catch empty descr during allocation
- [#14042](#): BUG: np.array cleared errors occurred in PyMemoryView_FromObject
- [#14043](#): BUG: Fixes for Undefined Behavior Sanitizer (UBSan) errors.
- [#14044](#): BUG: ensure that casting to/from structured is properly checked.
- [#14045](#): MAINT: fix histogram*d dispatchers
- [#14046](#): BUG: further fixup to histogram2d dispatcher.
- [#14052](#): BUG: Replace contextlib.suppress for Python 2.7
- [#14056](#): BUG: fix compilation of 3rd party modules with Py_LIMITED_API...
- [#14057](#): BUG: Fix memory leak in dtype from dict constructor
- [#14058](#): DOC: Document array_function at a higher level.
- [#14084](#): BUG, DOC: add new recfunctions to `__all__`
- [#14162](#): BUG: Remove stray print that causes a SystemError on python 3.7
- [#14297](#): TST: Pin pytest version to 5.0.1.
- [#14322](#): ENH: Enable huge pages in all Linux builds
- [#14346](#): BUG: fix behavior of structured_to_unstructured on non-trivial...
- [#14382](#): REL: Prepare for the NumPy 1.16.5 release.

15.38 NumPy 1.16.4 Release Notes

The NumPy 1.16.4 release fixes bugs reported against the 1.16.3 release, and also backports several enhancements from master that seem appropriate for a release series that is the last to support Python 2.7. The wheels on PyPI are linked with OpenBLAS v0.3.7-dev, which should fix issues on Skylake series cpus.

Downstream developers building this release should use Cython `>= 0.29.2` and, if using OpenBLAS, `OpenBLAS > v0.3.7`. The supported Python versions are 2.7 and 3.5-3.7.

15.38.1 New deprecations

Writeable flag of C-API wrapped arrays

When an array is created from the C-API to wrap a pointer to data, the only indication we have of the read-write nature of the data is the `writeable` flag set during creation. It is dangerous to force the flag to writeable. In the future it will not be possible to switch the writeable flag to `True` from python. This deprecation should not affect many users since arrays created in such a manner are very rare in practice and only available through the NumPy C-API.

15.38.2 Compatibility notes

Potential changes to the random stream

Due to bugs in the application of log to random floating point numbers, the stream may change when sampling from `np.random.beta`, `np.random.binomial`, `np.random.laplace`, `np.random.logistic`, `np.random.logseries` or `np.random.multinomial` if a 0 is generated in the underlying MT19937 random stream. There is a 1 in 10^{53} chance of this occurring, and so the probability that the stream changes for any given seed is extremely small. If a 0 is encountered in the underlying generator, then the incorrect value produced (either `np.inf` or `np.nan`) is now dropped.

15.38.3 Changes

`numpy.lib.recfunctions.structured_to_unstructured` does not squeeze single-field views

Previously `structured_to_unstructured(arr[['a']])` would produce a squeezed result inconsistent with `structured_to_unstructured(arr[['a', b]])`. This was accidental. The old behavior can be retained with `structured_to_unstructured(arr[['a']]).squeeze(axis=-1)` or far more simply, `arr['a']`.

15.38.4 Contributors

A total of 10 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Eric Wieser
- Dennis Zollo +
- Hunter Damron +
- Jingbei Li +
- Kevin Sheppard
- Matti Picus
- Nicola Soranzo +
- Sebastian Berg
- Tyler Reddy

15.38.5 Pull requests merged

A total of 16 pull requests were merged for this release.

- [#13392](#): BUG: Some PyPy versions lack `PyStructSequence_InitType2`.
- [#13394](#): MAINT, DEP: Fix deprecated `assertEquals()`
- [#13396](#): BUG: Fix `structured_to_unstructured` on single-field types (backport)
- [#13549](#): BLD: Make CI pass again with `pytest 4.5`
- [#13552](#): TST: Register markers in `conftest.py`.
- [#13559](#): BUG: Removes `ValueError` for empty `kwargs` in `arraymultiter_new`

- [#13560](#): BUG: Add `TypeError` to accepted exceptions in `crackfortran`.
- [#13561](#): BUG: Handle subarrays in `descr_to_dtype`
- [#13562](#): BUG: Protect generators from `log(0.0)`
- [#13563](#): BUG: Always return views from `structured_to_unstructured` when...
- [#13564](#): BUG: Catch `stderr` when checking compiler version
- [#13565](#): BUG: `longdouble(int)` does not work
- [#13587](#): BUG: `distutils/system_info.py` fix missing `subprocess` import ([#13523](#))
- [#13620](#): BUG,DEP: Fix writeable flag setting for arrays without base
- [#13641](#): MAINT: Prepare for the 1.16.4 release.
- [#13644](#): BUG: special case object arrays when printing `rel-`, `abs-error`

15.39 NumPy 1.16.3 Release Notes

The NumPy 1.16.3 release fixes bugs reported against the 1.16.2 release, and also backports several enhancements from master that seem appropriate for a release series that is the last to support Python 2.7. The wheels on PyPI are linked with OpenBLAS v0.3.4+, which should fix the known threading issues found in previous OpenBLAS versions.

Downstream developers building this release should use Cython `>= 0.29.2` and, if using OpenBLAS, `OpenBLAS > v0.3.4`.

The most noticeable change in this release is that unpickling object arrays when loading `*.npy` or `*.npz` files now requires an explicit opt-in. This backwards incompatible change was made in response to [CVE-2019-6446](#).

15.39.1 Compatibility notes

Unpickling while loading requires explicit opt-in

The functions `np.load`, and `np.lib.format.read_array` take an *allow_pickle* keyword which now defaults to `False` in response to [CVE-2019-6446](#).

15.39.2 Improvements

Covariance in *random.mvnormal* cast to double

This should make the tolerance used when checking the singular values of the covariance matrix more meaningful.

15.39.3 Changes

`__array_interface__` `offset` now works as documented

The interface may use an `offset` value that was previously mistakenly ignored.

15.40 NumPy 1.16.2 Release Notes

NumPy 1.16.2 is a quick release fixing several problems encountered on Windows. The Python versions supported are 2.7 and 3.5-3.7. The Windows problems addressed are:

- DLL load problems for NumPy wheels on Windows,
- distutils command line parsing on Windows.

There is also a regression fix correcting signed zeros produced by `divmod`, see below for details.

Downstream developers building this release should use Cython `>= 0.29.2` and, if using OpenBLAS, `OpenBLAS > v0.3.4`.

If you are installing using pip, you may encounter a problem with older installed versions of NumPy that pip did not delete becoming mixed with the current version, resulting in an `ImportError`. That problem is particularly common on Debian derived distributions due to a modified pip. The fix is to make sure all previous NumPy versions installed by pip have been removed. See [#12736](#) for discussion of the issue.

15.40.1 Compatibility notes

Signed zero when using `divmod`

Starting in version 1.12.0, numpy incorrectly returned a negatively signed zero when using the `divmod` and `floor_divide` functions when the result was zero. For example:

```
>>> np.zeros(10)//1
array([-0., -0., -0., -0., -0., -0., -0., -0., -0., -0.])
```

With this release, the result is correctly returned as a positively signed zero:

```
>>> np.zeros(10)//1
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

15.40.2 Contributors

A total of 5 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Eric Wieser
- Matti Pícus
- Tyler Reddy
- Tony LaTorre +

15.40.3 Pull requests merged

A total of 7 pull requests were merged for this release.

- [#12909](#): TST: fix vmImage dispatch in Azure
- [#12923](#): MAINT: remove complicated test of multiarray import failure mode
- [#13020](#): BUG: fix signed zero behavior in npy_divmod
- [#13026](#): MAINT: Add functions to parse shell-strings in the platform-native...
- [#13028](#): BUG: Fix regression in parsing of F90 and F77 environment variables
- [#13038](#): BUG: parse shell escaping in extra_compile_args and extra_link_args
- [#13041](#): BLD: Windows absolute path DLL loading

15.41 NumPy 1.16.1 Release Notes

The NumPy 1.16.1 release fixes bugs reported against the 1.16.0 release, and also backports several enhancements from master that seem appropriate for a release series that is the last to support Python 2.7. The wheels on PyPI are linked with OpenBLAS v0.3.4+, which should fix the known threading issues found in previous OpenBLAS versions.

Downstream developers building this release should use Cython `>= 0.29.2` and, if using OpenBLAS, `OpenBLAS > v0.3.4`.

If you are installing using pip, you may encounter a problem with older installed versions of NumPy that pip did not delete becoming mixed with the current version, resulting in an `ImportError`. That problem is particularly common on Debian derived distributions due to a modified pip. The fix is to make sure all previous NumPy versions installed by pip have been removed. See [#12736](#) for discussion of the issue. Note that previously this problem resulted in an `AttributeError`.

15.41.1 Contributors

A total of 16 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Antoine Pitrou
- Arcesio Castaneda Medina +
- Charles Harris
- Chris Markiewicz +
- Christoph Gohlke
- Christopher J. Markiewicz +
- Daniel Hrisca +
- EelcoPeacs +
- Eric Wieser
- Kevin Sheppard
- Matti Pícus
- OBATA Akio +
- Ralf Gommers
- Sebastian Berg

- Stephan Hoyer
- Tyler Reddy

15.41.2 Enhancements

- [#12767](#): ENH: add `mm->q` `floordiv`
- [#12768](#): ENH: port `np.core.overrides` to C for speed
- [#12769](#): ENH: Add `np.ctypeslib.as_ctypes_type(dtype)`, improve `np.ctypeslib.as_ctypes`
- [#12773](#): ENH: add “max difference” messages to `np.testing.assert_array_equal...`
- [#12820](#): ENH: Add `mm->qm` `divmod`
- [#12890](#): ENH: add `_dtype_ctype` to namespace for freeze analysis

15.41.3 Compatibility notes

- The changed error message emitted by array comparison testing functions may affect doctests. See below for detail.
- Casting from double and single denormals to float16 has been corrected. In some rare cases, this may result in results being rounded up instead of down, changing the last bit (ULP) of the result.

15.41.4 New Features

divmod operation is now supported for two `timedelta64` operands

The `divmod` operator now handles two `np.timedelta64` operands, with type signature `mm->qm`.

15.41.5 Improvements

Further improvements to `ctypes` support in `np.ctypeslib`

A new `numpy.ctypeslib.as_ctypes_type` function has been added, which can be used to convert a *dtype* into a best-guess `ctypes` type. Thanks to this new function, `numpy.ctypeslib.as_ctypes` now supports a much wider range of array types, including structures, booleans, and integers of non-native endianness.

Array comparison assertions include maximum differences

Error messages from array comparison tests such as `np.testing.assert_allclose` now include “max absolute difference” and “max relative difference,” in addition to the previous “mismatch” percentage. This information makes it easier to update absolute and relative error tolerances.

15.41.6 Changes

`timedelta64 % 0` behavior adjusted to return `NaT`

The modulus operation with two `np.timedelta64` operands now returns `NaT` in the case of division by zero, rather than returning zero

15.42 NumPy 1.16.0 Release Notes

This NumPy release is the last one to support Python 2.7 and will be maintained as a long term release with bug fixes until 2020. Support for Python 3.4 been dropped, the supported Python versions are 2.7 and 3.5-3.7. The wheels on PyPI are linked with OpenBLAS v0.3.4+, which should fix the known threading issues found in previous OpenBLAS versions.

Downstream developers building this release should use Cython `>= 0.29` and, if using OpenBLAS, `OpenBLAS > v0.3.4`.

This release has seen a lot of refactoring and features many bug fixes, improved code organization, and better cross platform compatibility. Not all of these improvements will be visible to users, but they should help make maintenance easier going forward.

15.42.1 Highlights

- Experimental (opt-in only) support for overriding numpy functions, see `__array_function__` below.
- The `matmul` function is now a `ufunc`. This provides better performance and allows overriding with `__array_ufunc__`.
- Improved support for the ARM and POWER architectures.
- Improved support for AIX and PyPy.
- Improved interop with ctypes.
- Improved support for PEP 3118.

15.42.2 New functions

- New functions added to the `numpy.lib.recfunctions` module to ease the structured assignment changes:
 - `assign_fields_by_name`
 - `structured_to_unstructured`
 - `unstructured_to_structured`
 - `apply_along_fields`
 - `require_fields`

See the user guide at <https://docs.scipy.org/doc/numpy/user/basics.rec.html> for more info.

15.42.3 New deprecations

- The type dictionaries `numpy.core.typeNA` and `numpy.core.sctypeNA` are deprecated. They were buggy and not documented and will be removed in the 1.18 release. Use `numpy.sctypeDict` instead.
- The `numpy.asscalar` function is deprecated. It is an alias to the more powerful `numpy.ndarray.item`, not tested, and fails for scalars.
- The `numpy.set_array_ops` and `numpy.get_array_ops` functions are deprecated. As part of *NEP 15*, they have been deprecated along with the C-API functions `PyArray_SetNumericOps` and `PyArray_GetNumericOps`. Users who wish to override the inner loop functions in built-in ufuncs should use `PyUFunc_ReplaceLoopBySignature`.
- The `numpy.unravel_index` keyword argument `dims` is deprecated, use `shape` instead.
- The `numpy.histogram` `normed` argument is deprecated. It was deprecated previously, but no warning was issued.
- The positive operator (+) applied to non-numerical arrays is deprecated. See below for details.
- Passing an iterator to the stack functions is deprecated

15.42.4 Expired deprecations

- `NaT` comparisons now return `False` without a warning, finishing a deprecation cycle begun in NumPy 1.11.
- `np.lib.function_base.unique` was removed, finishing a deprecation cycle begun in NumPy 1.4. Use `numpy.unique` instead.
- multi-field indexing now returns views instead of copies, finishing a deprecation cycle begun in NumPy 1.7. The change was previously attempted in NumPy 1.14 but reverted until now.
- `np.PackageLoader` and `np.pkgload` have been removed. These were deprecated in 1.10, had no tests, and seem to no longer work in 1.15.

15.42.5 Future changes

- NumPy 1.17 will drop support for Python 2.7.

15.42.6 Compatibility notes

f2py script on Windows

On Windows, the installed script for running `f2py` is now an `.exe` file rather than a `*.py` file and should be run from the command line as `f2py` whenever the `Scripts` directory is in the path. Running `f2py` as a module `python -m numpy.f2py [...]` will work without path modification in any version of NumPy.

NaT comparisons

Consistent with the behavior of NaN, all comparisons other than inequality checks with datetime64 or timedelta64 NaT (“not-a-time”) values now always return `False`, and inequality checks with NaT now always return `True`. This includes comparisons between NaT values. For compatibility with the old behavior, use `np.isnat` to explicitly check for NaT or convert datetime64/timedelta64 arrays with `.astype(np.int64)` before making comparisons.

complex64/128 alignment has changed

The memory alignment of complex types is now the same as a C-struct composed of two floating point values, while before it was equal to the size of the type. For many users (for instance on x64/unix/gcc) this means that complex64 is now 4-byte aligned instead of 8-byte aligned. An important consequence is that aligned structured dtypes may now have a different size. For instance, `np.dtype('c8,u1', align=True)` used to have an itemsize of 16 (on x64/gcc) but now it is 12.

More in detail, the complex64 type now has the same alignment as a C-struct `struct {float r, i;}`, according to the compiler used to compile numpy, and similarly for the complex128 and complex256 types.

nd_grid __len__ removal

`len(np.mgrid)` and `len(np.ogrid)` are now considered nonsensical and raise a `TypeError`.

np.unravel_index now accepts shape keyword argument

Previously, only the `dims` keyword argument was accepted for specification of the shape of the array to be used for unraveling. `dims` remains supported, but is now deprecated.

multi-field views return a view instead of a copy

Indexing a structured array with multiple fields, e.g., `arr[['f1', 'f3']]`, returns a view into the original array instead of a copy. The returned view will often have extra padding bytes corresponding to intervening fields in the original array, unlike before, which will affect code such as `arr[['f1', 'f3']].view('float64')`. This change has been planned since numpy 1.7. Operations hitting this path have emitted `FutureWarnings` since then. Additional `FutureWarnings` about this change were added in 1.12.

To help users update their code to account for these changes, a number of functions have been added to the `numpy.lib.recfunctions` module which safely allow such operations. For instance, the code above can be replaced with `structured_to_unstructured(arr[['f1', 'f3']], dtype='float64')`. See the “accessing multiple fields” section of the [user guide](#).

15.42.7 C API changes

The `NPY_FEATURE_VERSION` was incremented to 0x0000D, due to the addition of:

- `PyUFuncObject.core_dim_flags`
- `PyUFuncObject.core_dim_sizes`
- `PyUFuncObject.identity_value`
- `PyUFunc_FromFuncAndDataAndSignatureAndIdentity`

15.42.8 New Features

Integrated squared error (ISE) estimator added to `histogram`

This method (`bins='stone'`) for optimizing the bin number is a generalization of the Scott's rule. The Scott's rule assumes the distribution is approximately Normal, while the ISE is a non-parametric method based on cross-validation.

`max_rows` keyword added for `np.loadtxt`

New keyword `max_rows` in `numpy.loadtxt` sets the maximum rows of the content to be read after `skiprows`, as in `numpy.genfromtxt`.

modulus operator support added for `np.timedelta64` operands

The modulus (remainder) operator is now supported for two operands of type `np.timedelta64`. The operands may have different units and the return value will match the type of the operands.

15.42.9 Improvements

no-copy pickling of numpy arrays

Up to protocol 4, numpy array pickling created 2 spurious copies of the data being serialized. With pickle protocol 5, and the `PickleBuffer` API, a large variety of numpy arrays can now be serialized without any copy using out-of-band buffers, and with one less copy using in-band buffers. This results, for large arrays, in an up to 66% drop in peak memory usage.

build shell independence

NumPy builds should no longer interact with the host machine shell directly. `exec_command` has been replaced with `subprocess.check_output` where appropriate.

`np.polynomial.Polynomial` classes render in LaTeX in Jupyter notebooks

When used in a front-end that supports it, *Polynomial* instances are now rendered through LaTeX. The current format is experimental, and is subject to change.

`randint` and `choice` now work on empty distributions

Even when no elements needed to be drawn, `np.random.randint` and `np.random.choice` raised an error when the arguments described an empty distribution. This has been fixed so that e.g. `np.random.choice([], 0) == np.array([], dtype=float64)`.

`linalg.lstsq`, `linalg.qr`, and `linalg.svd` now work with empty arrays

Previously, a `LinAlgError` would be raised when an empty matrix/empty matrices (with zero rows and/or columns) is/are passed in. Now outputs of appropriate shapes are returned.

Chain exceptions to give better error messages for invalid PEP3118 format strings

This should help track down problems.

Einsum optimization path updates and efficiency improvements

Einsum was synchronized with the current upstream work.

`numpy.angle` and `numpy.expand_dims` now work on ndarray subclasses

In particular, they now work for masked arrays.

`NPY_NO_DEPRECATED_API` compiler warning suppression

Setting `NPY_NO_DEPRECATED_API` to a value of 0 will suppress the current compiler warnings when the deprecated numpy API is used.

`np.diff` Added kwargs `prepend` and `append`

New kwargs `prepend` and `append`, allow for values to be inserted on either end of the differences. Similar to options for `ediff1d`. Now the inverse of `cumsum` can be obtained easily via `prepend=0`.

ARM support updated

Support for ARM CPUs has been updated to accommodate 32 and 64 bit targets, and also big and little endian byte ordering. AARCH32 memory alignment issues have been addressed. CI testing has been expanded to include AARCH64 targets via the services of shippable.com.

Appending to build flags

`numpy.distutils` has always overridden rather than appended to `LDFLAGS` and other similar such environment variables for compiling Fortran extensions. Now, if the `NPY_DISTUTILS_APPEND_FLAGS` environment variable is set to 1, the behavior will be appending. This applied to: `LDFLAGS`, `F77FLAGS`, `F90FLAGS`, `FREEFLAGS`, `FOPT`, `FDEBUG`, and `FFLAGS`. See [gh-11525](#) for more details.

Generalized ufunc signatures now allow fixed-size dimensions

By using a numerical value in the signature of a generalized ufunc, one can indicate that the given function requires input or output to have dimensions with the given size. E.g., the signature of a function that converts a polar angle to a two-dimensional cartesian unit vector would be $() \rightarrow (2)$; that for one that converts two spherical angles to a three-dimensional unit vector would be $() , () \rightarrow (3)$; and that for the cross product of two three-dimensional vectors would be $(3) , (3) \rightarrow (3)$.

Note that to the elementary function these dimensions are not treated any differently from variable ones indicated with a name starting with a letter; the loop still is passed the corresponding size, but it can now count on that size being equal to the fixed one given in the signature.

Generalized ufunc signatures now allow flexible dimensions

Some functions, in particular numpy's implementation of `@` as `matmul`, are very similar to generalized ufuncs in that they operate over core dimensions, but one could not present them as such because they were able to deal with inputs in which a dimension is missing. To support this, it is now allowed to postfix a dimension name with a question mark to indicate that the dimension does not necessarily have to be present.

With this addition, the signature for `matmul` can be expressed as $(m?, n) , (n, p?) \rightarrow (m?, p?)$. This indicates that if, e.g., the second operand has only one dimension, for the purposes of the elementary function it will be treated as if that input has core shape $(n, 1)$, and the output has the corresponding core shape of $(m, 1)$. The actual output array, however, has the flexible dimension removed, i.e., it will have shape $(..., m)$. Similarly, if both arguments have only a single dimension, the inputs will be presented as having shapes $(1, n)$ and $(n, 1)$ to the elementary function, and the output as $(1, 1)$, while the actual output array returned will have shape $()$. In this way, the signature allows one to use a single elementary function for four related but different signatures, $(m, n) , (n, p) \rightarrow (m, p)$, $(n) , (n, p) \rightarrow (p)$, $(m, n) , (n) \rightarrow (m)$ and $(n) , (n) \rightarrow ()$.

`np.clip` and the `clip` method check for memory overlap

The `out` argument to these functions is now always tested for memory overlap to avoid corrupted results when memory overlap occurs.

New value `unscaled` for option `cov` in `np.polyfit`

A further possible value has been added to the `cov` parameter of the `np.polyfit` function. With `cov='unscaled'` the scaling of the covariance matrix is disabled completely (similar to setting `absolute_sigma=True` in `scipy.optimize.curve_fit`). This would be useful in occasions, where the weights are given by $1/\sigma$ with σ being the (known) standard errors of (Gaussian distributed) data points, in which case the unscaled matrix is already a correct estimate for the covariance matrix.

Detailed docstrings for scalar numeric types

The `help` function, when applied to numeric types such as `numpy.intc`, `numpy.int_`, and `numpy.longlong`, now lists all of the aliased names for that type, distinguishing between platform -dependent and -independent aliases.

`__module__` attribute now points to public modules

The `__module__` attribute on most NumPy functions has been updated to refer to the preferred public module from which to access a function, rather than the module in which the function happens to be defined. This produces more informative displays for functions in tools such as IPython, e.g., instead of `<function 'numpy.core.fromnumeric.sum'>` you now see `<function 'numpy.sum'>`.

Large allocations marked as suitable for transparent hugepages

On systems that support transparent hugepages over the `madvise` system call numpy now marks that large memory allocations can be backed by hugepages which reduces page fault overhead and can in some fault heavy cases improve performance significantly. On Linux the setting for huge pages to be used, `/sys/kernel/mm/transparent_hugepage/enabled`, must be at least *madvise*. Systems which already have it set to *always* will not see much difference as the kernel will automatically use huge pages where appropriate.

Users of very old Linux kernels (~3.x and older) should make sure that `/sys/kernel/mm/transparent_hugepage/defrag` is not set to *always* to avoid performance problems due to concurrency issues in the memory defragmentation.

Alpine Linux (and other musl c library distros) support

We now default to use *fenv.h* for floating point status error reporting. Previously we had a broken default that sometimes would not report underflow, overflow, and invalid floating point operations. Now we can support non-glibc distributions like Alpine Linux as long as they ship *fenv.h*.

Speedup `np.block` for large arrays

Large arrays (greater than $512 * 512$) now use a blocking algorithm based on copying the data directly into the appropriate slice of the resulting array. This results in significant speedups for these large arrays, particularly for arrays being blocked along more than 2 dimensions.

`arr.ctypes.data_as(...)` holds a reference to `arr`

Previously the caller was responsible for keeping the array alive for the lifetime of the pointer.

Speedup `np.take` for read-only arrays

The implementation of `np.take` no longer makes an unnecessary copy of the source array when its `writable` flag is set to `False`.

Support path-like objects for more functions

The `np.core.records.fromfile` function now supports `pathlib.Path` and other path-like objects in addition to a file object. Furthermore, the `np.load` function now also supports path-like objects when using memory mapping (`mmap_mode` keyword argument).

Better behaviour of ufunc identities during reductions

Universal functions have an `.identity` which is used when `.reduce` is called on an empty axis.

As of this release, the logical binary ufuncs, `logical_and`, `logical_or`, and `logical_xor`, now have `identity`s of type `bool`, where previously they were of type `int`. This restores the 1.14 behavior of getting `bool`s when reducing empty object arrays with these ufuncs, while also keeping the 1.15 behavior of getting `int`s when reducing empty object arrays with arithmetic ufuncs like `add` and `multiply`.

Additionally, `logaddexp` now has an identity of `-inf`, allowing it to be called on empty sequences, where previously it could not be.

This is possible thanks to the new `PyUFunc_FromFuncAndDataAndSignatureAndIdentity`, which allows arbitrary values to be used as identities now.

Improved conversion from ctypes objects

Numpy has always supported taking a value or type from `ctypes` and converting it into an array or dtype, but only behaved correctly for simpler types. As of this release, this caveat is lifted - now:

- The `__pack__` attribute of `ctypes.Structure`, used to emulate C's `__attribute__((packed))`, is respected.
- Endianness of all `ctypes` objects is preserved
- `ctypes.Union` is supported
- Non-representable constructs raise exceptions, rather than producing dangerously incorrect results:
 - Bitfields are no longer interpreted as sub-arrays
 - Pointers are no longer replaced with the type that they point to

A new `ndpointer.contents` member

This matches the `.contents` member of normal `ctypes` arrays, and can be used to construct an `np.array` around the pointers contents. This replaces `np.array(some_nd_pointer)`, which stopped working in 1.15. As a side effect of this change, `ndpointer` now supports dtypes with overlapping fields and padding.

`matmul` is now a ufunc

`numpy.matmul` is now a ufunc which means that both the function and the `__matmul__` operator can now be overridden by `__array_ufunc__`. Its implementation has also changed. It uses the same BLAS routines as `numpy.dot`, ensuring its performance is similar for large matrices.

Start and stop arrays for `linspace`, `logspace` and `geomspace`

These functions used to be limited to scalar stop and start values, but can now take arrays, which will be properly broadcast and result in an output which has one axis prepended. This can be used, e.g., to obtain linearly interpolated points between sets of points.

CI extended with additional services

We now use additional free CI services, thanks to the companies that provide:

- Codecoverage testing via codecov.io
- Arm testing via shippable.com
- Additional test runs on azure pipelines

These are in addition to our continued use of travis, appveyor (for wheels) and LGTM

15.42.10 Changes

Comparison ufuncs will now error rather than return `NotImplemented`

Previously, comparison ufuncs such as `np.equal` would return *NotImplemented* if their arguments had structured dtypes, to help comparison operators such as `__eq__` deal with those. This is no longer needed, as the relevant logic has moved to the comparison operators proper (which thus do continue to return *NotImplemented* as needed). Hence, like all other ufuncs, the comparison ufuncs will now error on structured dtypes.

Positive will now raise a deprecation warning for non-numerical arrays

Previously, `+array` unconditionally returned a copy. Now, it will raise a `DeprecationWarning` if the array is not numerical (i.e., if `np.positive(array)` raises a `TypeError`. For `ndarray` subclasses that override the default `__array_ufunc__` implementation, the `TypeError` is passed on.

`NDArrayOperatorsMixin` now implements matrix multiplication

Previously, `np.lib.mixins.NDArrayOperatorsMixin` did not implement the special methods for Python's matrix multiplication operator (`@`). This has changed now that `matmul` is a ufunc and can be overridden using `__array_ufunc__`.

The scaling of the covariance matrix in `np.polyfit` is different

So far, `np.polyfit` used a non-standard factor in the scaling of the the covariance matrix. Namely, rather than using the standard $\text{chisq}/(M-N)$, it scaled it with $\text{chisq}/(M-N-2)$ where M is the number of data points and N is the number of parameters. This scaling is inconsistent with other fitting programs such as e.g. `scipy.optimize.curve_fit` and was changed to $\text{chisq}/(M-N)$.

`maximum` and `minimum` no longer emit warnings

As part of code introduced in 1.10, `float32` and `float64` set invalid float status when a `Nan` is encountered in `numpy.maximum` and `numpy.minimum`, when using SSE2 semantics. This caused a *RuntimeWarning* to sometimes be emitted. In 1.15 we fixed the inconsistencies which caused the warnings to become more conspicuous. Now no warnings will be emitted.

Umath and multiarray c-extension modules merged into a single module

The two modules were merged, according to [NEP 15](#). Previously `np.core.umath` and `np.core.multiarray` were separate c-extension modules. They are now python wrappers to the single `np.core/_multiarray_math` c-extension module.

`getfield` validity checks extended

`numpy.ndarray.getfield` now checks the dtype and offset arguments to prevent accessing invalid memory locations.

NumPy functions now support overrides with `__array_function__`

NumPy has a new experimental mechanism for overriding the implementation of almost all NumPy functions on non-NumPy arrays by defining an `__array_function__` method, as described in [NEP 18](#).

This feature is not yet been enabled by default, but has been released to facilitate experimentation by potential users. See the NEP for details on setting the appropriate environment variable. We expect the NumPy 1.17 release will enable overrides by default, which will also be more performant due to a new implementation written in C.

Arrays based off readonly buffers cannot be set writeable

We now disallow setting the `writable` flag `True` on arrays created from `fromstring(readonly-buffer)`.

15.43 NumPy 1.15.4 Release Notes

This is a bugfix release for bugs and regressions reported following the 1.15.3 release. The Python versions supported by this release are 2.7, 3.4-3.7. The wheels are linked with OpenBLAS v0.3.0, which should fix some of the linalg problems reported for NumPy 1.14.

15.43.1 Compatibility Note

The NumPy 1.15.x OS X wheels released on PyPI no longer contain 32-bit binaries. That will also be the case in future releases. See [#11625](#) for the related discussion. Those needing 32-bit support should look elsewhere or build from source.

15.43.2 Contributors

A total of 4 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Matti Pícus
- Sebastian Berg
- bbbbbbba +

15.43.3 Pull requests merged

A total of 4 pull requests were merged for this release.

- [#12296](#): BUG: Dealloc cached buffer info
- [#12297](#): BUG: Fix fill value in masked array ‘==’ and ‘!=’ ops.
- [#12307](#): DOC: Correct the default value of *optimize* in `numpy.einsum`
- [#12320](#): REL: Prepare for the NumPy 1.15.4 release

15.44 NumPy 1.15.3 Release Notes

This is a bugfix release for bugs and regressions reported following the 1.15.2 release. The Python versions supported by this release are 2.7, 3.4-3.7. The wheels are linked with OpenBLAS v0.3.0, which should fix some of the linalg problems reported for NumPy 1.14.

15.44.1 Compatibility Note

The NumPy 1.15.x OS X wheels released on PyPI no longer contain 32-bit binaries. That will also be the case in future releases. See [#11625](#) for the related discussion. Those needing 32-bit support should look elsewhere or build from source.

15.44.2 Contributors

A total of 7 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Allan Haldane
- Charles Harris
- Jeroen Demeyer
- Kevin Sheppard
- Matthew Bowden +
- Matti Pícus
- Tyler Reddy

15.44.3 Pull requests merged

A total of 12 pull requests were merged for this release.

- [#12080](#): MAINT: Blacklist some MSVC complex functions.
- [#12083](#): TST: Add azure CI testing to 1.15.x branch.
- [#12084](#): BUG: `test_path()` now uses `Path.resolve()`
- [#12085](#): TST, MAINT: Fix some failing tests on azure-pipelines mac and...
- [#12187](#): BUG: Fix memory leak in `mapping.c`
- [#12188](#): BUG: Allow boolean subtract in histogram
- [#12189](#): BUG: Fix in-place permutation

- [#12190](#): BUG: limit default for `get_num_build_jobs()` to 8
- [#12191](#): BUG: `OBJECT_to_*` should check for errors
- [#12192](#): DOC: Prepare for NumPy 1.15.3 release.
- [#12237](#): BUG: Fix `MaskedArray fill_value` type conversion.
- [#12238](#): TST: Backport azure-pipeline testing fixes for Mac

15.45 NumPy 1.15.2 Release Notes

This is a bugfix release for bugs and regressions reported following the 1.15.1 release.

- The matrix `PendingDeprecationWarning` is now suppressed in `pytest 3.8`.
- The new cached allocations machinery has been fixed to be thread safe.
- The boolean indexing of subclasses now works correctly.
- A small memory leak in `PyArray_AdaptFlexibleDType` has been fixed.

The Python versions supported by this release are 2.7, 3.4-3.7. The wheels are linked with OpenBLAS v0.3.0, which should fix some of the linalg problems reported for NumPy 1.14.

15.45.1 Compatibility Note

The NumPy 1.15.x OS X wheels released on PyPI no longer contain 32-bit binaries. That will also be the case in future releases. See [#11625](#) for the related discussion. Those needing 32-bit support should look elsewhere or build from source.

15.45.2 Contributors

A total of 4 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Julian Taylor
- Marten van Kerkwijk
- Matti Picus

15.45.3 Pull requests merged

A total of 4 pull requests were merged for this release.

- [#11902](#): BUG: Fix matrix `PendingDeprecationWarning` suppression for `pytest...`
- [#11981](#): BUG: fix cached allocations without the GIL for 1.15.x
- [#11982](#): BUG: fix refcount leak in `PyArray_AdaptFlexibleDType`
- [#11992](#): BUG: Ensure boolean indexing of subclasses sets base correctly.

15.46 NumPy 1.15.1 Release Notes

This is a bugfix release for bugs and regressions reported following the 1.15.0 release.

- The annoying but harmless RuntimeWarning that “numpy.dtype size changed” has been suppressed. The long standing suppression was lost in the transition to pytest.
- The update to Cython 0.28.3 exposed a problematic use of a gcc attribute used to prefer code size over speed in module initialization, possibly resulting in incorrect compiled code. This has been fixed in latest Cython but has been disabled here for safety.
- Support for big-endian and ARMv8 architectures has been improved.

The Python versions supported by this release are 2.7, 3.4-3.7. The wheels are linked with OpenBLAS v0.3.0, which should fix some of the linalg problems reported for NumPy 1.14.

15.46.1 Compatibility Note

The NumPy 1.15.x OS X wheels released on PyPI no longer contain 32-bit binaries. That will also be the case in future releases. See [#11625](#) for the related discussion. Those needing 32-bit support should look elsewhere or build from source.

15.46.2 Contributors

A total of 7 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Chris Billington
- Elliott Sales de Andrade +
- Eric Wieser
- Jeremy Manning +
- Matti Picus
- Ralf Gommers

15.46.3 Pull requests merged

A total of 24 pull requests were merged for this release.

- [#11647](#): MAINT: Filter Cython warnings in `__init__.py`
- [#11648](#): BUG: Fix doc source links to unwrap decorators
- [#11657](#): BUG: Ensure singleton dimensions are not dropped when converting...
- [#11661](#): BUG: Warn on Nan in minimum,maximum for scalars
- [#11665](#): BUG: cython sometimes emits invalid gcc attribute
- [#11682](#): BUG: Fix regression in `void_getitem`
- [#11698](#): BUG: Make `matrix_power` again work for object arrays.
- [#11700](#): BUG: Add missing `PyErr_NoMemory` after failing `malloc`
- [#11719](#): BUG: Fix undefined functions on big-endian systems.

- [#11720](#): MAINT: Make einsum optimize default to False.
- [#11746](#): BUG: Fix regression in loadtxt for bz2 text files in Python 2.
- [#11757](#): BUG: Revert use of *console_scripts*.
- [#11758](#): BUG: Fix Fortran kind detection for aarch64 & s390x.
- [#11759](#): BUG: Fix printing of longdouble on ppc64le.
- [#11760](#): BUG: Fixes for unicode field names in Python 2
- [#11761](#): BUG: Increase required cython version on python 3.7
- [#11763](#): BUG: check return value of `_buffer_format_string`
- [#11775](#): MAINT: Make `assert_array_compare` more generic.
- [#11776](#): TST: Fix `urlopen` stubbing.
- [#11777](#): BUG: Fix regression in `intersect1d`.
- [#11779](#): BUG: Fix test sensitive to platform byte order.
- [#11781](#): BUG: Avoid signed overflow in histogram
- [#11785](#): BUG: Fix pickle and memoryview for `datetime64`, `timedelta64` scalars
- [#11786](#): BUG: Deprecation triggers `segfault`

15.47 NumPy 1.15.0 Release Notes

NumPy 1.15.0 is a release with an unusual number of cleanups, many deprecations of old functions, and improvements to many existing functions. Please read the detailed descriptions below to see if you are affected.

For testing, we have switched to `pytest` as a replacement for the no longer maintained `nose` framework. The old `nose` based interface remains for downstream projects who may still be using it.

The Python versions supported by this release are 2.7, 3.4-3.7. The wheels are linked with OpenBLAS v0.3.0, which should fix some of the `linalg` problems reported for NumPy 1.14.

15.47.1 Highlights

- NumPy has switched to `pytest` for testing.
- A new `numpy.printoptions` context manager.
- Many improvements to the histogram functions.
- Support for unicode field names in python 2.7.
- Improved support for PyPy.
- Fixes and improvements to `numpy.einsum`.

15.47.2 New functions

- `numpy.gcd` and `numpy.lcm`, to compute the greatest common divisor and least common multiple.
- `numpy.ma.stack`, the `numpy.stack` array-joining function generalized to masked arrays.
- `numpy.quantile` function, an interface to `percentile` without factors of 100
- `numpy.nanquantile` function, an interface to `nanpercentile` without factors of 100
- `numpy.printoptions`, a context manager that sets print options temporarily for the scope of the `with` block:

```
>>> with np.printoptions(precision=2):  
...     print(np.array([2.0]) / 3)  
[0.67]
```

- `numpy.histogram_bin_edges`, a function to get the edges of the bins used by a histogram without needing to calculate the histogram.
- C functions `npy_get_floatstatus_barrier` and `npy_clear_floatstatus_barrier` have been added to deal with compiler optimization changing the order of operations. See below for details.

15.47.3 Deprecations

- Aliases of builtin `pickle` functions are deprecated, in favor of their unaliased `pickle.<func>` names:
 - `numpy.loads`
 - `numpy.core.numeric.load`
 - `numpy.core.numeric.loads`
 - `numpy.ma.loads`, `numpy.ma.dumps`
 - `numpy.ma.load`, `numpy.ma.dump` - these functions already failed on python 3 when called with a string.
- Multidimensional indexing with anything but a tuple is deprecated. This means that the index list in `ind = [slice(None), 0]`; `arr[ind]` should be changed to a tuple, e.g., `ind = [slice(None), 0]`; `arr[tuple(ind)]` or `arr[(slice(None), 0)]`. That change is necessary to avoid ambiguity in expressions such as `arr[[[0, 1], [0, 1]]]`, currently interpreted as `arr[array([0, 1]), array([0, 1])]`, that will be interpreted as `arr[array([[0, 1], [0, 1]])]` in the future.
- Imports from the following sub-modules are deprecated, they will be removed at some future date.
 - `numpy.testing.utils`
 - `numpy.testing.decorators`
 - `numpy.testing.nosetester`
 - `numpy.testing.noseclasses`
 - `numpy.core.umath_tests`
- Giving a generator to `numpy.sum` is now deprecated. This was undocumented behavior, but worked. Previously, it would calculate the sum of the generator expression. In the future, it might return a different result. Use `np.sum(np.from_iter(generator))` or the built-in Python `sum` instead.
- Users of the C-API should call `PyArrayResolveWriteBackIfCopy` or `PyArray_DiscardWritebackIfCopy` on any array with the `WRITEBACKIFCOPY` flag set, before deallocating the array. A deprecation warning will be emitted if those calls are not used when needed.

- Users of `nditer` should use the `nditer` object as a context manager anytime one of the iterator operands is writeable, so that numpy can manage writeback semantics, or should call `it.close()`. A *RuntimeWarning* may be emitted otherwise in these cases.
- The `normed` argument of `np.histogram`, deprecated long ago in 1.6.0, now emits a *DeprecationWarning*.

15.47.4 Future Changes

- NumPy 1.16 will drop support for Python 3.4.
- NumPy 1.17 will drop support for Python 2.7.

15.47.5 Compatibility notes

Compiled testing modules renamed and made private

The following compiled modules have been renamed and made private:

- `umath_tests` -> `_umath_tests`
- `test_rational` -> `_rational_tests`
- `multiarray_tests` -> `_multiarray_tests`
- `struct_ufunc_test` -> `_struct_ufunc_tests`
- `operand_flag_tests` -> `_operand_flag_tests`

The `umath_tests` module is still available for backwards compatibility, but will be removed in the future.

The `NpzFile` returned by `np.savez` is now a `collections.abc.Mapping`

This means it behaves like a readonly dictionary, and has a new `.values()` method and `len()` implementation.

For python 3, this means that `.iteritems()`, `.iterkeys()` have been deprecated, and `.keys()` and `.items()` now return views and not lists. This is consistent with how the builtin `dict` type changed between python 2 and python 3.

Under certain conditions, `nditer` must be used in a context manager

When using an `numpy.nditer` with the "writeonly" or "readwrite" flags, there are some circumstances where `nditer` doesn't actually give you a view of the writable array. Instead, it gives you a copy, and if you make changes to the copy, `nditer` later writes those changes back into your actual array. Currently, this writeback occurs when the array objects are garbage collected, which makes this API error-prone on CPython and entirely broken on PyPy. Therefore, `nditer` should now be used as a context manager whenever it is used with writeable arrays, e.g., with `np.nditer(...)` as `it: ...`. You may also explicitly call `it.close()` for cases where a context manager is unusable, for instance in generator expressions.

Numpy has switched to using pytest instead of nose for testing

The last nose release was 1.3.7 in June, 2015, and development of that tool has ended, consequently NumPy has now switched to using pytest. The old decorators and nose tools that were previously used by some downstream projects remain available, but will not be maintained. The standard testing utilities, `assert_almost_equal` and such, are not be affected by this change except for the nose specific functions `import_nose` and `raises`. Those functions are not used in numpy, but are kept for downstream compatibility.

Numpy no longer monkey-patches ctypes with `__array_interface__`

Previously numpy added `__array_interface__` attributes to all the integer types from ctypes.

`np.ma.notmasked_contiguous` and `np.ma.flatnotmasked_contiguous` always return lists

This is the documented behavior, but previously the result could be any of slice, None, or list.

All downstream users seem to check for the None result from `flatnotmasked_contiguous` and replace it with `[]`. Those callers will continue to work as before.

`np.squeeze` restores old behavior of objects that cannot handle an `axis` argument

Prior to version 1.7.0, `numpy.squeeze` did not have an `axis` argument and all empty axes were removed by default. The incorporation of an `axis` argument made it possible to selectively squeeze single or multiple empty axes, but the old API expectation was not respected because axes could still be selectively removed (silent success) from an object expecting all empty axes to be removed. That silent, selective removal of empty axes for objects expecting the old behavior has been fixed and the old behavior restored.

unstructured void array's `.item` method now returns a bytes object

`.item` now returns a `bytes` object instead of a buffer or byte array. This may affect code which assumed the return value was mutable, which is no longer the case.

`copy.copy` and `copy.deepcopy` no longer turn masked into an array

Since `np.ma.masked` is a readonly scalar, copying should be a no-op. These functions now behave consistently with `np.copy()`.

Multifield Indexing of Structured Arrays will still return a copy

The change that multi-field indexing of structured arrays returns a view instead of a copy is pushed back to 1.16. A new method `numpy.lib.recfunctions.repack_fields` has been introduced to help mitigate the effects of this change, which can be used to write code compatible with both numpy 1.15 and 1.16. For more information on how to update code to account for this future change see the “accessing multiple fields” section of the [user guide](#).

15.47.6 C API changes

New functions `numpy_get_floatstatus_barrier` and `numpy_clear_floatstatus_barrier`

Functions `numpy_get_floatstatus_barrier` and `numpy_clear_floatstatus_barrier` have been added and should be used in place of the `numpy_get_floatstatus``` and ```numpy_clear_status` functions. Optimizing compilers like GCC 8.1 and Clang were rearranging the order of operations when the previous functions were used in the ufunc SIMD functions, resulting in the floatstatus flags being checked before the operation whose status we wanted to check was run. See [#10339](#).

Changes to `PyArray_GetDTypeTransferFunction`

`PyArray_GetDTypeTransferFunction` now defaults to using user-defined `copyswapn / copyswap` for user-defined dtypes. If this causes a significant performance hit, consider implementing `copyswapn` to reflect the implementation of `PyArray_GetStridedCopyFn`. See [#10898](#).

15.47.7 New Features

`np.gcd` and `np.lcm` ufuncs added for integer and objects types

These compute the greatest common divisor, and lowest common multiple, respectively. These work on all the numpy integer types, as well as the builtin arbitrary-precision `Decimal` and `long` types.

Support for cross-platform builds for iOS

The build system has been modified to add support for the `_PYTHON_HOST_PLATFORM` environment variable, used by `distutils` when compiling on one platform for another platform. This makes it possible to compile NumPy for iOS targets.

This only enables you to compile NumPy for one specific platform at a time. Creating a full iOS-compatible NumPy package requires building for the 5 architectures supported by iOS (i386, x86_64, armv7, armv7s and arm64), and combining these 5 compiled builds products into a single “fat” binary.

`return_indices` keyword added for `np.intersect1d`

New keyword `return_indices` returns the indices of the two input arrays that correspond to the common elements.

`np.quantile` and `np.nanquantile`

Like `np.percentile` and `np.nanpercentile`, but takes quantiles in `[0, 1]` rather than percentiles in `[0, 100]`. `np.percentile` is now a thin wrapper around `np.quantile` with the extra step of dividing by 100.

Build system

Added experimental support for the 64-bit RISC-V architecture.

15.47.8 Improvements

`np.einsum` updates

Syncs einsum path optimization tech between `numpy` and `opt_einsum`. In particular, the *greedy* path has received many enhancements by @jcmgray. A full list of issues fixed are:

- Arbitrary memory can be passed into the *greedy* path. Fixes gh-11210.
- The greedy path has been updated to contain more dynamic programming ideas preventing a large number of duplicate (and expensive) calls that figure out the actual pair contraction that takes place. Now takes a few seconds on several hundred input tensors. Useful for matrix product state theories.
- Reworks the broadcasting dot error catching found in gh-11218 gh-10352 to be a bit earlier in the process.
- Enhances the *can_dot* functionality that previous missed an edge case (part of gh-11308).

`np.ufunc.reduce` and related functions now accept an initial value

`np.ufunc.reduce`, `np.sum`, `np.prod`, `np.min` and `np.max` all now accept an `initial` keyword argument that specifies the value to start the reduction with.

`np.flip` can operate over multiple axes

`np.flip` now accepts `None`, or tuples of `int`, in its `axis` argument. If `axis` is `None`, it will flip over all the axes.

`histogram` and `histogramdd` functions have moved to `np.lib.histograms`

These were originally found in `np.lib.function_base`. They are still available under their unscoped `np.histogram(dd)` names, and to maintain compatibility, aliased at `np.lib.function_base.histogram(dd)`.

Code that does `from np.lib.function_base import *` will need to be updated with the new location, and should consider not using `import *` in future.

`histogram` will accept NaN values when explicit bins are given

Previously it would fail when trying to compute a finite range for the data. Since the range is ignored anyway when the bins are given explicitly, this error was needless.

Note that calling `histogram` on NaN values continues to raise the `RuntimeWarning`s typical of working with nan values, which can be silenced as usual with `errstate`.

histogram works on datetime types, when explicit bin edges are given

Dates, times, and timedeltas can now be histogrammed. The bin edges must be passed explicitly, and are not yet computed automatically.

histogram “auto” estimator handles limited variance better

No longer does an IQR of 0 result in `n_bins=1`, rather the number of bins chosen is related to the data size in this situation.

The edges returned by *histogram*’ and *histogramdd* now match the data float type

When passed `np.float16`, `np.float32`, or `np.longdouble` data, the returned edges are now of the same dtype. Previously, *histogram* would only return the same type if explicit bins were given, and *histogram* would produce `float64` bins no matter what the inputs.

histogramdd allows explicit ranges to be given in a subset of axes

The `range` argument of `numpy.histogramdd` can now contain `None` values to indicate that the range for the corresponding axis should be computed from the data. Previously, this could not be specified on a per-axis basis.

The normed arguments of *histogramdd* and *histogram2d* have been renamed

These arguments are now called `density`, which is consistent with *histogram*. The old argument continues to work, but the new name should be preferred.

np.r_ works with 0d arrays, and np.ma.mr_ works with np.ma.masked

0d arrays passed to the `r_` and `mr_` concatenation helpers are now treated as though they are arrays of length 1. Previously, passing these was an error. As a result, `numpy.ma.mr_` now works correctly on the `masked` constant.

np.ptp accepts a keepdims argument, and extended axis tuples

`np.ptp` (peak-to-peak) can now work over multiple axes, just like `np.max` and `np.min`.

MaskedArray.astype now is identical to ndarray.astype

This means it takes all the same arguments, making more code written for `ndarray` work for masked array too.

Enable AVX2/AVX512 at compile time

Change to `simd.inc.src` to allow use of AVX2 or AVX512 at compile time. Previously compilation for avx2 (or 512) with `-march=native` would still use the SSE code for the `simd` functions even when the rest of the code got AVX2.

`nan_to_num` always returns scalars when receiving scalar or 0d inputs

Previously an array was returned for integer scalar inputs, which is inconsistent with the behavior for float inputs, and that of `ufuncs` in general. For all types of scalar or 0d input, the result is now a scalar.

`np.flatnonzero` works on numpy-convertible types

`np.flatnonzero` now uses `np.ravel(a)` instead of `a.ravel()`, so it works for lists, tuples, etc.

`np.interp` returns numpy scalars rather than builtin scalars

Previously `np.interp(0.5, [0, 1], [10, 20])` would return a `float`, but now it returns a `np.float64` object, which more closely matches the behavior of other functions.

Additionally, the special case of `np.interp(object_array_0d, ...)` is no longer supported, as `np.interp(object_array_nd)` was never supported anyway.

As a result of this change, the `period` argument can now be used on 0d arrays.

Allow dtype field names to be unicode in Python 2

Previously `np.dtype([(u'name', float)])` would raise a `TypeError` in Python 2, as only bytestrings were allowed in field names. Now any unicode string field names will be encoded with the `ascii` codec, raising a `UnicodeEncodeError` upon failure.

This change makes it easier to write Python 2/3 compatible code using `from __future__ import unicode_literals`, which previously would cause string literal field names to raise a `TypeError` in Python 2.

Comparison ufuncs accept `dtype=object`, overriding the default `bool`

This allows object arrays of symbolic types, which override `==` and other operators to return expressions, to be compared elementwise with `np.equal(a, b, dtype=object)`.

sort functions accept `kind='stable'`

Up until now, to perform a stable sort on the data, the user must do:

```
>>> np.sort([5, 2, 6, 2, 1], kind='mergesort')
[1, 2, 2, 5, 6]
```

because merge sort is the only stable sorting algorithm available in NumPy. However, having `kind='mergesort'` does not make it explicit that the user wants to perform a stable sort thus harming the readability.

This change allows the user to specify `kind='stable'` thus clarifying the intent.

Do not make temporary copies for in-place accumulation

When ufuncs perform accumulation they no longer make temporary copies because of the overlap between input and output, that is, the next element accumulated is added before the accumulated result is stored in its place, hence the overlap is safe. Avoiding the copy results in faster execution.

`linalg.matrix_power` can now handle stacks of matrices

Like other functions in `linalg`, `matrix_power` can now deal with arrays of dimension larger than 2, which are treated as stacks of matrices. As part of the change, to further improve consistency, the name of the first argument has been changed to `a` (from `M`), and the exceptions for non-square matrices have been changed to `LinAlgError` (from `ValueError`).

Increased performance in `random.permutation` for multidimensional arrays

`permutation` uses the fast path in `random.shuffle` for all input array dimensions. Previously the fast path was only used for 1-d arrays.

Generalized ufuncs now accept `axes`, `axis` and `keepdims` arguments

One can control over which axes a generalized ufunc operates by passing in an `axes` argument, a list of tuples with indices of particular axes. For instance, for a signature of $(i, j), (j, k) \rightarrow (i, k)$ appropriate for matrix multiplication, the base elements are two-dimensional matrices and these are taken to be stored in the two last axes of each argument. The corresponding axes keyword would be `[(-2, -1), (-2, -1), (-2, -1)]`. If one wanted to use leading dimensions instead, one would pass in `[(0, 1), (0, 1), (0, 1)]`.

For simplicity, for generalized ufuncs that operate on 1-dimensional arrays (vectors), a single integer is accepted instead of a single-element tuple, and for generalized ufuncs for which all outputs are scalars, the (empty) output tuples can be omitted. Hence, for a signature of $(i), (i) \rightarrow ()$ appropriate for an inner product, one could pass in `axes=[0, 0]` to indicate that the vectors are stored in the first dimensions of the two inputs arguments.

As a short-cut for generalized ufuncs that are similar to reductions, i.e., that act on a single, shared core dimension such as the inner product example above, one can pass an `axis` argument. This is equivalent to passing in `axes` with identical entries for all arguments with that core dimension (e.g., for the example above, `axes=[(axis,), (axis,)]`).

Furthermore, like for reductions, for generalized ufuncs that have inputs that all have the same number of core dimensions and outputs with no core dimension, one can pass in `keepdims` to leave a dimension with size 1 in the outputs, thus allowing proper broadcasting against the original inputs. The location of the extra dimension can be controlled with `axes`. For instance, for the inner-product example, `keepdims=True, axes=[-2, -2, -2]` would act on the inner-product example, `keepdims=True, axis=-2` would act on the one-but-last dimension of the input arguments, and leave a size 1 dimension in that place in the output.

float128 values now print correctly on ppc systems

Previously printing float128 values was buggy on ppc, since the special double-double floating-point-format on these systems was not accounted for. float128s now print with correct rounding and uniqueness.

Warning to ppc users: You should upgrade glibc if it is version ≤ 2.23 , especially if using float128. On ppc, glibc's malloc in these version often misaligns allocated memory which can crash numpy when using float128 values.

New `np.take_along_axis` and `np.put_along_axis` functions

When used on multidimensional arrays, `argsort`, `argmin`, `argmax`, and `argpartition` return arrays that are difficult to use as indices. `take_along_axis` provides an easy way to use these indices to lookup values within an array, so that:

```
np.take_along_axis(a, np.argsort(a, axis=axis), axis=axis)
```

is the same as:

```
np.sort(a, axis=axis)
```

`np.put_along_axis` acts as the dual operation for writing to these indices within an array.

15.48 NumPy 1.14.6 Release Notes

This is a bugfix release for bugs reported following the 1.14.5 release. The most significant fixes are:

- Fix for behavior change in `ma.masked_values(shrink=True)`
- Fix the new cached allocations machinery to be thread safe.

The Python versions supported in this release are 2.7 and 3.4 - 3.7. The Python 3.6 wheels on PyPI should be compatible with all Python 3.6 versions.

15.48.1 Contributors

A total of 4 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Eric Wieser
- Julian Taylor
- Matti Picus

15.48.2 Pull requests merged

A total of 4 pull requests were merged for this release.

- [#11985](#): BUG: fix cached allocations without the GIL
- [#11986](#): BUG: Undo behavior change in `ma.masked_values(shrink=True)`
- [#11987](#): BUG: fix refcount leak in `PyArray_AdaptFlexibleDType`
- [#11995](#): TST: Add Python 3.7 testing to NumPy 1.14.

15.49 NumPy 1.14.5 Release Notes

This is a bugfix release for bugs reported following the 1.14.4 release. The most significant fixes are:

- fixes for compilation errors on alpine and NetBSD

The Python versions supported in this release are 2.7 and 3.4 - 3.6. The Python 3.6 wheels available from PIP are built with Python 3.6.2 and should be compatible with all previous versions of Python 3.6. The source releases were cythonized with Cython 0.28.2 and should work for the upcoming Python 3.7.

15.49.1 Contributors

A total of 1 person contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris

15.49.2 Pull requests merged

A total of 2 pull requests were merged for this release.

- [#11274](#): BUG: Correct use of NPY_UNUSED.
- [#11294](#): BUG: Remove extra trailing parentheses.

15.50 NumPy 1.14.4 Release Notes

This is a bugfix release for bugs reported following the 1.14.3 release. The most significant fixes are:

- fixes for compiler instruction reordering that resulted in NaN’s not being properly propagated in *np.max* and *np.min*,
- fixes for bus faults on SPARC and older ARM due to incorrect alignment checks.

There are also improvements to printing of long doubles on PPC platforms. All is not yet perfect on that platform, the whitespace padding is still incorrect and is to be fixed in numpy 1.15, consequently NumPy still fails some printing-related (and other) unit tests on ppc systems. However, the printed values are now correct.

Note that NumPy will error on import if it detects incorrect float32 *dot* results. This problem has been seen on the Mac when working in the Anaconda environment and is due to a subtle interaction between MKL and PyQt5. It is not strictly a NumPy problem, but it is best that users be aware of it. See the [gh-8577](#) NumPy issue for more information.

The Python versions supported in this release are 2.7 and 3.4 - 3.6. The Python 3.6 wheels available from PIP are built with Python 3.6.2 and should be compatible with all previous versions of Python 3.6. The source releases were cythonized with Cython 0.28.2 and should work for the upcoming Python 3.7.

15.50.1 Contributors

A total of 7 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Allan Haldane
- Charles Harris
- Marten van Kerkwijk
- Matti Picus

- Pauli Virtanen
- Ryan Soklaski +
- Sebastian Berg

15.50.2 Pull requests merged

A total of 11 pull requests were merged for this release.

- [#11104](#): BUG: str of DOUBLE_DOUBLE format wrong on ppc64
- [#11170](#): TST: linalg: add regression test for gh-8577
- [#11174](#): MAINT: add sanity-checks to be run at import time
- [#11181](#): BUG: void dtype setup checked offset not actual pointer for alignment
- [#11194](#): BUG: Python2 doubles don't print correctly in interactive shell.
- [#11198](#): BUG: optimizing compilers can reorder call to `numpy.get_floatstatus`
- [#11199](#): BUG: reduce using SSE only warns if inside SSE loop
- [#11203](#): BUG: Bytes delimiter/comments in `genfromtxt` should be decoded
- [#11211](#): BUG: Fix reference count/memory leak exposed by better testing
- [#11219](#): BUG: Fixes einsum broadcasting bug when `optimize=True`
- [#11251](#): DOC: Document 1.14.4 release.

15.51 NumPy 1.14.3 Release Notes

This is a bugfix release for a few bugs reported following the 1.14.2 release:

- `np.lib.recfunctions.fromrecords` accepts a list-of-lists, until 1.15
- In python2, float types use the new print style when printing to a file
- style arg in “legacy” print mode now works for 0d arrays

The Python versions supported in this release are 2.7 and 3.4 - 3.6. The Python 3.6 wheels available from PIP are built with Python 3.6.2 and should be compatible with all previous versions of Python 3.6. The source releases were cythonized with Cython 0.28.2.

15.51.1 Contributors

A total of 6 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Allan Haldane
- Charles Harris
- Jonathan March +
- Malcolm Smith +
- Matti Picus
- Pauli Virtanen

15.51.2 Pull requests merged

A total of 8 pull requests were merged for this release.

- [#10862](#): BUG: floating types should override tp_print (1.14 backport)
- [#10905](#): BUG: for 1.14 back-compat, accept list-of-lists in fromrecords
- [#10947](#): BUG: ‘style’ arg to array2string broken in legacy mode (1.14...
- [#10959](#): BUG: test, fix for missing flags[“WRITEBACKIFCOPY”] key
- [#10960](#): BUG: Add missing underscore to prototype in check_embedded_lapack
- [#10961](#): BUG: Fix encoding regression in ma/bench.py (Issue #10868)
- [#10962](#): BUG: core: fix NPY_TITLE_KEY macro on pypy
- [#10974](#): BUG: test, fix PyArray_DiscardWritebackIfCopy...

15.52 NumPy 1.14.2 Release Notes

This is a bugfix release for some bugs reported following the 1.14.1 release. The major problems dealt with are as follows.

- Residual bugs in the new array printing functionality.
- Regression resulting in a relocation problem with shared library.
- Improved PyPy compatibility.

The Python versions supported in this release are 2.7 and 3.4 - 3.6. The Python 3.6 wheels available from PIP are built with Python 3.6.2 and should be compatible with all previous versions of Python 3.6. The source releases were cythonized with Cython 0.26.1, which is known to **not** support the upcoming Python 3.7 release. People who wish to run Python 3.7 should check out the NumPy repo and try building with the, as yet, unreleased master branch of Cython.

15.52.1 Contributors

A total of 4 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Allan Haldane
- Charles Harris
- Eric Wieser
- Pauli Virtanen

15.52.2 Pull requests merged

A total of 5 pull requests were merged for this release.

- [#10674](#): BUG: Further back-compat fix for subclassed array repr
- [#10725](#): BUG: dragon4 fractional output mode adds too many trailing zeros
- [#10726](#): BUG: Fix f2py generated code to work on PyPy
- [#10727](#): BUG: Fix missing NPY_VISIBILITY_HIDDEN on npy_longdouble_to_PyLong
- [#10729](#): DOC: Create 1.14.2 notes and changelog.

15.53 NumPy 1.14.1 Release Notes

This is a bugfix release for some problems reported following the 1.14.0 release. The major problems fixed are the following.

- Problems with the new array printing, particularly the printing of complex values, Please report any additional problems that may turn up.
- Problems with `np.einsum` due to the new `optimized=True` default. Some fixes for optimization have been applied and `optimize=False` is now the default.
- The sort order in `np.unique` when `axis=<some-number>` will now always be lexicographic in the subarray elements. In previous NumPy versions there was an optimization that could result in sorting the subarrays as unsigned byte strings.
- The change in 1.14.0 that multi-field indexing of structured arrays returns a view instead of a copy has been reverted but remains on track for NumPy 1.15. Affected users should read the 1.14.1 Numpy User Guide section “basics/structured arrays/accessing multiple fields” for advice on how to manage this transition.

The Python versions supported in this release are 2.7 and 3.4 - 3.6. The Python 3.6 wheels available from PIP are built with Python 3.6.2 and should be compatible with all previous versions of Python 3.6. The source releases were cythonized with Cython 0.26.1, which is known to **not** support the upcoming Python 3.7 release. People who wish to run Python 3.7 should check out the NumPy repo and try building with the, as yet, unreleased master branch of Cython.

15.53.1 Contributors

A total of 14 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Allan Haldane
- Charles Harris
- Daniel Smith
- Dennis Weyland +
- Eric Larson
- Eric Wieser
- Jarrod Millman
- Kenichi Maehashi +
- Marten van Kerkwijk
- Mathieu Lamarre
- Sebastian Berg
- Simon Conseil
- Simon Gibbons
- xoviat

15.53.2 Pull requests merged

A total of 36 pull requests were merged for this release.

- [#10339](#): BUG: restrict the `__config__` modifications to win32
- [#10368](#): MAINT: Adjust type promotion in `linalg.norm`
- [#10375](#): BUG: add missing paren and remove quotes from repr of fieldless...
- [#10395](#): MAINT: Update download URL in `setup.py`.
- [#10396](#): BUG: fix einsum issue with unicode input and py2
- [#10397](#): BUG: fix error message not formatted in einsum
- [#10398](#): DOC: add documentation about how to handle new array printing
- [#10403](#): BUG: Set einsum optimize parameter default to *False*.
- [#10424](#): ENH: Fix repr of `np.record` objects to match `np.void` types [#10412](#)
- [#10425](#): MAINT: Update zesty to artful for i386 testing
- [#10431](#): REL: Add 1.14.1 release notes template
- [#10435](#): MAINT: Use `ValueError` for duplicate field names in lookup (backport)
- [#10534](#): BUG: Provide a better error message for out-of-order fields
- [#10536](#): BUG: Resize bytes columns in `genfromtxt` (backport of [#10401](#))
- [#10537](#): BUG: multifield-indexing adds padding bytes: revert for 1.14.1
- [#10539](#): BUG: fix `np.save` issue with python 2.7.5
- [#10540](#): BUG: Add missing `DECREF` in Py2 `int()` cast
- [#10541](#): TST: Add circleci document testing to maintenance/1.14.x
- [#10542](#): BUG: complex repr has extra spaces, missing + (1.14 backport)
- [#10550](#): BUG: Set missing exception after `malloc`
- [#10557](#): BUG: In `numpy.i`, clear `CARRAY` flag if wrapped buffer is not `C_CONTIGUOUS`.
- [#10558](#): DEP: Issue `FutureWarning` when malformed records detected.
- [#10559](#): BUG: Fix einsum optimize logic for singleton dimensions
- [#10560](#): BUG: Fix calling ufuncs with a positional output argument.
- [#10561](#): BUG: Fix various Big-Endian test failures (ppc64)
- [#10562](#): BUG: Make `dtype.descr` error for out-of-order fields.
- [#10563](#): BUG: arrays not being flattened in *union1d*
- [#10607](#): MAINT: Update sphinxext submodule hash.
- [#10608](#): BUG: Revert sort optimization in `np.unique`.
- [#10609](#): BUG: infinite recursion in str of `0d` subclasses
- [#10610](#): BUG: Align type definition with generated lapack
- [#10612](#): BUG/ENH: Improve output for structured non-void types
- [#10622](#): BUG: deallocate recursive closure in `arrayprint.py` (1.14 backport)
- [#10624](#): BUG: Correctly identify comma separated dtype strings

- [#10629](#): BUG: deallocate recursive closure in arrayprint.py (backport...
- [#10630](#): REL: Prepare for 1.14.1 release.

15.54 NumPy 1.14.0 Release Notes

NumPy 1.14.0 is the result of seven months of work and contains a large number of bug fixes and new features, along with several changes with potential compatibility issues. The major change that users will notice are the stylistic changes in the way numpy arrays and scalars are printed, a change that will affect doctests. See below for details on how to preserve the old style printing when needed.

A major decision affecting future development concerns the schedule for dropping Python 2.7 support in the runup to 2020. The decision has been made to support 2.7 for all releases made in 2018, with the last release being designated a long term release with support for bug fixes extending through 2019. In 2019 support for 2.7 will be dropped in all new releases. More details can be found in [NEP 12](#).

This release supports Python 2.7 and 3.4 - 3.6.

15.54.1 Highlights

- The `np.einsum` function uses BLAS when possible
- `genfromtxt`, `loadtxt`, `fromregex` and `savetxt` can now handle files with arbitrary Python supported encoding.
- Major improvements to printing of NumPy arrays and scalars.

15.54.2 New functions

- `parametrize`: decorator added to `numpy.testing`
- `chebinterpolate`: Interpolate function at Chebyshev points.
- `format_float_positional` and `format_float_scientific`: format floating-point scalars unambiguously with control of rounding and padding.
- `PyArray_ResolveWritebackIfCopy` and `PyArray_SetWritebackIfCopyBase`, new C-API functions useful in achieving PyPy compatibility.

15.54.3 Deprecations

- Using `np.bool_` objects in place of integers is deprecated. Previously `operator.index(np.bool_)` was legal and allowed constructs such as `[1, 2, 3][np.True_]`. That was misleading, as it behaved differently from `np.array([1, 2, 3])[np.True_]`.
- Truth testing of an empty array is deprecated. To check if an array is not empty, use `array.size > 0`.
- Calling `np.bincount` with `minlength=None` is deprecated. `minlength=0` should be used instead.
- Calling `np.fromstring` with the default value of the `sep` argument is deprecated. When that argument is not provided, a broken version of `np.frombuffer` is used that silently accepts unicode strings and – after encoding them as either utf-8 (python 3) or the default encoding (python 2) – treats them as binary data. If reading binary data is desired, `np.frombuffer` should be used directly.
- The `style` option of `array2string` is deprecated in non-legacy printing mode.

- `PyArray_SetUpdateIfCopyBase` has been deprecated. For NumPy versions ≥ 1.14 use `PyArray_SetWritebackIfCopyBase` instead, see *C API changes* below for more details.
- The use of `UPDATEIFCOPY` arrays is deprecated, see *C API changes* below for details. We will not be dropping support for those arrays, but they are not compatible with PyPy.

15.54.4 Future Changes

- `np.issubdtype` will stop downcasting dtype-like arguments. It might be expected that `issubdtype(np.float32, 'float64')` and `issubdtype(np.float32, np.float64)` mean the same thing - however, there was an undocumented special case that translated the former into `issubdtype(np.float32, np.floating)`, giving the surprising result of `True`.

This translation now gives a warning that explains what translation is occurring. In the future, the translation will be disabled, and the first example will be made equivalent to the second.

- `np.linalg.lstsq` default for `rcond` will be changed. The `rcond` parameter to `np.linalg.lstsq` will change its default to machine precision times the largest of the input array dimensions. A `FutureWarning` is issued when `rcond` is not passed explicitly.
- `a.flat.__array__()` will return a writeable copy of `a` when `a` is non-contiguous. Previously it returned an `UPDATEIFCOPY` array when `a` was writeable. Currently it returns a non-writeable copy. See [gh-7054](#) for a discussion of the issue.
- Unstructured void array's `.item` method will return a bytes object. In the future, calling `.item()` on arrays or scalars of `np.void` datatype will return a `bytes` object instead of a buffer or int array, the same as returned by `bytes(void_scalar)`. This may affect code which assumed the return value was mutable, which will no longer be the case. A `FutureWarning` is now issued when this would occur.

15.54.5 Compatibility notes

The mask of a masked array view is also a view rather than a copy

There was a `FutureWarning` about this change in NumPy 1.11.x. In short, it is now the case that, when changing a view of a masked array, changes to the mask are propagated to the original. That was not previously the case. This change affects slices in particular. Note that this does not yet work properly if the mask of the original array is `nomask` and the mask of the view is changed. See [gh-5580](#) for an extended discussion. The original behavior of having a copy of the mask can be obtained by calling the `unshare_mask` method of the view.

`np.ma.masked` is no longer writeable

Attempts to mutate the masked constant now error, as the underlying arrays are marked readonly. In the past, it was possible to get away with:

```
# emulating a function that sometimes returns np.ma.masked
val = random.choice([np.ma.masked, 10])
var_arr = np.asarray(val)
var_arr += 1 # now errors, previously changed np.ma.masked.data
```

np.ma functions producing fill_value s have changed

Previously, `np.ma.default_fill_value` would return a 0d array, but `np.ma.minimum_fill_value` and `np.ma.maximum_fill_value` would return a tuple of the fields. Instead, all three methods return a structured `np.void` object, which is what you would already find in the `.fill_value` attribute.

Additionally, the dtype guessing now matches that of `np.array` - so when passing a python scalar `x`, `maximum_fill_value(x)` is always the same as `maximum_fill_value(np.array(x))`. Previously `x = long(1)` on Python 2 violated this assumption.

a.flat.__array__() returns non-writeable arrays when a is non-contiguous

The intent is that the `UPDATEIFCOPY` array previously returned when `a` was non-contiguous will be replaced by a writeable copy in the future. This temporary measure is aimed to notify folks who expect the underlying array be modified in this situation that that will no longer be the case. The most likely places for this to be noticed is when expressions of the form `np.asarray(a.flat)` are used, or when `a.flat` is passed as the `out` parameter to a ufunc.

np.tensordot now returns zero array when contracting over 0-length dimension

Previously `np.tensordot` raised a `ValueError` when contracting over 0-length dimension. Now it returns a zero array, which is consistent with the behaviour of `np.dot` and `np.einsum`.

numpy.testing reorganized

This is not expected to cause problems, but possibly something has been left out. If you experience an unexpected import problem using `numpy.testing` let us know.

np.asfarray no longer accepts non-dtypes through the dtype argument

This previously would accept `dtype=some_array`, with the implied semantics of `dtype=some_array.dtype`. This was undocumented, unique across the numpy functions, and if used would likely correspond to a typo.

1D np.linalg.norm preserves float input types, even for arbitrary orders

Previously, this would promote to `float64` when arbitrary orders were passed, despite not doing so under the simple cases:

```
>>> f32 = np.float32([[1, 2]])
>>> np.linalg.norm(f32, 2.0, axis=-1).dtype
dtype('float32')
>>> np.linalg.norm(f32, 2.0001, axis=-1).dtype
dtype('float64') # numpy 1.13
dtype('float32') # numpy 1.14
```

This change affects only `float32` and `float16` arrays.

`count_nonzero(arr, axis=())` now counts over no axes, not all axes

Elsewhere, `axis==()` is always understood as “no axes”, but `count_nonzero` had a special case to treat this as “all axes”. This was inconsistent and surprising. The correct way to count over all axes has always been to pass `axis == None`.

`__init__.py` files added to test directories

This is for pytest compatibility in the case of duplicate test file names in the different directories. As a result, `run_module_suite` no longer works, i.e., `python <path-to-test-file>` results in an error.

`.astype(bool)` on unstructured void arrays now calls `bool` on each element

On Python 2, `void_array.astype(bool)` would always return an array of `True`, unless the dtype is `V0`. On Python 3, this operation would usually crash. Going forwards, `astype` matches the behavior of `bool(np.void)`, considering a buffer of all zeros as false, and anything else as true. Checks for `V0` can still be done with `arr.dtype.itemsize == 0`.

`MaskedArray.squeeze` never returns `np.ma.masked`

`np.squeeze` is documented as returning a view, but the masked variant would sometimes return `masked`, which is not a view. This has been fixed, so that the result is always a view on the original masked array. This breaks any code that used `masked_arr.squeeze() is np.ma.masked`, but fixes code that writes to the result of `squeeze()`.

Renamed first parameter of `can_cast` from `from` to `from_`

The previous parameter name `from` is a reserved keyword in Python, which made it difficult to pass the argument by name. This has been fixed by renaming the parameter to `from_`.

`isnat` raises `TypeError` when passed wrong type

The ufunc `isnat` used to raise a `ValueError` when it was not passed variables of type `datetime` or `timedelta`. This has been changed to raising a `TypeError`.

`dtype.__getitem__` raises `TypeError` when passed wrong type

When indexed with a float, the dtype object used to raise `ValueError`.

User-defined types now need to implement `__str__` and `__repr__`

Previously, user-defined types could fall back to a default implementation of `__str__` and `__repr__` implemented in numpy, but this has now been removed. Now user-defined types will fall back to the python default object `__str__` and object `__repr__`.

Many changes to array printing, disableable with the new “legacy” printing mode

The `str` and `repr` of `ndarrays` and `numpy` scalars have been changed in a variety of ways. These changes are likely to break downstream user’s doctests.

These new behaviors can be disabled to mostly reproduce `numpy 1.13` behavior by enabling the new 1.13 “legacy” printing mode. This is enabled by calling `np.set_printoptions(legacy="1.13")`, or using the new `legacy` argument to `np.array2string`, as `np.array2string(arr, legacy='1.13')`.

In summary, the major changes are:

- For floating-point types:
 - The `repr` of float arrays often omits a space previously printed in the sign position. See the new `sign` option to `np.set_printoptions`.
 - Floating-point arrays and scalars use a new algorithm for decimal representations, giving the shortest unique representation. This will usually shorten `float16` fractional output, and sometimes `float32` and `float128` output. `float64` should be unaffected. See the new `floatmode` option to `np.set_printoptions`.
 - Float arrays printed in scientific notation no longer use fixed-precision, and now instead show the shortest unique representation.
 - The `str` of floating-point scalars is no longer truncated in `python2`.
- For other data types:
 - Non-finite complex scalars print like `nanj` instead of `nan*j`.
 - `NaT` values in `datetime` arrays are now properly aligned.
 - Arrays and scalars of `np.void` datatype are now printed using hex notation.
- For line-wrapping:
 - The “dtype” part of `ndarray` reprs will now be printed on the next line if there isn’t space on the last line of array output.
 - The `linewidth` format option is now always respected. The `repr` or `str` of an array will never exceed this, unless a single element is too wide.
 - The last line of an array string will never have more elements than earlier lines.
 - An extra space is no longer inserted on the first line if the elements are too wide.
- For summarization (the use of `...` to shorten long arrays):
 - A trailing comma is no longer inserted for `str`. Previously, `str(np.arange(1001))` gave `'[0 1 2 ..., 998 999 1000]'`, which has an extra comma.
 - For arrays of 2-D and beyond, when `...` is printed on its own line in order to summarize any but the last axis, newlines are now appended to that line to match its leading newlines and a trailing space character is removed.
- `MaskedArray` arrays now separate printed elements with commas, always print the dtype, and correctly wrap the elements of long arrays to multiple lines. If there is more than 1 dimension, the array attributes are now printed in a new “left-justified” printing style.
- `recarray` arrays no longer print a trailing space before their dtype, and wrap to the right number of columns.
- `0d` arrays no longer have their own idiosyncratic implementations of `str` and `repr`. The `style` argument to `np.array2string` is deprecated.
- Arrays of `bool` datatype will omit the datatype in the `repr`.

- User-defined dtypes (subclasses of `np.generic`) now need to implement `__str__` and `__repr__`.

Some of these changes are described in more detail below. If you need to retain the previous behavior for doctests or other reasons, you may want to do something like:

```
# FIXME: We need the str/repr formatting used in Numpy < 1.14.
try:
    np.set_printoptions(legacy='1.13')
except TypeError:
    pass
```

15.54.6 C API changes

PyPy compatible alternative to `UPDATEIFCOPY` arrays

`UPDATEIFCOPY` arrays are contiguous copies of existing arrays, possibly with different dimensions, whose contents are copied back to the original array when their refcount goes to zero and they are deallocated. Because PyPy does not use refcounts, they do not function correctly with PyPy. NumPy is in the process of eliminating their use internally and two new C-API functions,

- `PyArray_SetWritebackIfCopyBase`
- `PyArray_ResolveWritebackIfCopy`,

have been added together with a complementary flag, `NPY_ARRAY_WRITEBACKIFCOPY`. Using the new functionality also requires that some flags be changed when new arrays are created, to wit: `NPY_ARRAY_INOUT_ARRAY` should be replaced by `NPY_ARRAY_INOUT_ARRAY2` and `NPY_ARRAY_INOUT_FARRAY` should be replaced by `NPY_ARRAY_INOUT_FARRAY2`. Arrays created with these new flags will then have the `WRITEBACKIFCOPY` semantics.

If PyPy compatibility is not a concern, these new functions can be ignored, although there will be a `DeprecationWarning`. If you do wish to pursue PyPy compatibility, more information on these functions and their use may be found in the [c-api](#) documentation and the example in [how-to-extend](#).

15.54.7 New Features

Encoding argument for text IO functions

`genfromtxt`, `loadtxt`, `fromregex` and `savetxt` can now handle files with arbitrary encoding supported by Python via the `encoding` argument. For backward compatibility the argument defaults to the special `bytes` value which continues to treat text as raw byte values and continues to pass latin1 encoded bytes to custom converters. Using any other value (including `None` for system default) will switch the functions to real text IO so one receives unicode strings instead of bytes in the resulting arrays.

External nose plugins are usable by `numpy.testing.Tester`

`numpy.testing.Tester` is now aware of nose plugins that are outside the nose built-in ones. This allows using, for example, nose-timer like so: `np.test(extra_argv=['--with-timer', '--timer-top-n', '20'])` to obtain the runtime of the 20 slowest tests. An extra keyword `timer` was also added to `Tester.test`, so `np.test(timer=20)` will also report the 20 slowest tests.

`parametrize` decorator added to `numpy.testing`

A basic `parametrize` decorator is now available in `numpy.testing`. It is intended to allow rewriting yield based tests that have been deprecated in pytest so as to facilitate the transition to pytest in the future. The nose testing framework has not been supported for several years and looks like abandonware.

The new `parametrize` decorator does not have the full functionality of the one in pytest. It doesn't work for classes, doesn't support nesting, and does not substitute variable names. Even so, it should be adequate to rewrite the NumPy tests.

`chebinterpolate` function added to `numpy.polynomial.chebyshev`

The new `chebinterpolate` function interpolates a given function at the Chebyshev points of the first kind. A new `Chebyshev.interpolate` class method adds support for interpolation over arbitrary intervals using the scaled and shifted Chebyshev points of the first kind.

Support for reading lzma compressed text files in Python 3

With Python versions containing the `lzma` module the text IO functions can now transparently read from files with `xz` or `lzma` extension.

sign option added to `np.setprintoptions` and `np.array2string`

This option controls printing of the sign of floating-point types, and may be one of the characters '-', '+', or ' '. With '+' numpy always prints the sign of positive values, with ' ' it always prints a space (whitespace character) in the sign position of positive values, and with '-' it will omit the sign character for positive values. The new default is ' '.

This new default changes the float output relative to numpy 1.13. The old behavior can be obtained in 1.13 "legacy" printing mode, see compatibility notes above.

hermitian option added to "`np.linalg.matrix_rank`"

The new `hermitian` option allows choosing between standard SVD based matrix rank calculation and the more efficient eigenvalue based method for symmetric/hermitian matrices.

threshold and edgeitems options added to np.array2string

These options could previously be controlled using `np.set_printoptions`, but now can be changed on a per-call basis as arguments to `np.array2string`.

concatenate and stack gained an out argument

A preallocated buffer of the desired dtype can now be used for the output of these functions.

Support for PGI flang compiler on Windows

The PGI flang compiler is a Fortran front end for LLVM released by NVIDIA under the Apache 2 license. It can be invoked by

```
python setup.py config --compiler=clang --fcompiler=flang install
```

There is little experience with this new compiler, so any feedback from people using it will be appreciated.

15.54.8 Improvements

Numerator degrees of freedom in `random.noncentral_f` need only be positive.

Prior to NumPy 1.14.0, the numerator degrees of freedom needed to be > 1 , but the distribution is valid for values > 0 , which is the new requirement.

The GIL is released for all `np.einsum` variations

Some specific loop structures which have an accelerated loop version did not release the GIL prior to NumPy 1.14.0. This oversight has been fixed.

The `np.einsum` function will use BLAS when possible and optimize by default

The `np.einsum` function will now call `np.tensordot` when appropriate. Because `np.tensordot` uses BLAS when possible, that will speed up execution. By default, `np.einsum` will also attempt optimization as the overhead is small relative to the potential improvement in speed.

f2py now handles arrays of dimension 0

`f2py` now allows for the allocation of arrays of dimension 0. This allows for more consistent handling of corner cases downstream.

`numpy.distutils` supports using MSVC and mingw64-gfortran together

NumPy `distutils` now supports using Mingw64 gfortran and MSVC compilers together. This enables the production of Python extension modules on Windows containing Fortran code while retaining compatibility with the binaries distributed by Python.org. Not all use cases are supported, but most common ways to wrap Fortran for Python are functional.

Compilation in this mode is usually enabled automatically, and can be selected via the `--fcompiler` and `--compiler` options to `setup.py`. Moreover, linking Fortran codes to static OpenBLAS is supported; by default a gfortran compatible static archive `openblas.a` is looked for.

`np.linalg.pinv` now works on stacked matrices

Previously it was limited to a single 2d array.

`numpy.save` aligns data to 64 bytes instead of 16

Saving NumPy arrays in the `npz` format with `numpy.save` inserts padding before the array data to align it at 64 bytes. Previously this was only 16 bytes (and sometimes less due to a bug in the code for version 2). Now the alignment is 64 bytes, which matches the widest SIMD instruction set commonly available, and is also the most common cache line size. This makes `npz` files easier to use in programs which open them with `mmap`, especially on Linux where an `mmap` offset must be a multiple of the page size.

NPZ files now can be written without using temporary files

In Python 3.6+ `numpy.savez` and `numpy.savez_compressed` now write directly to a ZIP file, without creating intermediate temporary files.

Better support for empty structured and string types

Structured types can contain zero fields, and string dtypes can contain zero characters. Zero-length strings still cannot be created directly, and must be constructed through structured dtypes:

```
str0 = np.empty(10, np.dtype([('v', str, N)]))['v']
void0 = np.empty(10, np.void)
```

It was always possible to work with these, but the following operations are now supported for these arrays:

- `arr.sort()`
- `arr.view(bytes)`
- `arr.resize(...)`
- `pickle.dumps(arr)`

Support for `decimal.Decimal` in `np.lib.financial`

Unless otherwise stated all functions within the `financial` package now support using the `decimal.Decimal` built-in type.

Float printing now uses “dragon4” algorithm for shortest decimal representation

The `str` and `repr` of floating-point values (16, 32, 64 and 128 bit) are now printed to give the shortest decimal representation which uniquely identifies the value from others of the same type. Previously this was only true for `float64` values. The remaining float types will now often be shorter than in numpy 1.13. Arrays printed in scientific notation now also use the shortest scientific representation, instead of fixed precision as before.

Additionally, the `str` of float scalars will no longer be truncated in python2, unlike python2 *float*'s. *np.double* scalars now have a `str` and `repr` identical to that of a python3 float.

New functions `np.format_float_scientific` and `np.format_float_positional` are provided to generate these decimal representations.

A new option `floatmode` has been added to `np.set_printoptions` and `np.array2string`, which gives control over uniqueness and rounding of printed elements in an array. The new default is `floatmode='maxprec'` with `precision=8`, which will print at most 8 fractional digits, or fewer if an element can be uniquely represented with fewer. A useful new mode is `floatmode="unique"`, which will output enough digits to specify the array elements uniquely.

Numpy complex-floating-scalars with values like `inf*j` or `nan*j` now print as `infj` and `nanj`, like the pure-python complex type.

The `FloatFormat` and `LongFloatFormat` classes are deprecated and should both be replaced by `FloatingFormat`. Similarly `ComplexFormat` and `LongComplexFormat` should be replaced by `ComplexFloatingFormat`.

void datatype elements are now printed in hex notation

A hex representation compatible with the python `bytes` type is now printed for unstructured `np.void` elements, e.g., `V4` datatype. Previously, in python2 the raw void data of the element was printed to stdout, or in python3 the integer byte values were shown.

printing style for void datatypes is now independently customizable

The printing style of `np.void` arrays is now independently customizable using the `formatter` argument to `np.set_printoptions`, using the `'void'` key, instead of the catch-all `numpystr` key as before.

Reduced memory usage of `np.loadtxt`

`np.loadtxt` now reads files in chunks instead of all at once which decreases its memory usage significantly for large files.

15.54.9 Changes

Multiple-field indexing/assignment of structured arrays

The indexing and assignment of structured arrays with multiple fields has changed in a number of ways, as warned about in previous releases.

First, indexing a structured array with multiple fields, e.g., `arr[['f1', 'f3']]`, returns a view into the original array instead of a copy. The returned view will have extra padding bytes corresponding to intervening fields in the original array, unlike the copy in 1.13, which will affect code such as `arr[['f1', 'f3']].view(newdtype)`.

Second, assignment between structured arrays will now occur “by position” instead of “by field name”. The Nth field of the destination will be set to the Nth field of the source regardless of field name, unlike in numpy versions 1.6 to 1.13 in which fields in the destination array were set to the identically-named field in the source array or to 0 if the source did not have a field.

Correspondingly, the order of fields in a structured dtypes now matters when computing dtype equality. For example, with the dtypes

```
x = dtype({'names': ['A', 'B'], 'formats': ['i4', 'f4'], 'offsets': [0, 4]})
y = dtype({'names': ['B', 'A'], 'formats': ['f4', 'i4'], 'offsets': [4, 0]})
```

the expression `x == y` will now return `False`, unlike before. This makes dictionary based dtype specifications like `dtype({'a': ('i4', 0), 'b': ('f4', 4)})` dangerous in python < 3.6 since dict key order is not preserved in those versions.

Assignment from a structured array to a boolean array now raises a `ValueError`, unlike in 1.13, where it always set the destination elements to `True`.

Assignment from structured array with more than one field to a non-structured array now raises a `ValueError`. In 1.13 this copied just the first field of the source to the destination.

Using field “titles” in multiple-field indexing is now disallowed, as is repeating a field name in a multiple-field index.

The documentation for structured arrays in the user guide has been significantly updated to reflect these changes.

Integer and Void scalars are now unaffected by `np.set_string_function`

Previously, unlike most other numpy scalars, the `str` and `repr` of integer and void scalars could be controlled by `np.set_string_function`. This is no longer possible.

0d array printing changed, `style` arg of `array2string` deprecated

Previously the `str` and `repr` of 0d arrays had idiosyncratic implementations which returned `str(a.item())` and `'array(' + repr(a.item()) + ')'` respectively for 0d array `a`, unlike both numpy scalars and higher dimension ndarrays.

Now, the `str` of a 0d array acts like a numpy scalar using `str(a[()])` and the `repr` acts like higher dimension arrays using `formatter(a[()])`, where `formatter` can be specified using `np.set_printoptions`. The `style` argument of `np.array2string` is deprecated.

This new behavior is disabled in 1.13 legacy printing mode, see compatibility notes above.

Seeding `RandomState` using an array requires a 1-d array

`RandomState` previously would accept empty arrays or arrays with 2 or more dimensions, which resulted in either a failure to seed (empty arrays) or for some of the passed values to be ignored when setting the seed.

`MaskedArray` objects show a more useful repr

The repr of a `MaskedArray` is now closer to the python code that would produce it, with arrays now being shown with commas and dtypes. Like the other formatting changes, this can be disabled with the 1.13 legacy printing mode in order to help transition doctests.

The repr of `np.polynomial` classes is more explicit

It now shows the domain and window parameters as keyword arguments to make them more clear:

```
>>> np.polynomial.Polynomial(range(4))
Polynomial([0., 1., 2., 3.], domain=[-1, 1], window=[-1, 1])
```

15.55 NumPy 1.13.3 Release Notes

This is a bugfix release for some problems found since 1.13.1. The most important fixes are for CVE-2017-12852 and temporary elision. Users of earlier versions of 1.13 should upgrade.

The Python versions supported are 2.7 and 3.4 - 3.6. The Python 3.6 wheels available from PIP are built with Python 3.6.2 and should be compatible with all previous versions of Python 3.6. It was cythonized with Cython 0.26.1, which should be free of the bugs found in 0.27 while also being compatible with Python 3.7-dev. The Windows wheels were built with OpenBlas instead ATLAS, which should improve the performance of the linear algebra functions.

The NumPy 1.13.3 release is a re-release of 1.13.2, which suffered from a bug in Cython 0.27.0.

15.55.1 Contributors

A total of 12 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Allan Haldane
- Brandon Carter
- Charles Harris
- Eric Wieser
- Iryna Shcherbina +
- James Bourbeau +
- Jonathan Helmus
- Julian Taylor
- Matti Picus
- Michael Lamparski +
- Michael Seifert
- Ralf Gommers

15.55.2 Pull requests merged

A total of 22 pull requests were merged for this release.

- #9390 BUG: Return the poly1d coefficients array directly
- #9555 BUG: Fix regression in 1.13.x in distutils.mingw32ccompiler.
- #9556 BUG: Fix true_divide when dtype=np.float64 specified.
- #9557 DOC: Fix some rst markup in numpy/doc/basics.py.
- #9558 BLD: Remove -xhost flag from IntelFCompiler.
- #9559 DOC: Removes broken docstring example (source code, png, pdf)...
- #9580 BUG: Add hypot and cabs functions to WIN32 blacklist.
- #9732 BUG: Make scalar function elision check if temp is writeable.
- #9736 BUG: Various fixes to np.gradient
- #9742 BUG: Fix np.pad for CVE-2017-12852
- #9744 BUG: Check for exception in sort functions, add tests
- #9745 DOC: Add whitespace after “versionadded:” directive so it actually...
- #9746 BUG: Memory leak in np.dot of size 0
- #9747 BUG: Adjust gfortran version search regex
- #9757 BUG: Cython 0.27 breaks NumPy on Python 3.
- #9764 BUG: Ensure `_np_scaled_cexp{f,l}` is defined when needed.
- #9765 BUG: PyArray_CountNonzero does not check for exceptions
- #9766 BUG: Fixes histogram monotonicity check for unsigned bin values
- #9767 BUG: Ensure consistent result dtype of count_nonzero
- #9771 BUG: MAINT: Fix mtrand for Cython 0.27.
- #9772 DOC: Create the 1.13.2 release notes.
- #9794 DOC: Create 1.13.3 release notes.

15.56 NumPy 1.13.2 Release Notes

This is a bugfix release for some problems found since 1.13.1. The most important fixes are for CVE-2017-12852 and temporary elision. Users of earlier versions of 1.13 should upgrade.

The Python versions supported are 2.7 and 3.4 - 3.6. The Python 3.6 wheels available from PIP are built with Python 3.6.2 and should be compatible with all previous versions of Python 3.6. The Windows wheels are now built with OpenBlas instead ATLAS, which should improve the performance of the linear algebra functions.

15.56.1 Contributors

A total of 12 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Allan Haldane
- Brandon Carter
- Charles Harris
- Eric Wieser
- Iryna Shcherbina +
- James Bourbeau +
- Jonathan Helmus
- Julian Taylor
- Matti Picus
- Michael Lamparski +
- Michael Seifert
- Ralf Gommers

15.56.2 Pull requests merged

A total of 20 pull requests were merged for this release.

- #9390 BUG: Return the poly1d coefficients array directly
- #9555 BUG: Fix regression in 1.13.x in distutils.mingw32ccompiler.
- #9556 BUG: Fix true_divide when dtype=np.float64 specified.
- #9557 DOC: Fix some rst markup in numpy/doc/basics.py.
- #9558 BLD: Remove -xhost flag from IntelFCompiler.
- #9559 DOC: Removes broken docstring example (source code, png, pdf)...
- #9580 BUG: Add hypot and cabs functions to WIN32 blacklist.
- #9732 BUG: Make scalar function elision check if temp is writeable.
- #9736 BUG: Various fixes to np.gradient
- #9742 BUG: Fix np.pad for CVE-2017-12852
- #9744 BUG: Check for exception in sort functions, add tests
- #9745 DOC: Add whitespace after “versionadded:” directive so it actually...
- #9746 BUG: Memory leak in np.dot of size 0
- #9747 BUG: Adjust gfortran version search regex
- #9757 BUG: Cython 0.27 breaks NumPy on Python 3.
- #9764 BUG: Ensure `_np_scaled_cexp{f,l}` is defined when needed.
- #9765 BUG: PyArray_CountNonzero does not check for exceptions
- #9766 BUG: Fixes histogram monotonicity check for unsigned bin values

- #9767 BUG: Ensure consistent result dtype of count_nonzero
- #9771 BUG, MAINT: Fix mtrand for Cython 0.27.

15.57 NumPy 1.13.1 Release Notes

This is a bugfix release for problems found in 1.13.0. The major changes are fixes for the new memory overlap detection and temporary elision as well as reversion of the removal of the boolean binary `-` operator. Users of 1.13.0 should upgrade.

The Python versions supported are 2.7 and 3.4 - 3.6. Note that the Python 3.6 wheels available from PIP are built against 3.6.1, hence will not work when used with 3.6.0 due to Python bug [29943](#). NumPy 1.13.2 will be released shortly after Python 3.6.2 is out to fix that problem. If you are using 3.6.0 the workaround is to upgrade to 3.6.1 or use an earlier Python version.

15.57.1 Pull requests merged

A total of 19 pull requests were merged for this release.

- #9240 DOC: BLD: fix lots of Sphinx warnings/errors.
- #9255 Revert “DEP: Raise TypeError for subtract(bool, bool).”
- #9261 BUG: don’t elide into readonly and updateifcopy temporaries for...
- #9262 BUG: fix missing keyword rename for common block in numpy.f2py
- #9263 BUG: handle resize of 0d array
- #9267 DOC: update f2py front page and some doc build metadata.
- #9299 BUG: Fix Intel compilation on Unix.
- #9317 BUG: fix wrong ndim used in empty where check
- #9319 BUG: Make extensions compilable with MinGW on Py2.7
- #9339 BUG: Prevent crash if ufunc doc string is null
- #9340 BUG: umath: un-break ufunc where= when no out= is given
- #9371 DOC: Add isnat/positive ufunc to documentation
- #9372 BUG: Fix error in fromstring function from numpy.core.records...
- #9373 BUG: ‘)’ is printed at the end pointer of the buffer in numpy.f2py.
- #9374 DOC: Create NumPy 1.13.1 release notes.
- #9376 BUG: Prevent hang traversing ufunc userloop linked list
- #9377 DOC: Use x1 and x2 in the heaviside docstring.
- #9378 DOC: Add \$PARAMS to the isnat docstring
- #9379 DOC: Update the 1.13.1 release notes

15.57.2 Contributors

A total of 12 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Andras Deak +
- Bob Eldering +
- Charles Harris
- Daniel Hrisca +
- Eric Wieser
- Joshua Leahy +
- Julian Taylor
- Michael Seifert
- Pauli Virtanen
- Ralf Gommers
- Roland Kaufmann
- Warren Weckesser

15.58 NumPy 1.13.0 Release Notes

This release supports Python 2.7 and 3.4 - 3.6.

15.58.1 Highlights

- Operations like `a + b + c` will reuse temporaries on some platforms, resulting in less memory use and faster execution.
- Inplace operations check if inputs overlap outputs and create temporaries to avoid problems.
- New `__array_ufunc__` attribute provides improved ability for classes to override default ufunc behavior.
- New `np.block` function for creating blocked arrays.

15.58.2 New functions

- New `np.positive` ufunc.
- New `np.divmod` ufunc provides more efficient `divmod`.
- New `np.isnat` ufunc tests for NaT special values.
- New `np.heaviside` ufunc computes the Heaviside function.
- New `np.isin` function, improves on `in1d`.
- New `np.block` function for creating blocked arrays.
- New `PyArray_MapIterArrayCopyIfOverlap` added to NumPy C-API.

See below for details.

15.58.3 Deprecations

- Calling `np.fix`, `np.isposinf`, and `np.isneginf` with `f(x, y=out)` is deprecated - the argument should be passed as `f(x, out=out)`, which matches other ufunc-like interfaces.
- Use of the C-API `NPY_CHAR` type number deprecated since version 1.7 will now raise deprecation warnings at runtime. Extensions built with older f2py versions need to be recompiled to remove the warning.
- `np.ma.argsort`, `np.ma.minimum.reduce`, and `np.ma.maximum.reduce` should be called with an explicit *axis* argument when applied to arrays with more than 2 dimensions, as the default value of this argument (`None`) is inconsistent with the rest of numpy (`-1`, `0`, and `0`, respectively).
- `np.ma.MaskedArray.min` is deprecated, as it almost duplicates the functionality of `np.MaskedArray.min`. Exactly equivalent behaviour can be obtained with `np.ma.minimum.reduce`.
- The single-argument form of `np.ma.minimum` and `np.ma.maximum` is deprecated. `np.ma.minimum`, `np.ma.minimum(x)` should now be spelt `np.ma.minimum.reduce(x)`, which is consistent with how this would be done with `np.minimum`.
- Calling `ndarray.conjugate` on non-numeric dtypes is deprecated (it should match the behavior of `np.conjugate`, which throws an error).
- Calling `expand_dims` when the `axis` keyword does not satisfy `-a.ndim - 1 <= axis <= a.ndim`, where `a` is the array being reshaped, is deprecated.

15.58.4 Future Changes

- Assignment between structured arrays with different field names will change in NumPy 1.14. Previously, fields in the `dst` would be set to the value of the identically-named field in the `src`. In numpy 1.14 fields will instead be assigned 'by position': The *n*-th field of the `dst` will be set to the *n*-th field of the `src` array. Note that the `FutureWarning` raised in NumPy 1.12 incorrectly reported this change as scheduled for NumPy 1.13 rather than NumPy 1.14.

15.58.5 Build System Changes

- `numpy.distutils` now automatically determines C-file dependencies with GCC compatible compilers.

15.58.6 Compatibility notes

Error type changes

- `numpy.hstack()` now throws `ValueError` instead of `IndexError` when input is empty.
- Functions taking an `axis` argument, when that argument is out of range, now throw `np.AxisError` instead of a mixture of `IndexError` and `ValueError`. For backwards compatibility, `AxisError` subclasses both of these.

Tuple object dtypes

Support has been removed for certain obscure dtypes that were unintentionally allowed, of the form `(old_dtype, new_dtype)`, where either of the dtypes is or contains the `object` dtype. As an exception, dtypes of the form `(object, [('name', object)])` are still supported due to evidence of existing use.

DeprecationWarning to error

See Changes section for more detail.

- `partition`, `TypeError` when non-integer partition index is used.
- `NpyIter_AdvancedNew`, `ValueError` when `oa_ndim == 0` and `op_axes` is `NULL`
- `negative(bool_)`, `TypeError` when `negative` applied to booleans.
- `subtract(bool_, bool_)`, `TypeError` when subtracting boolean from boolean.
- `np.equal`, `np.not_equal`, object identity doesn't override failed comparison.
- `np.equal`, `np.not_equal`, object identity doesn't override non-boolean comparison.
- Deprecated boolean indexing behavior dropped. See Changes below for details.
- Deprecated `np.alterdot()` and `np.restoredot()` removed.

FutureWarning to changed behavior

See Changes section for more detail.

- `numpy.average` preserves subclasses
- `array == None` and `array != None` do element-wise comparison.
- `np.equal`, `np.not_equal`, object identity doesn't override comparison result.

dtypes are now always true

Previously `bool(dtype)` would fall back to the default python implementation, which checked if `len(dtype) > 0`. Since dtype objects implement `__len__` as the number of record fields, `bool` of scalar dtypes would evaluate to `False`, which was unintuitive. Now `bool(dtype) == True` for all dtypes.

__getslice__ and __setslice__ are no longer needed in ndarray subclasses

When subclassing `np.ndarray` in Python 2.7, it is no longer `_necessary_` to implement `__*slice__` on the derived class, as `__*item__` will intercept these calls correctly.

Any code that did implement these will work exactly as before. Code that invokes “`ndarray.__getslice__`” (e.g. through `super(...).__getslice__`) will now issue a `DeprecationWarning` - `.__getitem__(slice(start, end))` should be used instead.

Indexing MaskedArrays/Constants with ... (ellipsis) now returns MaskedArray

This behavior mirrors that of `np.ndarray`, and accounts for nested arrays in `MaskedArrays` of object dtype, and ellipsis combined with other forms of indexing.

15.58.7 C API changes

GUfuncs on empty arrays and NpyIter axis removal

It is now allowed to remove a zero-sized axis from `NpyIter`. Which may mean that code removing axes from `NpyIter` has to add an additional check when accessing the removed dimensions later on.

The largest followup change is that gufuncs are now allowed to have zero-sized inner dimensions. This means that a gufunc now has to anticipate an empty inner dimension, while this was never possible and an error raised instead.

For most gufuncs no change should be necessary. However, it is now possible for gufuncs with a signature such as `(..., N, M) -> (... , M)` to return a valid result if `N=0` without further wrapping code.

PyArray_MapIterArrayCopyIfOverlap added to NumPy C-API

Similar to `PyArray_MapIterArray` but with an additional `copy_if_overlap` argument. If `copy_if_overlap != 0`, checks if input has memory overlap with any of the other arrays and make copies as appropriate to avoid problems if the input is modified during the iteration. See the documentation for more complete documentation.

15.58.8 New Features

`__array_ufunc__` added

This is the renamed and redesigned `__numpy_ufunc__`. Any class, `ndarray` subclass or not, can define this method or set it to `None` in order to override the behavior of NumPy's ufuncs. This works quite similarly to Python's `__mul__` and other binary operation routines. See the documentation for a more detailed description of the implementation and behavior of this new option. The API is provisional, we do not yet guarantee backward compatibility as modifications may be made pending feedback. See [NEP 13](#) and [documentation](#) for more details.

New `positive` ufunc

This ufunc corresponds to unary `+`, but unlike `+` on an `ndarray` it will raise an error if array values do not support numeric operations.

New `divmod` ufunc

This ufunc corresponds to the Python builtin `divmod`, and is used to implement `divmod` when called on numpy arrays. `np.divmod(x, y)` calculates a result equivalent to `(np.floor_divide(x, y), np.remainder(x, y))` but is approximately twice as fast as calling the functions separately.

`np.isnat` ufunc tests for NaT special datetime and timedelta values

The new ufunc `np.isnat` finds the positions of special NaT values within datetime and timedelta arrays. This is analogous to `np.isnan`.

`np.heaviside` ufunc computes the Heaviside function

The new function `np.heaviside(x, h0)` (a ufunc) computes the Heaviside function:

```

heaviside(x, h0) = { 0   if x < 0,
                    { h0  if x == 0,
                    { 1   if x > 0.

```

`np.block` function for creating blocked arrays

Add a new `block` function to the current stacking functions `vstack`, `hstack`, and `stack`. This allows concatenation across multiple axes simultaneously, with a similar syntax to array creation, but where elements can themselves be arrays. For instance:

```

>>> A = np.eye(2) * 2
>>> B = np.eye(3) * 3
>>> np.block([
...     [A,          np.zeros((2, 3))],
...     [np.ones((3, 2)), B
...     ])
array([[ 2.,  0.,  0.,  0.,  0.],
       [ 0.,  2.,  0.,  0.,  0.],
       [ 1.,  1.,  3.,  0.,  0.],
       [ 1.,  1.,  0.,  3.,  0.],
       [ 1.,  1.,  0.,  0.,  3.]])

```

While primarily useful for block matrices, this works for arbitrary dimensions of arrays.

It is similar to Matlab's square bracket notation for creating block matrices.

`isin` function, improving on `in1d`

The new function `isin` tests whether each element of an N-dimensional array is present anywhere within a second array. It is an enhancement of `in1d` that preserves the shape of the first array.

Temporary elision

On platforms providing the `backtrace` function NumPy will try to avoid creating temporaries in expression involving basic numeric types. For example `d = a + b + c` is transformed to `d = a + b; d += c` which can improve performance for large arrays as less memory bandwidth is required to perform the operation.

axes argument for unique

In an N-dimensional array, the user can now choose the axis along which to look for duplicate N-1-dimensional elements using `numpy.unique`. The original behaviour is recovered if `axis=None` (default).

np.gradient now supports unevenly spaced data

Users can now specify a not-constant spacing for data. In particular `np.gradient` can now take:

1. A single scalar to specify a sample distance for all dimensions.
2. N scalars to specify a constant sample distance for each dimension. i.e. `dx, dy, dz, ...`
3. N arrays to specify the coordinates of the values along each dimension of F. The length of the array must match the size of the corresponding dimension
4. Any combination of N scalars/arrays with the meaning of 2. and 3.

This means that, e.g., it is now possible to do the following:

```
>>> f = np.array([[1, 2, 6], [3, 4, 5]], dtype=np.float_)
>>> dx = 2.
>>> y = [1., 1.5, 3.5]
>>> np.gradient(f, dx, y)
[array([[ 1. ,  1. , -0.5], [ 1. ,  1. , -0.5]]),
 array([[ 2. ,  2. ,  2. ], [ 2. ,  1.7,  0.5]])]
```

Support for returning arrays of arbitrary dimensions in apply_along_axis

Previously, only scalars or 1D arrays could be returned by the function passed to `apply_along_axis`. Now, it can return an array of any dimensionality (including 0D), and the shape of this array replaces the axis of the array being iterated over.

.ndim property added to dtype to complement .shape

For consistency with `ndarray` and `broadcast`, `d.ndim` is a shorthand for `len(d.shape)`.

Support for tracemalloc in Python 3.6

NumPy now supports memory tracing with `tracemalloc` module of Python 3.6 or newer. Memory allocations from NumPy are placed into the domain defined by `numpy.lib.tracemalloc_domain`. Note that NumPy allocation will not show up in `tracemalloc` of earlier Python versions.

NumPy may be built with relaxed stride checking debugging

Setting `NPY_RELAXED_STRIDES_DEBUG=1` in the environment when relaxed stride checking is enabled will cause NumPy to be compiled with the affected strides set to the maximum value of `numpy_intp` in order to help detect invalid usage of the strides in downstream projects. When enabled, invalid usage often results in an error being raised, but the exact type of error depends on the details of the code. `TypeError` and `OverflowError` have been observed in the wild.

It was previously the case that this option was disabled for releases and enabled in master and changing between the two required editing the code. It is now disabled by default but can be enabled for test builds.

15.58.9 Improvements

Ufunc behavior for overlapping inputs

Operations where ufunc input and output operands have memory overlap produced undefined results in previous NumPy versions, due to data dependency issues. In NumPy 1.13.0, results from such operations are now defined to be the same as for equivalent operations where there is no memory overlap.

Operations affected now make temporary copies, as needed to eliminate data dependency. As detecting these cases is computationally expensive, a heuristic is used, which may in rare cases result to needless temporary copies. For operations where the data dependency is simple enough for the heuristic to analyze, temporary copies will not be made even if the arrays overlap, if it can be deduced copies are not necessary. As an example, “`np.add(a, b, out=a)`” will not involve copies.

To illustrate a previously undefined operation:

```
>>> x = np.arange(16).astype(float)
>>> np.add(x[1:], x[:-1], out=x[1:])
```

In NumPy 1.13.0 the last line is guaranteed to be equivalent to:

```
>>> np.add(x[1:].copy(), x[:-1].copy(), out=x[1:])
```

A similar operation with simple non-problematic data dependence is:

```
>>> x = np.arange(16).astype(float)
>>> np.add(x[1:], x[:-1], out=x[:-1])
```

It will continue to produce the same results as in previous NumPy versions, and will not involve unnecessary temporary copies.

The change applies also to in-place binary operations, for example:

```
>>> x = np.random.rand(500, 500)
>>> x += x.T
```

This statement is now guaranteed to be equivalent to `x[...] = x + x.T`, whereas in previous NumPy versions the results were undefined.

Partial support for 64-bit f2py extensions with MinGW

Extensions that incorporate Fortran libraries can now be built using the free [MinGW](#) toolset, also under Python 3.5. This works best for extensions that only do calculations and uses the runtime modestly (reading and writing from files, for instance). Note that this does not remove the need for Mingwpy; if you make extensive use of the runtime, you will most likely run into [issues](#). Instead, it should be regarded as a band-aid until Mingwpy is fully functional.

Extensions can also be compiled using the MinGW toolset using the runtime library from the (moveable) WinPython 3.4 distribution, which can be useful for programs with a PySide1/Qt4 front-end.

Performance improvements for `packbits` and `unpackbits`

The functions `numpy.packbits` with boolean input and `numpy.unpackbits` have been optimized to be a significantly faster for contiguous data.

Fix for PPC long double floating point information

In previous versions of NumPy, the `finfo` function returned invalid information about the `double double` format of the `longdouble` float type on Power PC (PPC). The invalid values resulted from the failure of the NumPy algorithm to deal with the variable number of digits in the significand that are a feature of *PPC long doubles*. This release by-passes the failing algorithm by using heuristics to detect the presence of the PPC double double format. A side-effect of using these heuristics is that the `finfo` function is faster than previous releases.

Better default repr for `ndarray` subclasses

Subclasses of `ndarray` with no `repr` specialization now correctly indent their data and type lines.

More reliable comparisons of masked arrays

Comparisons of masked arrays were buggy for masked scalars and failed for structured arrays with dimension higher than one. Both problems are now solved. In the process, it was ensured that in getting the result for a structured array, masked fields are properly ignored, i.e., the result is equal if all fields that are non-masked in both are equal, thus making the behaviour identical to what one gets by comparing an unstructured masked array and then doing `.all()` over some axis.

`np.matrix` with booleans elements can now be created using the string syntax

`np.matrix` failed whenever one attempts to use it with booleans, e.g., `np.matrix('True')`. Now, this works as expected.

More `linalg` operations now accept empty vectors and matrices

All of the following functions in `np.linalg` now work when given input arrays with a 0 in the last two dimensions: `det`, `slogdet`, `pinv`, `eigvals`, `eigvalsh`, `eig`, `eigh`.

Bundled version of LAPACK is now 3.2.2

NumPy comes bundled with a minimal implementation of lapack for systems without a lapack library installed, under the name of `lapack_lite`. This has been upgraded from LAPACK 3.0.0 (June 30, 1999) to LAPACK 3.2.2 (June 30, 2010). See the [LAPACK changelogs](#) for details on the all the changes this entails.

While no new features are exposed through `numpy`, this fixes some bugs regarding “workspace” sizes, and in some places may use faster algorithms.

reduce of `np.hypot`, `reduce` and `np.logical_xor` allowed in more cases

This now works on empty arrays, returning 0, and can reduce over multiple axes. Previously, a `ValueError` was thrown in these cases.

Better repr of object arrays

Object arrays that contain themselves no longer cause a recursion error.

Object arrays that contain `list` objects are now printed in a way that makes clear the difference between a 2d object array, and a 1d object array of lists.

15.58.10 Changes**`argsort` on masked arrays takes the same default arguments as `sort`**

By default, `argsort` now places the masked values at the end of the sorted array, in the same way that `sort` already did. Additionally, the `end_with` argument is added to `argsort`, for consistency with `sort`. Note that this argument is not added at the end, so breaks any code that passed `fill_value` as a positional argument.

`average` now preserves subclasses

For ndarray subclasses, `numpy.average` will now return an instance of the subclass, matching the behavior of most other NumPy functions such as `mean`. As a consequence, also calls that returned a scalar may now return a subclass array scalar.

`array == None` and `array != None` do element-wise comparison

Previously these operations returned scalars `False` and `True` respectively.

`np.equal`, `np.not_equal` for object arrays ignores object identity

Previously, these functions always treated identical objects as equal. This had the effect of overriding comparison failures, comparison of objects that did not return booleans, such as `np.arrays`, and comparison of objects where the results differed from object identity, such as NaNs.

Boolean indexing changes

- Boolean array-likes (such as lists of python bools) are always treated as boolean indexes.
- Boolean scalars (including python `True`) are legal boolean indexes and never treated as integers.
- Boolean indexes must match the dimension of the axis that they index.
- Boolean indexes used on the lhs of an assignment must match the dimensions of the rhs.
- Boolean indexing into scalar arrays return a new 1-d array. This means that `array(1)[array(True)]` gives `array([1])` and not the original array.

`np.random.multivariate_normal` behavior with bad covariance matrix

It is now possible to adjust the behavior the function will have when dealing with the covariance matrix by using two new keyword arguments:

- `tol` can be used to specify a tolerance to use when checking that the covariance matrix is positive semidefinite.
- `check_valid` can be used to configure what the function will do in the presence of a matrix that is not positive semidefinite. Valid options are `ignore`, `warn` and `raise`. The default value, `warn` keeps the the behavior used on previous releases.

`assert_array_less` compares `np.inf` and `-np.inf` now

Previously, `np.testing.assert_array_less` ignored all infinite values. This is not the expected behavior both according to documentation and intuitively. Now, $-\infty < x < \infty$ is considered `True` for any real number `x` and all other cases fail.

`assert_array_` and masked arrays `assert_equal` hide less warnings

Some warnings that were previously hidden by the `assert_array_` functions are not hidden anymore. In most cases the warnings should be correct and, should they occur, will require changes to the tests using these functions. For the masked array `assert_equal` version, warnings may occur when comparing `NaT`. The function presently does not handle `NaT` or `NaN` specifically and it may be best to avoid it at this time should a warning show up due to this change.

`offset` attribute value in `memmap` objects

The `offset` attribute in a `memmap` object is now set to the offset into the file. This is a behaviour change only for offsets greater than `mmmap.ALLOCATIONGRANULARITY`.

`np.real` and `np.imag` return scalars for scalar inputs

Previously, `np.real` and `np.imag` used to return array objects when provided a scalar input, which was inconsistent with other functions like `np.angle` and `np.conj`.

The polynomial convenience classes cannot be passed to ufuncs

The `ABCPolyBase` class, from which the convenience classes are derived, sets `__array_ufunc__ = None` in order of opt out of ufuncs. If a polynomial convenience class instance is passed as an argument to a ufunc, a `TypeError` will now be raised.

Output arguments to ufuncs can be tuples also for ufunc methods

For calls to ufuncs, it was already possible, and recommended, to use an `out` argument with a tuple for ufuncs with multiple outputs. This has now been extended to output arguments in the `reduce`, `accumulate`, and `reduceat` methods. This is mostly for compatibility with `__array_ufunc`; there are no ufuncs yet that have more than one output.

15.59 NumPy 1.12.1 Release Notes

NumPy 1.12.1 supports Python 2.7 and 3.4 - 3.6 and fixes bugs and regressions found in NumPy 1.12.0. In particular, the regression in f2py constant parsing is fixed. Wheels for Linux, Windows, and OSX can be found on PyPI,

15.59.1 Bugs Fixed

- BUG: Fix wrong future nat warning and equiv type logic error...
- BUG: Fix wrong masked median for some special cases
- DOC: Place np.average in inline code
- TST: Work around isfinite inconsistency on i386
- BUG: Guard against replacing constants without '_' spec in f2py.
- BUG: Fix mean for float 16 non-array inputs for 1.12
- BUG: Fix calling python api with error set and minor leaks for...
- BUG: Make iscomplexobj compatible with custom dtypes again
- BUG: Fix undefined behaviour induced by bad __array_wrap__
- BUG: Fix MaskedArray.__setitem__
- BUG: PPC64el machines are POWER for Fortran in f2py
- BUG: Look up methods on MaskedArray in *_frommethod*
- BUG: Remove extra digit in binary_repr at limit
- BUG: Fix deepcopy regression for empty arrays.
- BUG: Fix ma.median for empty ndarrays

15.60 NumPy 1.12.0 Release Notes

This release supports Python 2.7 and 3.4 - 3.6.

15.60.1 Highlights

The NumPy 1.12.0 release contains a large number of fixes and improvements, but few that stand out above all others. That makes picking out the highlights somewhat arbitrary but the following may be of particular interest or indicate areas likely to have future consequences.

- Order of operations in `np.einsum` can now be optimized for large speed improvements.
- New `signature` argument to `np.vectorize` for vectorizing with core dimensions.
- The `keepdims` argument was added to many functions.
- New context manager for testing warnings
- Support for BLIS in `numpy.distutils`
- Much improved support for PyPy (not yet finished)

15.60.2 Dropped Support

- Support for Python 2.6, 3.2, and 3.3 has been dropped.

15.60.3 Added Support

- Support for PyPy 2.7 v5.6.0 has been added. While not complete (`nditer.updateifcopy` is not supported yet), this is a milestone for PyPy's C-API compatibility layer.

15.60.4 Build System Changes

- Library order is preserved, instead of being reordered to match that of the directories.

15.60.5 Deprecations

Assignment of `ndarray` object's `data` attribute

Assigning the `'data'` attribute is an inherently unsafe operation as pointed out in gh-7083. Such a capability will be removed in the future.

Unsafe int casting of the `num` attribute in `linspace`

`np.linspace` now raises `DeprecationWarning` when `num` cannot be safely interpreted as an integer.

Insufficient bit width parameter to `binary_repr`

If a `'width'` parameter is passed into `binary_repr` that is insufficient to represent the number in base 2 (positive) or 2's complement (negative) form, the function used to silently ignore the parameter and return a representation using the minimal number of bits needed for the form in question. Such behavior is now considered unsafe from a user perspective and will raise an error in the future.

15.60.6 Future Changes

- In 1.13 `NAT` will always compare `False` except for `NAT != NAT`, which will be `True`. In short, `NAT` will behave like `NaN`
- In 1.13 `np.average` will preserve subclasses, to match the behavior of most other numpy functions such as `np.mean`. In particular, this means calls which returned a scalar may return a 0-d subclass object instead.

Multiple-field manipulation of structured arrays

In 1.13 the behavior of structured arrays involving multiple fields will change in two ways:

First, indexing a structured array with multiple fields (eg, `arr[['f1', 'f3']]`) will return a view into the original array in 1.13, instead of a copy. Note the returned view will have extra padding bytes corresponding to intervening fields in the original array, unlike the copy in 1.12, which will affect code such as `arr[['f1', 'f3']].view(newdtype)`.

Second, for numpy versions 1.6 to 1.12 assignment between structured arrays occurs “by field name”: Fields in the destination array are set to the identically-named field in the source array or to 0 if the source does not have a field:

```
>>> a = np.array([(1,2), (3,4)], dtype=[('x', 'i4'), ('y', 'i4')])
>>> b = np.ones(2, dtype=[('z', 'i4'), ('y', 'i4'), ('x', 'i4')])
>>> b[:] = a
>>> b
array([(0, 2, 1), (0, 4, 3)],
      dtype=[('z', '<i4'), ('y', '<i4'), ('x', '<i4')])
```

In 1.13 assignment will instead occur “by position”: The Nth field of the destination will be set to the Nth field of the source regardless of field name. The old behavior can be obtained by using indexing to reorder the fields before assignment, e.g., `b[['x', 'y']] = a[['y', 'x']]`.

15.60.7 Compatibility notes

DeprecationWarning to error

- Indexing with floats raises `IndexError`, e.g., `a[0, 0.0]`.
- Indexing with non-integer array_like raises `IndexError`, e.g., `a['1', '2']`
- Indexing with multiple ellipsis raises `IndexError`, e.g., `a[..., ...]`.
- Non-integers used as index values raise `TypeError`, e.g., in `reshape`, `take`, and specifying reduce axis.

FutureWarning to changed behavior

- `np.full` now returns an array of the fill-value’s dtype if no dtype is given, instead of defaulting to float.
- `np.average` will emit a warning if the argument is a subclass of `ndarray`, as the subclass will be preserved starting in 1.13. (see Future Changes)

power and ** raise errors for integer to negative integer powers

The previous behavior depended on whether numpy scalar integers or numpy integer arrays were involved.

For arrays

- Zero to negative integer powers returned least integral value.
- Both 1, -1 to negative integer powers returned correct values.
- The remaining integers returned zero when raised to negative integer powers.

For scalars

- Zero to negative integer powers returned least integral value.
- Both 1, -1 to negative integer powers returned correct values.

- The remaining integers sometimes returned zero, sometimes the correct float depending on the integer type combination.

All of these cases now raise a `ValueError` except for those integer combinations whose common type is float, for instance `uint64` and `int8`. It was felt that a simple rule was the best way to go rather than have special exceptions for the integer units. If you need negative powers, use an inexact type.

Relaxed stride checking is the default

This will have some impact on code that assumed that `F_CONTIGUOUS` and `C_CONTIGUOUS` were mutually exclusive and could be set to determine the default order for arrays that are now both.

The `np.percentile` ‘midpoint’ interpolation method fixed for exact indices

The ‘midpoint’ interpolator now gives the same result as ‘lower’ and ‘higher’ when the two coincide. Previous behavior of ‘lower’ + 0.5 is fixed.

`keepdims` kwarg is passed through to user-class methods

numpy functions that take a `keepdims` kwarg now pass the value through to the corresponding methods on ndarray sub-classes. Previously the `keepdims` keyword would be silently dropped. These functions now have the following behavior:

1. If user does not provide `keepdims`, no keyword is passed to the underlying method.
2. Any user-provided value of `keepdims` is passed through as a keyword argument to the method.

This will raise in the case where the method does not support a `keepdims` kwarg and the user explicitly passes in `keepdims`.

The following functions are changed: `sum`, `product`, `sometrue`, `alltrue`, `any`, `all`, `amax`, `amin`, `prod`, `mean`, `std`, `var`, `nanmin`, `nanmax`, `nansum`, `nanprod`, `nanmean`, `nanmedian`, `nanvar`, `nanstd`

`bitwise_and` identity changed

The previous identity was 1, it is now -1. See entry in Improvements for more explanation.

`ma.median` warns and returns nan when unmasked invalid values are encountered

Similar to unmasked median the masked median `ma.median` now emits a Runtime warning and returns `NaN` in slices where an unmasked `NaN` is present.

Greater consistency in `assert_almost_equal`

The precision check for scalars has been changed to match that for arrays. It is now:

```
abs(actual - desired) < 1.5 * 10**(-decimal)
```

Note that this is looser than previously documented, but agrees with the previous implementation used in `assert_array_almost_equal`. Due to the change in implementation some very delicate tests may fail that did not fail before.

NoseTester behaviour of warnings during testing

When `raise_warnings="develop"` is given, all uncaught warnings will now be considered a test failure. Previously only selected ones were raised. Warnings which are not caught or raised (mostly when in release mode) will be shown once during the test cycle similar to the default python settings.

`assert_warns` and `deprecated` decorator more specific

The `assert_warns` function and context manager are now more specific to the given warning category. This increased specificity leads to them being handled according to the outer warning settings. This means that no warning may be raised in cases where a wrong category warning is given and ignored outside the context. Alternatively the increased specificity may mean that warnings that were incorrectly ignored will now be shown or raised. See also the new `suppress_warnings` context manager. The same is true for the `deprecated` decorator.

C API

No changes.

15.60.8 New Features

Writeable keyword argument for `as_strided`

`np.lib.stride_tricks.as_strided` now has a `writable` keyword argument. It can be set to `False` when no write operation to the returned array is expected to avoid accidental unpredictable writes.

`axes` keyword argument for `rot90`

The `axes` keyword argument in `rot90` determines the plane in which the array is rotated. It defaults to `axes=(0, 1)` as in the original function.

Generalized `flip`

`flipud` and `fliplr` reverse the elements of an array along `axis=0` and `axis=1` respectively. The newly added `flip` function reverses the elements of an array along any given axis.

- `np.count_nonzero` now has an `axis` parameter, allowing non-zero counts to be generated on more than just a flattened array object.

BLIS support in `numpy.distutils`

Building against the BLAS implementation provided by the BLIS library is now supported. See the `[blis]` section in `site.cfg.example` (in the root of the numpy repo or source distribution).

Hook in `numpy/__init__.py` to run distribution-specific checks

Binary distributions of numpy may need to run specific hardware checks or load specific libraries during numpy initialization. For example, if we are distributing numpy with a BLAS library that requires SSE2 instructions, we would like to check the machine on which numpy is running does have SSE2 in order to give an informative error.

Add a hook in `numpy/__init__.py` to import a `numpy/_distributor_init.py` file that will remain empty (bar a docstring) in the standard numpy source, but that can be overwritten by people making binary distributions of numpy.

New nanfunctions `nancumsum` and `nancumprod` added

Nan-functions `nancumsum` and `nancumprod` have been added to compute `cumsum` and `cumprod` by ignoring nans.

`np.interp` can now interpolate complex values

`np.lib.interp(x, xp, fp)` now allows the interpolated array `fp` to be complex and will interpolate at `complex128` precision.

New polynomial evaluation function `polyvalfromroots` added

The new function `polyvalfromroots` evaluates a polynomial at given points from the roots of the polynomial. This is useful for higher order polynomials, where expansion into polynomial coefficients is inaccurate at machine precision.

New array creation function `geomspace` added

The new function `geomspace` generates a geometric sequence. It is similar to `logspace`, but with `start` and `stop` specified directly: `geomspace(start, stop)` behaves the same as `logspace(log10(start), log10(stop))`.

New context manager for testing warnings

A new context manager `suppress_warnings` has been added to the testing utils. This context manager is designed to help reliably test warnings. Specifically to reliably filter/ignore warnings. Ignoring warnings by using an “ignore” filter in Python versions before 3.4.x can quickly result in these (or similar) warnings not being tested reliably.

The context manager allows to filter (as well as record) warnings similar to the `catch_warnings` context, but allows for easier specificity. Also printing warnings that have not been filtered or nesting the context manager will work as expected. Additionally, it is possible to use the context manager as a decorator which can be useful when multiple tests give need to hide the same warning.

New masked array functions `ma.convolve` and `ma.correlate` added

These functions wrapped the non-masked versions, but propagate through masked values. There are two different propagation modes. The default causes masked values to contaminate the result with masks, but the other mode only outputs masks if there is no alternative.

New `float_power` ufunc

The new `float_power` ufunc is like the `power` function except all computation is done in a minimum precision of float64. There was a long discussion on the numpy mailing list of how to treat integers to negative integer powers and a popular proposal was that the `__pow__` operator should always return results of at least float64 precision. The `float_power` function implements that option. Note that it does not support object arrays.

`np.loadtxt` now supports a single integer as `usecol` argument

Instead of using `usecol=(n,)` to read the *n*th column of a file it is now allowed to use `usecol=n`. Also the error message is more user friendly when a non-integer is passed as a column index.

Improved automated bin estimators for `histogram`

Added 'doane' and 'sqrt' estimators to `histogram` via the `bins` argument. Added support for range-restricted histograms with automated bin estimation.

`np.roll` can now roll multiple axes at the same time

The `shift` and `axis` arguments to `roll` are now broadcast against each other, and each specified axis is shifted accordingly.

The `__complex__` method has been implemented for the `ndarrays`

Calling `complex()` on a size 1 array will now cast to a python complex.

`pathlib.Path` objects now supported

The standard `np.load`, `np.save`, `np.loadtxt`, `np.savez`, and similar functions can now take `pathlib.Path` objects as an argument instead of a filename or open file object.

New `bits` attribute for `np.finfo`

This makes `np.finfo` consistent with `np.iinfo` which already has that attribute.

New signature argument to `np.vectorize`

This argument allows for vectorizing user defined functions with core dimensions, in the style of NumPy's generalized universal functions. This allows for vectorizing a much broader class of functions. For example, an arbitrary distance metric that combines two vectors to produce a scalar could be vectorized with `signature='(n), (n) -> ()'`. See `np.vectorize` for full details.

Emit `py3kwarnings` for division of integer arrays

To help people migrate their code bases from Python 2 to Python 3, the python interpreter has a handy option `-3`, which issues warnings at runtime. One of its warnings is for integer division:

```
$ python -3 -c "2/3"
-c:1: DeprecationWarning: classic int division
```

In Python 3, the new integer division semantics also apply to numpy arrays. With this version, numpy will emit a similar warning:

```
$ python -3 -c "import numpy as np; np.array(2)/np.array(3)"
-c:1: DeprecationWarning: numpy: classic int division
```

Previously, it included `str` (bytes) and `unicode` on Python2, but only `str` (unicode) on Python3.

15.60.9 Improvements

`bitwise_and` identity changed

The previous identity was 1 with the result that all bits except the LSB were masked out when the `reduce` method was used. The new identity is -1, which should work properly on twos complement machines as all bits will be set to one.

Generalized Ufuncs will now unlock the GIL

Generalized Ufuncs, including most of the `linalg` module, will now unlock the Python global interpreter lock.

Caches in `np.fft` are now bounded in total size and item count

The caches in `np.fft` that speed up successive FFTs of the same length can no longer grow without bounds. They have been replaced with LRU (least recently used) caches that automatically evict no longer needed items if either the memory size or item count limit has been reached.

Improved handling of zero-width string/unicode dtypes

Fixed several interfaces that explicitly disallowed arrays with zero-width string dtypes (i.e. `dtype('S0')` or `dtype('U0')`), and fixed several bugs where such dtypes were not handled properly. In particular, changed `ndarray.__new__` to not implicitly convert `dtype('S0')` to `dtype('S1')` (and likewise for unicode) when creating new arrays.

Integer ufuncs vectorized with AVX2

If the cpu supports it at runtime the basic integer ufuncs now use AVX2 instructions. This feature is currently only available when compiled with GCC.

Order of operations optimization in `np.einsum`

`np.einsum` now supports the `optimize` argument which will optimize the order of contraction. For example, `np.einsum` would complete the chain dot example `np.einsum('ij,jk,kl->il', a, b, c)` in a single pass which would scale like N^4 ; however, when `optimize=True` `np.einsum` will create an intermediate array to reduce this scaling to N^3 or effectively `np.dot(a, b).dot(c)`. Usage of intermediate tensors to reduce scaling has been applied to the general einsum summation notation. See `np.einsum_path` for more details.

quicksort has been changed to an introsort

The quicksort kind of `np.sort` and `np.argsort` is now an introsort which is regular quicksort but changing to a heapsort when not enough progress is made. This retains the good quicksort performance while changing the worst case runtime from $O(N^2)$ to $O(N \log(N))$.

`ediff1d` improved performance and subclass handling

The `ediff1d` function uses an array instead on a flat iterator for the subtraction. When `to_begin` or `to_end` is not `None`, the subtraction is performed in place to eliminate a copy operation. A side effect is that certain subclasses are handled better, namely `astropy.Quantity`, since the complete array is created, wrapped, and then begin and end values are set, instead of using concatenate.

Improved precision of `ndarray.mean` for float16 arrays

The computation of the mean of float16 arrays is now carried out in float32 for improved precision. This should be useful in packages such as Theano where the precision of float16 is adequate and its smaller footprint is desirable.

15.60.10 Changes

All array-like methods are now called with keyword arguments in `fromnumeric.py`

Internally, many array-like methods in `fromnumeric.py` were being called with positional arguments instead of keyword arguments as their external signatures were doing. This caused a complication in the downstream 'pandas' library that encountered an issue with 'numpy' compatibility. Now, all array-like methods in this module are called with keyword arguments instead.

Operations on np.memmap objects return numpy arrays in most cases

Previously operations on a memmap object would misleadingly return a memmap instance even if the result was actually not memmapped. For example, `arr + 1` or `arr + arr` would return memmap instances, although no memory from the output array is memmapped. Version 1.12 returns ordinary numpy arrays from these operations.

Also, reduction of a memmap (e.g. `.sum(axis=None)`) now returns a numpy scalar instead of a 0d memmap.

stacklevel of warnings increased

The stacklevel for python based warnings was increased so that most warnings will report the offending line of the user code instead of the line the warning itself is given. Passing of stacklevel is now tested to ensure that new warnings will receive the `stacklevel` argument.

This causes warnings with the “default” or “module” filter to be shown once for every offending user code line or user module instead of only once. On python versions before 3.4, this can cause warnings to appear that were falsely ignored before, which may be surprising especially in test suits.

15.61 NumPy 1.11.3 Release Notes

Numpy 1.11.3 fixes a bug that leads to file corruption when very large files opened in append mode are used in `ndarray.tofile`. It supports Python versions 2.6 - 2.7 and 3.2 - 3.5. Wheels for Linux, Windows, and OS X can be found on PyPI.

15.61.1 Contributors to maintenance/1.11.3

A total of 2 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Charles Harris
- Pavel Potocek +

15.61.2 Pull Requests Merged

- [#8341](#): BUG: Fix `ndarray.tofile` large file corruption in append mode.
- [#8346](#): TST: Fix tests in PR [#8341](#) for NumPy 1.11.x

15.62 NumPy 1.11.2 Release Notes

Numpy 1.11.2 supports Python 2.6 - 2.7 and 3.2 - 3.5. It fixes bugs and regressions found in Numpy 1.11.1 and includes several build related improvements. Wheels for Linux, Windows, and OS X can be found on PyPI.

15.62.1 Pull Requests Merged

Fixes overridden by later merges and release notes updates are omitted.

- #7736 BUG: Many functions silently drop 'keepdims' kwarg.
- #7738 ENH: Add extra kwargs and update doc of many MA methods.
- #7778 DOC: Update Numpy 1.11.1 release notes.
- #7793 BUG: MaskedArray.count treats negative axes incorrectly.
- #7816 BUG: Fix array too big error for wide dtypes.
- #7821 BUG: Make sure npy_mul_with_overflow_<type> detects overflow.
- #7824 MAINT: Allocate fewer bytes for empty arrays.
- #7847 MAINT,DOC: Fix some imp module uses and update f2py.compile docstring.
- #7849 MAINT: Fix remaining uses of deprecated Python imp module.
- #7851 BLD: Fix ATLAS version detection.
- #7896 BUG: Construct ma.array from np.array which contains padding.
- #7904 BUG: Fix float16 type not being called due to wrong ordering.
- #7917 BUG: Production install of numpy should not require nose.
- #7919 BLD: Fixed MKL detection for recent versions of this library.
- #7920 BUG: Fix for issue #7835 (ma.median of 1d).
- #7932 BUG: Monkey-patch _msvccompile.gen_lib_option like other compilers.
- #7939 BUG: Check for HAVE_LDOUBLE_DOUBLE_DOUBLE_LE in npy_math_complex.
- #7953 BUG: Guard against buggy comparisons in generic quicksort.
- #7954 BUG: Use keyword arguments to initialize Extension base class.
- #7955 BUG: Make sure numpy globals keep identity after reload.
- #7972 BUG: MSVCCompiler grows 'lib' & 'include' env strings exponentially.
- #8005 BLD: Remove __NUMPY_SETUP__ from builtins at end of setup.py.
- #8010 MAINT: Remove leftover imp module imports.
- #8020 BUG: Fix return of np.ma.count if keepdims is True and axis is None.
- #8024 BUG: Fix numpy.ma.median.
- #8031 BUG: Fix np.ma.median with only one non-masked value.
- #8044 BUG: Fix bug in NpyIter buffering with discontinuous arrays.

15.63 NumPy 1.11.1 Release Notes

NumPy 1.11.1 supports Python 2.6 - 2.7 and 3.2 - 3.5. It fixes bugs and regressions found in NumPy 1.11.0 and includes several build related improvements. Wheels for Linux, Windows, and OSX can be found on PyPI.

15.63.1 Fixes Merged

- #7506 BUG: Make sure numpy imports on python 2.6 when nose is unavailable.
- #7530 BUG: Floating exception with invalid axis in np.lexsort.
- #7535 BUG: Extend glibc complex trig functions blacklist to glibc < 2.18.
- #7551 BUG: Allow graceful recovery for no compiler.
- #7558 BUG: Constant padding expected wrong type in constant_values.
- #7578 BUG: Fix OverflowError in Python 3.x. in swig interface.
- #7590 BLD: Fix configparser.InterpolationSyntaxError.
- #7597 BUG: Make np.ma.take work on scalars.
- #7608 BUG: linalg.norm(): Don't convert object arrays to float.
- #7638 BLD: Correct C compiler customization in system_info.py.
- #7654 BUG: ma.median of 1d array should return a scalar.
- #7656 BLD: Remove hardcoded Intel compiler flag -xSSE4.2.
- #7660 BUG: Temporary fix for str(mvoid) for object field types.
- #7665 BUG: Fix incorrect printing of 1D masked arrays.
- #7670 BUG: Correct initial index estimate in histogram.
- #7671 BUG: Boolean assignment no GIL release when transfer needs API.
- #7676 BUG: Fix handling of right edge of final histogram bin.
- #7680 BUG: Fix np.clip bug NaN handling for Visual Studio 2015.
- #7724 BUG: Fix segfaults in np.random.shuffle.
- #7731 MAINT: Change mkl_info.dir_env_var from MKL to MKLROOT.
- #7737 BUG: Fix issue on OS X with Python 3.x, npymath.ini not installed.

15.64 NumPy 1.11.0 Release Notes

This release supports Python 2.6 - 2.7 and 3.2 - 3.5 and contains a number of enhancements and improvements. Note also the build system changes listed below as they may have subtle effects.

No Windows (TM) binaries are provided for this release due to a broken toolchain. One of the providers of Python packages for Windows (TM) is your best bet.

15.64.1 Highlights

Details of these improvements can be found below.

- The `datetime64` type is now timezone naive.
- A `dtype` parameter has been added to `randint`.
- Improved detection of two arrays possibly sharing memory.
- Automatic bin size estimation for `np.histogram`.
- Speed optimization of `A @ A.T` and `dot(A, A.T)`.
- New function `np.moveaxis` for reordering array axes.

15.64.2 Build System Changes

- Numpy now uses `setuptools` for its builds instead of plain `distutils`. This fixes usage of `install_requires='numpy'` in the `setup.py` files of projects that depend on Numpy (see [gh-6551](#)). It potentially affects the way that build/install methods for Numpy itself behave though. Please report any unexpected behavior on the Numpy issue tracker.
- Bento build support and related files have been removed.
- Single file build support and related files have been removed.

15.64.3 Future Changes

The following changes are scheduled for Numpy 1.12.0.

- Support for Python 2.6, 3.2, and 3.3 will be dropped.
- Relaxed stride checking will become the default. See the 1.8.0 release notes for a more extended discussion of what this change implies.
- The behavior of the `datetime64` “not a time” (NaT) value will be changed to match that of floating point “not a number” (NaN) values: all comparisons involving NaT will return `False`, except for `NaT != NaT` which will return `True`.
- Indexing with floats will raise `IndexError`, e.g., `a[0, 0.0]`.
- Indexing with non-integer `array_like` will raise `IndexError`, e.g., `a['1', '2']`
- Indexing with multiple ellipsis will raise `IndexError`, e.g., `a[..., ...]`.
- Non-integers used as index values will raise `TypeError`, e.g., in `reshape`, `take`, and specifying `reduce` axis.

In a future release the following changes will be made.

- The `rand` function exposed in `numpy.testing` will be removed. That function is left over from early Numpy and was implemented using the Python `random` module. The random number generators from `numpy.random` should be used instead.
- The `ndarray.view` method will only allow `c_contiguous` arrays to be viewed using a `dtype` of different size causing the last dimension to change. That differs from the current behavior where arrays that are `f_contiguous` but not `c_contiguous` can be viewed as a `dtype` type of different size causing the first dimension to change.
- Slicing a `MaskedArray` will return views of both data **and** mask. Currently the mask is copy-on-write and changes to the mask in the slice do not propagate to the original mask. See the `FutureWarnings` section below for details.

15.64.4 Compatibility notes

datetime64 changes

In prior versions of NumPy the experimental datetime64 type always stored times in UTC. By default, creating a datetime64 object from a string or printing it would convert from or to local time:

```
# old behavior
>>> np.datetime64('2000-01-01T00:00:00')
numpy.datetime64('2000-01-01T00:00:00-0800') # note the timezone offset -08:00
```

A consensus of datetime64 users agreed that this behavior is undesirable and at odds with how datetime64 is usually used (e.g., by `pandas`). For most use cases, a timezone naive datetime type is preferred, similar to the `datetime.datetime` type in the Python standard library. Accordingly, datetime64 no longer assumes that input is in local time, nor does it print local times:

```
>>> np.datetime64('2000-01-01T00:00:00')
numpy.datetime64('2000-01-01T00:00:00')
```

For backwards compatibility, datetime64 still parses timezone offsets, which it handles by converting to UTC. However, the resulting datetime is timezone naive:

```
>>> np.datetime64('2000-01-01T00:00:00-08')
DeprecationWarning: parsing timezone aware datetimes is deprecated;
this will raise an error in the future
numpy.datetime64('2000-01-01T08:00:00')
```

As a corollary to this change, we no longer prohibit casting between datetimes with date units and datetimes with time units. With timezone naive datetimes, the rule for casting from dates to times is no longer ambiguous.

linalg.norm return type changes

The return type of the `linalg.norm` function is now floating point without exception. Some of the norm types previously returned integers.

polynomial fit changes

The various fit functions in the numpy polynomial package no longer accept non-integers for degree specification.

np.dot now raises `TypeError` instead of `ValueError`

This behaviour mimics that of other functions such as `np.inner`. If the two arguments cannot be cast to a common type, it could have raised a `TypeError` or `ValueError` depending on their order. Now, `np.dot` will now always raise a `TypeError`.

FutureWarning to changed behavior

- In `np.lib.split` an empty array in the result always had dimension `(0,)` no matter the dimensions of the array being split. This has been changed so that the dimensions will be preserved. A `FutureWarning` for this change has been in place since Numpy 1.9 but, due to a bug, sometimes no warning was raised and the dimensions were already preserved.

% and // operators

These operators are implemented with the `remainder` and `floor_divide` functions respectively. Those functions are now based around `fmod` and are computed together so as to be compatible with each other and with the Python versions for float types. The results should be marginally more accurate or outright bug fixes compared to the previous results, but they may differ significantly in cases where roundoff makes a difference in the integer returned by `floor_divide`. Some corner cases also change, for instance, NaN is always returned for both functions when the divisor is zero, `divmod(1.0, inf)` returns `(0.0, 1.0)` except on MSVC 2008, and `divmod(-1.0, inf)` returns `(-1.0, inf)`.

C API

Removed the `check_return` and `inner_loop_selector` members of the `PyUFuncObject` struct (replacing them with `reserved` slots to preserve struct layout). These were never used for anything, so it's unlikely that any third-party code is using them either, but we mention it here for completeness.

object dtype detection for old-style classes

In python 2, objects which are instances of old-style user-defined classes no longer automatically count as 'object' type in the dtype-detection handler. Instead, as in python 3, they may potentially count as sequences, but only if they define both a `__len__` and a `__getitem__` method. This fixes a segfault and inconsistency between python 2 and 3.

15.64.5 New Features

- `np.histogram` now provides plugin estimators for automatically estimating the optimal number of bins. Passing one of `['auto', 'fd', 'scott', 'rice', 'sturges']` as the argument to 'bins' results in the corresponding estimator being used.
- A benchmark suite using [Airspeed Velocity](#) has been added, converting the previous `vbench`-based one. You can run the suite locally via `python runtests.py --bench`. For more details, see `benchmarks/README.rst`.
- A new function `np.shares_memory` that can check exactly whether two arrays have memory overlap is added. `np.may_share_memory` also now has an option to spend more effort to reduce false positives.
- `SkipTest` and `KnownFailureException` exception classes are exposed in the `numpy.testing` namespace. Raise them in a test function to mark the test to be skipped or mark it as a known failure, respectively.
- `f2py.compile` has a new extension keyword parameter that allows the fortran extension to be specified for generated temp files. For instance, the files can be specified to be `*.f90`. The `verbose` argument is also activated, it was previously ignored.
- A `dtype` parameter has been added to `np.random.randint` Random ndarrays of the following types can now be generated:
 - `np.bool_`,
 - `np.int8`, `np.uint8`,

```
- np.int16, np.uint16,  
- np.int32, np.uint32,  
- np.int64, np.uint64,  
- np.int_ `` , `` np.intp
```

The specification is by precision rather than by C type. Hence, on some platforms `np.int64` may be a `long` instead of `long long` even if the specified dtype is `long long` because the two may have the same precision. The resulting type depends on which C type numpy uses for the given precision. The byteorder specification is also ignored, the generated arrays are always in native byte order.

- A new `np.moveaxis` function allows for moving one or more array axes to a new position by explicitly providing source and destination axes. This function should be easier to use than the current `rollaxis` function as well as providing more functionality.
- The `deg` parameter of the various `numpy.polynomial` fits has been extended to accept a list of the degrees of the terms to be included in the fit, the coefficients of all other terms being constrained to zero. The change is backward compatible, passing a scalar `deg` will behave as before.
- A `divmod` function for float types modeled after the Python version has been added to the `numpy.math` library.

15.64.6 Improvements

`np.gradient` now supports an axis argument

The `axis` parameter was added to `np.gradient` for consistency. It allows to specify over which axes the gradient is calculated.

`np.lexsort` now supports arrays with object data-type

The function now internally calls the generic `numpy.amergesort` when the type does not implement a merge-sort kind of `argsort` method.

`np.ma.core.MaskedArray` now supports an order argument

When constructing a new `MaskedArray` instance, it can be configured with an `order` argument analogous to the one when calling `np.ndarray`. The addition of this argument allows for the proper processing of an `order` argument in several `MaskedArray`-related utility functions such as `np.ma.core.array` and `np.ma.core.asarray`.

Memory and speed improvements for masked arrays

Creating a masked array with `mask=True` (resp. `mask=False`) now uses `np.ones` (resp. `np.zeros`) to create the mask, which is faster and avoid a big memory peak. Another optimization was done to avoid a memory peak and useless computations when printing a masked array.

ndarray.tofile now uses fallocate on linux

The function now uses the `fallocate` system call to reserve sufficient disk space on file systems that support it.

Optimizations for operations of the form $A.T @ A$ and $A @ A.T$

Previously, `gemm` BLAS operations were used for all matrix products. Now, if the matrix product is between a matrix and its transpose, it will use `syrk` BLAS operations for a performance boost. This optimization has been extended to `@`, `numpy.dot`, `numpy.inner`, and `numpy.matmul`.

Note: Requires the transposed and non-transposed matrices to share data.

np.testing.assert_warns can now be used as a context manager

This matches the behavior of `assert_raises`.

Speed improvement for np.random.shuffle

`np.random.shuffle` is now much faster for 1d ndarrays.

15.64.7 Changes**Pyrex support was removed from numpy.distutils**

The method `build_src.generate_a_pyrex_source` will remain available; it has been monkeypatched by users to support Cython instead of Pyrex. It's recommended to switch to a better supported method of build Cython extensions though.

np.broadcast can now be called with a single argument

The resulting object in that case will simply mimic iteration over a single array. This change obsoletes distinctions like

```
if len(x) == 1:
    shape = x[0].shape
else:
    shape = np.broadcast(*x).shape
```

Instead, `np.broadcast` can be used in all cases.

np.trace now respects array subclasses

This behaviour mimics that of other functions such as `np.diagonal` and ensures, e.g., that for masked arrays `np.trace(ma)` and `ma.trace()` give the same result.

`np.dot` now raises `TypeError` instead of `ValueError`

This behaviour mimics that of other functions such as `np.inner`. If the two arguments cannot be cast to a common type, it could have raised a `TypeError` or `ValueError` depending on their order. Now, `np.dot` will now always raise a `TypeError`.

`linalg.norm` return type changes

The `linalg.norm` function now does all its computations in floating point and returns floating results. This change fixes bugs due to integer overflow and the failure of `abs` with signed integers of minimum value, e.g., `int8(-128)`. For consistency, floats are used even where an integer might work.

15.64.8 Deprecations

Views of arrays in Fortran order

The `F_CONTIGUOUS` flag was used to signal that views using a dtype that changed the element size would change the first index. This was always problematical for arrays that were both `F_CONTIGUOUS` and `C_CONTIGUOUS` because `C_CONTIGUOUS` took precedence. Relaxed stride checking results in more such dual contiguous arrays and breaks some existing code as a result. Note that this also affects changing the dtype by assigning to the dtype attribute of an array. The aim of this deprecation is to restrict views to `C_CONTIGUOUS` arrays at some future time. A work around that is backward compatible is to use `a.T.view(...).T` instead. A parameter may also be added to the view method to explicitly ask for Fortran order views, but that will not be backward compatible.

Invalid arguments for array ordering

It is currently possible to pass in arguments for the `order` parameter in methods like `array.flatten` or `array.ravel` that were not one of the following: `'C'`, `'F'`, `'A'`, `'K'` (note that all of these possible values are both unicode and case insensitive). Such behavior will not be allowed in future releases.

Random number generator in the `testing` namespace

The Python standard library random number generator was previously exposed in the `testing` namespace as `testing.rand`. Using this generator is not recommended and it will be removed in a future release. Use generators from `numpy.random` namespace instead.

Random integer generation on a closed interval

In accordance with the Python C API, which gives preference to the half-open interval over the closed one, `np.random.random_integers` is being deprecated in favor of calling `np.random.randint`, which has been enhanced with the `dtype` parameter as described under “New Features”. However, `np.random.random_integers` will not be removed anytime soon.

15.64.9 FutureWarnings

Assigning to slices/views of `MaskedArray`

Currently a slice of a masked array contains a view of the original data and a copy-on-write view of the mask. Consequently, any changes to the slice's mask will result in a copy of the original mask being made and that new mask being changed rather than the original. For example, if we make a slice of the original like so, `view = original[:]`, then modifications to the data in one array will affect the data of the other but, because the mask will be copied during assignment operations, changes to the mask will remain local. A similar situation occurs when explicitly constructing a masked array using `MaskedArray(data, mask)`, the returned array will contain a view of `data` but the mask will be a copy-on-write view of `mask`.

In the future, these cases will be normalized so that the data and mask arrays are treated the same way and modifications to either will propagate between views. In 1.11, numpy will issue a `MaskedArrayFutureWarning` warning whenever user code modifies the mask of a view that in the future may cause values to propagate back to the original. To silence these warnings and make your code robust against the upcoming changes, you have two options: if you want to keep the current behavior, call `masked_view.unshare_mask()` before modifying the mask. If you want to get the future behavior early, use `masked_view._sharedmask = False`. However, note that setting the `_sharedmask` attribute will break following explicit calls to `masked_view.unshare_mask()`.

15.65 NumPy 1.10.4 Release Notes

This release is a bugfix source release motivated by a segfault regression. No windows binaries are provided for this release, as there appear to be bugs in the toolchain we use to generate those files. Hopefully that problem will be fixed for the next release. In the meantime, we suggest using one of the providers of windows binaries.

15.65.1 Compatibility notes

- The trace function now calls the trace method on subclasses of `ndarray`, except for `matrix`, for which the current behavior is preserved. This is to help with the units package of `AstroPy` and hopefully will not cause problems.

15.65.2 Issues Fixed

- gh-6922 BUG: `numpy.recarray.sort` segfaults on Windows.
- gh-6937 BUG: `busday_offset` does the wrong thing with modified preceding roll.
- gh-6949 BUG: Type is lost when slicing a subclass of `recarray`.

15.65.3 Merged PRs

The following PRs have been merged into 1.10.4. When the PR is a backport, the PR number for the original PR against master is listed.

- gh-6840 TST: Update travis testing script in 1.10.x
- gh-6843 BUG: Fix use of python 3 only `FileNotFoundError` in `test_f2py`.
- gh-6884 REL: Update `pavement.py` and `setup.py` to reflect current version.
- gh-6916 BUG: Fix `test_f2py` so it runs correctly in `runtests.py`.
- gh-6924 BUG: Fix segfault gh-6922.

- gh-6942 Fix datetime roll='modifiedpreceding' bug.
- gh-6943 DOC,BUG: Fix some latex generation problems.
- gh-6950 BUG trace is not subclass aware, `np.trace(ma) != ma.trace()`.
- gh-6952 BUG recarray slices should preserve subclass.

15.66 NumPy 1.10.3 Release Notes

N/A this release did not happen due to various screwups involving PyPI.

15.67 NumPy 1.10.2 Release Notes

This release deals with a number of bugs that turned up in 1.10.1 and adds various build and release improvements.

Numpy 1.10.1 supports Python 2.6 - 2.7 and 3.2 - 3.5.

15.67.1 Compatibility notes

Relaxed stride checking is no longer the default

There were back compatibility problems involving views changing the dtype of multidimensional Fortran arrays that need to be dealt with over a longer timeframe.

Fix swig bug in `numpy.i`

Relaxed stride checking revealed a bug in `array_is_fortran(a)`, that was using `PyArray_ISFORTRAN` to check for Fortran contiguity instead of `PyArray_IS_F_CONTIGUOUS`. You may want to regenerate swigged files using the updated `numpy.i`

Deprecate views changing dimensions in fortran order

This deprecates assignment of a new descriptor to the dtype attribute of a non-C-contiguous array if it result in changing the shape. This effectively bars viewing a multidimensional Fortran array using a dtype that changes the element size along the first axis.

The reason for the deprecation is that, when relaxed strides checking is enabled, arrays that are both C and Fortran contiguous are always treated as C contiguous which breaks some code that depended the two being mutually exclusive for non-scalar arrays of `ndim > 1`. This deprecation prepares the way to always enable relaxed stride checking.

15.67.2 Issues Fixed

- gh-6019 Masked array repr fails for structured array with multi-dimensional column.
- gh-6462 Median of empty array produces IndexError.
- gh-6467 Performance regression for record array access.
- gh-6468 numpy.interp uses 'left' value even when `x[0]==xp[0]`.
- gh-6475 np.allclose returns a memmap when one of its arguments is a memmap.
- gh-6491 Error in broadcasting stride_tricks array.
- gh-6495 Unrecognized command line option '-ffpe-summary' in gfortran.
- gh-6497 Failure of reduce operation on recarrays.
- gh-6498 Mention change in default casting rule in 1.10 release notes.
- gh-6530 The partition function errors out on empty input.
- gh-6532 numpy.inner return wrong inaccurate value sometimes.
- gh-6563 Intent(out) broken in recent versions of f2py.
- gh-6569 Cannot run tests after 'python setup.py build_ext -i'
- gh-6572 Error in broadcasting stride_tricks array component.
- gh-6575 BUG: Split produces empty arrays with wrong number of dimensions
- gh-6590 Fortran Array problem in numpy 1.10.
- gh-6602 Random __all__ missing choice and dirichlet.
- gh-6611 ma.dot no longer always returns a masked array in 1.10.
- gh-6618 NPY_FortranOrder in make_fortran() in numpy.i
- gh-6636 Memory leak in nested dtypes in numpy.recarray
- gh-6641 Subsetting recarray by fields yields a structured array.
- gh-6667 ma.make_mask handles ma.nomask input incorrectly.
- gh-6675 Optimized blas detection broken in master and 1.10.
- gh-6678 Getting unexpected error from: `X.dtype = complex` (or `Y = X.view(complex)`)
- gh-6718 f2py test fail in pip installed numpy-1.10.1 in virtualenv.
- gh-6719 Error compiling Cython file: Pythonic division not allowed without gil.
- gh-6771 Numpy.rec.fromarrays losing dtype metadata between versions 1.9.2 and 1.10.1
- gh-6781 The travis-ci script in maintenance/1.10.x needs fixing.
- gh-6807 Windows testing errors for 1.10.2

15.67.3 Merged PRs

The following PRs have been merged into 1.10.2. When the PR is a backport, the PR number for the original PR against master is listed.

- gh-5773 MAINT: Hide testing helper tracebacks when using them with pytest.
- gh-6094 BUG: Fixed a bug with string representation of masked structured arrays.
- gh-6208 MAINT: Speedup field access by removing unneeded safety checks.
- gh-6460 BUG: Replacing the `os.environ.clear` by less invasive procedure.
- gh-6470 BUG: Fix `AttributeError` in `numpy.distutils`.
- gh-6472 MAINT: Use Python 3.5 instead of 3.5-dev for travis 3.5 testing.
- gh-6474 REL: Update Paver script for sdist and auto-switch test warnings.
- gh-6478 BUG: Fix Intel compiler flags for OS X build.
- gh-6481 MAINT: `LIBPATH` with spaces is now supported Python 2.7+ and Win32.
- gh-6487 BUG: Allow nested use of parameters in definition of arrays in `f2py`.
- gh-6488 BUG: Extend common blocks rather than overwriting in `f2py`.
- gh-6499 DOC: Mention that default casting for inplace operations has changed.
- gh-6500 BUG: Recarrays viewed as subarrays don't convert to `np.record` type.
- gh-6501 REL: Add "make upload" command for built docs, update "make dist".
- gh-6526 BUG: Fix use of `__doc__` in `setup.py` for `-OO` mode.
- gh-6527 BUG: Fix the `IndexError` when taking the median of an empty array.
- gh-6537 BUG: Make `ma.atleast_*` with scalar argument return arrays.
- gh-6538 BUG: Fix `ma.masked_values` does not shrink mask if requested.
- gh-6546 BUG: Fix inner product regression for non-contiguous arrays.
- gh-6553 BUG: Fix partition and `argpartition` error for empty input.
- gh-6556 BUG: Error in `broadcast_arrays` with `as_strided` array.
- gh-6558 MAINT: Minor update to "make upload" doc build command.
- gh-6562 BUG: Disable view safety checks in `recarray`.
- gh-6567 BUG: Revert some import * fixes in `f2py`.
- gh-6574 DOC: Release notes for Numpy 1.10.2.
- gh-6577 BUG: Fix for #6569, allowing `build_ext -inplace`
- gh-6579 MAINT: Fix mistake in doc upload rule.
- gh-6596 BUG: Fix `swig` for relaxed stride checking.
- gh-6606 DOC: Update 1.10.2 release notes.
- gh-6614 BUG: Add choice and `dirichlet` to `numpy.random.__all__`.
- gh-6621 BUG: Fix `swig` `make_fortran` function.
- gh-6628 BUG: Make `allclose` return python bool.
- gh-6642 BUG: Fix memleak in `_convert_from_dict`.

- gh-6643 ENH: make recarray.getitem return a recarray.
- gh-6653 BUG: Fix ma.dot to always return masked array.
- gh-6668 BUG: ma.make_mask should always return nomask for nomask argument.
- gh-6686 BUG: Fix a bug in assert_string_equal.
- gh-6695 BUG: Fix removing tempdirs created during build.
- gh-6697 MAINT: Fix spurious semicolon in macro definition of PyArray_FROM_OT.
- gh-6698 TST: test np rint bug for large integers.
- gh-6717 BUG: Readd fallback CBLAS detection on linux.
- gh-6721 BUG: Fix for #6719.
- gh-6726 BUG: Fix bugs exposed by relaxed stride rollback.
- gh-6757 BUG: link cblas library if cblas is detected.
- gh-6756 TST: only test f2py, not f2py2.7 etc, fixes #6718.
- gh-6747 DEP: Deprecate changing shape of non-C-contiguous array via descr.
- gh-6775 MAINT: Include from __future__ boilerplate in some files missing it.
- gh-6780 BUG: metadata is not copied to base_dtype.
- gh-6783 BUG: Fix travis ci testing for new google infrastructure.
- gh-6785 BUG: Quick and dirty fix for interp.
- gh-6813 TST,BUG: Make test_mvoid_multidim_print work for 32 bit systems.
- gh-6817 BUG: Disable 32-bit msvc9 compiler optimizations for npy_rint.
- gh-6819 TST: Fix test_mvoid_multidim_print failures on Python 2.x for Windows.

Initial support for mingwpy was reverted as it was causing problems for non-windows builds.

- gh-6536 BUG: Revert gh-5614 to fix non-windows build problems

A fix for np.lib.split was reverted because it resulted in “fixing” behavior that will be present in the Numpy 1.11 and that was already present in Numpy 1.9. See the discussion of the issue at gh-6575 for clarification.

- gh-6576 BUG: Revert gh-6376 to fix split behavior for empty arrays.

Relaxed stride checking was reverted. There were back compatibility problems involving views changing the dtype of multidimensional Fortran arrays that need to be dealt with over a longer timeframe.

- gh-6735 MAINT: Make no relaxed stride checking the default for 1.10.

15.67.4 Notes

A bug in the Numpy 1.10.1 release resulted in exceptions being raised for `RuntimeWarning` and `DeprecationWarning` in projects depending on Numpy. That has been fixed.

15.68 NumPy 1.10.1 Release Notes

This release deals with a few build problems that showed up in 1.10.0. Most users would not have seen these problems. The differences are:

- Compiling with msvc9 or msvc10 for 32 bit Windows now requires SSE2. This was the easiest fix for what looked to be some miscompiled code when SSE2 was not used. If you need to compile for 32 bit Windows systems without SSE2 support, mingw32 should still work.
- Make compiling with VS2008 python2.7 SDK easier
- Change Intel compiler options so that code will also be generated to support systems without SSE4.2.
- Some `_config` test functions needed an explicit integer return in order to avoid the openSUSE rpmlinter erring out.
- We ran into a problem with pipy not allowing reuse of filenames and a resulting proliferation of `..*.postN` releases. Not only were the names getting out of hand, some packages were unable to work with the postN suffix.

NumPy 1.10.1 supports Python 2.6 - 2.7 and 3.2 - 3.5.

Commits:

45a3d84 DEP: Remove warning for *full* when dtype is set. 0c1a5df BLD: import setuptools to allow compile with VS2008 python2.7 sdk 04211c6 BUG: mask nan to 1 in ordered compare 826716f DOC: Document the reason msvc requires SSE2 on 32 bit platforms. 49fa187 BLD: enable SSE2 for 32-bit msvc 9 and 10 compilers dcbc4cc MAINT: remove Wreturn-type warnings from config checks d6564cb BLD: do not build exclusively for SSE4.2 processors 15cb66f BLD: do not build exclusively for SSE4.2 processors c38bc08 DOC: fix var. reference in percentile docstring 78497f4 DOC: Sync 1.10.0-notes.rst in 1.10.x branch with master.

15.69 NumPy 1.10.0 Release Notes

This release supports Python 2.6 - 2.7 and 3.2 - 3.5.

15.69.1 Highlights

- `numpy.distutils` now supports parallel compilation via the `--parallel/-j` argument passed to `setup.py build`
- `numpy.distutils` now supports additional customization via `site.cfg` to control compilation parameters, i.e. runtime libraries, extra linking/compilation flags.
- Addition of `np.linalg.multi_dot`: compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order.
- The new function `np.stack` provides a general interface for joining a sequence of arrays along a new axis, complementing `np.concatenate` for joining along an existing axis.
- Addition of `nanprod` to the set of nanfunctions.
- Support for the `@` operator in Python 3.5.

15.69.2 Dropped Support

- The `_dotblas` module has been removed. CBLAS Support is now in Multiarray.
- The `testcalcs.py` file has been removed.
- The `polytemplate.py` file has been removed.
- `numpy_PyFile_Dup` and `numpy_PyFile_DupClose` have been removed from `numpy_3kcompat.h`.
- `splitcmdline` has been removed from `numpy/distutils/exec_command.py`.
- `try_run` and `get_output` have been removed from `numpy/distutils/command/config.py`.
- The `a_format` attribute is no longer supported for array printing.
- Keywords `skiprows` and `missing` removed from `np.genfromtxt`.
- Keyword `old_behavior` removed from `np.correlate`.

15.69.3 Future Changes

- In array comparisons like `arr1 == arr2`, many corner cases involving strings or structured dtypes that used to return scalars now issue `FutureWarning` or `DeprecationWarning`, and in the future will be change to either perform elementwise comparisons or raise an error.
- In `np.lib.split` an empty array in the result always had dimension `(0,)` no matter the dimensions of the array being split. In Numpy 1.11 that behavior will be changed so that the dimensions will be preserved. A `FutureWarning` for this change has been in place since Numpy 1.9 but, due to a bug, sometimes no warning was raised and the dimensions were already preserved.
- The `SafeEval` class will be removed in Numpy 1.11.
- The `alterdot` and `restoredot` functions will be removed in Numpy 1.11.

See below for more details on these changes.

15.69.4 Compatibility notes

Default casting rule change

Default casting for inplace operations has changed to `'same_kind'`. For instance, if `n` is an array of integers, and `f` is an array of floats, then `n += f` will result in a `TypeError`, whereas in previous Numpy versions the floats would be silently cast to ints. In the unlikely case that the example code is not an actual bug, it can be updated in a backward compatible way by rewriting it as `np.add(n, f, out=n, casting='unsafe')`. The old `'unsafe'` default has been deprecated since Numpy 1.7.

numpy version string

The numpy version string for development builds has been changed from `x.y.z.dev-githash` to `x.y.z.dev0+githash` (note the `+`) in order to comply with PEP 440.

relaxed stride checking

NPY_RELAXED_STRIDE_CHECKING is now true by default.

UPDATE: In 1.10.2 the default value of NPY_RELAXED_STRIDE_CHECKING was changed to false for back compatibility reasons. More time is needed before it can be made the default. As part of the roadmap a deprecation of dimension changing views of f_contiguous not c_contiguous arrays was also added.

Concatenation of 1d arrays along any but `axis=0` raises `IndexError`

Using `axis != 0` has raised a `DeprecationWarning` since NumPy 1.7, it now raises an error.

np.ravel, *np.diagonal* and *np.diag* now preserve subtypes

There was inconsistent behavior between *x.ravel()* and *np.ravel(x)*, as well as between *x.diagonal()* and *np.diagonal(x)*, with the methods preserving subtypes while the functions did not. This has been fixed and the functions now behave like the methods, preserving subtypes except in the case of matrices. Matrices are special cased for backward compatibility and still return 1-D arrays as before. If you need to preserve the matrix subtype, use the methods instead of the functions.

rollaxis and *swapaxes* always return a view

Previously, a view was returned except when no change was made in the order of the axes, in which case the input array was returned. A view is now returned in all cases.

nonzero now returns base ndarrays

Previously, an inconsistency existed between 1-D inputs (returning a base ndarray) and higher dimensional ones (which preserved subclasses). Behavior has been unified, and the return will now be a base ndarray. Subclasses can still override this behavior by providing their own *nonzero* method.

C API

The changes to *swapaxes* also apply to the *PyArray_SwapAxes* C function, which now returns a view in all cases.

The changes to *nonzero* also apply to the *PyArray_Nonzero* C function, which now returns a base ndarray in all cases.

The dtype structure (*PyArray_Descr*) has a new member at the end to cache its hash value. This shouldn't affect any well-written applications.

The change to the concatenation function `DeprecationWarning` also affects *PyArray_ConcatenateArrays*,

recarray field return types

Previously the returned types for recarray fields accessed by attribute and by index were inconsistent, and fields of string type were returned as chararrays. Now, fields accessed by either attribute or indexing will return an ndarray for fields of non-structured type, and a recarray for fields of structured type. Notably, this affects recarrays containing strings with whitespace, as trailing whitespace is trimmed from chararrays but kept in ndarrays of string type. Also, the `dtype.type` of nested structured fields is now inherited.

recarray views

Viewing an ndarray as a recarray now automatically converts the dtype to np.record. See new record array documentation. Additionally, viewing a recarray with a non-structured dtype no longer converts the result's type to ndarray - the result will remain a recarray.

'out' keyword argument of ufuncs now accepts tuples of arrays

When using the 'out' keyword argument of a ufunc, a tuple of arrays, one per ufunc output, can be provided. For ufuncs with a single output a single array is also a valid 'out' keyword argument. Previously a single array could be provided in the 'out' keyword argument, and it would be used as the first output for ufuncs with multiple outputs, is deprecated, and will result in a *DeprecationWarning* now and an error in the future.

byte-array indices now raises an IndexError

Indexing an ndarray using a byte-string in Python 3 now raises an IndexError instead of a ValueError.

Masked arrays containing objects with arrays

For such (rare) masked arrays, getting a single masked item no longer returns a corrupted masked array, but a fully masked version of the item.

Median warns and returns nan when invalid values are encountered

Similar to mean, median and percentile now emits a Runtime warning and returns *NaN* in slices where a *NaN* is present. To compute the median or percentile while ignoring invalid values use the new *nanmedian* or *nanpercentile* functions.

Functions available from numpy.ma.testutils have changed

All functions from numpy.testing were once available from numpy.ma.testutils but not all of them were redefined to work with masked arrays. Most of those functions have now been removed from numpy.ma.testutils with a small subset retained in order to preserve backward compatibility. In the long run this should help avoid mistaken use of the wrong functions, but it may cause import problems for some.

15.69.5 New Features

Reading extra flags from site.cfg

Previously customization of compilation of dependency libraries and numpy itself was only accomplishable via code changes in the distutils package. Now numpy.distutils reads in the following extra flags from each group of the *site.cfg*:

- **runtime_library_dirs/rpath**, sets runtime library directories to override

LD_LIBRARY_PATH

- **extra_compile_args**, add extra flags to the compilation of sources
- **extra_link_args**, add extra flags when linking libraries

This should, at least partially, complete user customization.

***np.cbrt* to compute cube root for real floats**

np.cbrt wraps the C99 cube root function *cbrt*. Compared to *np.power(x, 1./3.)* it is well defined for negative real floats and a bit faster.

By passing *-parallel=n* or *-jn* to *setup.py build* the compilation of extensions is now performed in *n* parallel processes. The parallelization is limited to files within one extension so projects using Cython will not profit because it builds extensions from single files.

***genfromtxt* has a new `max_rows` argument**

A `max_rows` argument has been added to *genfromtxt* to limit the number of rows read in a single call. Using this functionality, it is possible to read in multiple arrays stored in a single file by making repeated calls to the function.

New function *np.broadcast_to* for invoking array broadcasting

np.broadcast_to manually broadcasts an array to a given shape according to numpy's broadcasting rules. The functionality is similar to *broadcast_arrays*, which in fact has been rewritten to use *broadcast_to* internally, but only a single array is necessary.

New context manager *clear_and_catch_warnings* for testing warnings

When Python emits a warning, it records that this warning has been emitted in the module that caused the warning, in a module attribute `__warningregistry__`. Once this has happened, it is not possible to emit the warning again, unless you clear the relevant entry in `__warningregistry__`. This makes it hard and fragile to test warnings, because if your test comes after another that has already caused the warning, you will not be able to emit the warning or test it. The context manager *clear_and_catch_warnings* clears warnings from the module registry on entry and resets them on exit, meaning that warnings can be re-raised.

***cov* has new `fweights` and `aweights` arguments**

The `fweights` and `aweights` arguments add new functionality to covariance calculations by applying two types of weighting to observation vectors. An array of `fweights` indicates the number of repeats of each observation vector, and an array of `aweights` provides their relative importance or probability.

Support for the '@' operator in Python 3.5+

Python 3.5 adds support for a matrix multiplication operator '@' proposed in PEP465. Preliminary support for that has been implemented, and an equivalent function *matmul* has also been added for testing purposes and use in earlier Python versions. The function is preliminary and the order and number of its optional arguments can be expected to change.

New argument `norm` to `fft` functions

The default normalization has the direct transforms unscaled and the inverse transforms are scaled by $1/n$. It is possible to obtain unitary transforms by setting the keyword argument `norm` to `"ortho"` (default is `None`) so that both direct and inverse transforms will be scaled by $1/\sqrt{n}$.

15.69.6 Improvements

`np.digitize` using binary search

`np.digitize` is now implemented in terms of `np.searchsorted`. This means that a binary search is used to bin the values, which scales much better for larger number of bins than the previous linear search. It also removes the requirement for the input array to be 1-dimensional.

`np.poly` now casts integer inputs to float

`np.poly` will now cast 1-dimensional input arrays of integer type to double precision floating point, to prevent integer overflow when computing the monic polynomial. It is still possible to obtain higher precision results by passing in an array of object type, filled e.g. with Python ints.

`np.interp` can now be used with periodic functions

`np.interp` now has a new parameter `period` that supplies the period of the input data `xp`. In such case, the input data is properly normalized to the given period and one end point is added to each extremity of `xp` in order to close the previous and the next period cycles, resulting in the correct interpolation behavior.

`np.pad` supports more input types for `pad_width` and `constant_values`

`constant_values` parameters now accepts NumPy arrays and float values. NumPy arrays are supported as input for `pad_width`, and an exception is raised if its values are not of integral type.

`np.argmax` and `np.argmin` now support an `out` argument

The `out` parameter was added to `np.argmax` and `np.argmin` for consistency with `ndarray.argmax` and `ndarray.argmin`. The new parameter behaves exactly as it does in those methods.

More system C99 complex functions detected and used

All of the functions in `complex.h` are now detected. There are new fallback implementations of the following functions.

- `npy_ctan`,
- `npy_cacos`, `npy_casin`, `npy_catan`
- `npy_ccosh`, `npy_csinh`, `npy_ctanh`,
- `npy_cacosh`, `npy_casinh`, `npy_catanh`

As a result of these improvements, there will be some small changes in returned values, especially for corner cases.

***np.loadtxt* support for the strings produced by the `float.hex` method**

The strings produced by `float.hex` look like `0x1.921fb54442d18p+1`, so this is not the hex used to represent unsigned integer types.

***np.isclose* properly handles minimal values of integer dtypes**

In order to properly handle minimal values of integer types, *np.isclose* will now cast to the float dtype during comparisons. This aligns its behavior with what was provided by *np.allclose*.

***np.allclose* uses *np.isclose* internally.**

np.allclose now uses *np.isclose* internally and inherits the ability to compare NaNs as equal by setting `equal_nan=True`. Subclasses, such as *np.ma.MaskedArray*, are also preserved now.

***np.genfromtxt* now handles large integers correctly**

np.genfromtxt now correctly handles integers larger than $2^{31}-1$ on 32-bit systems and larger than $2^{63}-1$ on 64-bit systems (it previously crashed with an `OverflowError` in these cases). Integers larger than $2^{63}-1$ are converted to floating-point values.

***np.load*, *np.save* have pickle backward compatibility flags**

The functions *np.load* and *np.save* have additional keyword arguments for controlling backward compatibility of pickled Python objects. This enables Numpy on Python 3 to load npy files containing object arrays that were generated on Python 2.

MaskedArray support for more complicated base classes

Built-in assumptions that the baseclass behaved like a plain array are being removed. In particular, setting and getting elements and ranges will respect baseclass overrides of `__setitem__` and `__getitem__`, and arithmetic will respect overrides of `__add__`, `__sub__`, etc.

15.69.7 Changes

dotblas functionality moved to multiarray

The cblas versions of `dot`, `inner`, and `vdot` have been integrated into the `multiarray` module. In particular, `vdot` is now a `multiarray` function, which it was not before.

stricter check of gufunc signature compliance

Inputs to generalized universal functions are now more strictly checked against the function's signature: all core dimensions are now required to be present in input arrays; core dimensions with the same label must have the exact same size; and output core dimension's must be specified, either by a same label input core dimension or by a passed-in output array.

views returned from *np.einsum* are writeable

Views returned by *np.einsum* will now be writeable whenever the input array is writeable.

np.argmin skips NaT values

np.argmin now skips NaT values in *datetime64* and *timedelta64* arrays, making it consistent with *np.min*, *np.argmax* and *np.max*.

15.69.8 Deprecations

Array comparisons involving strings or structured dtypes

Normally, comparison operations on arrays perform elementwise comparisons and return arrays of booleans. But in some corner cases, especially involving strings or structured dtypes, NumPy has historically returned a scalar instead. For example:

```

### Current behaviour

np.arange(2) == "foo"
# -> False

np.arange(2) < "foo"
# -> True on Python 2, error on Python 3

np.ones(2, dtype="i4,i4") == np.ones(2, dtype="i4,i4,i4")
# -> False

```

Continuing work started in 1.9, in 1.10 these comparisons will now raise *FutureWarning* or *DeprecationWarning*, and in the future they will be modified to behave more consistently with other comparison operations, e.g.:

```

### Future behaviour

np.arange(2) == "foo"
# -> array([False, False])

np.arange(2) < "foo"
# -> error, strings and numbers are not orderable

np.ones(2, dtype="i4,i4") == np.ones(2, dtype="i4,i4,i4")
# -> [False, False]

```

SafeEval

The SafeEval class in `numpy/lib/utls.py` is deprecated and will be removed in the next release.

alterdot, restoredot

The `alterdot` and `restoredot` functions no longer do anything, and are deprecated.

pkgload, PackageLoader

These ways of loading packages are now deprecated.

bias, ddof arguments to corrcoef

The values for the `bias` and `ddof` arguments to the `corrcoef` function canceled in the division implied by the correlation coefficient and so had no effect on the returned values.

We now deprecate these arguments to `corrcoef` and the masked array version `ma.corrcoef`.

Because we are deprecating the `bias` argument to `ma.corrcoef`, we also deprecate the use of the `allow_masked` argument as a positional argument, as its position will change with the removal of `bias`. `allow_masked` will in due course become a keyword-only argument.

dtype string representation changes

Since 1.6, creating a dtype object from its string representation, e.g. `'f4'`, would issue a deprecation warning if the size did not correspond to an existing type, and default to creating a dtype of the default size for the type. Starting with this release, this will now raise a `TypeError`.

The only exception is object dtypes, where both `'O4'` and `'O8'` will still issue a deprecation warning. This platform-dependent representation will raise an error in the next release.

In preparation for this upcoming change, the string representation of an object dtype, i.e. `np.dtype(object).str`, no longer includes the item size, i.e. will return `'|O'` instead of `'|O4'` or `'|O8'` as before.

15.70 NumPy 1.9.2 Release Notes

This is a bugfix only release in the 1.9.x series.

15.70.1 Issues fixed

- [#5316](#): fix too large dtype alignment of strings and complex types
- [#5424](#): fix `ma.median` when used on ndarrays
- [#5481](#): Fix astype for structured array fields of different byte order
- [#5354](#): fix segfault when clipping complex arrays
- [#5524](#): allow `np.argpartition` on non ndarrays
- [#5612](#): Fixes `ndarray.fill` to accept full range of `uint64`

- [#5155](#): Fix loadtxt with comments=None and a string None data
- [#4476](#): Masked array view fails if structured dtype has datetime component
- [#5388](#): Make RandomState.set_state and RandomState.get_state threadsafe
- [#5390](#): make seed, randint and shuffle threadsafe
- [#5374](#): Fixed incorrect assert_array_almost_equal_nulp documentation
- [#5393](#): Add support for ATLAS > 3.9.33.
- [#5313](#): PyArray_AsCArray caused segfault for 3d arrays
- [#5492](#): handle out of memory in rfftf
- [#4181](#): fix a few bugs in the random.pareto docstring
- [#5359](#): minor changes to linspace docstring
- [#4723](#): fix a compile issues on AIX

15.71 NumPy 1.9.1 Release Notes

This is a bugfix only release in the 1.9.x series.

15.71.1 Issues fixed

- [gh-5184](#): restore linear edge behaviour of gradient to as it was in < 1.9. The second order behaviour is available via the `edge_order` keyword
- [gh-4007](#): workaround Accelerate sgemv crash on OSX 10.9
- [gh-5100](#): restore object dtype inference from iterable objects without `len()`
- [gh-5163](#): avoid gcc-4.1.2 (red hat 5) miscompilation causing a crash
- [gh-5138](#): fix nanmedian on arrays containing inf
- [gh-5240](#): fix not returning out array from ufuncs with subok=False set
- [gh-5203](#): copy inherited masks in MaskedArray.__array_finalize__
- [gh-2317](#): genfromtxt did not handle filling_values=0 correctly
- [gh-5067](#): restore api of npy_PyFile_DupClose in python2
- [gh-5063](#): cannot convert invalid sequence index to tuple
- [gh-5082](#): Segmentation fault with argmin() on unicode arrays
- [gh-5095](#): don't propagate subtypes from np.where
- [gh-5104](#): np.inner segfaults with SciPy's sparse matrices
- [gh-5251](#): Issue with fromarrays not using correct format for unicode arrays
- [gh-5136](#): Import dummy_threading if importing threading fails
- [gh-5148](#): Make numpy import when run with Python flag '-OO'
- [gh-5147](#): Einsum double contraction in particular order causes ValueError
- [gh-479](#): Make f2py work with intent(in out)

- gh-5170: Make python2 .npy files readable in python3
- gh-5027: Use 'l' as the default length specifier for long long
- gh-4896: fix build error with MSVC 2013 caused by C99 complex support
- gh-4465: Make PyArray_PutTo respect writeable flag
- gh-5225: fix crash when using arange on datetime without dtype set
- gh-5231: fix build in c99 mode

15.72 NumPy 1.9.0 Release Notes

This release supports Python 2.6 - 2.7 and 3.2 - 3.4.

15.72.1 Highlights

- Numerous performance improvements in various areas, most notably indexing and operations on small arrays are significantly faster. Indexing operations now also release the GIL.
- Addition of *nanmedian* and *nanpercentile* rounds out the nanfunction set.

15.72.2 Dropped Support

- The oldnumeric and numarray modules have been removed.
- The doc/pyrex and doc/cython directories have been removed.
- The doc/numpybook directory has been removed.
- The numpy/testing/numpytest.py file has been removed together with the importall function it contained.

15.72.3 Future Changes

- The numpy/polynomial/polytemplate.py file will be removed in NumPy 1.10.0.
- Default casting for inplace operations will change to 'same_kind' in Numpy 1.10.0. This will certainly break some code that is currently ignoring the warning.
- Relaxed stride checking will be the default in 1.10.0
- String version checks will break because, e.g., '1.9' > '1.10' is True. A NumpyVersion class has been added that can be used for such comparisons.
- The diagonal and diag functions will return writeable views in 1.10.0
- The S and/or a dtypes may be changed to represent Python strings instead of bytes, in Python 3 these two types are very different.

15.72.4 Compatibility notes

The diagonal and diag functions return readonly views.

In NumPy 1.8, the diagonal and diag functions returned readonly copies, in NumPy 1.9 they return readonly views, and in 1.10 they will return writeable views.

Special scalar float values don't cause upcast to double anymore

In previous numpy versions operations involving floating point scalars containing special values `NaN`, `Inf` and `-Inf` caused the result type to be at least `float64`. As the special values can be represented in the smallest available floating point type, the upcast is not performed anymore.

For example the dtype of:

```
np.array([1.], dtype=np.float32) * float('nan')
```

now remains `float32` instead of being cast to `float64`. Operations involving non-special values have not been changed.

Percentile output changes

If given more than one percentile to compute `numpy.percentile` returns an array instead of a list. A single percentile still returns a scalar. The array is equivalent to converting the list returned in older versions to an array via `np.array`.

If the `overwrite_input` option is used the input is only partially instead of fully sorted.

ndarray.tofile exception type

All `tofile` exceptions are now `IOError`, some were previously `ValueError`.

Invalid fill value exceptions

Two changes to `numpy.ma.core._check_fill_value`:

- When the fill value is a string and the array type is not one of 'OSUV', `TypeError` is raised instead of the default fill value being used.
- When the fill value overflows the array type, `TypeError` is raised instead of `OverflowError`.

Polynomial Classes no longer derived from PolyBase

This may cause problems with folks who depended on the polynomial classes being derived from `PolyBase`. They are now all derived from the abstract base class `ABCPolyBase`. Strictly speaking, there should be a deprecation involved, but no external code making use of the old baseclass could be found.

Using `numpy.random.binomial` may change the RNG state vs. `numpy < 1.9`

A bug in one of the algorithms to generate a binomial random variate has been fixed. This change will likely alter the number of random draws performed, and hence the sequence location will be different after a call to `distribution.c::rk_binomial_btpe`. Any tests which rely on the RNG being in a known state should be checked and/or updated as a result.

Random seed enforced to be a 32 bit unsigned integer

`np.random.seed` and `np.random.RandomState` now throw a `ValueError` if the seed cannot safely be converted to 32 bit unsigned integers. Applications that now fail can be fixed by masking the higher 32 bit values to zero: `seed = seed & 0xFFFFFFFF`. This is what is done silently in older versions so the random stream remains the same.

Argmin and argmax out argument

The `out` argument to `np.argmin` and `np.argmax` and their equivalent C-API functions is now checked to match the desired output shape exactly. If the check fails a `ValueError` instead of `TypeError` is raised.

Einsum

Remove unnecessary broadcasting notation restrictions. `np.einsum('ijk,j->ijk', A, B)` can also be written as `np.einsum('ij...,j->ij...', A, B)` (ellipsis is no longer required on 'j')

Indexing

The NumPy indexing has seen a complete rewrite in this version. This makes most advanced integer indexing operations much faster and should have no other implications. However some subtle changes and deprecations were introduced in advanced indexing operations:

- Boolean indexing into scalar arrays will always return a new 1-d array. This means that `array(1)[array(True)]` gives `array([1])` and not the original array.
- Advanced indexing into one dimensional arrays used to have (undocumented) special handling regarding repeating the value array in assignments when the shape of the value array was too small or did not match. Code using this will raise an error. For compatibility you can use `arr.flat[index] = values`, which uses the old code branch. (for example `a = np.ones(10); a[np.arange(10)] = [1, 2, 3]`)
- The iteration order over advanced indexes used to be always C-order. In NumPy 1.9, the iteration order adapts to the inputs and is not guaranteed (with the exception of a *single* advanced index which is never reversed for compatibility reasons). This means that the result is undefined if multiple values are assigned to the same element. An example for this is `arr[[0, 0], [1, 1]] = [1, 2]`, which may set `arr[0, 1]` to either 1 or 2.
- Equivalent to the iteration order, the memory layout of the advanced indexing result is adapted for faster indexing and cannot be predicted.
- All indexing operations return a view or a copy. No indexing operation will return the original array object. (For example `arr[...]`)
- In the future Boolean array-likes (such as lists of python bools) will always be treated as Boolean indexes and Boolean scalars (including python `True`) will be a legal *boolean* index. At this time, this is already the case for scalar arrays to allow the general `positive = a[a > 0]` to work when `a` is zero dimensional.

- In NumPy 1.8 it was possible to use `array(True)` and `array(False)` equivalent to 1 and 0 if the result of the operation was a scalar. This will raise an error in NumPy 1.9 and, as noted above, treated as a boolean index in the future.
- All non-integer array-likes are deprecated, object arrays of custom integer like objects may have to be cast explicitly.
- The error reporting for advanced indexing is more informative, however the error type has changed in some cases. (Broadcasting errors of indexing arrays are reported as `IndexError`)
- Indexing with more than one ellipsis (`. . .`) is deprecated.

Non-integer reduction axis indexes are deprecated

Non-integer axis indexes to reduction ufuncs like `add.reduce` or `sum` are deprecated.

`promote_types` and string dtype

`promote_types` function now returns a valid string length when given an integer or float dtype as one argument and a string dtype as another argument. Previously it always returned the input string dtype, even if it wasn't long enough to store the max integer/float value converted to a string.

`can_cast` and string dtype

`can_cast` function now returns `False` in “safe” casting mode for integer/float dtype and string dtype if the string dtype length is not long enough to store the max integer/float value converted to a string. Previously `can_cast` in “safe” mode returned `True` for integer/float dtype and a string dtype of any length.

`astype` and string dtype

The `astype` method now returns an error if the string dtype to cast to is not long enough in “safe” casting mode to hold the max value of integer/float array that is being casted. Previously the casting was allowed even if the result was truncated.

`numpyio.recfromcsv` keyword arguments change

`numpyio.recfromcsv` no longer accepts the undocumented `update` keyword, which used to override the `dtype` keyword.

The `doc/swig` directory moved

The `doc/swig` directory has been moved to `tools/swig`.

The `numpy_3kcompat.h` header changed

The unused `simple_capsule_dtor` function has been removed from `numpy_3kcompat.h`. Note that this header is not meant to be used outside of numpy; other projects should be using their own copy of this file when needed.

Negative indices in C-API `sq_item` and `sq_ass_item` sequence methods

When directly accessing the `sq_item` or `sq_ass_item` PyObject slots for item getting, negative indices will not be supported anymore. `PySequence_GetItem` and `PySequence_SetItem` however fix negative indices so that they can be used there.

NDIter

When `NpyIter_RemoveAxis` is now called, the iterator range will be reset.

When a multi index is being tracked and an iterator is not buffered, it is possible to use `NpyIter_RemoveAxis`. In this case an iterator can shrink in size. Because the total size of an iterator is limited, the iterator may be too large before these calls. In this case its size will be set to `-1` and an error issued not at construction time but when removing the multi index, setting the iterator range, or getting the next function.

This has no effect on currently working code, but highlights the necessity of checking for an error return if these conditions can occur. In most cases the arrays being iterated are as large as the iterator so that such a problem cannot occur.

This change was already applied to the 1.8.1 release.

`zeros_like` for string dtypes now returns empty strings

To match the `zeros` function `zeros_like` now returns an array initialized with empty strings instead of an array filled with `0`.

15.72.5 New Features

Percentile supports more interpolation options

`np.percentile` now has the interpolation keyword argument to specify in which way points should be interpolated if the percentiles fall between two values. See the documentation for the available options.

Generalized axis support for median and percentile

`np.median` and `np.percentile` now support generalized axis arguments like ufunc reductions do since 1.7. One can now say `axis=(index, index)` to pick a list of axes for the reduction. The `keepdims` keyword argument was also added to allow convenient broadcasting to arrays of the original shape.

Dtype parameter added to `np.linspace` and `np.logspace`

The returned data type from the `linspace` and `logspace` functions can now be specified using the `dtype` parameter.

More general `np.triu` and `np.tril` broadcasting

For arrays with `ndim` exceeding 2, these functions will now apply to the final two axes instead of raising an exception.

`tobytes` alias for `tostring` method

`ndarray.tobytes` and `MaskedArray.tobytes` have been added as aliases for `tostring` which exports arrays as bytes. This is more consistent in Python 3 where `str` and `bytes` are not the same.

Build system

Added experimental support for the ppc64le and OpenRISC architecture.

Compatibility to python `numbers` module

All numerical numpy types are now registered with the type hierarchy in the python `numbers` module.

`increasing` parameter added to `np.vander`

The ordering of the columns of the Vandermonde matrix can be specified with this new boolean argument.

`unique_counts` parameter added to `np.unique`

The number of times each unique item comes up in the input can now be obtained as an optional return value.

Support for median and percentile in nanfunctions

The `np.nanmedian` and `np.nanpercentile` functions behave like the `median` and `percentile` functions except that NaNs are ignored.

NumpyVersion class added

The class may be imported from `numpy.lib` and can be used for version comparison when the numpy version goes to 1.10.devel. For example:

```
>>> from numpy.lib import NumpyVersion
>>> if NumpyVersion(np.__version__) < '1.10.0':
...     print('Wow, that is an old NumPy version!')
```

Allow saving arrays with large number of named columns

The numpy storage format 1.0 only allowed the array header to have a total size of 65535 bytes. This can be exceeded by structured arrays with a large number of columns. A new format 2.0 has been added which extends the header size to 4 GiB. `np.save` will automatically save in 2.0 format if the data requires it, else it will always use the more compatible 1.0 format.

Full broadcasting support for `np.cross`

`np.cross` now properly broadcasts its two input arrays, even if they have different number of dimensions. In earlier versions this would result in either an error being raised, or wrong results computed.

15.72.6 Improvements

Better numerical stability for sum in some cases

Pairwise summation is now used in the sum method, but only along the fast axis and for groups of the values ≤ 8192 in length. This should also improve the accuracy of var and std in some common cases.

Percentile implemented in terms of `np.partition`

`np.percentile` has been implemented in terms of `np.partition` which only partially sorts the data via a selection algorithm. This improves the time complexity from $O(n \log(n))$ to $O(n)$.

Performance improvement for `np.array`

The performance of converting lists containing arrays to arrays using `np.array` has been improved. It is now equivalent in speed to `np.vstack(list)`.

Performance improvement for `np.searchsorted`

For the built-in numeric types, `np.searchsorted` no longer relies on the data type's `compare` function to perform the search, but is now implemented by type specific functions. Depending on the size of the inputs, this can result in performance improvements over 2x.

Optional reduced verbosity for `np.distutils`

Set `numpy.distutils.system_info.system_info.verbosity = 0` and then calls to `numpy.distutils.system_info.get_info('blas_opt')` will not print anything on the output. This is mostly for other packages using `numpy.distutils`.

Covariance check in `np.random.multivariate_normal`

A `RuntimeWarning` warning is raised when the covariance matrix is not positive-semidefinite.

Polynomial Classes no longer template based

The polynomial classes have been refactored to use an abstract base class rather than a template in order to implement a common interface. This makes importing the polynomial package faster as the classes do not need to be compiled on import.

More GIL releases

Several more functions now release the Global Interpreter Lock allowing more efficient parallelization using the `threading` module. Most notably the GIL is now released for fancy indexing, `np.where` and the `random` module now uses a per-state lock instead of the GIL.

MaskedArray support for more complicated base classes

Built-in assumptions that the baseclass behaved like a plain array are being removed. In particular, `repr` and `str` should now work more reliably.

C-API

15.72.7 Deprecations

Non-integer scalars for sequence repetition

Using non-integer numpy scalars to repeat python sequences is deprecated. For example `np.float_(2) * [1]` will be an error in the future.

`select` input deprecations

The integer and empty input to `select` is deprecated. In the future only boolean arrays will be valid conditions and an empty `condlist` will be considered an input error instead of returning the default.

`rank` function

The `rank` function has been deprecated to avoid confusion with `numpy.linalg.matrix_rank`.

Object array equality comparisons

In the future object array comparisons both `==` and `np.equal` will not make use of identity checks anymore. For example:

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

will consistently return `False` (and in the future an error) even if the array in *a* and *b* was the same object.

The equality operator `==` will in the future raise errors like `np.equal` if broadcasting or element comparisons, etc. fails.

Comparison with `arr == None` will in the future do an elementwise comparison instead of just returning `False`. Code should be using `arr is None`.

All of these changes will give Deprecation- or FutureWarnings at this time.

C-API

The utility function `npy_PyFile_Dup` and `npy_PyFile_DupClose` are broken by the internal buffering python 3 applies to its file objects. To fix this two new functions `npy_PyFile_Dup2` and `npy_PyFile_DupClose2` are declared in `npy_3kcompat.h` and the old functions are deprecated. Due to the fragile nature of these functions it is recommended to instead use the python API when possible.

This change was already applied to the 1.8.1 release.

15.73 NumPy 1.8.2 Release Notes

This is a bugfix only release in the 1.8.x series.

15.73.1 Issues fixed

- gh-4836: partition produces wrong results for multiple selections in equal ranges
- gh-4656: Make `fftpack._raw_fft` threadsafe
- gh-4628: incorrect argument order to `_copyto` in `np.nanmax`, `np.nanmin`
- gh-4642: Hold GIL for converting dtypes types with fields
- gh-4733: fix `np.linalg.svd(b, compute_uv=False)`
- gh-4853: avoid unaligned simd load on reductions on i386
- gh-4722: Fix seg fault converting empty string to object
- gh-4613: Fix lack of NULL check in `array_richcompare`
- gh-4774: avoid unaligned access for strided byteswap
- gh-650: Prevent division by zero when creating arrays from some buffers
- gh-4602: ifort has issues with optimization flag `O2`, use `O1`

15.74 NumPy 1.8.1 Release Notes

This is a bugfix only release in the 1.8.x series.

15.74.1 Issues fixed

- gh-4276: Fix mean, var, std methods for object arrays
- gh-4262: remove insecure mktemp usage
- gh-2385: `absolute(complex(inf))` raises invalid warning in python3
- gh-4024: Sequence assignment doesn't raise exception on shape mismatch
- gh-4027: Fix chunked reading of strings longer than `BUFFERSIZE`
- gh-4109: Fix object scalar return type of 0-d array indices
- gh-4018: fix missing check for memory allocation failure in ufuncs
- gh-4156: high order `linalg.norm` discards imaginary elements of complex arrays
- gh-4144: `linalg: norm` fails on `longdouble`, signed int
- gh-4094: fix NaT handling in `_strided_to_strided_string_to_datetime`
- gh-4051: fix uninitialized use in `_strided_to_strided_string_to_datetime`
- gh-4093: Loading compressed .npz file fails under Python 2.6.6
- gh-4138: segfault with non-native endian memoryview in python 3.4
- gh-4123: Fix missing NULL check in `lexsort`
- gh-4170: fix native-only long long check in memoryviews
- gh-4187: Fix large file support on 32 bit
- gh-4152: `fromfile`: ensure file handle positions are in sync in python3
- gh-4176: clang compatibility: Typos in `conversion_utils`
- gh-4223: Fetching a non-integer item caused array return
- gh-4197: fix minor memory leak in memoryview failure case
- gh-4206: fix build with single-threaded python
- gh-4220: add `versionadded:: 1.8.0` to `ufunc.at` docstring
- gh-4267: improve handling of memory allocation failure
- gh-4267: fix use of `capi` without `gil` in `ufunc.at`
- gh-4261: Detect vendor versions of GNU Compilers
- gh-4253: `IRR` was returning nan instead of valid negative answer
- gh-4254: fix unnecessary byte order flag change for byte arrays
- gh-3263: `numpy.random.shuffle` clobbers mask of a `MaskedArray`
- gh-4270: `np.random.shuffle` not work with flexible dtypes
- gh-3173: Segmentation fault when 'size' argument to `random.multinomial`
- gh-2799: allow using `unique` with lists of complex

- gh-3504: fix linspace truncation for integer array scalar
- gh-4191: get_info('openblas') does not read libraries key
- gh-3348: Access violation in _descriptor_from_pep3118_format
- gh-3175: segmentation fault with numpy.array() from bytearray
- gh-4266: histogramdd - wrong result for entries very close to last boundary
- gh-4408: Fix stride_stricks.as_strided function for object arrays
- gh-4225: fix log1p and expm1 return for np.inf on windows compiler builds
- gh-4359: Fix infinite recursion in str.format of flex arrays
- gh-4145: Incorrect shape of broadcast result with the exponent operator
- gh-4483: Fix commutativity of {dot,multiply,inner}(scalar, matrix_of_objs)
- gh-4466: Delay npyiter size check when size may change
- gh-4485: Buffered stride was erroneously marked fixed
- gh-4354: byte_bounds fails with datetime dtypes
- gh-4486: segfault/error converting from/to high-precision datetime64 objects
- gh-4428: einsum(None, None, None, None) causes segfault
- gh-4134: uninitialized use for for size 1 object reductions

15.74.2 Changes

NDIter

When `NpyIter_RemoveAxis` is now called, the iterator range will be reset.

When a multi index is being tracked and an iterator is not buffered, it is possible to use `NpyIter_RemoveAxis`. In this case an iterator can shrink in size. Because the total size of an iterator is limited, the iterator may be too large before these calls. In this case its size will be set to `-1` and an error issued not at construction time but when removing the multi index, setting the iterator range, or getting the next function.

This has no effect on currently working code, but highlights the necessity of checking for an error return if these conditions can occur. In most cases the arrays being iterated are as large as the iterator so that such a problem cannot occur.

Optional reduced verbosity for `np.distutils`

Set `numpy.distutils.system_info.system_info.verbosity = 0` and then calls to `numpy.distutils.system_info.get_info('blas_opt')` will not print anything on the output. This is mostly for other packages using `numpy.distutils`.

15.74.3 Deprecations

C-API

The utility function `numpy.PyFile_Dup` and `numpy.PyFile_DupClose` are broken by the internal buffering python 3 applies to its file objects. To fix this two new functions `numpy.PyFile_Dup2` and `numpy.PyFile_DupClose2` are declared in `numpy_3kcompat.h` and the old functions are deprecated. Due to the fragile nature of these functions it is recommended to instead use the python API when possible.

15.75 NumPy 1.8.0 Release Notes

This release supports Python 2.6 -2.7 and 3.2 - 3.3.

15.75.1 Highlights

- New, no 2to3, Python 2 and Python 3 are supported by a common code base.
- New, `gufuncs` for linear algebra, enabling operations on stacked arrays.
- New, inplace fancy indexing for ufuncs with the `.at` method.
- New, `partition` function, partial sorting via selection for fast median.
- New, `nanmean`, `nanvar`, and `nanstd` functions skipping NaNs.
- New, `full` and `full_like` functions to create value initialized arrays.
- New, `PyUFunc_RegisterLoopForDescr`, better ufunc support for user dtypes.
- Numerous performance improvements in many areas.

15.75.2 Dropped Support

Support for Python versions 2.4 and 2.5 has been dropped,

Support for SCons has been removed.

15.75.3 Future Changes

The `Datetime64` type remains experimental in this release. In 1.9 there will probably be some changes to make it more usable.

The `diagonal` method currently returns a new array and raises a `FutureWarning`. In 1.9 it will return a readonly view.

Multiple field selection from an array of structured type currently returns a new array and raises a `FutureWarning`. In 1.9 it will return a readonly view.

The `numpy/oldnumeric` and `numpy/numarray` compatibility modules will be removed in 1.9.

15.75.4 Compatibility notes

The doc/sphinxext content has been moved into its own github repository, and is included in numpy as a submodule. See the instructions in doc/HOWTO_BUILD_DOCS.rst.txt for how to access the content.

The hash function of numpy.void scalars has been changed. Previously the pointer to the data was hashed as an integer. Now, the hash function uses the tuple-hash algorithm to combine the hash functions of the elements of the scalar, but only if the scalar is read-only.

Numpy has switched its build system to using ‘separate compilation’ by default. In previous releases this was supported, but not default. This should produce the same results as the old system, but if you’re trying to do something complicated like link numpy statically or using an unusual compiler, then it’s possible you will encounter problems. If so, please file a bug and as a temporary workaround you can re-enable the old build system by exporting the shell variable `NPY_SEPARATE_COMPILATION=0`.

For the AdvancedNew iterator the `oa_ndim` flag should now be -1 to indicate that no `op_axes` and `itershape` are passed in. The `oa_ndim == 0` case, now indicates a 0-D iteration and `op_axes` being NULL and the old usage is deprecated. This does not effect the `NpyIter_New` or `NpyIter_MultiNew` functions.

The functions `nanargmin` and `nanargmax` now return `np.iinfo['intp'].min` for the index in all-NaN slices. Previously the functions would raise a `ValueError` for array returns and `NaN` for scalar returns.

NPY_RELAXED_STRIDES_CHECKING

There is a new compile time environment variable `NPY_RELAXED_STRIDES_CHECKING`. If this variable is set to 1, then numpy will consider more arrays to be C- or F-contiguous – for example, it becomes possible to have a column vector which is considered both C- and F-contiguous simultaneously. The new definition is more accurate, allows for faster code that makes fewer unnecessary copies, and simplifies numpy’s code internally. However, it may also break third-party libraries that make too-strong assumptions about the stride values of C- and F-contiguous arrays. (It is also currently known that this breaks Cython code using memoryviews, which will be fixed in Cython.) THIS WILL BECOME THE DEFAULT IN A FUTURE RELEASE, SO PLEASE TEST YOUR CODE NOW AGAINST NUMPY BUILT WITH:

```
NPY_RELAXED_STRIDES_CHECKING=1 python setup.py install
```

You can check whether `NPY_RELAXED_STRIDES_CHECKING` is in effect by running:

```
np.ones((10, 1), order="C").flags.f_contiguous
```

This will be `True` if relaxed strides checking is enabled, and `False` otherwise. The typical problem we’ve seen so far is C code that works with C-contiguous arrays, and assumes that the `itemsz` can be accessed by looking at the last element in the `PyArray_STRIDES(arr)` array. When relaxed strides are in effect, this is not true (and in fact, it never was true in some corner cases). Instead, use `PyArray_ITEMSIZE(arr)`.

For more information check the “Internal memory layout of an ndarray” section in the documentation.

Binary operations with non-arrays as second argument

Binary operations of the form `<array-or-subclass> * <non-array-subclass>` where `<non-array-subclass>` declares an `__array_priority__` higher than that of `<array-or-subclass>` will now unconditionally return *NotImplemented*, giving `<non-array-subclass>` a chance to handle the operation. Previously, *NotImplemented* would only be returned if `<non-array-subclass>` actually implemented the reversed operation, and after a (potentially expensive) array conversion of `<non-array-subclass>` had been attempted. (bug, pull request)

Function *median* used with *overwrite_input* only partially sorts array

If *median* is used with *overwrite_input* option the input array will now only be partially sorted instead of fully sorted.

Fix to *financial.npv*

The *npv* function had a bug. Contrary to what the documentation stated, it summed from indexes 1 to *M* instead of from 0 to *M* - 1. The fix changes the returned value. The *mirr* function called the *npv* function, but worked around the problem, so that was also fixed and the return value of the *mirr* function remains unchanged.

Runtime warnings when comparing NaN numbers

Comparing NaN floating point numbers now raises the *invalid* runtime warning. If a NaN is expected the warning can be ignored using *np.errstate*. E.g.:

```
with np.errstate(invalid='ignore'):
    operation()
```

15.75.5 New Features

Support for linear algebra on stacked arrays

The *gufunc* machinery is now used for *np.linalg*, allowing operations on stacked arrays and vectors. For example:

```
>>> a
array([[[ 1.,  1.],
        [ 0.,  1.]],

       [[ 1.,  1.],
        [ 0.,  1.]])

>>> np.linalg.inv(a)
array([[[ 1., -1.],
        [ 0.,  1.]],

       [[ 1., -1.],
        [ 0.,  1.]])
```

In place fancy indexing for ufuncs

The function *at* has been added to *ufunc* objects to allow in place *ufuncs* with no buffering when fancy indexing is used. For example, the following will increment the first and second items in the array, and will increment the third item twice: `numpy.add.at(arr, [0, 1, 2, 2], 1)`

This is what many have mistakenly thought `arr[[0, 1, 2, 2]] += 1` would do, but that does not work as the incremented value of `arr[2]` is simply copied into the third slot in `arr` twice, not incremented twice.

New functions *partition* and *argpartition*

New functions to partially sort arrays via a selection algorithm.

A `partition` by index k moves the k smallest element to the front of an array. All elements before k are then smaller or equal than the value in position k and all elements following k are then greater or equal than the value in position k . The ordering of the values within these bounds is undefined. A sequence of indices can be provided to sort all of them into their sorted position at once iterative partitioning. This can be used to efficiently obtain order statistics like median or percentiles of samples. `partition` has a linear time complexity of $O(n)$ while a full sort has $O(n \log(n))$.

New functions *nanmean*, *nanvar* and *nanstd*

New nan aware statistical functions are added. In these functions the results are what would be obtained if nan values were omitted from all computations.

New functions *full* and *full_like*

New convenience functions to create arrays filled with a specific value; complementary to the existing `zeros` and `zeros_like` functions.

IO compatibility with large files

Large NPZ files >2GB can be loaded on 64-bit systems.

Building against OpenBLAS

It is now possible to build numpy against OpenBLAS by editing `site.cfg`.

New constant

Euler's constant is now exposed in numpy as `euler_gamma`.

New modes for *qr*

New modes 'complete', 'reduced', and 'raw' have been added to the `qr` factorization and the old 'full' and 'economic' modes are deprecated. The 'reduced' mode replaces the old 'full' mode and is the default as was the 'full' mode, so backward compatibility can be maintained by not specifying the mode.

The 'complete' mode returns a full dimensional factorization, which can be useful for obtaining a basis for the orthogonal complement of the range space. The 'raw' mode returns arrays that contain the Householder reflectors and scaling factors that can be used in the future to apply q without needing to convert to a matrix. The 'economic' mode is simply deprecated, there isn't much use for it and it isn't any more efficient than the 'raw' mode.

New *invert* argument to *in1d*

The function *in1d* now accepts a *invert* argument which, when *True*, causes the returned array to be inverted.

Advanced indexing using *np.newaxis*

It is now possible to use *np.newaxis/None* together with index arrays instead of only in simple indices. This means that `array[np.newaxis, [0, 1]]` will now work as expected and select the first two rows while prepending a new axis to the array.

C-API

New ufuncs can now be registered with builtin input types and a custom output type. Before this change, NumPy wouldn't be able to find the right ufunc loop function when the ufunc was called from Python, because the ufunc loop signature matching logic wasn't looking at the output operand type. Now the correct ufunc loop is found, as long as the user provides an output argument with the correct output type.

runtests.py

A simple test runner script *runtests.py* was added. It also builds Numpy via *setup.py build* and can be used to run tests easily during development.

15.75.6 Improvements

IO performance improvements

Performance in reading large files was improved by chunking (see also IO compatibility).

Performance improvements to *pad*

The *pad* function has a new implementation, greatly improving performance for all inputs except *mode=* (retained for backwards compatibility). Scaling with dimensionality is dramatically improved for rank ≥ 4 .

Performance improvements to *isnan*, *isinf*, *isfinite* and *byteswap*

isnan, *isinf*, *isfinite* and *byteswap* have been improved to take advantage of compiler builtins to avoid expensive calls to *libc*. This improves performance of these operations by about a factor of two on *gnu libc* systems.

Performance improvements via SSE2 vectorization

Several functions have been optimized to make use of SSE2 CPU SIMD instructions.

- **Float32 and float64:**
 - base math (*add*, *subtract*, *divide*, *multiply*)
 - *sqrt*
 - *minimum/maximum*
 - *absolute*

- **Bool:**

- *logical_or*
- *logical_and*
- *logical_not*

This improves performance of these operations up to 4x/2x for float32/float64 and up to 10x for bool depending on the location of the data in the CPU caches. The performance gain is greatest for in-place operations.

In order to use the improved functions the SSE2 instruction set must be enabled at compile time. It is enabled by default on x86_64 systems. On x86_32 with a capable CPU it must be enabled by passing the appropriate flag to the CFLAGS build variable (-msse2 with gcc).

Performance improvements to *median*

median is now implemented in terms of *partition* instead of *sort* which reduces its time complexity from $O(n \log(n))$ to $O(n)$. If used with the *overwrite_input* option the array will now only be partially sorted instead of fully sorted.

Overridable operand flags in ufunc C-API

When creating a ufunc, the default ufunc operand flags can be overridden via the new *op_flags* attribute of the ufunc object. For example, to set the operand flag for the first input to read/write:

```
PyObject *ufunc = PyUFunc_FromFuncAndData(...); ufunc->op_flags[0] = NPY_ITER_READWRITE;
```

This allows a ufunc to perform an operation in place. Also, global *nditer* flags can be overridden via the new *iter_flags* attribute of the ufunc object. For example, to set the reduce flag for a ufunc:

```
ufunc->iter_flags = NPY_ITER_REDUCE_OK;
```

15.75.7 Changes

General

The function `np.take` now allows 0-d arrays as indices.

The separate compilation mode is now enabled by default.

Several changes to `np.insert` and `np.delete`:

- Previously, negative indices and indices that pointed past the end of the array were simply ignored. Now, this will raise a Future or Deprecation Warning. In the future they will be treated like normal indexing treats them – negative indices will wrap around, and out-of-bound indices will generate an error.
- Previously, boolean indices were treated as if they were integers (always referring to either the 0th or 1st item in the array). In the future, they will be treated as masks. In this release, they raise a FutureWarning warning of this coming change.
- In Numpy 1.7. `np.insert` already allowed the syntax `np.insert(arr, 3, [1,2,3])` to insert multiple items at a single position. In Numpy 1.8. this is also possible for `np.insert(arr, [3], [1, 2, 3])`.

Padded regions from `np.pad` are now correctly rounded, not truncated.

C-API Array Additions

Four new functions have been added to the array C-API.

- `PyArray_Partition`
- `PyArray_ArgPartition`
- `PyArray_SelectkindConverter`
- `PyDataMem_NEW_ZEROED`

C-API Ufunc Additions

One new function has been added to the ufunc C-API that allows to register an inner loop for user types using the descr.

- `PyUFunc_RegisterLoopForDescr`

C-API Developer Improvements

The `PyArray_Type` instance creation function `tp_new` now uses `tp_basicsize` to determine how much memory to allocate. In previous releases only `sizeof(PyArrayObject)` bytes of memory were allocated, often requiring C-API subtypes to reimplement `tp_new`.

15.75.8 Deprecations

The ‘full’ and ‘economic’ modes of qr factorization are deprecated.

General

The use of non-integer for indices and most integer arguments has been deprecated. Previously float indices and function arguments such as axes or shapes were truncated to integers without warning. For example `arr.reshape(3., -1)` or `arr[0.]` will trigger a deprecation warning in NumPy 1.8., and in some future version of NumPy they will raise an error.

15.75.9 Authors

This release contains work by the following people who contributed at least one patch to this release. The names are in alphabetical order by first name:

- 87
- Adam Ginsburg +
- Adam Griffiths +
- Alexander Belopolsky +
- Alex Barth +
- Alex Ford +
- Andreas Hilboll +
- Andreas Kloeckner +
- Andreas Schwab +

- Andrew Horton +
- argriffing +
- Arink Verma +
- Bago Amirbekian +
- Bartosz Telenczuk +
- bebert218 +
- Benjamin Root +
- Bill Spotz +
- Bradley M. Froehle
- Carwyn Pelley +
- Charles Harris
- Chris
- Christian Brueffer +
- Christoph Dann +
- Christoph Gohlke
- Dan Hipschman +
- Daniel +
- Dan Miller +
- daveydave400 +
- David Cournapeau
- David Warde-Farley
- Denis Laxalde
- dmuellner +
- Edward Catmur +
- Egor Zindy +
- endolith
- Eric Firing
- Eric Fode
- Eric Moore +
- Eric Price +
- Fazlul Shahriar +
- Félix Hartmann +
- Fernando Perez
- Frank B +
- Frank Breitling +
- Frederic

- Gabriel
- GaelVaroquaux
- Guillaume Gay +
- Han Genuit
- HaroldMills +
- hklemm +
- jamestwebber +
- Jason Madden +
- Jay Bourque
- jeromekelleher +
- Jesús Gómez +
- jmozmoz +
- jnothman +
- Johannes Schönberger +
- John Benediktsson +
- John Salvatier +
- John Stechschulte +
- Jonathan Waltman +
- Joon Ro +
- Jos de Kloe +
- Joseph Martinot-Lagarde +
- Josh Warner (Mac) +
- Jostein Bø Fløystad +
- Juan Luis Cano Rodríguez +
- Julian Taylor +
- Julien Phalip +
- K.-Michael Aye +
- Kumar Appaiah +
- Lars Buitinck
- Leon Weber +
- Luis Pedro Coelho
- Marcin Juskiewicz
- Mark Wiebe
- Marten van Kerkwijk +
- Martin Baeuml +
- Martin Spacek

- Martin Teichmann +
- Matt Davis +
- Matthew Brett
- Maximilian Albert +
- m-d-w +
- Michael Droettboom
- mwtoews +
- Nathaniel J. Smith
- Nicolas Scheffer +
- Nils Werner +
- ochoadavid +
- Ondřej Čertík
- ovillellas +
- Paul Ivanov
- Pauli Virtanen
- peterjc
- Ralf Gommers
- Raul Cota +
- Richard Hattersley +
- Robert Costa +
- Robert Kern
- Rob Ruana +
- Ronan Lamy
- Sandro Tosi
- Sascha Peilicke +
- Sebastian Berg
- Skipper Seabold
- Stefan van der Walt
- Steve +
- Takafumi Arakaki +
- Thomas Robitaille +
- Tomas Tomecek +
- Travis E. Oliphant
- Valentin Haenel
- Vladimir Rutsky +
- Warren Weckesser

- Yaroslav Halchenko
- Yury V. Zaytsev +

A total of 119 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

15.76 NumPy 1.7.2 Release Notes

This is a bugfix only release in the 1.7.x series. It supports Python 2.4 - 2.7 and 3.1 - 3.3 and is the last series that supports Python 2.4 - 2.5.

15.76.1 Issues fixed

- gh-3153: Do not reuse nditer buffers when not filled enough
- gh-3192: f2py crashes with UnboundLocalError exception
- gh-442: Concatenate with axis=None now requires equal number of array elements
- gh-2485: Fix for astype('S') string truncate issue
- gh-3312: bug in count_nonzero
- gh-2684: numpy.ma.average casts complex to float under certain conditions
- gh-2403: masked array with named components does not behave as expected
- gh-2495: np.ma.compress treated inputs in wrong order
- gh-576: add __len__ method to ma.mvoid
- gh-3364: reduce performance regression of mmap slicing
- gh-3421: fix non-swapping strided copies in GetStridedCopySwap
- gh-3373: fix small leak in datetime metadata initialization
- gh-2791: add platform specific python include directories to search paths
- gh-3168: fix undefined function and add integer divisions
- gh-3301: memmap does not work with TemporaryFile in python3
- gh-3057: distutils.misc_util.get_shared_lib_extension returns wrong debug extension
- gh-3472: add module extensions to load_library search list
- gh-3324: Make comparison function (gt, ge, ...) respect __array_priority__
- gh-3497: np.insert behaves incorrectly with argument 'axis=-1'
- gh-3541: make preprocessor tests consistent in halffloat.c
- gh-3458: array_ass_boolean_subscript() writes 'non-existent' data to array
- gh-2892: Regression in ufunc.reduceat with zero-sized index array
- gh-3608: Regression when filling struct from tuple
- gh-3701: add support for Python 3.4 ast.NameConstant
- gh-3712: do not assume that GIL is enabled in xerbla
- gh-3712: fix LAPACK error handling in lapack_litemodule

- gh-3728: f2py fix decref on wrong object
- gh-3743: Hash changed signature in Python 3.3
- gh-3793: scalar int hashing broken on 64 bit python3
- gh-3160: SandboxViolation easyinstalling 1.7.0 on Mac OS X 10.8.3
- gh-3871: npy_math.h has invalid isinf for Solaris with SUNWsp12.2
- gh-2561: Disable check for oldstyle classes in python3
- gh-3900: Ensure NotImplemented is passed on in MaskedArray ufunc's
- gh-2052: del scalar subscript causes segfault
- gh-3832: fix a few uninitialized uses and memleaks
- gh-3971: f2py changed string.lowercase to string.ascii_lowercase for python3
- gh-3480: numpy.random.binomial raised ValueError for n == 0
- gh-3992: hypot(inf, 0) shouldn't raise a warning, hypot(inf, inf) wrong result
- gh-4018: Segmentation fault dealing with very large arrays
- gh-4094: fix NaT handling in _strided_to_strided_string_to_datetime
- gh-4051: fix uninitialized use in _strided_to_strided_string_to_datetime
- gh-4123: lexsort segfault
- gh-4141: Fix a few issues that show up with python 3.4b1

15.77 NumPy 1.7.1 Release Notes

This is a bugfix only release in the 1.7.x series. It supports Python 2.4 - 2.7 and 3.1 - 3.3 and is the last series that supports Python 2.4 - 2.5.

15.77.1 Issues fixed

- gh-2973: Fix *I* is printed during numpy.test()
- gh-2983: BUG: gh-2969: Backport memory leak fix 80b3a34.
- gh-3007: Backport gh-3006
- gh-2984: Backport fix complex polynomial fit
- gh-2982: BUG: Make nansum work with booleans.
- gh-2985: Backport large sort fixes
- gh-3039: Backport object take
- gh-3105: Backport nditer fix op axes initialization
- gh-3108: BUG: npy-pkg-config ini files were missing after Bento build.
- gh-3124: BUG: PyArray_LexSort allocates too much temporary memory.
- gh-3131: BUG: Exported f2py_size symbol prevents linking multiple f2py modules.
- gh-3117: Backport gh-2992

- gh-3135: DOC: Add mention of `PyArray_SetBaseObject` stealing a reference
- gh-3134: DOC: Fix typo in fft docs (the indexing variable is 'm', not 'n').
- gh-3136: Backport #3128

15.78 NumPy 1.7.0 Release Notes

This release includes several new features as well as numerous bug fixes and refactorings. It supports Python 2.4 - 2.7 and 3.1 - 3.3 and is the last release that supports Python 2.4 - 2.5.

15.78.1 Highlights

- `where=` parameter to ufuncs (allows the use of boolean arrays to choose where a computation should be done)
- `vectorize` improvements (added 'excluded' and 'cache' keyword, general cleanup and bug fixes)
- `numpy.random.choice` (random sample generating function)

15.78.2 Compatibility notes

In a future version of numpy, the functions `np.diag`, `np.diagonal`, and the `diagonal` method of `ndarrays` will return a view onto the original array, instead of producing a copy as they do now. This makes a difference if you write to the array returned by any of these functions. To facilitate this transition, numpy 1.7 produces a `FutureWarning` if it detects that you may be attempting to write to such an array. See the documentation for `np.diagonal` for details.

Similar to `np.diagonal` above, in a future version of numpy, indexing a record array by a list of field names will return a view onto the original array, instead of producing a copy as they do now. As with `np.diagonal`, numpy 1.7 produces a `FutureWarning` if it detects that you may be attempting to write to such an array. See the documentation for array indexing for details.

In a future version of numpy, the default casting rule for UFunc `out=` parameters will be changed from 'unsafe' to 'same_kind'. (This also applies to in-place operations like `a += b`, which is equivalent to `np.add(a, b, out=a)`.) Most usages which violate the 'same_kind' rule are likely bugs, so this change may expose previously undetected errors in projects that depend on NumPy. In this version of numpy, such usages will continue to succeed, but will raise a `DeprecationWarning`.

Full-array boolean indexing has been optimized to use a different, optimized code path. This code path should produce the same results, but any feedback about changes to your code would be appreciated.

Attempting to write to a read-only array (one with `arr.flags.writeable` set to `False`) used to raise either a `RuntimeError`, `ValueError`, or `TypeError` inconsistently, depending on which code path was taken. It now consistently raises a `ValueError`.

The `<ufunc>.reduce` functions evaluate some reductions in a different order than in previous versions of NumPy, generally providing higher performance. Because of the nature of floating-point arithmetic, this may subtly change some results, just as linking NumPy to a different BLAS implementations such as MKL can.

If upgrading from 1.5, then generally in 1.6 and 1.7 there have been substantial code added and some code paths altered, particularly in the areas of type resolution and buffered iteration over universal functions. This might have an impact on your code particularly if you relied on accidental behavior in the past.

15.78.3 New features

Reduction UFuncs Generalize axis= Parameter

Any `ufunc.reduce` function call, as well as other reductions like `sum`, `prod`, `any`, `all`, `max` and `min` support the ability to choose a subset of the axes to reduce over. Previously, one could say `axis=None` to mean all the axes or `axis=#` to pick a single axis. Now, one can also say `axis=(#,#)` to pick a list of axes for reduction.

Reduction UFuncs New keepdims= Parameter

There is a new `keepdims=` parameter, which if set to `True`, doesn't throw away the reduction axes but instead sets them to have size one. When this option is set, the reduction result will broadcast correctly to the original operand which was reduced.

Datetime support

Note: The datetime API is *experimental* in 1.7.0, and may undergo changes in future versions of NumPy.

There have been a lot of fixes and enhancements to `datetime64` compared to NumPy 1.6:

- the parser is quite strict about only accepting ISO 8601 dates, with a few convenience extensions
- converts between units correctly
- datetime arithmetic works correctly
- business day functionality (allows the datetime to be used in contexts where only certain days of the week are valid)

The notes in [doc/source/reference/arrays.datetime.rst](#) (also available in the online docs at [arrays.datetime.html](#)) should be consulted for more details.

Custom formatter for printing arrays

See the new `formatter` parameter of the `numpy.set_printoptions` function.

New function `numpy.random.choice`

A generic sampling function has been added which will generate samples from a given array-like. The samples can be with or without replacement, and with uniform or given non-uniform probabilities.

New function `isclose`

Returns a boolean array where two arrays are element-wise equal within a tolerance. Both relative and absolute tolerance can be specified.

Preliminary multi-dimensional support in the polynomial package

Axis keywords have been added to the integration and differentiation functions and a tensor keyword was added to the evaluation functions. These additions allow multi-dimensional coefficient arrays to be used in those functions. New functions for evaluating 2-D and 3-D coefficient arrays on grids or sets of points were added together with 2-D and 3-D pseudo-Vandermonde matrices that can be used for fitting.

Ability to pad rank-n arrays

A pad module containing functions for padding n-dimensional arrays has been added. The various private padding functions are exposed as options to a public ‘pad’ function. Example:

```
pad(a, 5, mode='mean')
```

Current modes are constant, edge, linear_ramp, maximum, mean, median, minimum, reflect, symmetric, wrap, and <function>.

New argument to searchsorted

The function searchsorted now accepts a ‘sorter’ argument that is a permutation array that sorts the array to search.

Build system

Added experimental support for the AArch64 architecture.

C API

New function PyArray_FailUnlessWriteable provides a consistent interface for checking array writeability – any C code which works with arrays whose WRITEABLE flag is not known to be True a priori, should make sure to call this function before writing.

NumPy C Style Guide added (doc/C_STYLE_GUIDE.rst.txt).

15.78.4 Changes

General

The function np.concatenate tries to match the layout of its input arrays. Previously, the layout did not follow any particular reason, and depended in an undesirable way on the particular axis chosen for concatenation. A bug was also fixed which silently allowed out of bounds axis arguments.

The ufuncs logical_or, logical_and, and logical_not now follow Python’s behavior with object arrays, instead of trying to call methods on the objects. For example the expression (3 and ‘test’) produces the string ‘test’, and now np.logical_and(np.array(3, ‘O’), np.array(‘test’, ‘O’)) produces ‘test’ as well.

The .base attribute on ndarrays, which is used on views to ensure that the underlying array owning the memory is not deallocated prematurely, now collapses out references when you have a view-of-a-view. For example:

```
a = np.arange(10)
b = a[1:]
c = b[1:]
```

In numpy 1.6, `c.base` is `b`, and `c.base.base` is `a`. In numpy 1.7, `c.base` is `a`.

To increase backwards compatibility for software which relies on the old behaviour of `.base`, we only ‘skip over’ objects which have exactly the same type as the newly created view. This makes a difference if you use `ndarray` subclasses. For example, if we have a mix of `ndarray` and `matrix` objects which are all views on the same original `ndarray`:

```
a = np.arange(10)
b = np.asmatrix(a)
c = b[0, 1:]
d = c[0, 1:]
```

then `d.base` will be `b`. This is because `d` is a `matrix` object, and so the collapsing process only continues so long as it encounters other `matrix` objects. It considers `c`, `b`, and `a` in that order, and `b` is the last entry in that list which is a `matrix` object.

Casting Rules

Casting rules have undergone some changes in corner cases, due to the NA-related work. In particular for combinations of scalar+scalar:

- the *longlong* type (*q*) now stays *longlong* for operations with any other number (*? b h i l q p B H I*), previously it was cast as *int_* (*l*). The *ulonglong* type (*Q*) now stays as *ulonglong* instead of *uint* (*L*).
- the *timedelta64* type (*m*) can now be mixed with any integer type (*b h i l q p B H I L Q P*), previously it raised *TypeError*.

For array + scalar, the above rules just broadcast except the case when the array and scalars are unsigned/signed integers, then the result gets converted to the array type (of possibly larger size) as illustrated by the following examples:

```
>>> (np.zeros((2,), dtype=np.uint8) + np.int16(257)).dtype
dtype('uint16')
>>> (np.zeros((2,), dtype=np.int8) + np.uint16(257)).dtype
dtype('int16')
>>> (np.zeros((2,), dtype=np.int16) + np.uint32(2**17)).dtype
dtype('int32')
```

Whether the size gets increased depends on the size of the scalar, for example:

```
>>> (np.zeros((2,), dtype=np.uint8) + np.int16(255)).dtype
dtype('uint8')
>>> (np.zeros((2,), dtype=np.uint8) + np.int16(256)).dtype
dtype('uint16')
```

Also a `complex128` scalar + `float32` array is cast to `complex64`.

In NumPy 1.7 the *datetime64* type (*M*) must be constructed by explicitly specifying the type as the second argument (e.g. `np.datetime64(2000, 'Y')`).

15.78.5 Deprecations

General

Specifying a custom string formatter with a `_format` array attribute is deprecated. The new `formatter` keyword in `numpy.set_printoptions` or `numpy.array2string` can be used instead.

The deprecated imports in the polynomial package have been removed.

`concatenate` now raises `DeprecationWarning` for 1D arrays if `axis != 0`. Versions of `numpy < 1.7.0` ignored axis argument value for 1D arrays. We allow this for now, but in due course we will raise an error.

C-API

Direct access to the fields of `PyArrayObject*` has been deprecated. Direct access has been recommended against for many releases. Expect similar deprecations for `PyArray_Descr*` and other core objects in the future as preparation for NumPy 2.0.

The macros in `old_defines.h` are deprecated and will be removed in the next major release (≥ 2.0). The sed script `tools/replace_old_macros.sed` can be used to replace these macros with the newer versions.

You can test your code against the deprecated C API by adding a line composed of `#define NPY_NO_DEPRECATED_API` and the target version number, such as `NPY_1_7_API_VERSION`, before including any NumPy headers.

The `NPY_CHAR` member of the `NPY_TYPES` enum is deprecated and will be removed in NumPy 1.8. See the discussion at [gh-2801](#) for more details.

15.79 NumPy 1.6.2 Release Notes

This is a bugfix release in the 1.6.x series. Due to the delay of the NumPy 1.7.0 release, this release contains far more fixes than a regular NumPy bugfix release. It also includes a number of documentation and build improvements.

15.79.1 Issues fixed

`numpy.core`

- #2063: make `unique()` return consistent index
- #1138: allow creating arrays from empty buffers or empty slices
- #1446: correct note about correspondence `vstack` and `concatenate`
- #1149: make `argmin()` work for `datetime`
- #1672: fix `allclose()` to work for scalar `inf`
- #1747: make `np.median()` work for 0-D arrays
- #1776: make complex division by zero to yield `inf` properly
- #1675: add scalar support for the `format()` function
- #1905: explicitly check for NaNs in `allclose()`
- #1952: allow floating `ddof` in `std()` and `var()`

- #1948: fix regression for indexing chararrays with empty list
- #2017: fix type hashing
- #2046: deleting array attributes causes segfault
- #2033: $a^{**2.0}$ has incorrect type
- #2045: make attribute/iterator_element deletions not segfault
- #2021: fix segfault in searchsorted()
- #2073: fix float16 `__array_interface__` bug

`numpy.lib`

- #2048: break reference cycle in NpzFile
- #1573: `savetxt()` now handles complex arrays
- #1387: allow `bincount()` to accept empty arrays
- #1899: fixed `histogramdd()` bug with empty inputs
- #1793: fix failing `npio` test under py3k
- #1936: fix extra nesting for subarray dtypes
- #1848: make `tril/triu` return the same dtype as the original array
- #1918: use `Py_TYPE` to access `ob_type`, so it works also on Py3

`numpy.distutils`

- #1261: change compile flag on AIX from `-O5` to `-O3`
- #1377: update HP compiler flags
- #1383: provide better support for C++ code on HP-UX
- #1857: fix build for py3k + pip
- BLD: raise a clearer warning in case of building without cleaning up first
- BLD: follow `build_ext` coding convention in `build_clib`
- BLD: fix up detection of Intel CPU on OS X in `system_info.py`
- BLD: add support for the new X11 directory structure on Ubuntu & co.
- BLD: add `ufsparse` to the libraries search path.
- BLD: add 'pgfortran' as a valid compiler in the Portland Group
- BLD: update version match regexp for IBM AIX Fortran compilers.

numpy.random

- BUG: Use npy_intp instead of long in mtrand

15.79.2 Changes

numpy.f2py

- ENH: Introduce new options extra_f77_compiler_args and extra_f90_compiler_args
- BLD: Improve reporting of fcompiler value
- BUG: Fix f2py test_kind.py test

numpy.poly

- ENH: Add some tests for polynomial printing
- ENH: Add companion matrix functions
- DOC: Rearrange the polynomial documents
- BUG: Fix up links to classes
- DOC: Add version added to some of the polynomial package modules
- DOC: Document xxxfit functions in the polynomial package modules
- BUG: The polynomial convenience classes let different types interact
- DOC: Document the use of the polynomial convenience classes
- DOC: Improve numpy reference documentation of polynomial classes
- ENH: Improve the computation of polynomials from roots
- STY: Code cleanup in polynomial [*]fromroots functions
- DOC: Remove references to cast and NA, which were added in 1.7

15.80 NumPy 1.6.1 Release Notes

This is a bugfix only release in the 1.6.x series.

15.80.1 Issues Fixed

- #1834: einsum fails for specific shapes
- #1837: einsum throws nan or freezes python for specific array shapes
- #1838: object <-> structured type arrays regression
- #1851: regression for SWIG based code in 1.6.0
- #1863: Buggy results when operating on array copied with astype()
- #1870: Fix corner case of object array assignment
- #1843: Py3k: fix error with recarray

- #1885: `nditer`: Error in detecting double reduction loop
- #1874: `f2py`: fix `-include_paths` bug
- #1749: Fix `ctypes.load_library()`
- #1895/1896: `iter`: writeonly operands weren't always being buffered correctly

15.81 NumPy 1.6.0 Release Notes

This release includes several new features as well as numerous bug fixes and improved documentation. It is backward compatible with the 1.5.0 release, and supports Python 2.4 - 2.7 and 3.1 - 3.2.

15.81.1 Highlights

- Re-introduction of `datetime` dtype support to deal with dates in arrays.
- A new 16-bit floating point type.
- A new iterator, which improves performance of many functions.

15.81.2 New features

New 16-bit floating point type

This release adds support for the IEEE 754-2008 `binary16` format, available as the data type `numpy.half`. Within Python, the type behaves similarly to *float* or *double*, and C extensions can add support for it with the exposed half-float API.

New iterator

A new iterator has been added, replacing the functionality of the existing iterator and multi-iterator with a single object and API. This iterator works well with general memory layouts different from C or Fortran contiguous, and handles both standard NumPy and customized broadcasting. The buffering, automatic data type conversion, and optional output parameters, offered by `ufuncs` but difficult to replicate elsewhere, are now exposed by this iterator.

Legendre, Laguerre, Hermite, HermiteE polynomials in `numpy.polynomial`

Extend the number of polynomials available in the polynomial package. In addition, a new `window` attribute has been added to the classes in order to specify the range the `domain` maps to. This is mostly useful for the Laguerre, Hermite, and HermiteE polynomials whose natural domains are infinite and provides a more intuitive way to get the correct mapping of values without playing unnatural tricks with the domain.

Fortran assumed shape array and size function support in `numpy.f2py`

F2py now supports wrapping Fortran 90 routines that use assumed shape arrays. Before such routines could be called from Python but the corresponding Fortran routines received assumed shape arrays as zero length arrays which caused unpredicted results. Thanks to Lorenz Hüpdepohl for pointing out the correct way to interface routines with assumed shape arrays.

In addition, f2py supports now automatic wrapping of Fortran routines that use two argument `size` function in dimension specifications.

Other new functions

`numpy.ravel_multi_index`: Converts a multi-index tuple into an array of flat indices, applying boundary modes to the indices.

`numpy.einsum`: Evaluate the Einstein summation convention. Using the Einstein summation convention, many common multi-dimensional array operations can be represented in a simple fashion. This function provides a way compute such summations.

`numpy.count_nonzero`: Counts the number of non-zero elements in an array.

`numpy.result_type` and `numpy.min_scalar_type`: These functions expose the underlying type promotion used by the ufuncs and other operations to determine the types of outputs. These improve upon the `numpy.common_type` and `numpy.mintypcode` which provide similar functionality but do not match the ufunc implementation.

15.81.3 Changes

default error handling

The default error handling has been change from `print` to `warn` for all except for `underflow`, which remains as `ignore`.

`numpy.distutils`

Several new compilers are supported for building Numpy: the Portland Group Fortran compiler on OS X, the PathScale compiler suite and the 64-bit Intel C compiler on Linux.

`numpy.testing`

The testing framework gained `numpy.testing.assert_allclose`, which provides a more convenient way to compare floating point arrays than `assert_almost_equal`, `assert_approx_equal` and `assert_array_almost_equal`.

C API

In addition to the APIs for the new iterator and half data type, a number of other additions have been made to the C API. The type promotion mechanism used by ufuncs is exposed via `PyArray_PromoteTypes`, `PyArray_ResultType`, and `PyArray_MinScalarType`. A new enumeration `NPY_CASTING` has been added which controls what types of casts are permitted. This is used by the new functions `PyArray_CanCastArrayTo` and `PyArray_CanCastTypeTo`. A more flexible way to handle conversion of arbitrary python objects into arrays is exposed by `PyArray_GetArrayParamsFromObject`.

15.81.4 Deprecated features

The “normed” keyword in `numpy.histogram` is deprecated. Its functionality will be replaced by the new “density” keyword.

15.81.5 Removed features

`numpy.fft`

The functions `refft`, `refft2`, `refft3`, `irefft`, `irefft2`, `irefft3`, which were aliases for the same functions without the ‘e’ in the name, were removed.

`numpy.memmap`

The `sync()` and `close()` methods of `memmap` were removed. Use `flush()` and “del `memmap`” instead.

`numpy.lib`

The deprecated functions `numpy.unique1d`, `numpy.setmember1d`, `numpy.intersect1d_nu` and `numpy.lib.ufunclike.log2` were removed.

`numpy.ma`

Several deprecated items were removed from the `numpy.ma` module:

```
* ``numpy.ma.MaskedArray`` "raw_data" method
* ``numpy.ma.MaskedArray`` constructor "flag" keyword
* ``numpy.ma.make_mask`` "flag" keyword
* ``numpy.ma.allclose`` "fill_value" keyword
```

`numpy.distutils`

The `numpy.get_numpy_include` function was removed, use `numpy.get_include` instead.

15.82 NumPy 1.5.0 Release Notes

15.82.1 Highlights

Python 3 compatibility

This is the first NumPy release which is compatible with Python 3. Support for Python 3 and Python 2 is done from a single code base. Extensive notes on changes can be found at <https://web.archive.org/web/20100814160313/http://projects.scipy.org/numpy/browser/trunk/doc/Py3K.txt>.

Note that the Numpy testing framework relies on nose, which does not have a Python 3 compatible release yet. A working Python 3 branch of nose can be found at <https://web.archive.org/web/20100817112505/http://bitbucket.org/jpellerin/nose3/> however.

Porting of SciPy to Python 3 is expected to be completed soon.

PEP 3118 compatibility

The new buffer protocol described by PEP 3118 is fully supported in this version of Numpy. On Python versions ≥ 2.6 Numpy arrays expose the buffer interface, and `array()`, `asarray()` and other functions accept new-style buffers as input.

15.82.2 New features

Warning on casting complex to real

Numpy now emits a `numpy.ComplexWarning` when a complex number is cast into a real number. For example:

```
>>> x = np.array([1,2,3])
>>> x[:2] = np.array([1+2j, 1-2j])
ComplexWarning: Casting complex values to real discards the imaginary part
```

The cast indeed discards the imaginary part, and this may not be the intended behavior in all cases, hence the warning. This warning can be turned off in the standard way:

```
>>> import warnings
>>> warnings.simplefilter("ignore", np.ComplexWarning)
```

Dot method for ndarrays

Ndarrays now have the dot product also as a method, which allows writing chains of matrix products as

```
>>> a.dot(b).dot(c)
```

instead of the longer alternative

```
>>> np.dot(a, np.dot(b, c))
```

linalg.slogdet function

The slogdet function returns the sign and logarithm of the determinant of a matrix. Because the determinant may involve the product of many small/large values, the result is often more accurate than that obtained by simple multiplication.

new header

The new header file ndarraytypes.h contains the symbols from ndarrayobject.h that do not depend on the PY_ARRAY_UNIQUE_SYMBOL and NO_IMPORT/_ARRAY macros. Broadly, these symbols are types, typedefs, and enumerations; the array function calls are left in ndarrayobject.h. This allows users to include array-related types and enumerations without needing to concern themselves with the macro expansions and their side- effects.

15.82.3 Changes

polynomial.polynomial

- The polyint and polyder functions now check that the specified number integrations or derivations is a non-negative integer. The number 0 is a valid value for both functions.
- A degree method has been added to the Polynomial class.
- A trimdeg method has been added to the Polynomial class. It operates like truncate except that the argument is the desired degree of the result, not the number of coefficients.
- Polynomial.fit now uses None as the default domain for the fit. The default Polynomial domain can be specified by using [] as the domain value.
- Weights can be used in both polyfit and Polynomial.fit
- A linspace method has been added to the Polynomial class to ease plotting.
- The polymulx function was added.

polynomial.chebyshev

- The chebint and chebder functions now check that the specified number integrations or derivations is a non-negative integer. The number 0 is a valid value for both functions.
- A degree method has been added to the Chebyshev class.
- A trimdeg method has been added to the Chebyshev class. It operates like truncate except that the argument is the desired degree of the result, not the number of coefficients.
- Chebyshev.fit now uses None as the default domain for the fit. The default Chebyshev domain can be specified by using [] as the domain value.
- Weights can be used in both chebfit and Chebyshev.fit
- A linspace method has been added to the Chebyshev class to ease plotting.
- The chebmulx function was added.
- Added functions for the Chebyshev points of the first and second kind.

histogram

After a two years transition period, the old behavior of the histogram function has been phased out, and the “new” keyword has been removed.

correlate

The old behavior of correlate was deprecated in 1.4.0, the new behavior (the usual definition for cross-correlation) is now the default.

15.83 NumPy 1.4.0 Release Notes

This minor includes numerous bug fixes, as well as a few new features. It is backward compatible with 1.3.0 release.

15.83.1 Highlights

- New datetime dtype support to deal with dates in arrays
- Faster import time
- Extended array wrapping mechanism for ufuncs
- New Neighborhood iterator (C-level only)
- C99-like complex functions in npymath

15.83.2 New features

Extended array wrapping mechanism for ufuncs

An `__array_prepare__` method has been added to `ndarray` to provide subclasses greater flexibility to interact with ufuncs and ufunc-like functions. `ndarray` already provided `__array_wrap__`, which allowed subclasses to set the array type for the result and populate metadata on the way out of the ufunc (as seen in the implementation of `MaskedArray`). For some applications it is necessary to provide checks and populate metadata *on the way in*. `__array_prepare__` is therefore called just after the ufunc has initialized the output array but before computing the results and populating it. This way, checks can be made and errors raised before operations which may modify data in place.

Automatic detection of forward incompatibilities

Previously, if an extension was built against a version `N` of NumPy, and used on a system with NumPy `M < N`, the `import_array` was successful, which could cause crashes because the version `M` does not have a function in `N`. Starting from NumPy 1.4.0, this will cause a failure in `import_array`, so the error will be caught early on.

New iterators

A new neighborhood iterator has been added to the C API. It can be used to iterate over the items in a neighborhood of an array, and can handle boundaries conditions automatically. Zero and one padding are available, as well as arbitrary constant value, mirror and circular padding.

New polynomial support

New modules `chebyshev` and `polynomial` have been added. The new polynomial module is not compatible with the current polynomial support in `numpy`, but is much like the new `chebyshev` module. The most noticeable difference to most will be that coefficients are specified from low to high power, that the low level functions do *not* work with the `Chebyshev` and `Polynomial` classes as arguments, and that the `Chebyshev` and `Polynomial` classes include a domain. Mapping between domains is a linear substitution and the two classes can be converted one to the other, allowing, for instance, a `Chebyshev` series in one domain to be expanded as a polynomial in another domain. The new classes should generally be used instead of the low level functions, the latter are provided for those who wish to build their own classes.

The new modules are not automatically imported into the `numpy` namespace, they must be explicitly brought in with an “`import numpy.polynomial`” statement.

New C API

The following C functions have been added to the C API:

1. `PyArray_GetNDArrayCFeatureVersion`: return the *API* version of the loaded `numpy`.
2. `PyArray_Correlate2` - like `PyArray_Correlate`, but implements the usual definition of correlation. Inputs are not swapped, and conjugate is taken for complex arrays.
3. `PyArray_NeighborhoodIterNew` - a new iterator to iterate over a neighborhood of a point, with automatic boundaries handling. It is documented in the iterators section of the C-API reference, and you can find some examples in the `multiarray_test.c.src` file in `numpy.core`.

New ufuncs

The following ufuncs have been added to the C API:

1. `copysign` - return the value of the first argument with the sign copied from the second argument.
2. `nextafter` - return the next representable floating point value of the first argument toward the second argument.

New defines

The alpha processor is now defined and available in `numpy/np_cpu.h`. The failed detection of the PARISC processor has been fixed. The defines are:

1. `NPY_CPU_HPPA`: PARISC
2. `NPY_CPU_ALPHA`: Alpha

Testing

1. `deprecated` decorator: this decorator may be used to avoid cluttering testing output while testing `DeprecationWarning` is effectively raised by the decorated test.
2. `assert_array_almost_equal_nulp`: new method to compare two arrays of floating point values. With this function, two values are considered close if there are not many representable floating point values in between, thus being more robust than `assert_array_almost_equal` when the values fluctuate a lot.
3. `assert_array_max_ulp`: raise an assertion if there are more than `N` representable numbers between two floating point values.
4. `assert_warns`: raise an `AssertionError` if a callable does not generate a warning of the appropriate class, without altering the warning state.

Reusing npymath

In 1.3.0, we started putting portable C math routines in `npymath` library, so that people can use those to write portable extensions. Unfortunately, it was not possible to easily link against this library: in 1.4.0, support has been added to `numpy.distutils` so that 3rd party can reuse this library. See `coremath` documentation for more information.

Improved set operations

In previous versions of NumPy some set functions (`intersect1d`, `setxor1d`, `setdiff1d` and `setmember1d`) could return incorrect results if the input arrays contained duplicate items. These now work correctly for input arrays with duplicates. `setmember1d` has been renamed to `in1d`, as with the change to accept arrays with duplicates it is no longer a set operation, and is conceptually similar to an elementwise version of the Python operator `'in'`. All of these functions now accept the boolean keyword `assume_unique`. This is `False` by default, but can be set `True` if the input arrays are known not to contain duplicates, which can increase the functions' execution speed.

15.83.3 Improvements

1. `numpy` import is noticeably faster (from 20 to 30 % depending on the platform and computer)
2. The sort functions now sort nans to the end.
 - Real sort order is `[R, nan]`
 - Complex sort order is `[R + Rj, R + nanj, nan + Rj, nan + nanj]`

Complex numbers with the same nan placements are sorted according to the non-nan part if it exists.
3. The type comparison functions have been made consistent with the new sort order of nans. `Searchsorted` now works with sorted arrays containing nan values.
4. Complex division has been made more resistant to overflow.
5. Complex floor division has been made more resistant to overflow.

15.83.4 Deprecations

The following functions are deprecated:

1. `correlate`: it takes a new keyword argument `old_behavior`. When `True` (the default), it returns the same result as before. When `False`, compute the conventional correlation, and take the conjugate for complex arrays. The old behavior will be removed in NumPy 1.5, and raises a `DeprecationWarning` in 1.4.
2. `unique1d`: use `unique` instead. `unique1d` raises a deprecation warning in 1.4, and will be removed in 1.5.
3. `intersect1d_nu`: use `intersect1d` instead. `intersect1d_nu` raises a deprecation warning in 1.4, and will be removed in 1.5.
4. `setmember1d`: use `in1d` instead. `setmember1d` raises a deprecation warning in 1.4, and will be removed in 1.5.

The following raise errors:

1. When operating on 0-d arrays, `numpy.max` and other functions accept only `axis=0`, `axis=-1` and `axis=None`. Using an out-of-bounds axes is an indication of a bug, so Numpy raises an error for these cases now.
2. Specifying `axis > MAX_DIMS` is no longer allowed; Numpy raises now an error instead of behaving similarly as for `axis=None`.

15.83.5 Internal changes

Use C99 complex functions when available

The numpy complex types are now guaranteed to be ABI compatible with C99 complex type, if available on the platform. Moreover, the complex ufunc now use the platform C99 functions instead of our own.

split multiarray and umath source code

The source code of `multiarray` and `umath` has been split into separate logic compilation units. This should make the source code more amenable for newcomers.

Separate compilation

By default, every file of `multiarray` (and `umath`) is merged into one for compilation as was the case before, but if `NPY_SEPARATE_COMPILATION` env variable is set to a non-negative value, experimental individual compilation of each file is enabled. This makes the compile/debug cycle much faster when working on core numpy.

Separate core math library

New functions which have been added:

- `np._copysign`
- `np._nextafter`
- `np._cpack`
- `np._creal`
- `np._cimag`
- `np._cabs`

- `numpy.cexp`
- `numpy.clog`
- `numpy.cpow`
- `numpy.csqr`
- `numpy.ccos`
- `numpy.csin`

15.84 NumPy 1.3.0 Release Notes

This minor includes numerous bug fixes, official python 2.6 support, and several new features such as generalized ufuncs.

15.84.1 Highlights

Python 2.6 support

Python 2.6 is now supported on all previously supported platforms, including windows.

<https://www.python.org/dev/peps/pep-0361/>

Generalized ufuncs

There is a general need for looping over not only functions on scalars but also over functions on vectors (or arrays), as explained on <http://scipy.org/scipy/numpy/wiki/GeneralLoopingFunctions>. We propose to realize this concept by generalizing the universal functions (ufuncs), and provide a C implementation that adds ~500 lines to the numpy code base. In current (specialized) ufuncs, the elementary function is limited to element-by-element operations, whereas the generalized version supports “sub-array” by “sub-array” operations. The Perl vector library PDL provides a similar functionality and its terms are re-used in the following.

Each generalized ufunc has information associated with it that states what the “core” dimensionality of the inputs is, as well as the corresponding dimensionality of the outputs (the element-wise ufuncs have zero core dimensions). The list of the core dimensions for all arguments is called the “signature” of a ufunc. For example, the ufunc `numpy.add` has signature “(,),->()” defining two scalar inputs and one scalar output.

Another example is (see the GeneralLoopingFunctions page) the function `inner1d(a,b)` with a signature of “(i),(i)->()”. This applies the inner product along the last axis of each input, but keeps the remaining indices intact. For example, where `a` is of shape (3,5,N) and `b` is of shape (5,N), this will return an output of shape (3,5). The underlying elementary function is called 3*5 times. In the signature, we specify one core dimension “(i)” for each input and zero core dimensions “()” for the output, since it takes two 1-d arrays and returns a scalar. By using the same name “i”, we specify that the two corresponding dimensions should be of the same size (or one of them is of size 1 and will be broadcasted).

The dimensions beyond the core dimensions are called “loop” dimensions. In the above example, this corresponds to (3,5).

The usual numpy “broadcasting” rules apply, where the signature determines how the dimensions of each input/output object are split into core and loop dimensions:

While an input array has a smaller dimensionality than the corresponding number of core dimensions, 1’s are pre-pended to its shape. The core dimensions are removed from all inputs and the remaining dimensions are broadcasted; defining the loop dimensions. The output is given by the loop dimensions plus the output core dimensions.

Experimental Windows 64 bits support

Numpy can now be built on windows 64 bits (amd64 only, not IA64), with both MS compilers and mingw-w64 compilers:

This is *highly experimental*: DO NOT USE FOR PRODUCTION USE. See INSTALL.txt, Windows 64 bits section for more information on limitations and how to build it by yourself.

15.84.2 New features

Formatting issues

Float formatting is now handled by numpy instead of the C runtime: this enables locale independent formatting, more robust fromstring and related methods. Special values (inf and nan) are also more consistent across platforms (nan vs IND/NaN, etc...), and more consistent with recent python formatting work (in 2.6 and later).

Nan handling in max/min

The maximum/minimum ufuncs now reliably propagate nans. If one of the arguments is a nan, then nan is returned. This affects np.min/np.max, amin/amax and the array methods max/min. New ufuncs fmax and fmin have been added to deal with non-propagating nans.

Nan handling in sign

The ufunc sign now returns nan for the sign of anan.

New ufuncs

1. fmax - same as maximum for integer types and non-nan floats. Returns the non-nan argument if one argument is nan and returns nan if both arguments are nan.
2. fmin - same as minimum for integer types and non-nan floats. Returns the non-nan argument if one argument is nan and returns nan if both arguments are nan.
3. deg2rad - converts degrees to radians, same as the radians ufunc.
4. rad2deg - converts radians to degrees, same as the degrees ufunc.
5. log2 - base 2 logarithm.
6. exp2 - base 2 exponential.
7. trunc - truncate floats to nearest integer towards zero.
8. logaddexp - add numbers stored as logarithms and return the logarithm of the result.
9. logaddexp2 - add numbers stored as base 2 logarithms and return the base 2 logarithm of the result.

Masked arrays

Several new features and bug fixes, including:

- structured arrays should now be fully supported by MaskedArray (r6463, r6324, r6305, r6300, r6294...)
- Minor bug fixes (r6356, r6352, r6335, r6299, r6298)
- Improved support for `__iter__` (r6326)
- made baseclass, sharedmask and hardmask accessible to the user (but read-only)
- doc update

gfortran support on windows

Gfortran can now be used as a fortran compiler for numpy on windows, even when the C compiler is Visual Studio (VS 2005 and above; VS 2003 will NOT work). Gfortran + Visual studio does not work on windows 64 bits (but gcc + gfortran does). It is unclear whether it will be possible to use gfortran and visual studio at all on x64.

Arch option for windows binary

Automatic arch detection can now be bypassed from the command line for the superpack installed:

```
numpy-1.3.0-superpack-win32.exe /arch=nosse
```

will install a numpy which works on any x86, even if the running computer supports SSE set.

15.84.3 Deprecated features

Histogram

The semantics of histogram has been modified to fix long-standing issues with outliers handling. The main changes concern

1. the definition of the bin edges, now including the rightmost edge, and
2. the handling of upper outliers, now ignored rather than tallied in the rightmost bin.

The previous behavior is still accessible using `new=False`, but this is deprecated, and will be removed entirely in 1.4.0.

15.84.4 Documentation changes

A lot of documentation has been added. Both user guide and references can be built from sphinx.

15.84.5 New C API

Multiarray API

The following functions have been added to the multiarray C API:

- `PyArray_GetEndianness`: to get runtime endianness

Ufunc API

The following functions have been added to the ufunc API:

- `PyUFunc_FromFuncAndDataAndSignature`: to declare a more general ufunc (generalized ufunc).

New defines

New public C defines are available for ARCH specific code through `numpy/np_cpu.h`:

- `NPY_CPU_X86`: x86 arch (32 bits)
- `NPY_CPU_AMD64`: amd64 arch (x86_64, NOT Itanium)
- `NPY_CPU_PPC`: 32 bits ppc
- `NPY_CPU_PPC64`: 64 bits ppc
- `NPY_CPU_SPARC`: 32 bits sparc
- `NPY_CPU_SPARC64`: 64 bits sparc
- `NPY_CPU_S390`: S390
- `NPY_CPU_IA64`: ia64
- `NPY_CPU_PARISC`: PARISC

New macros for CPU endianness has been added as well (see internal changes below for details):

- `NPY_BYTE_ORDER`: integer
- `NPY_LITTLE_ENDIAN`/`NPY_BIG_ENDIAN` defines

Those provide portable alternatives to glibc `endian.h` macros for platforms without it.

Portable NAN, INFINITY, etc...

`npymath.h` now makes available several portable macro to get NAN, INFINITY:

- `NPY_NAN`: equivalent to NAN, which is a GNU extension
- `NPY_INFINITY`: equivalent to C99 INFINITY
- `NPY_PZERO`, `NPY_NZERO`: positive and negative zero respectively

Corresponding single and extended precision macros are available as well. All references to NAN, or home-grown computation of NAN on the fly have been removed for consistency.

15.84.6 Internal changes

This should make the porting to new platforms easier, and more robust. In particular, the configuration stage does not need to execute any code on the target platform, which is a first step toward cross-compilation.

https://www.numpy.org/neps/nep-0003-math_config_clean.html

umath refactor

A lot of code cleanup for umath/ufunc code (charris).

Improvements to build warnings

Numpy can now build with -W -Wall without warnings

<https://www.numpy.org/neps/nep-0002-warnfix.html>

Separate core math library

The core math functions (sin, cos, etc... for basic C types) have been put into a separate library; it acts as a compatibility layer, to support most C99 maths functions (real only for now). The library includes platform-specific fixes for various maths functions, such as using those versions should be more robust than using your platform functions directly. The API for existing functions is exactly the same as the C99 math functions API; the only difference is the npy prefix (npy_cos vs cos).

The core library will be made available to any extension in 1.4.0.

CPU arch detection

npy_cpu.h defines numpy specific CPU defines, such as NPY_CPU_X86, etc... Those are portable across OS and toolchains, and set up when the header is parsed, so that they can be safely used even in the case of cross-compilation (the values is not set when numpy is built), or for multi-arch binaries (e.g. fat binaries on Max OS X).

npy_endian.h defines numpy specific endianness defines, modeled on the glibc endian.h. NPY_BYTE_ORDER is equivalent to BYTE_ORDER, and one of NPY_LITTLE_ENDIAN or NPY_BIG_ENDIAN is defined. As for CPU archs, those are set when the header is parsed by the compiler, and as such can be used for cross-compilation and multi-arch binaries.

NUMPY LICENSE

Copyright (c) 2005-2022, NumPy Developers.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of the NumPy Developers nor the names of any contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

PYTHON MODULE INDEX

n

`numpy.f2py`, [249](#)

`numpy.lib.recfunctions`, [111](#)

Non-alphabetical

. . ., [301](#)
 (n,), [301](#)
 :, [301](#)
 <, [302](#)
 >, [302](#)
 -1, [301](#)
 __array_finalize__ (ndarray attribute), [219](#)
 __array_priority__ (ndarray attribute), [220](#)
 __array_wrap__ (ndarray attribute), [220](#)

A

accumulate
 ufunc methods, [318](#)
 adding new
 dtype, [216](#), [218](#)
 ufunc, [198](#), [199](#), [202](#), [204](#), [213](#)
 advanced indexing, [302](#)
 along an axis, [302](#)
 append_fields() (in module *numpy.lib.recfunctions*),
 [111](#)
 apply_along_fields() (in module
 numpy.lib.recfunctions), [112](#)
 array, [303](#)
 array iterator, [214](#), [216](#), [312](#)
 array scalar, [303](#)
 array scalars, [313](#)
 array_like, [303](#)
 assign_fields_by_name() (in module
 numpy.lib.recfunctions), [112](#)
 axis, [303](#)

B

.base, [304](#)
 big-endian, [304](#)
 BLAS, [304](#)
 Boost.Python, [198](#)
 broadcast, [304](#)
 broadcastable, [94](#)
 broadcasting, [142](#), [216](#), [313](#)
 buffers, [144](#)
 built-in function

ndpointer(), [193](#)

C

C order, [305](#)
 castfunc (*C function*), [217](#)
 casting rules
 ufunc, [142](#)
 column-major, [305](#)
 compile() (in module *numpy.f2py*), [249](#)
 contiguous, [305](#)
 copy, [305](#)
 ctypes, [191](#), [196](#)
 cython, [188](#), [190](#)

D

dimension, [305](#)
 drop_fields() (in module *numpy.lib.recfunctions*),
 [113](#)
 dtype, [305](#), [312](#)
 adding new, [216](#), [218](#)

E

ellipsis, [70](#)
 error handling, [144](#)
 extension module, [179](#), [185](#)

F

f2py, [188](#)
 fancy indexing, [305](#)
 field, [306](#)
 find_duplicates() (in module
 numpy.lib.recfunctions), [114](#)
 flatten_descr() (in module *numpy.lib.recfunctions*),
 [114](#)
 flattened, [306](#)
 Fortran order, [306](#)

G

get_fieldstructure() (in module
 numpy.lib.recfunctions), [115](#)
 get_include() (in module *numpy.f2py*), [250](#)
 get_names() (in module *numpy.lib.recfunctions*), [115](#)

`get_names_flat()` (in `numpy.lib.recfunctions`), 115
`getitem`
 ndarray special methods, 70

H

homogeneous, 306

I

indexing, 69, 79, 313
itemsizes, 306

J

`join_by()` (in module `numpy.lib.recfunctions`), 116

L

little-endian, 306

M

mask, 306
masked array, 306
matrix, 307
memory model
 ndarray, 311
`merge_arrays()` (in module `numpy.lib.recfunctions`), 117
methods
 `accumulate`, ufunc, 318
 `reduce`, ufunc, 317
 `reduceat`, ufunc, 318
module
 `numpy.f2py`, 249
 `numpy.lib.recfunctions`, 111

N

`ndarray`, 79, 307
 memory model, 311
 special methods `getitem`, 70
 special methods `setitem`, 70
 subtyping, 218, 220
 view, 71
`ndpointer()`
 built-in function, 193
`newaxis`, 70
`numpy.f2py`
 module, 249
`numpy.lib.recfunctions`
 module, 111

O

object array, 307

P

`PyModule_AddIntConstant` (C function), 180

module `PyModule_AddObject` (C function), 180
`PyModule_AddStringConstant` (C function), 180
Python Enhancement Proposals
 PEP 440, 235
 PEP 585, 340
 PEP 646, 355
 PEP 3118, 543

R

`ravel`, 307
`rec_append_fields()` (in module `numpy.lib.recfunctions`), 118
`rec_drop_fields()` (in module `numpy.lib.recfunctions`), 118
`rec_join()` (in module `numpy.lib.recfunctions`), 118
record array, 307
`recursive_fill_fields()` (in module `numpy.lib.recfunctions`), 119
`reduce`
 ufunc methods, 317
`reduceat`
 ufunc methods, 318
reference counting, 182, 183
`rename_fields()` (in module `numpy.lib.recfunctions`), 119
`repack_fields()` (in module `numpy.lib.recfunctions`), 119
`require_fields()` (in module `numpy.lib.recfunctions`), 120
row-major, 307
`run_main()` (in module `numpy.f2py`), 251

S

scalar, 307
`setitem`
 ndarray special methods, 70
`shape`, 307
SIP, 197
slicing, 69
special methods
 `getitem`, ndarray, 70
 `setitem`, ndarray, 70
`stack_arrays()` (in module `numpy.lib.recfunctions`), 121
`stride`, 307
structured array, 307
structured data type, 307
`structured_to_unstructured()` (in module `numpy.lib.recfunctions`), 122
`subarray`, 307
subarray data type, 308
subtyping
 ndarray, 218, 220
swig, 197

T

title, [308](#)

type, [308](#)

U

ufunc, [308](#), [315](#), [318](#)

 adding new, [198](#), [199](#), [202](#), [204](#), [213](#)

 casting rules, [142](#)

 methods accumulate, [318](#)

 methods reduce, [317](#)

 methods reduceat, [318](#)

unstructured_to_structured() (*in module*
 numpy.lib.recfunctions), [123](#)

V

vectorization, [308](#)

view, [308](#)

 ndarray, [71](#)