NumPy Reference

Release 1.11.0

Written by the NumPy community

March 27, 2016

Release
1.11

Date
March 27, 2016

This reference manual details functions, modules, and objects included in Numpy, describing what they are and what they do. For learning how to use NumPy, see also user.

CHAPTER

ONE

ARRAY OBJECTS

NumPy provides an N-dimensional array type, the *ndarray*, which describes a collection of "items" of the same type. The items can be *indexed* using for example N integers.

All ndarrays are homogenous: every item takes up the same size block of memory, and all blocks are interpreted in exactly the same way. How each item in the array is to be interpreted is specified by a separate *data-type object*, one of which is associated with every array. In addition to basic types (integers, floats, *etc.*), the data type objects can also represent data structures.

An item extracted from an array, *e.g.*, by indexing, is represented by a Python object whose type is one of the *array scalar types* built in Numpy. The array scalars allow easy manipulation of also more complicated arrangements of data.

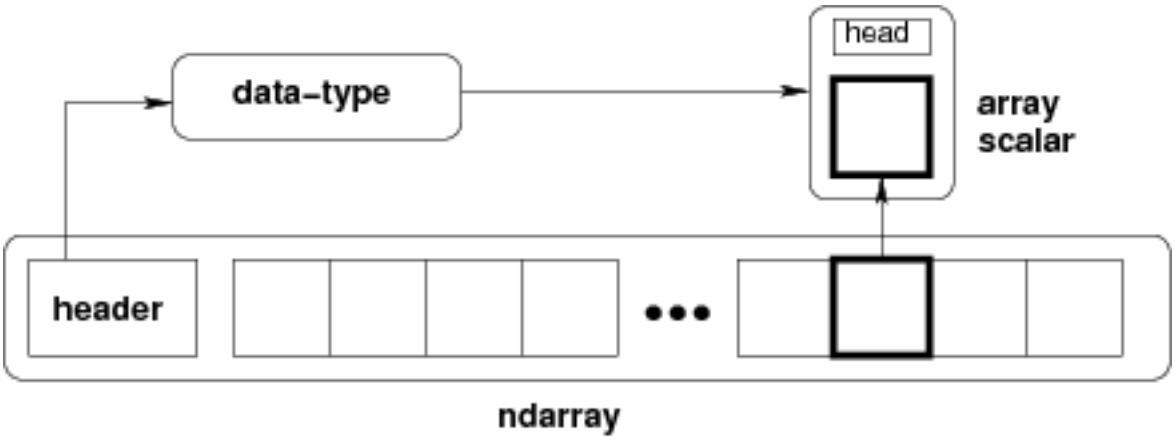


Fig. 1.1: **Figure** Conceptual diagram showing the relationship between the three fundamental objects used to describe the data in an array: 1) the ndarray itself, 2) the data-type object that describes the layout of a single fixed-size element of the array, 3) the array-scalar Python object that is returned when a single element of the array is accessed.

1.1 The N-dimensional array (`ndarray`)

An `ndarray` is a (usually fixed-size) multidimensional container of items of the same type and size. The number of dimensions and items in an array is defined by its `shape`, which is a `tuple` of *N* positive integers that specify the sizes of each dimension. The type of items in the array is specified by a separate *data-type object (dtype)*, one of which is associated with each ndarray.

As with other container objects in Python, the contents of an `ndarray` can be accessed and modified by *indexing or slicing* the array (using, for example, *N* integers), and via the methods and attributes of the `ndarray`.

Different *ndarrays* can share the same data, so that changes made in one *ndarray* may be visible in another. That is, an ndarray can be a *"view"* to another ndarray, and the data it is referring to is taken care of by the *"base"* ndarray. ndarrays can also be views to memory owned by Python `strings` or objects implementing the `buffer` or *array* interfaces.

Example

A 2-dimensional array of size 2 x 3, composed of 4-byte integer elements:

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], np.int32)
>>> type(x)
<type 'numpy.ndarray'>
>>> x.shape
(2, 3)
>>> x.dtype
dtype('int32')
```

The array can be indexed using Python container-like syntax:

```
>>> # The element of x in the *second* row, *third* column, namely, 6.
>>> x[1, 2]
```

For example *slicing* can produce views of the array:

```
>>> y = x[:,1]
>>> y
array([2, 5])
>>> y[0] = 9 # this also changes the corresponding element in x
>>> y
array([9, 5])
>>> x
array([[1, 9, 3],
       [4, 5, 6]])
```

1.1.1 Constructing arrays

New arrays can be constructed using the routines detailed in *Array creation routines*, and also by using the low-level *ndarray* constructor:

ndarray	An array object represents a multidimensional, homogeneous array of fixed-size items.

class numpy.**ndarray**

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 the methods and attributes of an array.

Parameters

(for the __new__ method; see Notes below)

shape : tuple of ints

Shape of created array.

dtype : data-type, optional

Any object that can be interpreted as a numpy data type.

buffer : object exposing buffer interface, optional

Used to fill the array with data.

offset : int, optional

Offset of array data in buffer.

strides : tuple of ints, optional

Strides of data in memory.

order : {'C', 'F'}, optional

Row-major (C-style) or column-major (Fortran-style) order.

See also:

`array`
Construct an array.

`zeros`
Create an array, each element of which is zero.

`empty`
Create an array, but leave its allocated memory unchanged (i.e., it contains "garbage").

`dtype`
Create a data-type.

Notes

There are two modes of creating an array using `__new__`:

1. If *buffer* is None, then only *`shape`*, *`dtype`*, and *order* are used.
2. If *buffer* is an object exposing the buffer interface, then all keywords are interpreted.

No `__init__` method is needed because the array is fully initialized after the `__new__` method.

Examples

These examples illustrate the low-level *`ndarray`* constructor. Refer to the *See Also* section above for easier ways of constructing an ndarray.

First mode, *buffer* is None:

```
>>> np.ndarray(shape=(2,2), dtype=float, order='F')
array([[ -1.13698227e+002,   4.25087011e-303],
       [  2.88528414e-306,   3.27025015e-309]])         #random
```

Second mode:

```
>>> np.ndarray((2,), buffer=np.array([1,2,3]),
...            offset=np.int_().itemsize,
...            dtype=int) # offset = 1*itemsize, i.e. skip first element
array([2, 3])
```

Attributes

T	Same as self.transpose(), except that self is returned if self.ndim < 2.
data	Python buffer object pointing to the start of the array's data.
dtype	Data-type of the array's elements.
flags	Information about the memory layout of the array.
flat	A 1-D iterator over the array.
imag	The imaginary part of the array.
real	The real part of the array.
size	Number of elements in the array.
itemsize	Length of one array element in bytes.
nbytes	Total bytes consumed by the elements of the array.
ndim	Number of array dimensions.
shape	Tuple of array dimensions.
strides	Tuple of bytes to step in each dimension when traversing an array.
ctypes	An object to simplify the interaction of the array with the ctypes module.
base	Base object if memory is from some other object.

ndarray.**T**

Same as self.transpose(), except that self is returned if self.ndim < 2.

Examples

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

ndarray.**data**

Python buffer object pointing to the start of the array's data.

ndarray.**dtype**

Data-type of the array's elements.

Parameters
None

Returns
d : numpy dtype object

See also:

numpy.dtype

Examples

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
```

```
>>> type(x.dtype)
<type 'numpy.dtype'>
```

ndarray.**flags**
Information about the memory layout of the array.

Notes

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling *ndarray.setflags*.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set `False`.
- ALIGNED can only be set `True` if the data is truly aligned.
- WRITEABLE can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

Attributes

C_CONTIGUOUS (C)	The data is in a single, C-style contiguous segment.
F_CONTIGUOUS (F)	The data is in a single, Fortran-style contiguous segment.
OWNDATA (O)	The array owns the memory it uses or borrows it from another object.
WRITEABLE (W)	The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.
ALIGNED (A)	The data and all elements are aligned appropriately for the hardware.
UPDATEIFCOPY (U)	This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.
FNC	F_CONTIGUOUS and not C_CONTIGUOUS.
FORC	F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).
BEHAVED (B)	ALIGNED and WRITEABLE.
CARRAY (CA)	BEHAVED and C_CONTIGUOUS.
FARRAY (FA)	BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

ndarray.**flat**
A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See also:

flatten
Return a copy of the array collapsed into one dimension.

flatiter

Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```

```
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

ndarray.**imag**
The imaginary part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.        ,  0.70710678])
>>> x.imag.dtype
dtype('float64')
```

ndarray.**real**
The real part of the array.

See also:

numpy.real
equivalent function

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.        ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

ndarray.**size**
Number of elements in the array.

Equivalent to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

ndarray.**itemsize**
Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

ndarray.**nbytes**
Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

ndarray.**ndim**
Number of array dimensions.

Examples

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

ndarray.**shape**
Tuple of array dimensions.

Notes

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

ndarray.**strides**

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the "ndarray.rst" file in the NumPy reference guide.

See also:

`numpy.lib.stride_tricks.as_strided`

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array *x* will be `(20, 4)`.

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> 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]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

ndarray.**ctypes**

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters
None

Returns
c : Python object

Possessing attributes data, shape, strides, etc.

See also:

`numpy.ctypeslib`

Notes

Below are the public attributes of this object which were documented in "Guide to NumPy" (we have omitted undocumented public attributes, as well as documented private attributes):

- data: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].
- shape (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.
- strides (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.
- data_as(obj): Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).
- shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).
- strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

Examples

```
>>> import ctypes
>>> x
```

```
        array([[0, 1],
               [2, 3]])
        >>> x.ctypes.data
        30439712
        >>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
        <ctypes.LP_c_long object at 0x01F01300>
        >>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
        c_long(0)
        >>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
        c_longlong(4294967296L)
        >>> x.ctypes.shape
        <numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
        >>> x.ctypes.shape_as(ctypes.c_long)
        <numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
        >>> x.ctypes.strides
        <numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
        >>> x.ctypes.strides_as(ctypes.c_longlong)
        <numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

ndarray.**base**

Base object if memory is from some other object.

Examples

The base of an array that owns its memory is None:

```
        >>> x = np.array([1,2,3,4])
        >>> x.base is None
        True
```

Slicing creates a view, whose memory is shared with x:

```
        >>> y = x[2:]
        >>> y.base is x
        True
```

Methods

all([axis, out, keepdims])	Returns True if all elements evaluate to True.
any([axis, out, keepdims])	Returns True if any of the elements of *a* evaluate to True.
argmax([axis, out])	Return indices of the maximum values along the given axis.
argmin([axis, out])	Return indices of the minimum values along the given axis of *a*.
argpartition(kth[, axis, kind, order])	Returns the indices that would partition this array.
argsort([axis, kind, order])	Returns the indices that would sort this array.
astype(dtype[, order, casting, subok, copy])	Copy of the array, cast to a specified type.
byteswap(inplace)	Swap the bytes of the array elements
choose(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
clip([min, max, out])	Return an array whose values are limited to `[min, max]`.
compress(condition[, axis, out])	Return selected slices of this array along given axis.
conj()	Complex-conjugate all elements.
conjugate()	Return the complex conjugate, element-wise.
copy([order])	Return a copy of the array.
cumprod([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
cumsum([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
diagonal([offset, axis1, axis2])	Return specified diagonals.

Table 1.3 – continued from previous page

dot(b[, out])	Dot product of two arrays.
dump(file)	Dump a pickle of the array to the specified file.
dumps()	Returns the pickle of the array as a string.
fill(value)	Fill the array with a scalar value.
flatten([order])	Return a copy of the array collapsed into one dimension.
getfield(dtype[, offset])	Returns a field of the given array as a certain type.
item(*args)	Copy an element of an array to a standard Python scalar and return it.
itemset(*args)	Insert scalar into an array (scalar is cast to array's dtype, if possible)
max([axis, out])	Return the maximum along a given axis.
mean([axis, dtype, out, keepdims])	Returns the average of the array elements along given axis.
min([axis, out, keepdims])	Return the minimum along a given axis.
newbyteorder([new_order])	Return the array with the same data viewed with a different byte order.
nonzero()	Return the indices of the elements that are non-zero.
partition(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that value of the element in kth po
prod([axis, dtype, out, keepdims])	Return the product of the array elements over the given axis
ptp([axis, out])	Peak to peak (maximum - minimum) value along a given axis.
put(indices, values[, mode])	Set `a.flat[n] = values[n]` for all *n* in indices.
ravel([order])	Return a flattened array.
repeat(repeats[, axis])	Repeat elements of an array.
reshape(shape[, order])	Returns an array containing the same data with a new shape.
resize(new_shape[, refcheck])	Change shape and size of array in-place.
round([decimals, out])	Return *a* with each element rounded to the given number of decimals.
searchsorted(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
setfield(val, dtype[, offset])	Put a value into a specified place in a field defined by a data-type.
setflags([write, align, uic])	Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.
sort([axis, kind, order])	Sort an array, in-place.
squeeze([axis])	Remove single-dimensional entries from the shape of *a*.
std([axis, dtype, out, ddof, keepdims])	Returns the standard deviation of the array elements along given axis.
sum([axis, dtype, out, keepdims])	Return the sum of the array elements over the given axis.
swapaxes(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
take(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.
tobytes([order])	Construct Python bytes containing the raw data bytes in the array.
tofile(fid[, sep, format])	Write array to a file as text or binary (default).
tolist()	Return the array as a (possibly nested) list.
tostring([order])	Construct Python bytes containing the raw data bytes in the array.
trace([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.
transpose(*axes)	Returns a view of the array with axes transposed.
var([axis, dtype, out, ddof, keepdims])	Returns the variance of the array elements, along given axis.
view([dtype, type])	New view of array with the same data.

ndarray.**all**(*axis=None*, *out=None*, *keepdims=False*)

Returns True if all elements evaluate to True.

Refer to `numpy.all` for full documentation.

See also:

numpy.all

equivalent function

ndarray.**any**(*axis=None*, *out=None*, *keepdims=False*)

Returns True if any of the elements of *a* evaluate to True.

Refer to `numpy.any` for full documentation.

See also:

numpy.any
equivalent function

`ndarray.`**argmax**(*axis=None*, *out=None*)
Return indices of the maximum values along the given axis.

Refer to `numpy.argmax` for full documentation.

See also:

numpy.argmax
equivalent function

`ndarray.`**argmin**(*axis=None*, *out=None*)
Return indices of the minimum values along the given axis of *a*.

Refer to `numpy.argmin` for detailed documentation.

See also:

numpy.argmin
equivalent function

`ndarray.`**argpartition**(*kth*, *axis=-1*, *kind='introselect'*, *order=None*)
Returns the indices that would partition this array.

Refer to `numpy.argpartition` for full documentation.

New in version 1.8.0.

See also:

numpy.argpartition
equivalent function

`ndarray.`**argsort**(*axis=-1*, *kind='quicksort'*, *order=None*)
Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

See also:

numpy.argsort
equivalent function

`ndarray.`**astype**(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)
Copy of the array, cast to a specified type.

Parameters
dtype : str or dtype

Typecode or data-type to which the array is cast.

order : {'C', 'F', 'A', 'K'}, optional

Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional

Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

subok : bool, optional

If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy : bool, optional

By default, astype always returns a newly allocated array. If this is set to false, and the `dtype`, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

Returns

arr_t : ndarray

Unless `copy` is False and the other conditions for returning the input array are satisfied (see description for `copy` input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by `dtype`, *order*.

Raises

ComplexWarning

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Notes

Starting in NumPy 1.9, 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.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

ndarray.**byteswap**(*inplace*)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

Parameters

inplace : bool, optional

If `True`, swap bytes in-place, default is `False`.

Returns

out : ndarray

The byteswapped array. If *inplace* is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([  256,     1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']
```

Arrays of strings are not swapped

```
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
      dtype='|S3')
```

`ndarray.`**`choose`**(*choices*, *out=None*, *mode='raise'*)

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See also:

`numpy.choose`
equivalent function

`ndarray.`**`clip`**(*min=None*, *max=None*, *out=None*)

Return an array whose values are limited to `[min, max]`. One of max or min must be given.

Refer to `numpy.clip` for full documentation.

See also:

`numpy.clip`
equivalent function

`ndarray.`**`compress`**(*condition*, *axis=None*, *out=None*)

Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See also:

`numpy.compress`
equivalent function

ndarray.**conj**()
Complex-conjugate all elements.

Refer to numpy.conjugate for full documentation.

See also:

numpy.conjugate
equivalent function

ndarray.**conjugate**()
Return the complex conjugate, element-wise.

Refer to numpy.conjugate for full documentation.

See also:

numpy.conjugate
equivalent function

ndarray.**copy**(*order='C'*)
Return a copy of the array.

Parameters
order : {'C', 'F', 'A', 'K'}, optional

Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and :func:numpy.copy are very similar, but have different default values for their order= arguments.)

See also:

numpy.copy, numpy.copyto

Examples

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

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

ndarray.**cumprod**(*axis=None*, *dtype=None*, *out=None*)
Return the cumulative product of the elements along the given axis.

Refer to numpy.cumprod for full documentation.

See also:

numpy.cumprod
equivalent function

ndarray.**cumsum**(*axis=None*, *dtype=None*, *out=None*)
Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See also:

numpy.cumsum
equivalent function

ndarray.**diagonal**(*offset=0*, *axis1=0*, *axis2=1*)
Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal` for full documentation.

See also:

numpy.diagonal
equivalent function

ndarray.**dot**(*b*, *out=None*)
Dot product of two arrays.

Refer to `numpy.dot` for full documentation.

See also:

numpy.dot
equivalent function

Examples

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

ndarray.**dump**(*file*)
Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters
file : str

A string naming the dump file.

ndarray.**dumps**()
Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

Parameters
None

ndarray.**fill**(*value*)
Fill the array with a scalar value.

Parameters
value : scalar

All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1.,  1.])
```

ndarray.**flatten**(*order='C'*)
Return a copy of the array collapsed into one dimension.

Parameters
order : {'C', 'F', 'A', 'K'}, optional

'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

Returns
y : ndarray

A copy of the input array, flattened to one dimension.

See also:

ravel
Return a flattened array.

flat
A 1-D flat iterator over the array.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

ndarray.**getfield**(*dtype*, *offset=0*)
Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype : str or dtype

The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset : int

Number of bytes to skip before beginning the element view.

Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[ 1.+1.j,  0.+0.j],
       [ 0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[ 1.,  0.],
       [ 0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[ 1.,  0.],
   [ 0.,  4.]])
```

ndarray.**item**(**args*)

Copy an element of an array to a standard Python scalar and return it.

Parameters

***args** : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z : Standard Python scalar object

A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
```

```
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

ndarray.**itemset**(**args*)

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

Parameters

***args** : Arguments

If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes

Compared to indexing syntax, *itemset* provides some speed increase for placing a scalar into a particular location in an *ndarray*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using *itemset* (and *item*) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

ndarray.**max**(*axis=None*, *out=None*)

Return the maximum along a given axis.

Refer to `numpy.amax` for full documentation.

See also:

numpy.amax
equivalent function

ndarray.**mean**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

See also:

numpy.mean
equivalent function

ndarray.**min**(*axis=None*, *out=None*, *keepdims=False*)
Return the minimum along a given axis.

Refer to numpy.amin for full documentation.

See also:

numpy.amin
equivalent function

ndarray.**newbyteorder**(*new_order='S'*)
Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

Parameters
new_order : string, optional

Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns
new_arr : array

New array object with the dtype reflecting given change to the byte order.

ndarray.**nonzero**()
Return the indices of the elements that are non-zero.

Refer to numpy.nonzero for full documentation.

See also:

numpy.nonzero
equivalent function

ndarray.**partition**(*kth*, *axis=-1*, *kind='introselect'*, *order=None*)
Rearranges the elements in the array in such a way that value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this

element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

Parameters

kth : int or sequence of ints

Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order all elements in the partitions is undefined. If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.

axis : int, optional

Axis along which to sort. Default is -1, which means sort along the last axis.

kind : {'introselect'}, optional

Selection algorithm. Default is 'introselect'.

order : str or list of str, optional

When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See also:

numpy.partition
Return a parititioned copy of an array.

argpartition
Indirect partition.

sort
Full sort.

Notes

See `np.partition` for notes on the different algorithms.

Examples

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

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

`ndarray.`**`prod`**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Return the product of the array elements over the given axis

Refer to `numpy.prod` for full documentation.

See also:

numpy.prod
equivalent function

ndarray.**ptp**(*axis=None*, *out=None*)
Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See also:

numpy.ptp
equivalent function

ndarray.**put**(*indices*, *values*, *mode='raise'*)
Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to `numpy.put` for full documentation.

See also:

numpy.put
equivalent function

ndarray.**ravel**([*order*])
Return a flattened array.

Refer to `numpy.ravel` for full documentation.

See also:

numpy.ravel
equivalent function

ndarray.flat
a flat iterator on the array.

ndarray.**repeat**(*repeats*, *axis=None*)
Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

See also:

numpy.repeat
equivalent function

ndarray.**reshape**(*shape*, *order='C'*)
Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

See also:

numpy.reshape
equivalent function

ndarray.**resize**(*new_shape*, *refcheck=True*)
Change shape and size of array in-place.

Parameters
new_shape : tuple of ints, or *n* ints

Shape of resized array.

refcheck : bool, optional

If False, reference count will not be checked. Default is True.

Returns

None

Raises

ValueError

If *a* does not own its own data or references or views to it exist, and the data memory must be changed.

SystemError

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

See also:

`resize`
Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

ndarray.**round**(*decimals=0*, *out=None*)

Return *a* with each element rounded to the given number of decimals.

Refer to `numpy.around` for full documentation.

See also:

numpy.around
equivalent function

ndarray.**searchsorted**(*v*, *side='left'*, *sorter=None*)

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see `numpy.searchsorted`

See also:

numpy.searchsorted
equivalent function

ndarray.**setfield**(*val*, *dtype*, *offset=0*)

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *dtype* and beginning *offset* bytes into the field.

Parameters

val : object

Value to be placed in field.

dtype : dtype object

Data-type of the field in which to place *val*.

offset : int, optional

The number of bytes into the field at which to place *val*.

Returns

None

See also:

getfield

Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
>>> x
array([[  1.00000000e+000,   1.48219694e-323,   1.48219694e-323],
       [  1.48219694e-323,   1.00000000e+000,   1.48219694e-323],
       [  1.48219694e-323,   1.48219694e-323,   1.00000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

ndarray.**setflags**(*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write : bool, optional

Describes whether or not *a* can be written to.

align : bool, optional

Describes whether or not *a* is aligned properly for its type.

uic : bool, optional

Describes whether or not *a* is a copy of another "base" array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

Examples

```
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

ndarray.**sort**(*axis=-1*, *kind='quicksort'*, *order=None*)

Sort an array, in-place.

Parameters

axis : int, optional

Axis along which to sort. Default is -1, which means sort along the last axis.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional

Sorting algorithm. Default is 'quicksort'.

order : str or list of str, optional

When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See also:

numpy.sort
Return a sorted copy of an array.

argsort
Indirect sort.

lexsort
Indirect stable sort on multiple keys.

searchsorted
Find elements in sorted array.

partition
Partial sort.

Notes

See `sort` for notes on the different sorting algorithms.

Examples

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

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

`ndarray.`**`squeeze`**(*axis=None*)
Remove single-dimensional entries from the shape of *a*.

Refer to `numpy.squeeze` for full documentation.

See also:

`numpy.squeeze`
equivalent function

`ndarray.`**`std`**(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)
Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

See also:

`numpy.std`
equivalent function

`ndarray.`**`sum`**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

See also:

`numpy.sum`
equivalent function

`ndarray.`**`swapaxes`**(*axis1*, *axis2*)
Return a view of the array with *axis1* and *axis2* interchanged.

Refer to `numpy.swapaxes` for full documentation.

See also:

numpy.swapaxes
equivalent function

ndarray.**take**(*indices*, *axis=None*, *out=None*, *mode='raise'*)
Return an array formed from the elements of *a* at the given indices.

Refer to `numpy.take` for full documentation.

See also:

numpy.take
equivalent function

ndarray.**tobytes**(*order='C'*)
Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

New in version 1.9.0.

Parameters
order : {'C', 'F', None}, optional

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns
s : bytes

Python bytes exhibiting a copy of *a*'s raw data.

Examples

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

ndarray.**tofile**(*fid*, *sep=""*, *format="%s"*)
Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function fromfile().

Parameters
fid : file or str

An open file object, or a string containing a filename.

sep : str

Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format : str

Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

ndarray.**tolist**()
Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

Parameters
none

Returns
y : list

The possibly nested list of array elements.

Notes

The array may be recreated, `a = np.array(a.tolist())`.

Examples

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

ndarray.**tostring**(*order='C'*)
Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

This function is a compatibility alias for tobytes. Despite its name it returns bytes not strings.

Parameters
order : {'C', 'F', None}, optional

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns
s : bytes

Python bytes exhibiting a copy of *a*'s raw data.

Examples

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
```

```
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

ndarray.**trace**(*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See also:

numpy.trace
equivalent function

ndarray.**transpose**(**axes*)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

Parameters

axes : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

Returns

out : ndarray

View of *a*, with axes suitably permuted.

See also:

ndarray.T
Array property returning the array transposed.

Examples

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

ndarray.**var**(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)
Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

See also:

numpy.var
equivalent function

ndarray.**view**(*dtype=None*, *type=None*)
New view of array with the same data.

Parameters

dtype : data-type or ndarray sub-class, optional

Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

type : Python type, optional

Type of the returned view, e.g., ndarray or matrix. Again, the default None results in type preservation.

Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrixlib.defmatrix.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
```

```
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([ 2.,  3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> print(x)
[(1, 20) (3, 4)]
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

1.1.2 Indexing arrays

Arrays can be indexed using an extended Python slicing syntax, `array[selection]`. Similar syntax is also used for accessing fields in a structured array.

See also:

Array Indexing.

1.1.3 Internal memory layout of an ndarray

An instance of class *ndarray* consists of a contiguous one-dimensional segment of computer memory (owned by the array, or by some other object), combined with an indexing scheme that maps *N* integers into the location of an item in the block. The ranges in which the indices can vary is specified by the *shape* of the array. How many bytes each item takes and how the bytes are interpreted is defined by the *data-type object* associated with the array.

A segment of memory is inherently 1-dimensional, and there are many different schemes for arranging the items of an *N*-dimensional array in a 1-dimensional block. Numpy is flexible, and *ndarray* objects can accommodate any

strided indexing scheme. In a strided scheme, the N-dimensional index $(n_0, n_1, ..., n_{N-1})$ corresponds to the offset (in bytes):

$$n_{\text{offset}} = \sum_{k=0}^{N-1} s_k n_k$$

from the beginning of the memory block associated with the array. Here, s_k are integers which specify the `strides` of the array. The column-major order (used, for example, in the Fortran language and in *Matlab*) and row-major order (used in C) schemes are just specific kinds of strided scheme, and correspond to memory that can be *addressed* by the strides:

$$s_k^{\text{column}} = \prod_{j=0}^{k-1} d_j, \quad s_k^{\text{row}} = \prod_{j=k+1}^{N-1} d_j.$$

where d_j = *self.itemsize* * *self.shape[j]*.

Both the C and Fortran orders are contiguous, *i.e.,* single-segment, memory layouts, in which every part of the memory block can be accessed by some combination of the indices.

While a C-style and Fortran-style contiguous array, which has the corresponding flags set, can be addressed with the above strides, the actual strides may be different. This can happen in two cases:

1. If `self.shape[k] == 1` then for any legal index `index[k] == 0`. This means that in the formula for the offset $n_k = 0$ and thus $s_k n_k = 0$ and the value of s_k = *self.strides[k]* is arbitrary.
2. If an array has no elements (`self.size == 0`) there is no legal index and the strides are never used. Any array with no elements may be considered C-style and Fortran-style contiguous.

Point 1. means that `self``and ``self.squeeze()` always have the same contiguity and aligned flags value. This also means that even a high dimensional array could be C-style and Fortran-style contiguous at the same time.

An array is considered aligned if the memory offsets for all elements and the base offset itself is a multiple of *self.itemsize*.

Note: Points (1) and (2) are not yet applied by default. Beginning with Numpy 1.8.0, they are applied consistently only if the environment variable `NPY_RELAXED_STRIDES_CHECKING=1` was defined when NumPy was built. Eventually this will become the default.

You can check whether this option was enabled when your NumPy was built by looking at the value of `np.ones((10,1), order='C').flags.f_contiguous`. If this is `True`, then your NumPy has relaxed strides checking enabled.

Warning: It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

Data in new `ndarrays` is in the row-major (C) order, unless otherwise specified, but, for example, *basic array slicing* often produces views in a different scheme.

Note: Several algorithms in NumPy work on arbitrarily strided arrays. However, some algorithms require single-segment arrays. When an irregularly strided array is passed in to such algorithms, a copy is automatically made.

1.1.4 Array attributes

Array attributes reflect information that is intrinsic to the array itself. Generally, accessing an array through its attributes allows you to get and sometimes set intrinsic properties of the array without creating a new array. The exposed attributes are the core parts of an array and only some of them can be reset meaningfully without creating a new array. Information on each attribute is given below.

Memory layout

The following attributes contain information about the memory layout of the array:

ndarray.flags	Information about the memory layout of the array.
ndarray.shape	Tuple of array dimensions.
ndarray.strides	Tuple of bytes to step in each dimension when traversing an array.
ndarray.ndim	Number of array dimensions.
ndarray.data	Python buffer object pointing to the start of the array's data.
ndarray.size	Number of elements in the array.
ndarray.itemsize	Length of one array element in bytes.
ndarray.nbytes	Total bytes consumed by the elements of the array.
ndarray.base	Base object if memory is from some other object.

ndarray.**flags**
Information about the memory layout of the array.

Notes

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling *ndarray.setflags*.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set `False`.
- ALIGNED can only be set `True` if the data is truly aligned.
- WRITEABLE can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

Attributes

C_CONTIGUOUS (C)	The data is in a single, C-style contiguous segment.
F_CONTIGUOUS (F)	The data is in a single, Fortran-style contiguous segment.
OWNDATA (O)	The array owns the memory it uses or borrows it from another object.
WRITEABLE (W)	The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.
ALIGNED (A)	The data and all elements are aligned appropriately for the hardware.
UPDATEIFCOPY (U)	This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.
FNC	F_CONTIGUOUS and not C_CONTIGUOUS.
FORC	F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).
BEHAVED (B)	ALIGNED and WRITEABLE.
CARRAY (CA)	BEHAVED and C_CONTIGUOUS.
FARRAY (FA)	BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

ndarray.**shape**
Tuple of array dimensions.

Notes

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

ndarray.**strides**
Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the "ndarray.rst" file in the NumPy reference guide.

See also:

`numpy.lib.stride_tricks.as_strided`

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array *x* will be `(20, 4)`.

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> 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]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

`ndarray.`**`ndim`**

Number of array dimensions.

Examples

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

ndarray.**data**
Python buffer object pointing to the start of the array's data.

ndarray.**size**
Number of elements in the array.

Equivalent to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

ndarray.**itemsize**
Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

ndarray.**nbytes**
Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

ndarray.**base**
Base object if memory is from some other object.

Examples

The base of an array that owns its memory is None:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```
>>> y = x[2:]
>>> y.base is x
True
```

Data type

See also:

Data type objects

The data type object associated with the array can be found in the `dtype` attribute:

`ndarray.dtype`	Data-type of the array's elements.

ndarray.**dtype**
Data-type of the array's elements.

Parameters
None

Returns
d : numpy dtype object

See also:

`numpy.dtype`

Examples

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

Other attributes

`ndarray.T`	Same as self.transpose(), except that self is returned if self.ndim < 2.
`ndarray.real`	The real part of the array.
`ndarray.imag`	The imaginary part of the array.
`ndarray.flat`	A 1-D iterator over the array.
`ndarray.ctypes`	An object to simplify the interaction of the array with the ctypes module.

ndarray.**T**
Same as self.transpose(), except that self is returned if self.ndim < 2.

Examples

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

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

ndarray.**real**
The real part of the array.

See also:

numpy.real
equivalent function

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.        ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

ndarray.**imag**
The imaginary part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.        ,  0.70710678])
>>> x.imag.dtype
dtype('float64')
```

ndarray.**flat**
A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See also:

flatten
Return a copy of the array collapsed into one dimension.

flatiter

Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

ndarray.**ctypes**

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters
None

Returns
c : Python object

Possessing attributes data, shape, strides, etc.

See also:

`numpy.ctypeslib`

Notes

Below are the public attributes of this object which were documented in "Guide to NumPy" (we have omitted undocumented public attributes, as well as documented private attributes):

- data: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].
- shape (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.
- strides (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.
- data_as(obj): Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).
- shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).
- strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this

problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

Examples

```
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

Array interface

See also:

The Array Interface.

__array_interface__	Python-side of the array interface
__array_struct__	C-side of the array interface

`ctypes` foreign function interface

ndarray.ctypes	An object to simplify the interaction of the array with the ctypes module.

`ndarray.`**`ctypes`**

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters
None

Returns
c : Python object

Possessing attributes data, shape, strides, etc.

See also:

`numpy.ctypeslib`

Notes

Below are the public attributes of this object which were documented in "Guide to NumPy" (we have omitted undocumented public attributes, as well as documented private attributes):

- data: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].
- shape (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.
- strides (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.
- data_as(obj): Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).
- shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).
- strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

Examples

```
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
```

```
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

1.1.5 Array methods

An *ndarray* object has many methods which operate on or with the array in some fashion, typically returning an array result. These methods are briefly explained below. (Each method's docstring has a more complete description.)

For the following methods there are also corresponding functions in *numpy*: `all`, `any`, `argmax`, `argmin`, `argpartition`, `argsort`, `choose`, `clip`, `compress`, `copy`, `cumprod`, `cumsum`, `diagonal`, `imag`, `max`, `mean`, `min`, `nonzero`, `partition`, `prod`, `ptp`, `put`, `ravel`, `real`, `repeat`, `reshape`, `round`, `searchsorted`, `sort`, `squeeze`, `std`, `sum`, `swapaxes`, `take`, `trace`, `transpose`, `var`.

Array conversion

ndarray.item(*args)	Copy an element of an array to a standard Python scalar and return it.
ndarray.tolist()	Return the array as a (possibly nested) list.
ndarray.itemset(*args)	Insert scalar into an array (scalar is cast to array's dtype, if possible)
ndarray.tostring([order])	Construct Python bytes containing the raw data bytes in the array.
ndarray.tobytes([order])	Construct Python bytes containing the raw data bytes in the array.
ndarray.tofile(fid[, sep, format])	Write array to a file as text or binary (default).
ndarray.dump(file)	Dump a pickle of the array to the specified file.
ndarray.dumps()	Returns the pickle of the array as a string.
ndarray.astype(dtype[, order, casting, ...])	Copy of the array, cast to a specified type.
ndarray.byteswap(inplace)	Swap the bytes of the array elements
ndarray.copy([order])	Return a copy of the array.
ndarray.view([dtype, type])	New view of array with the same data.
ndarray.getfield(dtype[, offset])	Returns a field of the given array as a certain type.
ndarray.setflags([write, align, uic])	Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.
ndarray.fill(value)	Fill the array with a scalar value.

ndarray.**item**(**args*)

Copy an element of an array to a standard Python scalar and return it.

Parameters

***args** : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z : Standard Python scalar object

A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

ndarray.**tolist**()

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

Parameters
none

Returns
y : list

The possibly nested list of array elements.

Notes

The array may be recreated, `a = np.array(a.tolist())`.

Examples

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

ndarray.**itemset**(**args*)

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

Parameters

***args** : Arguments

If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes

Compared to indexing syntax, *itemset* provides some speed increase for placing a scalar into a particular location in an *ndarray*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using *itemset* (and *item*) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

ndarray.**tostring**(*order='C'*)

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

This function is a compatibility alias for tobytes. Despite its name it returns bytes not strings.

Parameters

order : {'C', 'F', None}, optional

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns

s : bytes

Python bytes exhibiting a copy of *a*'s raw data.

Examples

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

ndarray.**tobytes**(*order='C'*)

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

New in version 1.9.0.

Parameters

order : {'C', 'F', None}, optional

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns

s : bytes

Python bytes exhibiting a copy of *a*'s raw data.

Examples

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

ndarray.**tofile**(*fid*, *sep=""*, *format="%s"*)

Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function fromfile().

Parameters

fid : file or str

An open file object, or a string containing a filename.

sep : str

Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format : str

Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

ndarray.**dump**(*file*)

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters

file : str

A string naming the dump file.

ndarray.**dumps**()
Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

Parameters
None

ndarray.**astype**(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)
Copy of the array, cast to a specified type.

Parameters
dtype : str or dtype

Typecode or data-type to which the array is cast.

order : {'C', 'F', 'A', 'K'}, optional

Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional

Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

subok : bool, optional

If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy : bool, optional

By default, astype always returns a newly allocated array. If this is set to false, and the `dtype`, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

Returns
arr_t : ndarray

Unless `copy` is False and the other conditions for returning the input array are satisfied (see description for `copy` input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by `dtype`, *order*.

Raises
ComplexWarning

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Notes

Starting in NumPy 1.9, 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.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

ndarray.**byteswap**(*inplace*)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

Parameters

inplace : bool, optional

If `True`, swap bytes in-place, default is `False`.

Returns

out : ndarray

The byteswapped array. If *inplace* is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([  256,     1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']
```

Arrays of strings are not swapped

```
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
      dtype='|S3')
```

ndarray.**copy**(*order='C'*)

Return a copy of the array.

Parameters

order : {'C', 'F', 'A', 'K'}, optional

Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and :func:numpy.copy are very similar, but have different default values for their order= arguments.)

See also:

`numpy.copy`, `numpy.copyto`

Examples

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

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

ndarray.**view**(*dtype=None*, *type=None*)

New view of array with the same data.

Parameters

dtype : data-type or ndarray sub-class, optional

Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

type : Python type, optional

Type of the returned view, e.g., ndarray or matrix. Again, the default None results in type preservation.

Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
```

```
>>> print(type(y))
<class 'numpy.matrixlib.defmatrix.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([ 2.,  3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> print(x)
[(1, 20) (3, 4)]
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

ndarray.**getfield**(*dtype*, *offset=0*)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype : str or dtype

The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset : int

Number of bytes to skip before beginning the element view.

Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[ 1.+1.j,  0.+0.j],
       [ 0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[ 1.,  0.],
       [ 0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[ 1.,  0.],
   [ 0.,  4.]])
```

ndarray.**setflags**(*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write : bool, optional

Describes whether or not *a* can be written to.

align : bool, optional

Describes whether or not *a* is aligned properly for its type.

uic : bool, optional

Describes whether or not *a* is a copy of another "base" array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

Examples

```
>>> y
array([[3, 1, 7],
       [2, 0, 0],
```

```
       [8, 5, 9]])
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

ndarray.**fill**(*value*)

Fill the array with a scalar value.

Parameters

value : scalar

All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1.,  1.])
```

Shape manipulation

For reshape, resize, and transpose, the single tuple argument may be replaced with `n` integers which will be interpreted as an n-tuple.

ndarray.reshape(shape[, order])	Returns an array containing the same data with a new shape.
ndarray.resize(new_shape[, refcheck])	Change shape and size of array in-place.
ndarray.transpose(*axes)	Returns a view of the array with axes transposed.
ndarray.swapaxes(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
ndarray.flatten([order])	Return a copy of the array collapsed into one dimension.
ndarray.ravel([order])	Return a flattened array.
ndarray.squeeze([axis])	Remove single-dimensional entries from the shape of *a*.

ndarray.**reshape**(*shape*, *order='C'*)

Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

See also:

numpy.reshape
equivalent function

ndarray.**resize**(*new_shape*, *refcheck=True*)
Change shape and size of array in-place.

Parameters
new_shape : tuple of ints, or *n* ints

Shape of resized array.

refcheck : bool, optional

If False, reference count will not be checked. Default is True.

Returns
None

Raises
ValueError

If *a* does not own its own data or references or views to it exist, and the data memory must be changed.

SystemError

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

See also:

resize
Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

ndarray.**transpose**(**axes*)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

Parameters

axes : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

Returns

out : ndarray

View of *a*, with axes suitably permuted.

See also:

ndarray.T

Array property returning the array transposed.

Examples

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

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

ndarray.**swapaxes**(*axis1*, *axis2*)

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to `numpy.swapaxes` for full documentation.

See also:

numpy.swapaxes
equivalent function

ndarray.**flatten**(*order='C'*)

Return a copy of the array collapsed into one dimension.

Parameters
order : {'C', 'F', 'A', 'K'}, optional

'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

Returns
y : ndarray

A copy of the input array, flattened to one dimension.

See also:

ravel
Return a flattened array.

flat
A 1-D flat iterator over the array.

Examples

```
    >>> a = np.array([[1,2], [3,4]])
    >>> a.flatten()
    array([1, 2, 3, 4])
    >>> a.flatten('F')
    array([1, 3, 2, 4])
```

ndarray.**ravel**([*order*])

Return a flattened array.

Refer to `numpy.ravel` for full documentation.

See also:

numpy.ravel
equivalent function

ndarray.flat
a flat iterator on the array.

ndarray.**squeeze**(*axis=None*)
Remove single-dimensional entries from the shape of *a*.

Refer to `numpy.squeeze` for full documentation.

See also:

numpy.squeeze
equivalent function

Item selection and manipulation

For array methods that take an *axis* keyword, it defaults to `None`. If axis is *None*, then the array is treated as a 1-D array. Any other value for *axis* represents the dimension along which the operation should proceed.

ndarray.take(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.
ndarray.put(indices, values[, mode])	Set `a.flat[n] = values[n]` for all *n* in indices.
ndarray.repeat(repeats[, axis])	Repeat elements of an array.
ndarray.choose(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
ndarray.sort([axis, kind, order])	Sort an array, in-place.
ndarray.argsort([axis, kind, order])	Returns the indices that would sort this array.
ndarray.partition(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that value of the element i
ndarray.argpartition(kth[, axis, kind, order])	Returns the indices that would partition this array.
ndarray.searchsorted(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
ndarray.nonzero()	Return the indices of the elements that are non-zero.
ndarray.compress(condition[, axis, out])	Return selected slices of this array along given axis.
ndarray.diagonal([offset, axis1, axis2])	Return specified diagonals.

ndarray.**take**(*indices*, *axis=None*, *out=None*, *mode='raise'*)
Return an array formed from the elements of *a* at the given indices.

Refer to `numpy.take` for full documentation.

See also:

numpy.take
equivalent function

ndarray.**put**(*indices*, *values*, *mode='raise'*)
Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to `numpy.put` for full documentation.

See also:

numpy.put
equivalent function

ndarray.**repeat**(*repeats*, *axis=None*)
Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

See also:

numpy.repeat
equivalent function

ndarray.**choose**(*choices*, *out=None*, *mode='raise'*)
Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See also:

numpy.choose
equivalent function

ndarray.**sort**(*axis=-1*, *kind='quicksort'*, *order=None*)
Sort an array, in-place.

Parameters
axis : int, optional

Axis along which to sort. Default is -1, which means sort along the last axis.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional

Sorting algorithm. Default is 'quicksort'.

order : str or list of str, optional

When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See also:

numpy.sort
Return a sorted copy of an array.

argsort
Indirect sort.

lexsort
Indirect stable sort on multiple keys.

searchsorted
Find elements in sorted array.

partition
Partial sort.

Notes

See `sort` for notes on the different sorting algorithms.

Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
```

```
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

ndarray.**argsort**(*axis=-1*, *kind='quicksort'*, *order=None*)

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

See also:

numpy.argsort
equivalent function

ndarray.**partition**(*kth*, *axis=-1*, *kind='introselect'*, *order=None*)

Rearranges the elements in the array in such a way that value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

Parameters

kth : int or sequence of ints

Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order all elements in the partitions is undefined. If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.

axis : int, optional

Axis along which to sort. Default is -1, which means sort along the last axis.

kind : {'introselect'}, optional

Selection algorithm. Default is 'introselect'.

order : str or list of str, optional

When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See also:

numpy.partition
Return a parititioned copy of an array.

argpartition
Indirect partition.

sort
Full sort.

Notes

See `np.partition` for notes on the different algorithms.

Examples

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

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

ndarray.**argpartition**(*kth*, *axis=-1*, *kind='introselect'*, *order=None*)
Returns the indices that would partition this array.

Refer to `numpy.argpartition` for full documentation.

New in version 1.8.0.

See also:

numpy.argpartition
equivalent function

ndarray.**searchsorted**(*v*, *side='left'*, *sorter=None*)
Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see `numpy.searchsorted`

See also:

numpy.searchsorted
equivalent function

ndarray.**nonzero**()
Return the indices of the elements that are non-zero.

Refer to `numpy.nonzero` for full documentation.

See also:

numpy.nonzero
equivalent function

ndarray.**compress**(*condition*, *axis=None*, *out=None*)
Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See also:

numpy.compress
equivalent function

ndarray.**diagonal**(*offset=0*, *axis1=0*, *axis2=1*)

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal` for full documentation.

See also:

numpy.diagonal
equivalent function

Calculation

Many of these methods take an argument named *axis*. In such cases,

- If *axis* is *None* (the default), the array is treated as a 1-D array and the operation is performed over the entire array. This behavior is also the default if self is a 0-dimensional array or array scalar. (An array scalar is an instance of the types/classes float32, float64, etc., whereas a 0-dimensional array is an ndarray instance containing precisely one array scalar.)
- If *axis* is an integer, then the operation is done over the given axis (for each 1-D subarray that can be created along the given axis).

Example of the *axis* argument

A 3-dimensional array of size 3 x 3 x 3, summed over each of its three axes

```
>>> 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]]])
>>> x.sum(axis=0)
array([[27, 30, 33],
       [36, 39, 42],
       [45, 48, 51]])
>>> # for sum, axis is the first keyword, so we may omit it,
>>> # specifying only its value
>>> x.sum(0), x.sum(1), x.sum(2)
(array([[27, 30, 33],
        [36, 39, 42],
        [45, 48, 51]]),
 array([[ 9, 12, 15],
        [36, 39, 42],
        [63, 66, 69]]),
 array([[ 3, 12, 21],
        [30, 39, 48],
        [57, 66, 75]]))
```

The parameter *dtype* specifies the data type over which a reduction operation (like summing) should take place. The default reduce data type is the same as the data type of *self*. To avoid overflow, it can be useful to perform the reduction using a larger data type.

For several methods, an optional *out* argument can also be provided and the result will be placed into the output array given. The *out* argument must be an *ndarray* and have the same number of elements. It can have a different data type in which case casting will be performed.

ndarray.argmax([axis, out])	Return indices of the maximum values along the given axis.
ndarray.min([axis, out, keepdims])	Return the minimum along a given axis.
ndarray.argmin([axis, out])	Return indices of the minimum values along the given axis of *a*.
ndarray.ptp([axis, out])	Peak to peak (maximum - minimum) value along a given axis.
ndarray.clip([min, max, out])	Return an array whose values are limited to `[min, max]`.
ndarray.conj()	Complex-conjugate all elements.
ndarray.round([decimals, out])	Return *a* with each element rounded to the given number of decimals.
ndarray.trace([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.
ndarray.sum([axis, dtype, out, keepdims])	Return the sum of the array elements over the given axis.
ndarray.cumsum([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
ndarray.mean([axis, dtype, out, keepdims])	Returns the average of the array elements along given axis.
ndarray.var([axis, dtype, out, ddof, keepdims])	Returns the variance of the array elements, along given axis.
ndarray.std([axis, dtype, out, ddof, keepdims])	Returns the standard deviation of the array elements along given axis.
ndarray.prod([axis, dtype, out, keepdims])	Return the product of the array elements over the given axis
ndarray.cumprod([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
ndarray.all([axis, out, keepdims])	Returns True if all elements evaluate to True.
ndarray.any([axis, out, keepdims])	Returns True if any of the elements of *a* evaluate to True.

ndarray.**argmax**(*axis=None*, *out=None*)
Return indices of the maximum values along the given axis.

Refer to `numpy.argmax` for full documentation.

See also:

numpy.argmax
equivalent function

ndarray.**min**(*axis=None*, *out=None*, *keepdims=False*)
Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

See also:

numpy.amin
equivalent function

ndarray.**argmin**(*axis=None*, *out=None*)
Return indices of the minimum values along the given axis of *a*.

Refer to `numpy.argmin` for detailed documentation.

See also:

numpy.argmin
equivalent function

ndarray.**ptp**(*axis=None*, *out=None*)
Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See also:

numpy.ptp
equivalent function

ndarray.**clip**(*min=None*, *max=None*, *out=None*)
Return an array whose values are limited to `[min, max]`. One of max or min must be given.

Refer to `numpy.clip` for full documentation.

See also:

numpy.clip
equivalent function

ndarray.**conj**()
Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

See also:

numpy.conjugate
equivalent function

ndarray.**round**(*decimals=0*, *out=None*)
Return *a* with each element rounded to the given number of decimals.

Refer to `numpy.around` for full documentation.

See also:

numpy.around
equivalent function

ndarray.**trace**(*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)
Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See also:

numpy.trace
equivalent function

ndarray.**sum**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

See also:

numpy.sum
equivalent function

ndarray.**cumsum**(*axis=None*, *dtype=None*, *out=None*)
Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See also:

numpy.cumsum
equivalent function

ndarray.**mean**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

See also:

numpy.mean
equivalent function

ndarray.**var**(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)
Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

See also:

numpy.var
equivalent function

ndarray.**std**(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)
Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

See also:

numpy.std
equivalent function

ndarray.**prod**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Return the product of the array elements over the given axis

Refer to `numpy.prod` for full documentation.

See also:

numpy.prod
equivalent function

ndarray.**cumprod**(*axis=None*, *dtype=None*, *out=None*)
Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

See also:

numpy.cumprod
equivalent function

ndarray.**all**(*axis=None*, *out=None*, *keepdims=False*)
Returns True if all elements evaluate to True.

Refer to `numpy.all` for full documentation.

See also:

numpy.all
equivalent function

ndarray.**any**(*axis=None*, *out=None*, *keepdims=False*)
Returns True if any of the elements of *a* evaluate to True.

Refer to `numpy.any` for full documentation.

See also:

numpy.any
equivalent function

1.1.6 Arithmetic, matrix multiplication, and comparison operations

Arithmetic and comparison operations on *ndarrays* are defined as element-wise operations, and generally yield *ndarray* objects as results.

Each of the arithmetic operations (`+`, `-`, `*`, `/`, `//`, `%`, `divmod()`, `**` or `pow()`, `<<`, `>>`, `&`, `^`, `|`, `~`) and the comparisons (`==`, `<`, `>`, `<=`, `>=`, `!=`) is equivalent to the corresponding universal function (or ufunc for short) in Numpy. For more information, see the section on *Universal Functions*.

Comparison operators:

ndarray.__lt__	x.__lt__(y) <==> x<y
ndarray.__le__	x.__le__(y) <==> x<=y
ndarray.__gt__	x.__gt__(y) <==> x>y
ndarray.__ge__	x.__ge__(y) <==> x>=y
ndarray.__eq__	x.__eq__(y) <==> x==y
ndarray.__ne__	x.__ne__(y) <==> x!=y

ndarray.**__lt__**
x.__lt__(y) <==> x<y

ndarray.**__le__**
x.__le__(y) <==> x<=y

ndarray.**__gt__**
x.__gt__(y) <==> x>y

ndarray.**__ge__**
x.__ge__(y) <==> x>=y

ndarray.**__eq__**
x.__eq__(y) <==> x==y

ndarray.**__ne__**
x.__ne__(y) <==> x!=y

Truth value of an array (`bool`):

ndarray.__nonzero__	x.__nonzero__() <==> x != 0

ndarray.**__nonzero__**
x.__nonzero__() <==> x != 0

Note: Truth-value testing of an array invokes *ndarray.__nonzero__*, which raises an error if the number of elements in the the array is larger than 1, because the truth value of such arrays is ambiguous. Use *.any()* and *.all()* instead to be clear about what is meant in such cases. (If the number of elements is 0, the array evaluates to `False`.)

Unary operations:

ndarray.__neg__	x.__neg__() <==> -x
ndarray.__pos__	x.__pos__() <==> +x
ndarray.__abs__() <==> abs(x)	
ndarray.__invert__	x.__invert__() <==> ~x

`ndarray.__neg__`
x.__neg__() <==> -x

`ndarray.__pos__`
x.__pos__() <==> +x

`ndarray.__abs__`*()* <==> *abs(x)*

`ndarray.__invert__`
x.__invert__() <==> ~x

Arithmetic:

ndarray.__add__	x.__add__(y) <==> x+y
ndarray.__sub__	x.__sub__(y) <==> x-y
ndarray.__mul__	x.__mul__(y) <==> x*y
ndarray.__div__	x.__div__(y) <==> x/y
ndarray.__truediv__	x.__truediv__(y) <==> x/y
ndarray.__floordiv__	x.__floordiv__(y) <==> x//y
ndarray.__mod__	x.__mod__(y) <==> x%y
ndarray.__divmod__(y) <==> divmod(x, y)	
ndarray.__pow__(y[, z]) <==> pow(x, y[, z])	
ndarray.__lshift__	x.__lshift__(y) <==> x<<y
ndarray.__rshift__	x.__rshift__(y) <==> x>>y
ndarray.__and__	x.__and__(y) <==> x&y
ndarray.__or__	x.__or__(y) <==> x|y
ndarray.__xor__	x.__xor__(y) <==> x^y

`ndarray.__add__`
x.__add__(y) <==> x+y

`ndarray.__sub__`
x.__sub__(y) <==> x-y

`ndarray.__mul__`
x.__mul__(y) <==> x*y

`ndarray.__div__`
x.__div__(y) <==> x/y

`ndarray.__truediv__`
x.__truediv__(y) <==> x/y

ndarray.**__floordiv__**
x.__floordiv__(y) <==> x//y

ndarray.**__mod__**
x.__mod__(y) <==> x%y

ndarray.**__divmod__**(*y*) <==> *divmod(x, y)*

ndarray.**__pow__**(*y[, z]*) <==> *pow(x, y[, z])*

ndarray.**__lshift__**
x.__lshift__(y) <==> x<<y

ndarray.**__rshift__**
x.__rshift__(y) <==> x>>y

ndarray.**__and__**
x.__and__(y) <==> x&y

ndarray.**__or__**
x.__or__(y) <==> x|y

ndarray.**__xor__**
x.__xor__(y) <==> x^y

Note:

- Any third argument to `pow` is silently ignored, as the underlying *`ufunc`* takes only two arguments.
- The three division operators are all defined; `div` is active by default, `truediv` is active when `__future__` division is in effect.
- Because *`ndarray`* is a built-in type (written in C), the `__r{op}__` special methods are not directly defined.
- The functions called to implement many arithmetic special methods for arrays can be modified using `set_numeric_ops`.

Arithmetic, in-place:

ndarray.__iadd__	x.__iadd__(y) <==> x+=y
ndarray.__isub__	x.__isub__(y) <==> x-=y
ndarray.__imul__	x.__imul__(y) <==> x*=y
ndarray.__idiv__	x.__idiv__(y) <==> x/=y
ndarray.__itruediv__	x.__itruediv__(y) <==> x/=y
ndarray.__ifloordiv__	x.__ifloordiv__(y) <==> x//=y
ndarray.__imod__	x.__imod__(y) <==> x%=y
ndarray.__ipow__	x.__ipow__(y) <==> x**=y
ndarray.__ilshift__	x.__ilshift__(y) <==> x<<=y
ndarray.__irshift__	x.__irshift__(y) <==> x>>=y
ndarray.__iand__	x.__iand__(y) <==> x&=y
ndarray.__ior__	x.__ior__(y) <==> x|=y
ndarray.__ixor__	x.__ixor__(y) <==> x^=y

ndarray.**__iadd__**
x.__iadd__(y) <==> x+=y

ndarray.**__isub__**

x.__isub__(y) <==> x-=y

ndarray.__**imul**__
x.__imul__(y) <==> x*=y

ndarray.__**idiv**__
x.__idiv__(y) <==> x/=y

ndarray.__**itruediv**__
x.__itruediv__(y) <==> x/=y

ndarray.__**ifloordiv**__
x.__ifloordiv__(y) <==> x//=y

ndarray.__**imod**__
x.__imod__(y) <==> x%=y

ndarray.__**ipow**__
x.__ipow__(y) <==> x**=y

ndarray.__**ilshift**__
x.__ilshift__(y) <==> x<<=y

ndarray.__**irshift**__
x.__irshift__(y) <==> x>>=y

ndarray.__**iand**__
x.__iand__(y) <==> x&=y

ndarray.__**ior**__
x.__ior__(y) <==> x|=y

ndarray.__**ixor**__
x.__ixor__(y) <==> x^=y

Warning: In place operations will perform the calculation using the precision decided by the data type of the two operands, but will silently downcast the result (if necessary) so it can fit back into the array. Therefore, for mixed precision calculations, `A {op}= B` can be different than `A = A {op} B`. For example, suppose `a = ones((3,3))`. Then, `a += 3j` is different than `a = a + 3j`: while they both perform the same computation, `a += 3` casts the result to fit back in `a`, whereas `a = a + 3j` re-binds the name `a` to the result.

Matrix Multiplication:

ndarray.__matmul__

Note: Matrix operators `@` and `@=` were introduced in Python 3.5 following PEP465. Numpy 1.10 has a preliminary implementation of `@` for testing purposes. Further documentation can be found in the `matmul` documentation.

1.1.7 Special methods

For standard library functions:

ndarray.__copy__([order])	Return a copy of the array.
ndarray.__deepcopy__(() -> Deep copy of array.)	Used if copy.deepcopy is called on an array.
ndarray.__reduce__()	For pickling.
ndarray.__setstate__(version, shape, dtype, ...)	For unpickling.

ndarray.**__copy__**([*order*])
Return a copy of the array.

Parameters
order : {'C', 'F', 'A'}, optional

If order is 'C' (False) then the result is contiguous (default). If order is 'Fortran' (True) then the result has fortran order. If order is 'Any' (None) then the result has fortran order only if the array already is in fortran order.

ndarray.**__deepcopy__**() → Deep copy of array.
Used if copy.deepcopy is called on an array.

ndarray.**__reduce__**()
For pickling.

ndarray.**__setstate__**(*version*, *shape*, *dtype*, *isfortran*, *rawdata*)
For unpickling.

Parameters
version : int

optional pickle version. If omitted defaults to 0.

shape : tuple

dtype : data-type

isFortran : bool

rawdata : string or list

a binary string with the data (or a list if 'a' is an object array)

Basic customization:

ndarray.__new__((S, ...)	
ndarray.__array__(...)	Returns either a new reference to self if dtype is not given or a new array of provided data type
ndarray.__array_wrap__(...)	

ndarray.**__new__**(*S*, ...) → a new object with type S, a subtype of T

ndarray.**__array__**(|*dtype*) → reference if type unchanged, copy otherwise.
Returns either a new reference to self if dtype is not given or a new array of provided data type if dtype is different from the current dtype of the array.

ndarray.**__array_wrap__**(*obj*) → Object of same type as ndarray object a.

Container customization: (see *Indexing*)

ndarray.__len__() <==> len(x)	
ndarray.__getitem__	x.__getitem__(y) <==> x[y]
ndarray.__setitem__	x.__setitem__(i, y) <==> x[i]=y
ndarray.__getslice__	x.__getslice__(i, j) <==> x[i:j]
ndarray.__setslice__	x.__setslice__(i, j, y) <==> x[i:j]=y
ndarray.__contains__	x.__contains__(y) <==> y in x

ndarray.**__len__**() <==> *len(x)*

ndarray.__**getitem**__
x.__getitem__(y) <==> x[y]

ndarray.__**setitem**__
x.__setitem__(i, y) <==> x[i]=y

ndarray.__**getslice**__
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

ndarray.__**setslice**__
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

ndarray.__**contains**__
x.__contains__(y) <==> y in x

Conversion; the operations `complex`, `int`, `long`, `float`, `oct`, and `hex`. They work only on arrays that have one element in them and return the appropriate scalar.

ndarray.__int__() <==> int(x)
ndarray.__long__() <==> long(x)
ndarray.__float__() <==> float(x)
ndarray.__oct__() <==> oct(x)
ndarray.__hex__() <==> hex(x)

ndarray.__**int**__() <==> *int(x)*

ndarray.__**long**__() <==> *long(x)*

ndarray.__**float**__() <==> *float(x)*

ndarray.__**oct**__() <==> *oct(x)*

ndarray.__**hex**__() <==> *hex(x)*

String representations:

ndarray.__str__() <==> str(x)
ndarray.__repr__() <==> repr(x)

ndarray.__**str**__() <==> *str(x)*

ndarray.__**repr**__() <==> *repr(x)*

1.2 Scalars

Python defines only one type of a particular data class (there is only one integer type, one floating-point type, etc.). This can be convenient in applications that don't need to be concerned with all the ways data can be represented in a computer. For scientific computing, however, more control is often needed.

In NumPy, there are 24 new fundamental Python types to describe different types of scalars. These type descriptors are mostly based on the types available in the C language that CPython is written in, with several additional types compatible with Python's types.

Array scalars have the same attributes and methods as *ndarrays*. [1] This allows one to treat items of an array partly on the same footing as arrays, smoothing out rough edges that result when mixing scalar and array operations.

Array scalars live in a hierarchy (see the Figure below) of data types. They can be detected using the hierarchy: For example, `isinstance(val, np.generic)` will return `True` if *val* is an array scalar object. Alternatively, what kind of array scalar is present can be determined using other members of the data type hierarchy. Thus, for example `isinstance(val, np.complexfloating)` will return `True` if *val* is a complex valued type, while `isinstance(val, np.flexible)` will return true if *val* is one of the flexible itemsize array types (`string`, `unicode`, `void`).

1.2.1 Built-in scalar types

The built-in scalar types are shown below. Along with their (mostly) C-derived names, the integer, float, and complex data-types are also available using a bit-width convention so that an array of the right size can always be ensured (e.g. `int8`, `float64`, `complex128`). Two aliases (`intp` and `uintp`) pointing to the integer type that is sufficiently large to hold a C pointer are also provided. The C-like names are associated with character codes, which are shown in the table. Use of the character codes, however, is discouraged.

Some of the scalar types are essentially equivalent to fundamental Python types and therefore inherit from them as well as from the generic array scalar type:

Array scalar type	Related Python type
`int_`	`IntType` (Python 2 only)
`float_`	`FloatType`
`complex_`	`ComplexType`
`str_`	`StringType`
`unicode_`	`UnicodeType`

The `bool_` data type is very similar to the Python `BooleanType` but does not inherit from it because Python's `BooleanType` does not allow itself to be inherited from, and on the C-level the size of the actual bool data is not the same as a Python Boolean scalar.

Warning: The `bool_` type is not a subclass of the `int_` type (the `bool_` is not even a number type). This is different than Python's default implementation of `bool` as a sub-class of int.

Warning: The `int_` type does **not** inherit from the `int` built-in under Python 3, because type `int` is no longer a fixed-width integer type.

Tip: The default data type in Numpy is `float_`.

[1] However, array scalars are immutable, so none of the array scalar attributes are settable.

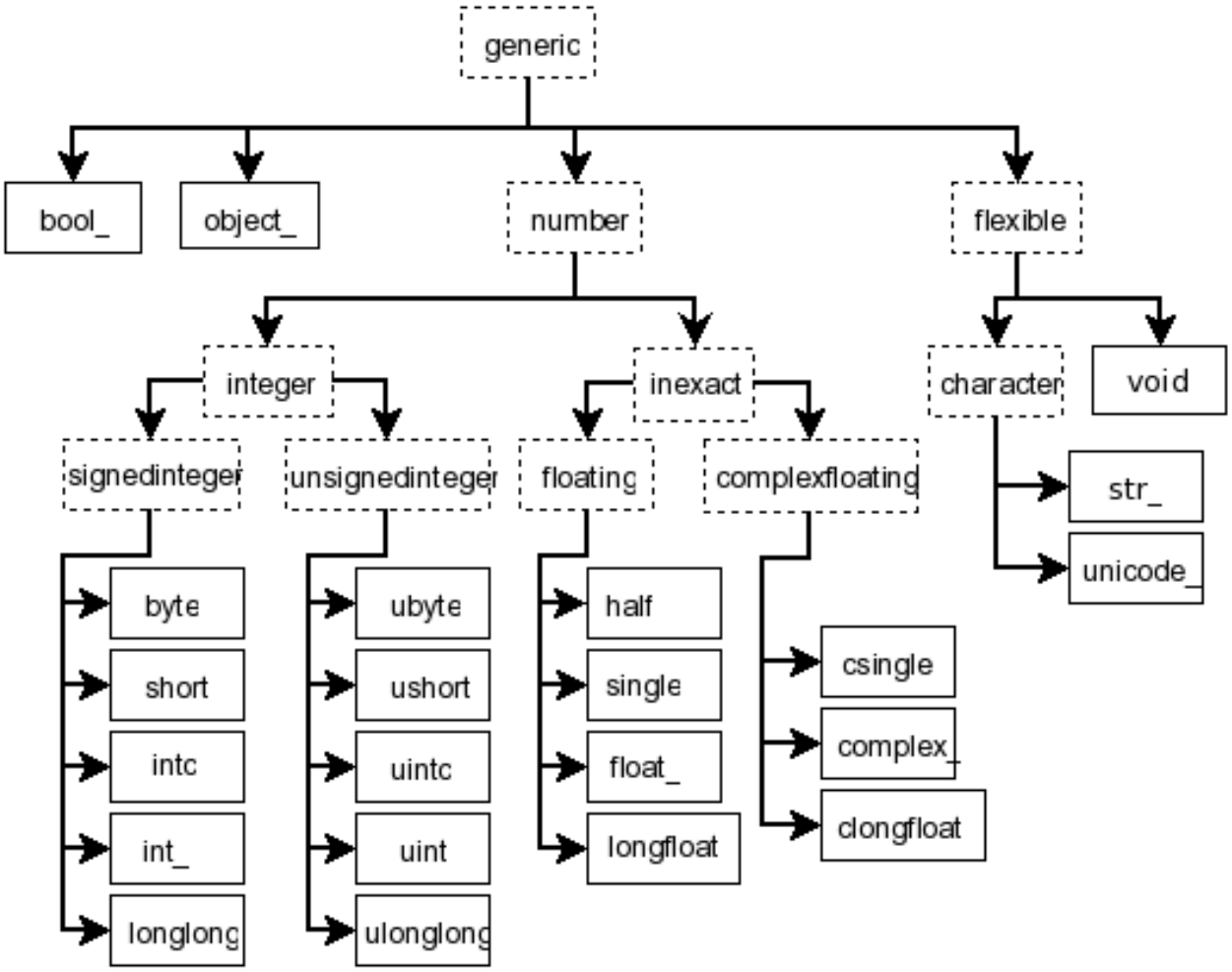


Fig. 1.2: **Figure:** Hierarchy of type objects representing the array data types. Not shown are the two integer types `intp` and `uintp` which just point to the integer type that holds a pointer for the platform. All the number types can be obtained using bit-width names as well.

In the tables below, `platform?` means that the type may not be available on all platforms. Compatibility with different C or Python types is indicated: two types are compatible if their data is of the same size and interpreted in the same way.

Booleans:

Type	Remarks	Character code
`bool_`	compatible: Python bool	`'?'`
`bool8`	8 bits	

Integers:

`byte`	compatible: C char	`'b'`
`short`	compatible: C short	`'h'`
`intc`	compatible: C int	`'i'`
`int_`	compatible: Python int	`'l'`
`longlong`	compatible: C long long	`'q'`
`intp`	large enough to fit a pointer	`'p'`
`int8`	8 bits	
`int16`	16 bits	
`int32`	32 bits	
`int64`	64 bits	

Unsigned integers:

`ubyte`	compatible: C unsigned char	`'B'`
`ushort`	compatible: C unsigned short	`'H'`
`uintc`	compatible: C unsigned int	`'I'`
`uint`	compatible: Python int	`'L'`
`ulonglong`	compatible: C long long	`'Q'`
`uintp`	large enough to fit a pointer	`'P'`
`uint8`	8 bits	
`uint16`	16 bits	
`uint32`	32 bits	
`uint64`	64 bits	

Floating-point numbers:

`half`		`'e'`
`single`	compatible: C float	`'f'`
`double`	compatible: C double	
`float_`	compatible: Python float	`'d'`
`longfloat`	compatible: C long float	`'g'`
`float16`	16 bits	
`float32`	32 bits	
`float64`	64 bits	
`float96`	96 bits, platform?	
`float128`	128 bits, platform?	

Complex floating-point numbers:

`csingle`		`'F'`
`complex_`	compatible: Python complex	`'D'`
`clongfloat`		`'G'`
`complex64`	two 32-bit floats	
`complex128`	two 64-bit floats	
`complex192`	two 96-bit floats, platform?	
`complex256`	two 128-bit floats, platform?	

Any Python object:

object_	any Python object	'O'

Note: The data actually stored in object arrays (*i.e.*, arrays having dtype object_) are references to Python objects, not the objects themselves. Hence, object arrays behave more like usual Python lists, in the sense that their contents need not be of the same Python type.

The object type is also special because an array containing object_ items does not return an object_ object on item access, but instead returns the actual object that the array item refers to.

The following data types are flexible. They have no predefined size: the data they describe can be of different length in different arrays. (In the character codes # is an integer denoting how many elements the data type consists of.)

str_	compatible: Python str	'S#'
unicode_	compatible: Python unicode	'U#'
void		'V#'

Warning: Numeric Compatibility: If you used old typecode characters in your Numeric code (which was never recommended), you will need to change some of them to the new characters. In particular, the needed changes are c -> S1, b -> B, 1 -> b, s -> h, w -> H, and u -> I. These changes make the type character convention more consistent with other Python modules such as the struct module.

1.2.2 Attributes

The array scalar objects have an array priority of *NPY_SCALAR_PRIORITY* (-1,000,000.0). They also do not (yet) have a *ctypes* attribute. Otherwise, they share the same attributes as arrays:

generic.flags	integer value of flags
generic.shape	tuple of array dimensions
generic.strides	tuple of bytes steps in each dimension
generic.ndim	number of array dimensions
generic.data	pointer to start of data
generic.size	number of elements in the gentype
generic.itemsize	length of one element in bytes
generic.base	base object
generic.dtype	get array data-descriptor
generic.real	real part of scalar
generic.imag	imaginary part of scalar
generic.flat	a 1-d view of scalar
generic.T	transpose
generic.__array_interface__	Array protocol: Python side
generic.__array_struct__	Array protocol: struct
generic.__array_priority__	Array priority.
generic.__array_wrap__	sc.__array_wrap__(obj) return scalar from array

generic.**flags**
: integer value of flags

generic.**shape**
: tuple of array dimensions

generic.**strides**

tuple of bytes steps in each dimension

`generic.`**`ndim`**
number of array dimensions

`generic.`**`data`**
pointer to start of data

`generic.`**`size`**
number of elements in the gentype

`generic.`**`itemsize`**
length of one element in bytes

`generic.`**`base`**
base object

`generic.`**`dtype`**
get array data-descriptor

`generic.`**`real`**
real part of scalar

`generic.`**`imag`**
imaginary part of scalar

`generic.`**`flat`**
a 1-d view of scalar

`generic.`**`T`**
transpose

`generic.`**`__array_interface__`**
Array protocol: Python side

`generic.`**`__array_struct__`**
Array protocol: struct

`generic.`**`__array_priority__`**
Array priority.

`generic.`**`__array_wrap__`**`()`
sc.__array_wrap__(obj) return scalar from array

1.2.3 Indexing

See also:

Indexing, *Data type objects (dtype)*

Array scalars can be indexed like 0-dimensional arrays: if *x* is an array scalar,

- `x[()]` returns a 0-dimensional *ndarray*
- `x['field-name']` returns the array scalar in the field *field-name*. (*x* can have fields, for example, when it corresponds to a structured data type.)

1.2.4 Methods

Array scalars have exactly the same methods as arrays. The default behavior of these methods is to internally convert the scalar to an equivalent 0-dimensional array and to call the corresponding array method. In addition, math operations

on array scalars are defined so that the same hardware flags are set and used to interpret the results as for *ufunc*, so that the error state used for ufuncs also carries over to the math on array scalars.

The exceptions to the above rules are given below:

generic	Base class for numpy scalar types.
generic.__array__	sc.__array__(ltype) return 0-dim array
generic.__array_wrap__	sc.__array_wrap__(obj) return scalar from array
generic.squeeze	Not implemented (virtual attribute)
generic.byteswap	Not implemented (virtual attribute)
generic.__reduce__	
generic.__setstate__	
generic.setflags	Not implemented (virtual attribute)

class numpy.**generic**

Base class for numpy scalar types.

Class from which most (all?) numpy scalar types are derived. For consistency, exposes the same API as *ndarray*, despite many consequent attributes being either "get-only," or completely irrelevant. This is the class from which it is strongly suggested users should derive custom scalar types.

Attributes

T	transpose
base	base object
data	pointer to start of data
dtype	get array data-descriptor
flags	integer value of flags
flat	a 1-d view of scalar
imag	imaginary part of scalar
itemsize	length of one element in bytes
nbytes	length of item in bytes
ndim	number of array dimensions
real	real part of scalar
shape	tuple of array dimensions
size	number of elements in the gentype
strides	tuple of bytes steps in each dimension

generic.**T**
: transpose

generic.**base**
: base object

generic.**data**
: pointer to start of data

generic.**dtype**
: get array data-descriptor

generic.**flags**
: integer value of flags

generic.**flat**
: a 1-d view of scalar

generic.**imag**
: imaginary part of scalar

generic.**itemsize**
: length of one element in bytes

generic.**nbytes**
: length of item in bytes

generic.**ndim**
: number of array dimensions

generic.**real**
: real part of scalar

generic.**shape**
: tuple of array dimensions

generic.**size**
: number of elements in the gentype

generic.**strides**
: tuple of bytes steps in each dimension

Methods

all	Not implemented (virtual attribute)
any	Not implemented (virtual attribute)
argmax	Not implemented (virtual attribute)
argmin	Not implemented (virtual attribute)
argsort	Not implemented (virtual attribute)
astype	Not implemented (virtual attribute)
byteswap	Not implemented (virtual attribute)
choose	Not implemented (virtual attribute)
clip	Not implemented (virtual attribute)
compress	Not implemented (virtual attribute)
conj	
conjugate	Not implemented (virtual attribute)
copy	Not implemented (virtual attribute)
cumprod	Not implemented (virtual attribute)
cumsum	Not implemented (virtual attribute)
diagonal	Not implemented (virtual attribute)
dump	Not implemented (virtual attribute)
dumps	Not implemented (virtual attribute)
fill	Not implemented (virtual attribute)
flatten	Not implemented (virtual attribute)
getfield	Not implemented (virtual attribute)
item	Not implemented (virtual attribute)
itemset	Not implemented (virtual attribute)
max	Not implemented (virtual attribute)
mean	Not implemented (virtual attribute)
min	Not implemented (virtual attribute)
newbyteorder([new_order])	Return a new *dtype* with a different byte order.
nonzero	Not implemented (virtual attribute)
prod	Not implemented (virtual attribute)

Continued on next page

Table 1.26 – continued from previous page

ptp	Not implemented (virtual attribute)
put	Not implemented (virtual attribute)
ravel	Not implemented (virtual attribute)
repeat	Not implemented (virtual attribute)
reshape	Not implemented (virtual attribute)
resize	Not implemented (virtual attribute)
round	Not implemented (virtual attribute)
searchsorted	Not implemented (virtual attribute)
setfield	Not implemented (virtual attribute)
setflags	Not implemented (virtual attribute)
sort	Not implemented (virtual attribute)
squeeze	Not implemented (virtual attribute)
std	Not implemented (virtual attribute)
sum	Not implemented (virtual attribute)
swapaxes	Not implemented (virtual attribute)
take	Not implemented (virtual attribute)
tobytes	
tofile	Not implemented (virtual attribute)
tolist	Not implemented (virtual attribute)
tostring	Not implemented (virtual attribute)
trace	Not implemented (virtual attribute)
transpose	Not implemented (virtual attribute)
var	Not implemented (virtual attribute)
view	Not implemented (virtual attribute)

generic.**all**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**any**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**argmax**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**argmin**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**argsort**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**astype**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**byteswap**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**choose**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**clip**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**compress**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**conj**()

generic.**conjugate**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**copy**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**cumprod**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**cumsum**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**diagonal**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**dump**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**dumps**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**fill**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**flatten**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**getfield**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**item**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**itemset**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**max**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**mean**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**min**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**newbyteorder**(*new_order='S'*)
Return a new *dtype* with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The *new_order* code can be any from the following:

- 'S' - swap dtype from current to opposite endian
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

Parameters
new_order : str, optional

Byte order to force; a value from the byte order specifications above. The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns
new_dtype : dtype

New *dtype* object with the given change to the byte order.

generic.**nonzero**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**prod**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**ptp**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**put**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**ravel**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**repeat**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**reshape**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**resize**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**round**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**searchsorted**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**setfield**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**setflags**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**sort**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**squeeze**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**std**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**sum**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**swapaxes**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**take**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**tobytes**()

generic.**tofile**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**tolist**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**tostring**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**trace**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**transpose**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**var**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**view**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**__array__**()
sc.__array__(|type) return 0-dim array

generic.**__array_wrap__**()
sc.__array_wrap__(obj) return scalar from array

generic.**squeeze**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**byteswap**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

generic.**__reduce__**()

generic.__setstate__()

generic.setflags()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

1.2.5 Defining new types

There are two ways to effectively define a new array scalar type (apart from composing structured types *dtypes* from the built-in scalar types): One way is to simply subclass the *ndarray* and overwrite the methods of interest. This will work to a degree, but internally certain behaviors are fixed by the data type of the array. To fully customize the data type of an array you need to define a new data-type, and register it with NumPy. Such new types can only be defined in C, using the *Numpy C-API*.

1.3 Data type objects (dtype)

A data type object (an instance of *numpy.dtype* class) describes how the bytes in the fixed-size block of memory corresponding to an array item should be interpreted. It describes the following aspects of the data:

1. Type of the data (integer, float, Python object, etc.)
2. Size of the data (how many bytes is in *e.g.* the integer)
3. Byte order of the data (little-endian or big-endian)
4. If the data type is structured, an aggregate of other data types, (*e.g.*, describing an array item consisting of an integer and a float),
   (a) what are the names of the "fields" of the structure, by which they can be *accessed*,
   (b) what is the data-type of each field, and
   (c) which part of the memory block each field takes.
5. If the data type is a sub-array, what is its shape and data type.

To describe the type of scalar data, there are several *built-in scalar types* in Numpy for various precision of integers, floating-point numbers, *etc*. An item extracted from an array, *e.g.*, by indexing, will be a Python object whose type is the scalar type associated with the data type of the array.

Note that the scalar types are not *dtype* objects, even though they can be used in place of one whenever a data type specification is needed in Numpy.

Structured data types are formed by creating a data type whose fields contain other data types. Each field has a name by which it can be *accessed*. The parent data type should be of sufficient size to contain all its fields; the parent is nearly always based on the void type which allows an arbitrary item size. Structured data types may also contain nested structured sub-array data types in their fields.

Finally, a data type can describe items that are themselves arrays of items of another data type. These sub-arrays must, however, be of a fixed size.

If an array is created using a data-type describing a sub-array, the dimensions of the sub-array are appended to the shape of the array when the array is created. Sub-arrays in a field of a structured type behave differently, see *Field Access*.

Sub-arrays always have a C-contiguous memory layout.

Example

A simple data type containing a 32-bit big-endian integer: (see *Specifying and constructing data types* for details on construction)

```
>>> dt = np.dtype('>i4')
>>> dt.byteorder
'>'
>>> dt.itemsize
4
>>> dt.name
'int32'
>>> dt.type is np.int32
True
```

The corresponding array scalar type is `int32`.

Example

A structured data type containing a 16-character string (in field 'name') and a sub-array of two 64-bit floating-point number (in field 'grades'):

```
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> dt['name']
dtype('|S16')
>>> dt['grades']
dtype(('float64',(2,)))
```

Items of an array of this data type are wrapped in an *array scalar* type that also has two fields:

```
>>> x = np.array([('Sarah', (8.0, 7.0)), ('John', (6.0, 7.0))], dtype=dt)
>>> x[1]
('John', [6.0, 7.0])
>>> x[1]['grades']
array([ 6.,  7.])
>>> type(x[1])
<type 'numpy.void'>
>>> type(x[1]['grades'])
<type 'numpy.ndarray'>
```

1.3.1 Specifying and constructing data types

Whenever a data-type is required in a NumPy function or method, either a *dtype* object or something that can be converted to one can be supplied. Such conversions are done by the *dtype* constructor:

dtype	Create a data type object.

class numpy.**dtype**

Create a data type object.

A numpy array is homogeneous, and contains elements described by a dtype object. A dtype object can be constructed from different combinations of fundamental numeric types.

Parameters

obj

Object to be converted to a data type object.

align : bool, optional

Add padding to the fields to match what a C compiler would output for a similar C-struct. Can be `True` only if *obj* is a dictionary or a comma-separated string. If a struct dtype is being created, this also sets a sticky alignment flag `isalignedstruct`.

copy : bool, optional

Make a new copy of the data-type object. If `False`, the result may just be a reference to a built-in data-type object.

See also:

`result_type`

Examples

Using array-scalar type:

```
>>> np.dtype(np.int16)
dtype('int16')
```

Structured type, one field name 'f1', containing int16:

```
>>> np.dtype([('f1', np.int16)])
dtype([('f1', '<i2')])
```

Structured type, one field named 'f1', in itself containing a structured type with one field:

```
>>> np.dtype([('f1', [('f1', np.int16)])])
dtype([('f1', [('f1', '<i2')])])
```

Structured type, two fields: the first field contains an unsigned int, the second an int32:

```
>>> np.dtype([('f1', np.uint), ('f2', np.int32)])
dtype([('f1', '<u4'), ('f2', '<i4')])
```

Using array-protocol type strings:

```
>>> np.dtype([('a','f8'),('b','S10')])
dtype([('a', '<f8'), ('b', '|S10')])
```

Using comma-separated field formats. The shape is (2,3):

```
>>> np.dtype("i4, (2,3)f8")
dtype([('f0', '<i4'), ('f1', '<f8', (2, 3))])
```

Using tuples. `int` is a fixed type, 3 the field's shape. `void` is a flexible type, here of size 10:

```
>>> np.dtype([('hello',(np.int,3)),('world',np.void,10)])
dtype([('hello', '<i4', 3), ('world', '|V10')])
```

Subdivide `int16` into 2 `int8`'s, called x and y. 0 and 1 are the offsets in bytes:

```
>>> np.dtype((np.int16, {'x':(np.int8,0), 'y':(np.int8,1)}))
dtype(('<i2', [('x', '|i1'), ('y', '|i1')]))
```

Using dictionaries. Two fields named 'gender' and 'age':

```
>>> np.dtype({'names':['gender','age'], 'formats':['S1',np.uint8]})
dtype([('gender', '|S1'), ('age', '|u1')])
```

Offsets in bytes, here 0 and 25:

```
>>> np.dtype({'surname':('S25',0),'age':(np.uint8,25)})
dtype([('surname', '|S25'), ('age', '|u1')])
```

Attributes

base	
descr	Array-interface compliant full description of the data-type.
fields	Dictionary of named fields defined for this data type, or None.
hasobject	Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.
isalignedstruct	Boolean indicating whether the dtype is a struct which maintains field alignment.
isbuiltin	Integer indicating how this dtype relates to the built-in dtypes.
isnative	Boolean indicating whether the byte order of this dtype is native to the platform.
metadata	
name	A bit-width name for this data-type.
names	Ordered list of field names, or None if there are no fields.
shape	Shape tuple of the sub-array if this data type describes a sub-array, and () otherwise.
str	The array-protocol typestring of this data-type object.
subdtype	Tuple (item_dtype, shape) if this *dtype* describes a sub-array, and None otherwise.

dtype.**base**

dtype.**descr**

Array-interface compliant full description of the data-type.

The format is that required by the 'descr' key in the *__array_interface__* attribute.

dtype.**fields**

Dictionary of named fields defined for this data type, or None.

The dictionary is indexed by keys that are the names of the fields. Each entry in the dictionary is a tuple fully describing the field:

```
(dtype, offset[, title])
```

If present, the optional title can be any object (if it is a string or unicode then it will also be a key in the fields dictionary, otherwise it's meta-data). Notice also that the first two elements of the tuple can be passed directly as arguments to the ndarray.getfield and ndarray.setfield methods.

See also:

ndarray.getfield, *ndarray.setfield*

Examples

```
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> print(dt.fields)
{'grades': (dtype(('float64',(2,))), 16), 'name': (dtype('|S16'), 0)}
```

dtype.**hasobject**
Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.

Recall that what is actually in the ndarray memory representing the Python object is the memory address of that object (a pointer). Special handling may be required, and this attribute is useful for distinguishing data types that may contain arbitrary Python objects and data-types that won't.

dtype.**isalignedstruct**
Boolean indicating whether the dtype is a struct which maintains field alignment. This flag is sticky, so when combining multiple structs together, it is preserved and produces new dtypes which are also aligned.

dtype.**isbuiltin**
Integer indicating how this dtype relates to the built-in dtypes.

Read-only.

0	if this is a structured array type, with fields
1	if this is a dtype compiled into numpy (such as ints, floats etc)
2	if the dtype is for a user-defined numpy type A user-defined type uses the numpy C-API machinery to extend numpy to handle a new array type. See user.user-defined-data-types in the Numpy manual.

Examples

```
>>> dt = np.dtype('i2')
>>> dt.isbuiltin
1
>>> dt = np.dtype('f8')
>>> dt.isbuiltin
1
>>> dt = np.dtype([('field1', 'f8')])
>>> dt.isbuiltin
0
```

dtype.**isnative**
Boolean indicating whether the byte order of this dtype is native to the platform.

dtype.**metadata**

dtype.**name**
A bit-width name for this data-type.

Un-sized flexible data-type objects do not have this attribute.

dtype.**names**
Ordered list of field names, or `None` if there are no fields.

The names are ordered according to increasing byte offset. This can be used, for example, to walk through all of the named fields in offset order.

Examples

```
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> dt.names
('name', 'grades')
```

dtype.**shape**
Shape tuple of the sub-array if this data type describes a sub-array, and `()` otherwise.

dtype.**str**
The array-protocol typestring of this data-type object.

dtype.**subdtype**
Tuple `(item_dtype, shape)` if this *dtype* describes a sub-array, and None otherwise.

The *shape* is the fixed shape of the sub-array described by this data type, and *item_dtype* the data type of the array.

If a field whose dtype object has this attribute is retrieved, then the extra dimensions implied by *shape* are tacked on to the end of the retrieved array.

Methods

newbyteorder([new_order])	Return a new dtype with a different byte order.

dtype.**newbyteorder**(*new_order='S'*)
Return a new dtype with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

Parameters
new_order : string, optional

Byte order to force; a value from the byte order specifications below. The default value ('S') results in swapping the current byte order. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

The code does a case-insensitive check on the first letter of *new_order* for these alternatives. For example, any of '>' or 'B' or 'b' or 'brian' are valid to specify big-endian.

Returns
new_dtype : dtype

New dtype object with the given change to the byte order.

Notes

Changes are also made in all fields and sub-arrays of the data type.

Examples

```
>>> import sys
>>> sys_is_le = sys.byteorder == 'little'
>>> native_code = sys_is_le and '<' or '>'
>>> swapped_code = sys_is_le and '>' or '<'
>>> native_dt = np.dtype(native_code+'i2')
>>> swapped_dt = np.dtype(swapped_code+'i2')
>>> native_dt.newbyteorder('S') == swapped_dt
True
>>> native_dt.newbyteorder() == swapped_dt
True
>>> native_dt == swapped_dt.newbyteorder('S')
True
>>> native_dt == swapped_dt.newbyteorder('=')
True
```

```
>>> native_dt == swapped_dt.newbyteorder('N')
True
>>> native_dt == native_dt.newbyteorder('|')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('<')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('L')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('>')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('B')
True
```

What can be converted to a data-type object is described below:

dtype object

Used as-is.

None

The default data type: `float_`.

Array-scalar types

The 24 built-in *array scalar type objects* all convert to an associated data-type object. This is true for their sub-classes as well.

Note that not all data-type information can be supplied with a type-object: for example, flexible data-types have a default *itemsize* of 0, and require an explicitly given size to be useful.

Example

```
>>> dt = np.dtype(np.int32)      # 32-bit integer
>>> dt = np.dtype(np.complex128) # 128-bit complex floating-point number
```

Generic types

The generic hierarchical type objects convert to corresponding type objects according to the associations:

number, inexact, floating	float
complexfloating	cfloat
integer, signedinteger	int_
unsignedinteger	uint
character	string
generic, flexible	void

Built-in Python types

Several python types are equivalent to a corresponding array scalar when used to generate a *dtype* object:

int	int_
bool	bool_
float	float_
complex	cfloat
str	string
unicode	unicode_
buffer	void
(all others)	object_

Example

```
>>> dt = np.dtype(float)    # Python-compatible floating-point number
>>> dt = np.dtype(int)      # Python-compatible integer
>>> dt = np.dtype(object)   # Python object
```

Types with `.dtype`

Any type object with a `dtype` attribute: The attribute will be accessed and used directly. The attribute must return something that is convertible into a dtype object.

Several kinds of strings can be converted. Recognized strings can be prepended with `'>'` (big-endian), `'<'` (little-endian), or `'='` (hardware-native, the default), to specify the byte order.

One-character strings

Each built-in data-type has a character code (the updated Numeric typecodes), that uniquely identifies it.

Example

```
>>> dt = np.dtype('b')  # byte, native byte order
>>> dt = np.dtype('>H') # big-endian unsigned short
>>> dt = np.dtype('<f') # little-endian single-precision float
>>> dt = np.dtype('d')  # double-precision floating-point number
```

Array-protocol type strings (see *The Array Interface*)

The first character specifies the kind of data and the remaining characters specify the number of bytes per item, except for Unicode, where it is interpreted as the number of characters. The item size must correspond to an existing type, or an error will be raised. The supported kinds are

`'b'`	boolean
`'i'`	(signed) integer
`'u'`	unsigned integer
`'f'`	floating-point
`'c'`	complex-floating point
`'m'`	timedelta
`'M'`	datetime
`'O'`	(Python) objects
`'S'`, `'a'`	(byte-)string
`'U'`	Unicode
`'V'`	raw data (`void`)

Example

```
>>> dt = np.dtype('i4')   # 32-bit signed integer
>>> dt = np.dtype('f8')   # 64-bit floating-point number
>>> dt = np.dtype('c16')  # 128-bit complex floating-point number
>>> dt = np.dtype('a25')  # 25-character string
```

String with comma-separated fields

A short-hand notation for specifying the format of a structured data type is a comma-separated string of basic formats.

A basic format in this context is an optional shape specifier followed by an array-protocol type string. Parenthesis are required on the shape if it has more than one dimension. NumPy allows a modification on the format in that any string that can uniquely identify the type can be used to specify the data-type in a field. The generated data-type fields are named `'f0'`, `'f1'`, ..., `'f<N-1>'` where N (>1) is the

number of comma-separated basic formats in the string. If the optional shape specifier is provided, then the data-type for the corresponding field describes a sub-array.

Example

- field named `f0` containing a 32-bit integer
- field named `f1` containing a 2 x 3 sub-array of 64-bit floating-point numbers
- field named `f2` containing a 32-bit floating-point number

```
>>> dt = np.dtype("i4, (2,3)f8, f4")
```

- field named `f0` containing a 3-character string
- field named `f1` containing a sub-array of shape (3,) containing 64-bit unsigned integers
- field named `f2` containing a 3 x 4 sub-array containing 10-character strings

```
>>> dt = np.dtype("a3, 3u8, (3,4)a10")
```

Type strings

Any string in `numpy.sctypeDict`.keys():

Example

```
>>> dt = np.dtype('uint32')   # 32-bit unsigned integer
>>> dt = np.dtype('Float64')  # 64-bit floating-point number
```

`(flexible_dtype, itemsize)`

The first argument must be an object that is converted to a zero-sized flexible data-type object, the second argument is an integer providing the desired itemsize.

Example

```
>>> dt = np.dtype((void, 10))   # 10-byte wide data block
>>> dt = np.dtype((str, 35))    # 35-character string
>>> dt = np.dtype(('U', 10))    # 10-character unicode string
```

`(fixed_dtype, shape)`

The first argument is any object that can be converted into a fixed-size data-type object. The second argument is the desired shape of this type. If the shape parameter is 1, then the data-type object is equivalent to fixed dtype. If *shape* is a tuple, then the new dtype defines a sub-array of the given shape.

Example

```
>>> dt = np.dtype((np.int32, (2,2)))          # 2 x 2 integer sub-array
>>> dt = np.dtype(('S10', 1))                 # 10-character string
>>> dt = np.dtype(('i4, (2,3)f8, f4', (2,3))) # 2 x 3 structured sub-array
```

`[(field_name, field_dtype, field_shape), ...]`

obj should be a list of fields where each field is described by a tuple of length 2 or 3. (Equivalent to the `descr` item in the *__array_interface__* attribute.)

The first element, *field_name*, is the field name (if this is `''` then a standard field name, `'f#'`, is assigned). The field name may also be a 2-tuple of strings where the first string is either a "title" (which

may be any string or unicode string) or meta-data for the field which can be any object, and the second string is the "name" which must be a valid Python identifier.

The second element, *field_dtype*, can be anything that can be interpreted as a data-type.

The optional third element *field_shape* contains the shape if this field represents an array of the data-type in the second element. Note that a 3-tuple with a third argument equal to 1 is equivalent to a 2-tuple.

This style does not accept *align* in the `dtype` constructor as it is assumed that all of the memory is accounted for by the array interface description.

Example

Data-type with fields `big` (big-endian 32-bit integer) and `little` (little-endian 32-bit integer):

```
>>> dt = np.dtype([('big', '>i4'), ('little', '<i4')])
```

Data-type with fields `R`, `G`, `B`, `A`, each being an unsigned 8-bit integer:

```
>>> dt = np.dtype([('R','u1'), ('G','u1'), ('B','u1'), ('A','u1')])
```

`{'names': ..., 'formats': ..., 'offsets': ..., 'titles': ..., 'itemsize': ...}`

This style has two required and three optional keys. The *names* and *formats* keys are required. Their respective values are equal-length lists with the field names and the field formats. The field names must be strings and the field formats can be any object accepted by `dtype` constructor.

When the optional keys *offsets* and *titles* are provided, their values must each be lists of the same length as the *names* and *formats* lists. The *offsets* value is a list of byte offsets (integers) for each field, while the *titles* value is a list of titles for each field (`None` can be used if no title is desired for that field). The *titles* can be any `string` or `unicode` object and will add another entry to the fields dictionary keyed by the title and referencing the same field tuple which will contain the title as an additional tuple member.

The *itemsize* key allows the total size of the dtype to be set, and must be an integer large enough so all the fields are within the dtype. If the dtype being constructed is aligned, the *itemsize* must also be divisible by the struct alignment.

Example

Data type with fields `r`, `g`, `b`, `a`, each being a 8-bit unsigned integer:

```
>>> dt = np.dtype({'names': ['r','g','b','a'],
...                'formats': [uint8, uint8, uint8, uint8]})
```

Data type with fields `r` and `b` (with the given titles), both being 8-bit unsigned integers, the first at byte position 0 from the start of the field and the second at position 2:

```
>>> dt = np.dtype({'names': ['r','b'], 'formats': ['u1', 'u1'],
...                'offsets': [0, 2],
...                'titles': ['Red pixel', 'Blue pixel']})
```

`{'field1': ..., 'field2': ..., ...}`

This usage is discouraged, because it is ambiguous with the other dict-based construction method. If you have a field called 'names' and a field called 'formats' there will be a conflict.

This style allows passing in the `fields` attribute of a data-type object.

obj should contain string or unicode keys that refer to `(data-type, offset)` or `(data-type, offset, title)` tuples.

Example

Data type containing field `col1` (10-character string at byte position 0), `col2` (32-bit float at byte position 10), and `col3` (integers at byte position 14):

```
>>> dt = np.dtype({'col1': ('S10', 0), 'col2': (float32, 10),
    'col3': (int, 14)})
```

`(base_dtype, new_dtype)`

In NumPy 1.7 and later, this form allows *base_dtype* to be interpreted as a structured dtype. Arrays created with this dtype will have underlying dtype *base_dtype* but will have fields and flags taken from *new_dtype*. This is useful for creating custom structured dtypes, as done in *record arrays*.

This form also makes it possible to specify struct dtypes with overlapping fields, functioning like the 'union' type in C. This usage is discouraged, however, and the union mechanism is preferred.

Both arguments must be convertible to data-type objects with the same total size. .. admonition:: Example

32-bit integer, whose first two bytes are interpreted as an integer via field `real`, and the following two bytes via field `imag`.

```
>>> dt = np.dtype((np.int32,{'real':(np.int16, 0),'imag':(np.int16, 2)}))
```

32-bit integer, which is interpreted as consisting of a sub-array of shape `(4,)` containing 8-bit integers:

```
>>> dt = np.dtype((np.int32, (np.int8, 4)))
```

32-bit integer, containing fields `r`, `g`, `b`, `a` that interpret the 4 bytes in the integer as four unsigned integers:

```
>>> dt = np.dtype(('i4', [('r','u1'),('g','u1'),('b','u1'),('a','u1')]))
```

1.3.2 `dtype`

Numpy data type descriptions are instances of the *dtype* class.

Attributes

The type of the data is described by the following *dtype* attributes:

dtype.type	The type object used to instantiate a scalar of this data-type.
dtype.kind	A character code (one of 'biufcmMOSUV') identifying the general kind of data.
dtype.char	A unique character code for each of the 21 different built-in types.
dtype.num	A unique number for each of the 21 different built-in types.
dtype.str	The array-protocol typestring of this data-type object.

`dtype.`**`type`**
The type object used to instantiate a scalar of this data-type.

`dtype.`**`kind`**
A character code (one of 'biufcmMOSUV') identifying the general kind of data.

b	boolean
i	signed integer
u	unsigned integer
f	floating-point
c	complex floating-point
m	timedelta
M	datetime
O	object
S	(byte-)string
U	Unicode
V	void

`dtype.`**`char`**
A unique character code for each of the 21 different built-in types.

`dtype.`**`num`**
A unique number for each of the 21 different built-in types.

These are roughly ordered from least-to-most precision.

`dtype.`**`str`**
The array-protocol typestring of this data-type object.

Size of the data is in turn described by:

dtype.name	A bit-width name for this data-type.
dtype.itemsize	The element size of this data-type object.

`dtype.`**`name`**
A bit-width name for this data-type.

Un-sized flexible data-type objects do not have this attribute.

`dtype.`**`itemsize`**
The element size of this data-type object.

For 18 of the 21 types this number is fixed by the data-type. For the flexible data-types, this number can be anything.

Endianness of this data:

dtype.byteorder	A character indicating the byte-order of this data-type object.

`dtype.`**`byteorder`**
A character indicating the byte-order of this data-type object.

One of:

'='	native
'<'	little-endian
'>'	big-endian
'I'	not applicable

All built-in data-type objects have byteorder either '=' or 'I'.

Examples

```
>>> dt = np.dtype('i2')
>>> dt.byteorder
```

```
    '='
    >>> # endian is not relevant for 8 bit numbers
    >>> np.dtype('i1').byteorder
    '|'
    >>> # or ASCII strings
    >>> np.dtype('S2').byteorder
    '|'
    >>> # Even if specific code is given, and it is native
    >>> # '=' is the byteorder
    >>> import sys
    >>> sys_is_le = sys.byteorder == 'little'
    >>> native_code = sys_is_le and '<' or '>'
    >>> swapped_code = sys_is_le and '>' or '<'
    >>> dt = np.dtype(native_code + 'i2')
    >>> dt.byteorder
    '='
    >>> # Swapped code shows up as itself
    >>> dt = np.dtype(swapped_code + 'i2')
    >>> dt.byteorder == swapped_code
    True
```

Information about sub-data-types in a structured data type:

dtype.fields	Dictionary of named fields defined for this data type, or None.
dtype.names	Ordered list of field names, or None if there are no fields.

dtype.**fields**

Dictionary of named fields defined for this data type, or None.

The dictionary is indexed by keys that are the names of the fields. Each entry in the dictionary is a tuple fully describing the field:

```
(dtype, offset[, title])
```

If present, the optional title can be any object (if it is a string or unicode then it will also be a key in the fields dictionary, otherwise it's meta-data). Notice also that the first two elements of the tuple can be passed directly as arguments to the ndarray.getfield and ndarray.setfield methods.

See also:

ndarray.getfield, *ndarray.setfield*

Examples

```
    >>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
    >>> print(dt.fields)
    {'grades': (dtype(('float64',(2,))), 16), 'name': (dtype('|S16'), 0)}
```

dtype.**names**

Ordered list of field names, or None if there are no fields.

The names are ordered according to increasing byte offset. This can be used, for example, to walk through all of the named fields in offset order.

Examples

```
    >>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
    >>> dt.names
    ('name', 'grades')
```

For data types that describe sub-arrays:

dtype.subdtype	Tuple `(item_dtype, shape)` if this *dtype* describes a sub-array, and None otherwise.
dtype.shape	Shape tuple of the sub-array if this data type describes a sub-array, and `()` otherwise.

`dtype.`**`subdtype`**
Tuple `(item_dtype, shape)` if this *dtype* describes a sub-array, and None otherwise.

The *shape* is the fixed shape of the sub-array described by this data type, and *item_dtype* the data type of the array.

If a field whose dtype object has this attribute is retrieved, then the extra dimensions implied by *shape* are tacked on to the end of the retrieved array.

`dtype.`**`shape`**
Shape tuple of the sub-array if this data type describes a sub-array, and `()` otherwise.

Attributes providing additional information:

dtype.hasobject	Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.
dtype.flags	Bit-flags describing how this data type is to be interpreted.
dtype.isbuiltin	Integer indicating how this dtype relates to the built-in dtypes.
dtype.isnative	Boolean indicating whether the byte order of this dtype is native to the platform.
dtype.descr	Array-interface compliant full description of the data-type.
dtype.alignment	The required alignment (bytes) of this data-type according to the compiler.

`dtype.`**`hasobject`**
Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.

Recall that what is actually in the ndarray memory representing the Python object is the memory address of that object (a pointer). Special handling may be required, and this attribute is useful for distinguishing data types that may contain arbitrary Python objects and data-types that won't.

`dtype.`**`flags`**
Bit-flags describing how this data type is to be interpreted.

Bit-masks are in `numpy.core.multiarray` as the constants *ITEM_HASOBJECT*, *LIST_PICKLE*, *ITEM_IS_POINTER*, *NEEDS_INIT*, *NEEDS_PYAPI*, *USE_GETITEM*, *USE_SETITEM*. A full explanation of these flags is in C-API documentation; they are largely useful for user-defined data-types.

`dtype.`**`isbuiltin`**
Integer indicating how this dtype relates to the built-in dtypes.

Read-only.

0	if this is a structured array type, with fields
1	if this is a dtype compiled into numpy (such as ints, floats etc)
2	if the dtype is for a user-defined numpy type A user-defined type uses the numpy C-API machinery to extend numpy to handle a new array type. See user.user-defined-data-types in the Numpy manual.

Examples

```
>>> dt = np.dtype('i2')
>>> dt.isbuiltin
1
>>> dt = np.dtype('f8')
>>> dt.isbuiltin
1
```

```
>>> dt = np.dtype([('field1', 'f8')])
>>> dt.isbuiltin
0
```

dtype.**isnative**
Boolean indicating whether the byte order of this dtype is native to the platform.

dtype.**descr**
Array-interface compliant full description of the data-type.

The format is that required by the 'descr' key in the *__array_interface__* attribute.

dtype.**alignment**
The required alignment (bytes) of this data-type according to the compiler.

More information is available in the C-API section of the manual.

Methods

Data types have the following method for changing the byte order:

dtype.newbyteorder([new_order])	Return a new dtype with a different byte order.

dtype.**newbyteorder**(*new_order='S'*)
Return a new dtype with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

Parameters
new_order : string, optional

Byte order to force; a value from the byte order specifications below. The default value ('S') results in swapping the current byte order. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

The code does a case-insensitive check on the first letter of *new_order* for these alternatives. For example, any of '>' or 'B' or 'b' or 'brian' are valid to specify big-endian.

Returns
new_dtype : dtype

New dtype object with the given change to the byte order.

Notes

Changes are also made in all fields and sub-arrays of the data type.

Examples

```
>>> import sys
>>> sys_is_le = sys.byteorder == 'little'
>>> native_code = sys_is_le and '<' or '>'
```

```
>>> swapped_code = sys_is_le and '>' or '<'
>>> native_dt = np.dtype(native_code+'i2')
>>> swapped_dt = np.dtype(swapped_code+'i2')
>>> native_dt.newbyteorder('S') == swapped_dt
True
>>> native_dt.newbyteorder() == swapped_dt
True
>>> native_dt == swapped_dt.newbyteorder('S')
True
>>> native_dt == swapped_dt.newbyteorder('=')
True
>>> native_dt == swapped_dt.newbyteorder('N')
True
>>> native_dt == native_dt.newbyteorder('|')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('<')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('L')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('>')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('B')
True
```

The following methods implement the pickle protocol:

dtype.__reduce__
dtype.__setstate__

dtype.**__reduce__**()

dtype.**__setstate__**()

1.4 Indexing

ndarrays can be indexed using the standard Python `x[obj]` syntax, where *x* is the array and *obj* the selection. There are three kinds of indexing available: field access, basic slicing, advanced indexing. Which one occurs depends on *obj*.

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

1.4.1 Basic Slicing and Indexing

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. In order to remain backward compatible with a common usage in Numeric, basic slicing is also initiated if the selection object is any non-ndarray sequence (such as a `list`) containing `slice` objects, the `Ellipsis` object, or the *newaxis* object, but not for integer arrays or other embedded sequences.

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.

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

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

  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.

  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 -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.

  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.

  Example

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

- `Ellipsis` expand to the number of `:` objects needed to make a selection tuple of the same length as `x.ndim`. There may only be a single ellipsis present.

  Example

```
>>> 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.

Example

```
>>> x[:,np.newaxis,:,:].shape
(2, 1, 3, 1)
```

- 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 *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*, ..., *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]`.

Note: Remember that 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 dimension.

`numpy.`**`newaxis`**
The *newaxis* object can be used in all slicing operations to create an axis of length one. :const: *newaxis* is an alias for 'None', and 'None' can be used in place of this with the same result.

1.4.2 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 `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 indexes into that dimension.

Purely integer array indexing

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

Advanced indexes 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 result shape is identical to the (broadcast) indexing array shapes `ind_1, ..., ind_N`.

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 = 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)
>>> 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, as was used in the previous example. This difference is the most important thing to remember about indexing with multiple advanced indexes.

Combining advanced and basic indexing

When there is at least one slice (`:`), ellipsis (`...`) or `np.newaxis` in the index (or the array has more dimensions than there are advanced indexes), 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. 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.

Example

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

The easiest way to understand the situation may be to think in terms of the result 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 indexes are separated by a slice, ellipsis or newaxis. For example `x[arr1, :, arr2]`.
- The advanced indexes 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`.

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`.

Example

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 straight forward. Care must only be taken to make sure that the boolean index has *exactly* as many dimensions as it is supposed to work with.

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]])
```

But if `rowsum` would have two dimensions as well:

```
>>> rowsum = x.sum(-1, keepdims=True)
>>> rowsum.shape
(3, 1)
>>> x[rowsum <= 2, :]     # fails
IndexError: too many indices
>>> x[rowsum <= 2]
array([0, 1])
```

The last one giving only the first elements because of the extra dimension. Compare `rowsum.nonzero()` to understand this example.

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 = 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], dtype=bool)
>>> columns = [0, 2]
>>> x[np.ix_(rows, columns)]
array([[ 3,  5],
       [ 9, 11]])
```

Without the `np.ix_` call or 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]])
```

1.4.3 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.

1.4.4 Field Access

See also:

Data type objects (dtype), *Scalars*

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']]`. Currently this returns a new array containing a copy of the values in the fields specified in the list. As of NumPy 1.7, returning a copy is being deprecated in favor of returning a view. A copy will continue to be returned for now, but a FutureWarning will be issued when writing to the copy. If you depend on the current behavior, then we suggest copying the returned array explicitly, i.e. use x[['field-name1','field-name2']].copy(). This will work with both past and future versions of NumPy.

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

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')
```

1.4.5 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.

1.5 Iterating Over Arrays

The iterator object *nditer*, introduced in NumPy 1.6, provides many flexible ways to visit all the elements of one or more arrays in a systematic fashion. This page introduces some basic ways to use the object for computations on arrays in Python, then concludes with how one can accelerate the inner loop in Cython. Since the Python exposure of *nditer* is a relatively straightforward mapping of the C array iterator API, these ideas will also provide help working with array iteration from C or C++.

1.5.1 Single Array Iteration

The most basic task that can be done with the *nditer* is to visit every element of an array. Each element is provided one by one using the standard Python iterator interface.

Example

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a):
...     print x,
...
0 1 2 3 4 5
```

An important thing to be aware of for this iteration is that the order is chosen to match the memory layout of the array instead of using a standard C or Fortran ordering. This is done for access efficiency, reflecting the idea that by default one simply wants to visit each element without concern for a particular ordering. We can see this by iterating over the transpose of our previous array, compared to taking a copy of that transpose in C order.

Example

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a.T):
...     print x,
...
0 1 2 3 4 5
```

```
>>> for x in np.nditer(a.T.copy(order='C')):
...     print x,
...
0 3 1 4 2 5
```

The elements of both *a* and *a.T* get traversed in the same order, namely the order they are stored in memory, whereas the elements of *a.T.copy(order='C')* get visited in a different order because they have been put into a different memory layout.

Controlling Iteration Order

There are times when it is important to visit the elements of an array in a specific order, irrespective of the layout of the elements in memory. The *nditer* object provides an *order* parameter to control this aspect of iteration. The default, having the behavior described above, is order='K' to keep the existing order. This can be overridden with order='C' for C order and order='F' for Fortran order.

Example

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a, order='F'):
...     print x,
...
0 3 1 4 2 5
>>> for x in np.nditer(a.T, order='C'):
...     print x,
...
0 3 1 4 2 5
```

Modifying Array Values

By default, the *nditer* treats the input array as a read-only object. To modify the array elements, you must specify either read-write or write-only mode. This is controlled with per-operand flags.

Regular assignment in Python simply changes a reference in the local or global variable dictionary instead of modifying an existing variable in place. This means that simply assigning to *x* will not place the value into the element of the array, but rather switch *x* from being an array element reference to being a reference to the value you assigned. To actually modify the element of the array, *x* should be indexed with the ellipsis.

Example

```
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> for x in np.nditer(a, op_flags=['readwrite']):
...     x[...] = 2 * x
...
>>> a
array([[ 0,  2,  4],
       [ 6,  8, 10]])
```

Using an External Loop

In all the examples so far, the elements of *a* are provided by the iterator one at a time, because all the looping logic is internal to the iterator. While this is simple and convenient, it is not very efficient. A better approach is to move the one-dimensional innermost loop into your code, external to the iterator. This way, NumPy's vectorized operations can be used on larger chunks of the elements being visited.

The *nditer* will try to provide chunks that are as large as possible to the inner loop. By forcing 'C' and 'F' order, we get different external loop sizes. This mode is enabled by specifying an iterator flag.

Observe that with the default of keeping native memory order, the iterator is able to provide a single one-dimensional chunk, whereas when forcing Fortran order, it has to provide three chunks of two elements each.

Example

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a, flags=['external_loop']):
...     print x,
...
[0 1 2 3 4 5]
```

```
>>> for x in np.nditer(a, flags=['external_loop'], order='F'):
...     print x,
...
[0 3] [1 4] [2 5]
```

Tracking an Index or Multi-Index

During iteration, you may want to use the index of the current element in a computation. For example, you may want to visit the elements of an array in memory order, but use a C-order, Fortran-order, or multidimensional index to look up values in a different array.

The Python iterator protocol doesn't have a natural way to query these additional values from the iterator, so we introduce an alternate syntax for iterating with an *nditer*. This syntax explicitly works with the iterator object itself, so its properties are readily accessible during iteration. With this looping construct, the current value is accessible by indexing into the iterator, and the index being tracked is the property *index* or *multi_index* depending on what was requested.

The Python interactive interpreter unfortunately prints out the values of expressions inside the while loop during each iteration of the loop. We have modified the output in the examples using this looping construct in order to be more readable.

Example

```
>>> a = np.arange(6).reshape(2,3)
>>> it = np.nditer(a, flags=['f_index'])
>>> while not it.finished:
...     print "%d <%d>" % (it[0], it.index),
...     it.iternext()
...
0 <0> 1 <2> 2 <4> 3 <1> 4 <3> 5 <5>
```

```
>>> it = np.nditer(a, flags=['multi_index'])
>>> while not it.finished:
...     print "%d <%s>" % (it[0], it.multi_index),
...     it.iternext()
...
0 <(0, 0)> 1 <(0, 1)> 2 <(0, 2)> 3 <(1, 0)> 4 <(1, 1)> 5 <(1, 2)>
```

```
>>> it = np.nditer(a, flags=['multi_index'], op_flags=['writeonly'])
>>> while not it.finished:
...     it[0] = it.multi_index[1] - it.multi_index[0]
...     it.iternext()
...
>>> a
array([[ 0,  1,  2],
       [-1,  0,  1]])
```

Tracking an index or multi-index is incompatible with using an external loop, because it requires a different index value per element. If you try to combine these flags, the *nditer* object will raise an exception

Example

```
>>> a = np.zeros((2,3))
>>> it = np.nditer(a, flags=['c_index', 'external_loop'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Iterator flag EXTERNAL_LOOP cannot be used if an index or multi-index is being tracked
```

Buffering the Array Elements

When forcing an iteration order, we observed that the external loop option may provide the elements in smaller chunks because the elements can't be visited in the appropriate order with a constant stride. When writing C code, this is generally fine, however in pure Python code this can cause a significant reduction in performance.

By enabling buffering mode, the chunks provided by the iterator to the inner loop can be made larger, significantly reducing the overhead of the Python interpreter. In the example forcing Fortran iteration order, the inner loop gets to see all the elements in one go when buffering is enabled.

Example

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a, flags=['external_loop'], order='F'):
...     print x,
...
[0 3] [1 4] [2 5]
```

```
>>> for x in np.nditer(a, flags=['external_loop','buffered'], order='F'):
...     print x,
...
[0 3 1 4 2 5]
```

Iterating as a Specific Data Type

There are times when it is necessary to treat an array as a different data type than it is stored as. For instance, one may want to do all computations on 64-bit floats, even if the arrays being manipulated are 32-bit floats. Except when writing low-level C code, it's generally better to let the iterator handle the copying or buffering instead of casting the data type yourself in the inner loop.

There are two mechanisms which allow this to be done, temporary copies and buffering mode. With temporary copies, a copy of the entire array is made with the new data type, then iteration is done in the copy. Write access is permitted through a mode which updates the original array after all the iteration is complete. The major drawback of temporary copies is that the temporary copy may consume a large amount of memory, particularly if the iteration data type has a larger itemsize than the original one.

Buffering mode mitigates the memory usage issue and is more cache-friendly than making temporary copies. Except for special cases, where the whole array is needed at once outside the iterator, buffering is recommended over temporary copying. Within NumPy, buffering is used by the ufuncs and other functions to support flexible inputs with minimal memory overhead.

In our examples, we will treat the input array with a complex data type, so that we can take square roots of negative numbers. Without enabling copies or buffering mode, the iterator will raise an exception if the data type doesn't match precisely.

Example

```
>>> a = np.arange(6).reshape(2,3) - 3
>>> for x in np.nditer(a, op_dtypes=['complex128']):
...     print np.sqrt(x),
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Iterator operand required copying or buffering, but neither copying nor buffering was enab
```

In copying mode, 'copy' is specified as a per-operand flag. This is done to provide control in a per-operand fashion. Buffering mode is specified as an iterator flag.

Example

```
>>> a = np.arange(6).reshape(2,3) - 3
>>> for x in np.nditer(a, op_flags=['readonly','copy'],
...                op_dtypes=['complex128']):
...     print np.sqrt(x),
```

```
...
1.73205080757j 1.41421356237j 1j 0j (1+0j) (1.41421356237+0j)
```

```
>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['complex128']):
...     print np.sqrt(x),
...
1.73205080757j 1.41421356237j 1j 0j (1+0j) (1.41421356237+0j)
```

The iterator uses NumPy's casting rules to determine whether a specific conversion is permitted. By default, it enforces 'safe' casting. This means, for example, that it will raise an exception if you try to treat a 64-bit float array as a 32-bit float array. In many cases, the rule 'same_kind' is the most reasonable rule to use, since it will allow conversion from 64 to 32-bit float, but not from float to int or from complex to float.

Example

```
>>> a = np.arange(6.)
>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['float32']):
...     print x,
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Iterator operand 0 dtype could not be cast from dtype('float64') to dtype('float32') acco
```

```
>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['float32'],
...                    casting='same_kind'):
...     print x,
...
0.0 1.0 2.0 3.0 4.0 5.0
```

```
>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['int32'], casting='same_kind'):
...     print x,
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Iterator operand 0 dtype could not be cast from dtype('float64') to dtype('int32') accordi
```

One thing to watch out for is conversions back to the original data type when using a read-write or write-only operand. A common case is to implement the inner loop in terms of 64-bit floats, and use 'same_kind' casting to allow the other floating-point types to be processed as well. While in read-only mode, an integer array could be provided, read-write mode will raise an exception because conversion back to the array would violate the casting rule.

Example

```
>>> a = np.arange(6)
>>> for x in np.nditer(a, flags=['buffered'], op_flags=['readwrite'],
...                    op_dtypes=['float64'], casting='same_kind'):
...     x[...] = x / 2.0
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: Iterator requested dtype could not be cast from dtype('float64') to dtype('int64'), the op
```

1.5.2 Broadcasting Array Iteration

NumPy has a set of rules for dealing with arrays that have differing shapes which are applied whenever functions take multiple operands which combine element-wise. This is called *broadcasting*. The `nditer` object can apply these rules for you when you need to write such a function.

As an example, we print out the result of broadcasting a one and a two dimensional array together.

Example

```
>>> a = np.arange(3)
>>> b = np.arange(6).reshape(2,3)
>>> for x, y in np.nditer([a,b]):
...     print "%d:%d" % (x,y),
...
0:0 1:1 2:2 0:3 1:4 2:5
```

When a broadcasting error occurs, the iterator raises an exception which includes the input shapes to help diagnose the problem.

Example

```
>>> a = np.arange(2)
>>> b = np.arange(6).reshape(2,3)
>>> for x, y in np.nditer([a,b]):
...     print "%d:%d" % (x,y),
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (2) (2,3)
```

Iterator-Allocated Output Arrays

A common case in NumPy functions is to have outputs allocated based on the broadcasting of the input, and additionally have an optional parameter called 'out' where the result will be placed when it is provided. The `nditer` object provides a convenient idiom that makes it very easy to support this mechanism.

We'll show how this works by creating a function `square` which squares its input. Let's start with a minimal function definition excluding 'out' parameter support.

Example

```
>>> def square(a):
...     it = np.nditer([a, None])
...     for x, y in it:
...         y[...] = x*x
...     return it.operands[1]
...
>>> square([1,2,3])
array([1, 4, 9])
```

By default, the `nditer` uses the flags 'allocate' and 'writeonly' for operands that are passed in as None. This means we were able to provide just the two operands to the iterator, and it handled the rest.

When adding the 'out' parameter, we have to explicitly provide those flags, because if someone passes in an array as 'out', the iterator will default to 'readonly', and our inner loop would fail. The reason 'readonly' is the default for input

arrays is to prevent confusion about unintentionally triggering a reduction operation. If the default were 'readwrite', any broadcasting operation would also trigger a reduction, a topic which is covered later in this document.

While we're at it, let's also introduce the 'no_broadcast' flag, which will prevent the output from being broadcast. This is important, because we only want one input value for each output. Aggregating more than one input value is a reduction operation which requires special handling. It would already raise an error because reductions must be explicitly enabled in an iterator flag, but the error message that results from disabling broadcasting is much more understandable for end-users. To see how to generalize the square function to a reduction, look at the sum of squares function in the section about Cython.

For completeness, we'll also add the 'external_loop' and 'buffered' flags, as these are what you will typically want for performance reasons.

Example

```
>>> def square(a, out=None):
...     it = np.nditer([a, out],
...             flags = ['external_loop', 'buffered'],
...             op_flags = [['readonly'],
...                         ['writeonly', 'allocate', 'no_broadcast']])
...     for x, y in it:
...         y[...] = x*x
...     return it.operands[1]
...
```

```
>>> square([1,2,3])
array([1, 4, 9])
```

```
>>> b = np.zeros((3,))
>>> square([1,2,3], out=b)
array([ 1.,  4.,  9.])
>>> b
array([ 1.,  4.,  9.])
```

```
>>> square(np.arange(6).reshape(2,3), out=b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in square
ValueError: non-broadcastable output operand with shape (3) doesn't match the broadcast shape (2,3)
```

Outer Product Iteration

Any binary operation can be extended to an array operation in an outer product fashion like in `outer`, and the *nditer* object provides a way to accomplish this by explicitly mapping the axes of the operands. It is also possible to do this with *newaxis* indexing, but we will show you how to directly use the nditer *op_axes* parameter to accomplish this with no intermediate views.

We'll do a simple outer product, placing the dimensions of the first operand before the dimensions of the second operand. The *op_axes* parameter needs one list of axes for each operand, and provides a mapping from the iterator's axes to the axes of the operand.

Suppose the first operand is one dimensional and the second operand is two dimensional. The iterator will have three dimensions, so *op_axes* will have two 3-element lists. The first list picks out the one axis of the first operand, and is -1 for the rest of the iterator axes, with a final result of [0, -1, -1]. The second list picks out the two axes of the second operand, but shouldn't overlap with the axes picked out in the first operand. Its list is [-1, 0, 1]. The output operand maps onto the iterator axes in the standard manner, so we can provide None instead of constructing another list.

The operation in the inner loop is a straightforward multiplication. Everything to do with the outer product is handled by the iterator setup.

Example

```
>>> a = np.arange(3)
>>> b = np.arange(8).reshape(2,4)
>>> it = np.nditer([a, b, None], flags=['external_loop'],
...             op_axes=[[0, -1, -1], [-1, 0, 1], None])
>>> for x, y, z in it:
...     z[...] = x*y
...
>>> it.operands[2]
array([[[ 0,  0,  0,  0],
        [ 0,  0,  0,  0]],
       [[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 0,  2,  4,  6],
        [ 8, 10, 12, 14]]])
```

Reduction Iteration

Whenever a writeable operand has fewer elements than the full iteration space, that operand is undergoing a reduction. The *nditer* object requires that any reduction operand be flagged as read-write, and only allows reductions when 'reduce_ok' is provided as an iterator flag.

For a simple example, consider taking the sum of all elements in an array.

Example

```
>>> a = np.arange(24).reshape(2,3,4)
>>> b = np.array(0)
>>> for x, y in np.nditer([a, b], flags=['reduce_ok', 'external_loop'],
...                       op_flags=[['readonly'], ['readwrite']]):
...     y[...] += x
...
>>> b
array(276)
>>> np.sum(a)
276
```

Things are a little bit more tricky when combining reduction and allocated operands. Before iteration is started, any reduction operand must be initialized to its starting values. Here's how we can do this, taking sums along the last axis of *a*.

Example

```
>>> a = np.arange(24).reshape(2,3,4)
>>> it = np.nditer([a, None], flags=['reduce_ok', 'external_loop'],
...             op_flags=[['readonly'], ['readwrite', 'allocate']],
...             op_axes=[None, [0,1,-1]])
>>> it.operands[1][...] = 0
>>> for x, y in it:
...     y[...] += x
...
>>> it.operands[1]
```

```
array([[ 6, 22, 38],
       [54, 70, 86]])
>>> np.sum(a, axis=2)
array([[ 6, 22, 38],
       [54, 70, 86]])
```

To do buffered reduction requires yet another adjustment during the setup. Normally the iterator construction involves copying the first buffer of data from the readable arrays into the buffer. Any reduction operand is readable, so it may be read into a buffer. Unfortunately, initialization of the operand after this buffering operation is complete will not be reflected in the buffer that the iteration starts with, and garbage results will be produced.

The iterator flag "delay_bufalloc" is there to allow iterator-allocated reduction operands to exist together with buffering. When this flag is set, the iterator will leave its buffers uninitialized until it receives a reset, after which it will be ready for regular iteration. Here's how the previous example looks if we also enable buffering.

Example

```
>>> a = np.arange(24).reshape(2,3,4)
>>> it = np.nditer([a, None], flags=['reduce_ok', 'external_loop',
...                                  'buffered', 'delay_bufalloc'],
...             op_flags=[['readonly'], ['readwrite', 'allocate']],
...             op_axes=[None, [0,1,-1]])
>>> it.operands[1][...] = 0
>>> it.reset()
>>> for x, y in it:
...     y[...] += x
...
>>> it.operands[1]
array([[ 6, 22, 38],
       [54, 70, 86]])
```

1.5.3 Putting the Inner Loop in Cython

Those who want really good performance out of their low level operations should strongly consider directly using the iteration API provided in C, but for those who are not comfortable with C or C++, Cython is a good middle ground with reasonable performance tradeoffs. For the *nditer* object, this means letting the iterator take care of broadcasting, dtype conversion, and buffering, while giving the inner loop to Cython.

For our example, we'll create a sum of squares function. To start, let's implement this function in straightforward Python. We want to support an 'axis' parameter similar to the numpy `sum` function, so we will need to construct a list for the *op_axes* parameter. Here's how this looks.

Example

```
>>> def axis_to_axeslist(axis, ndim):
...     if axis is None:
...         return [-1] * ndim
...     else:
...         if type(axis) is not tuple:
...             axis = (axis,)
...         axeslist = [1] * ndim
...         for i in axis:
...             axeslist[i] = -1
...         ax = 0
...         for i in range(ndim):
```

```
...             if axeslist[i] != -1:
...                 axeslist[i] = ax
...                 ax += 1
...         return axeslist
...
>>> def sum_squares_py(arr, axis=None, out=None):
...     axeslist = axis_to_axeslist(axis, arr.ndim)
...     it = np.nditer([arr, out], flags=['reduce_ok', 'external_loop',
...                                       'buffered', 'delay_bufalloc'],
...                 op_flags=[['readonly'], ['readwrite', 'allocate']],
...                 op_axes=[None, axeslist],
...                 op_dtypes=['float64', 'float64'])
...     it.operands[1][...] = 0
...     it.reset()
...     for x, y in it:
...         y[...] += x*x
...     return it.operands[1]
...
>>> a = np.arange(6).reshape(2,3)
>>> sum_squares_py(a)
array(55.0)
>>> sum_squares_py(a, axis=-1)
array([  5.,  50.])
```

To Cython-ize this function, we replace the inner loop (y[...] += x*x) with Cython code that's specialized for the float64 dtype. With the 'external_loop' flag enabled, the arrays provided to the inner loop will always be one-dimensional, so very little checking needs to be done.

Here's the listing of sum_squares.pyx:

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

def axis_to_axeslist(axis, ndim):
    if axis is None:
        return [-1] * ndim
    else:
        if type(axis) is not tuple:
            axis = (axis,)
        axeslist = [1] * ndim
        for i in axis:
            axeslist[i] = -1
        ax = 0
        for i in range(ndim):
            if axeslist[i] != -1:
                axeslist[i] = ax
                ax += 1
        return axeslist

@cython.boundscheck(False)
def sum_squares_cy(arr, axis=None, out=None):
    cdef np.ndarray[double] x
    cdef np.ndarray[double] y
    cdef int size
    cdef double value

    axeslist = axis_to_axeslist(axis, arr.ndim)
```

```
    it = np.nditer([arr, out], flags=['reduce_ok', 'external_loop',
                                      'buffered', 'delay_bufalloc'],
                op_flags=[['readonly'], ['readwrite', 'allocate']],
                op_axes=[None, axeslist],
                op_dtypes=['float64', 'float64'])
    it.operands[1][...] = 0
    it.reset()
    for xarr, yarr in it:
        x = xarr
        y = yarr
        size = x.shape[0]
        for i in range(size):
           value = x[i]
           y[i] = y[i] + value * value
    return it.operands[1]
```

On this machine, building the .pyx file into a module looked like the following, but you may have to find some Cython tutorials to tell you the specifics for your system configuration.:

```
$ cython sum_squares.pyx
$ gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -I/usr/include/python2.7 -fno-strict-aliasing -o sum_s
```

Running this from the Python interpreter produces the same answers as our native Python/NumPy code did.

Example

```
>>> from sum_squares import sum_squares_cy
>>> a = np.arange(6).reshape(2,3)
>>> sum_squares_cy(a)
array(55.0)
>>> sum_squares_cy(a, axis=-1)
array([  5.,  50.])
```

Doing a little timing in IPython shows that the reduced overhead and memory allocation of the Cython inner loop is providing a very nice speedup over both the straightforward Python code and an expression using NumPy's built-in sum function.:

```
>>> a = np.random.rand(1000,1000)

>>> timeit sum_squares_py(a, axis=-1)
10 loops, best of 3: 37.1 ms per loop

>>> timeit np.sum(a*a, axis=-1)
10 loops, best of 3: 20.9 ms per loop

>>> timeit sum_squares_cy(a, axis=-1)
100 loops, best of 3: 11.8 ms per loop

>>> np.all(sum_squares_cy(a, axis=-1) == np.sum(a*a, axis=-1))
True

>>> np.all(sum_squares_py(a, axis=-1) == np.sum(a*a, axis=-1))
True
```

1.6 Standard array subclasses

The *ndarray* in NumPy is a "new-style" Python built-in-type. Therefore, it can be inherited from (in Python or in C) if desired. Therefore, it can form a foundation for many useful classes. Often whether to sub-class the array object or to simply use the core array component as an internal part of a new class is a difficult decision, and can be simply a matter of choice. NumPy has several tools for simplifying how your new object interacts with other array objects, and so the choice may not be significant in the end. One way to simplify the question is by asking yourself if the object you are interested in can be replaced as a single array or does it really require two or more arrays at its core.

Note that `asarray` always returns the base-class ndarray. If you are confident that your use of the array object can handle any subclass of an ndarray, then `asanyarray` can be used to allow subclasses to propagate more cleanly through your subroutine. In principal a subclass could redefine any aspect of the array and therefore, under strict guidelines, `asanyarray` would rarely be useful. However, most subclasses of the array object will not redefine certain aspects of the array object such as the buffer interface, or the attributes of the array. One important example, however, of why your subroutine may not be able to handle an arbitrary subclass of an array is that matrices redefine the "*" operator to be matrix-multiplication, rather than element-by-element multiplication.

1.6.1 Special attributes and methods

See also:

Subclassing ndarray

Numpy provides several hooks that classes can customize:

`class.__numpy_ufunc__`(*ufunc*, *method*, *i*, *inputs*, ***kwargs*)

New in version 1.11.

Any class (ndarray subclass or not) can define this method to override behavior of Numpy's ufuncs. This works quite similarly to Python's `__mul__` and other binary operation routines.

- *ufunc* is the ufunc object that was called.
- *method* is a string indicating which Ufunc method was called (one of `"__call__"`, `"reduce"`, `"reduceat"`, `"accumulate"`, `"outer"`, `"inner"`).
- *i* is the index of *self* in *inputs*.
- *inputs* is a tuple of the input arguments to the `ufunc`
- *kwargs* is a dictionary containing the optional input arguments of the ufunc. The `out` argument is always contained in *kwargs*, if given. See the discussion in *Universal functions (ufunc)* for details.

The method should return either the result of the operation, or `NotImplemented` if the operation requested is not implemented.

If one of the arguments has a *__numpy_ufunc__* method, it is executed *instead* of the ufunc. If more than one of the input arguments implements *__numpy_ufunc__*, they are tried in the order: subclasses before superclasses, otherwise left to right. The first routine returning something else than `NotImplemented` determines the result. If all of the *__numpy_ufunc__* operations return `NotImplemented`, a `TypeError` is raised.

If an *ndarray* subclass defines the *__numpy_ufunc__* method, this disables the *__array_wrap__*, *__array_prepare__*, *__array_priority__* mechanism described below.

Note: In addition to ufuncs, *__numpy_ufunc__* also overrides the behavior of `numpy.dot` even though it is not an Ufunc.

Note: If you also define right-hand binary operator override methods (such as `__rmul__`) or comparison operations (such as `__gt__`) in your class, they take precedence over the *`__numpy_ufunc__`* mechanism when resolving results of binary operations (such as `ndarray_obj * your_obj`).

The technical special case is: `ndarray.__mul__` returns `NotImplemented` if the other object is *not* a subclass of *`ndarray`*, and defines both `__numpy_ufunc__` and `__rmul__`. Similar exception applies for the other operations than multiplication.

In such a case, when computing a binary operation such as `ndarray_obj * your_obj`, your `__numpy_ufunc__` method *will not* be called. Instead, the execution passes on to your right-hand `__rmul__` operation, as per standard Python operator override rules.

Similar special case applies to *in-place operations*: If you define `__rmul__`, then `ndarray_obj *= your_obj` *will not* call your `__numpy_ufunc__` implementation. Instead, the default Python behavior `ndarray_obj = ndarray_obj * your_obj` occurs.

Note that the above discussion applies only to Python's builtin binary operation mechanism. `np.multiply(ndarray_obj, your_obj)` always calls only your `__numpy_ufunc__`, as expected.

`class.`**`__array_finalize__`**(*obj*)
This method is called whenever the system internally allocates a new array from *obj*, where *obj* is a subclass (subtype) of the *`ndarray`*. It can be used to change attributes of *self* after construction (so as to ensure a 2-d matrix for example), or to update meta-information from the "parent." Subclasses inherit a default implementation of this method that does nothing.

`class.`**`__array_prepare__`**(*array*, *context=None*)
At the beginning of every *ufunc*, this method is called on the input object with the highest array priority, or the output object if one was specified. The output array is passed in and whatever is returned is passed to the ufunc. Subclasses inherit a default implementation of this method which simply returns the output array unmodified. 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 ufunc for computation.

`class.`**`__array_wrap__`**(*array*, *context=None*)
At the end of every *ufunc*, this method is called on the input object with the highest array priority, or the output object if one was specified. The ufunc-computed array is passed in and whatever is returned is passed to the user. Subclasses inherit a default implementation of this method, which transforms the array into a new instance of the object's class. 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.

`class.`**`__array_priority__`**
The value of this 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. Subclasses inherit a default value of 0.0 for this attribute.

`class.`**`__array__`**(*[dtype]*)
If a class (ndarray subclass or not) having the *`__array__`* method is used as the output object of an *ufunc*, results will be written to the object returned by *`__array__`*. Similar conversion is done on input arrays.

1.6.2 Matrix objects

`matrix` objects inherit from the ndarray and therefore, they have the same attributes and methods of ndarrays. There are six important differences of matrix objects, however, that may lead to unexpected results when you use matrices but expect them to act like arrays:

1. Matrix objects can be created using a string notation to allow Matlab-style syntax where spaces separate columns and semicolons (';') separate rows.

2. Matrix objects are always two-dimensional. This has far-reaching implications, in that m.ravel() is still two-dimensional (with a 1 in the first dimension) and item selection returns two-dimensional objects so that sequence behavior is fundamentally different than arrays.
3. Matrix objects over-ride multiplication to be matrix-multiplication. **Make sure you understand this for functions that you may want to receive matrices. Especially in light of the fact that asanyarray(m) returns a matrix when m is a matrix.**
4. Matrix objects over-ride power to be matrix raised to a power. The same warning about using power inside a function that uses asanyarray(...) to get an array object holds for this fact.
5. The default __array_priority__ of matrix objects is 10.0, and therefore mixed operations with ndarrays always produce matrices.
6. Matrices have special attributes which make calculations easier. These are

matrix.T	Returns the transpose of the matrix.
matrix.H	Returns the (complex) conjugate transpose of *self*.
matrix.I	Returns the (multiplicative) inverse of invertible *self*.
matrix.A	Return *self* as an *ndarray* object.

matrix.**T**

Returns the transpose of the matrix.

Does *not* conjugate! For the complex conjugate transpose, use `.H`.

Parameters
None

Returns
ret : matrix object

The (non-conjugated) transpose of the matrix.

See also:

`transpose`, `getH`

Examples

```
>>> m = np.matrix('[1, 2; 3, 4]')
>>> m
matrix([[1, 2],
        [3, 4]])
>>> m.getT()
matrix([[1, 3],
        [2, 4]])
```

matrix.**H**

Returns the (complex) conjugate transpose of *self*.

Equivalent to `np.transpose(self)` if *self* is real-valued.

Parameters
None

Returns
ret : matrix object

complex conjugate transpose of *self*

Examples

```
>>> x = np.matrix(np.arange(12).reshape((3,4)))
>>> z = x - 1j*x; z
matrix([[  0. +0.j,   1. -1.j,   2. -2.j,   3. -3.j],
        [  4. -4.j,   5. -5.j,   6. -6.j,   7. -7.j],
        [  8. -8.j,   9. -9.j,  10.-10.j,  11.-11.j]])
>>> z.getH()
matrix([[  0. +0.j,   4. +4.j,   8. +8.j],
        [  1. +1.j,   5. +5.j,   9. +9.j],
        [  2. +2.j,   6. +6.j,  10.+10.j],
        [  3. +3.j,   7. +7.j,  11.+11.j]])
```

`matrix.I`

Returns the (multiplicative) inverse of invertible *self*.

Parameters

None

Returns

ret : matrix object

If *self* is non-singular, *ret* is such that `ret * self == self * ret == np.matrix(np.eye(self[0,:].size)` all return `True`.

Raises

numpy.linalg.LinAlgError: Singular matrix

If *self* is singular.

See also:

`linalg.inv`

Examples

```
>>> m = np.matrix('[1, 2; 3, 4]'); m
matrix([[1, 2],
        [3, 4]])
>>> m.getI()
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
>>> m.getI() * m
matrix([[ 1.,  0.],
        [ 0.,  1.]])
```

`matrix.A`

Return *self* as an *ndarray* object.

Equivalent to `np.asarray(self)`.

Parameters

None

Returns

ret : ndarray

self as an *ndarray*

Examples

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.getA()
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Warning: Matrix objects over-ride multiplication, '*', and power, '**', to be matrix-multiplication and matrix power, respectively. If your subroutine can accept sub-classes and you do not convert to base- class arrays, then you must use the ufuncs multiply and power to be sure that you are performing the correct operation for all inputs.

The matrix class is a Python subclass of the ndarray and can be used as a reference for how to construct your own subclass of the ndarray. Matrices can be created from other matrices, strings, and anything else that can be converted to an `ndarray` . The name "mat "is an alias for "matrix "in NumPy.

Example 1: Matrix creation from a string

```
>>> a=mat('1 2 3; 4 5 3')
>>> print (a*a.T).I
[[ 0.2924 -0.1345]
 [-0.1345  0.0819]]
```

Example 2: Matrix creation from nested sequence

```
>>> mat([[1,5,10],[1.0,3,4j]])
matrix([[  1.+0.j,   5.+0.j,  10.+0.j],
        [  1.+0.j,   3.+0.j,   0.+4.j]])
```

Example 3: Matrix creation from an array

```
>>> mat(random.rand(3,3)).T
matrix([[ 0.7699,  0.7922,  0.3294],
        [ 0.2792,  0.0101,  0.9219],
        [ 0.3398,  0.7571,  0.8197]])
```

1.6.3 Memory-mapped file arrays

Memory-mapped files are useful for reading and/or modifying small segments of a large file with regular layout, without reading the entire file into memory. A simple subclass of the ndarray uses a memory-mapped file for the data buffer of the array. For small files, the over-head of reading the entire file into memory is typically not significant, however for large files using memory mapping can save considerable resources.

Memory-mapped-file arrays have one additional method (besides those they inherit from the ndarray): `.flush()` which must be called manually by the user to ensure that any changes to the array actually get written to disk.

Example:

```
>>> a = memmap('newfile.dat', dtype=float, mode='w+', shape=1000)
>>> a[10] = 10.0
```

```
>>> a[30] = 30.0
>>> del a
>>> b = fromfile('newfile.dat', dtype=float)
>>> print b[10], b[30]
10.0 30.0
>>> a = memmap('newfile.dat', dtype=float)
>>> print a[10], a[30]
10.0 30.0
```

1.6.4 Character arrays (`numpy.char`)

See also:

Creating character arrays (numpy.char)

Note: The `chararray` class exists for backwards compatibility with Numarray, it is not recommended for new development. Starting from numpy 1.4, if one needs arrays of strings, it is recommended to use arrays of *`dtype`* `object_`, `string_` or `unicode_`, and use the free functions in the `numpy.char` module for fast vectorized string operations.

These are enhanced arrays of either `string_` type or `unicode_` type. These arrays inherit from the *`ndarray`*, but specially-define the operations `+`, `*`, and `%` on a (broadcasting) element-by-element basis. These operations are not available on the standard *`ndarray`* of character type. In addition, the `chararray` has all of the standard `string` (and `unicode`) methods, executing them on an element-by-element basis. Perhaps the easiest way to create a chararray is to use *`self.view(chararray)`* where *self* is an ndarray of str or unicode data-type. However, a chararray can also be created using the `numpy.chararray` constructor, or via the *`numpy.char.array`* function:

`core.defchararray.array`(obj[, itemsize, ...])	Create a `chararray`.

numpy.core.defchararray.**array**(*obj*, *itemsize=None*, *copy=True*, *unicode=None*, *order=None*)

Create a *`chararray`*.

Note: This class is provided for numarray backward-compatibility. New code (not concerned with numarray compatibility) should use arrays of type `string_` or `unicode_` and use the free functions in `numpy.char` for fast vectorized string operations instead.

Versus a regular Numpy array of type *str* or *unicode*, this class adds the following functionality:

1. values automatically have whitespace removed from the end when indexed
2. comparison operators automatically remove whitespace from the end when comparing values
3. vectorized string operations are provided as methods (e.g. *str.endswith*) and infix operators (e.g. `+, *, %`)

Parameters

obj : array of str or unicode-like

itemsize : int, optional

itemsize is the number of characters per scalar in the resulting array. If *itemsize* is None, and *obj* is an object array or a Python list, the *itemsize* will be automatically determined. If *itemsize* is provided and *obj* is of type str or unicode, then the *obj* string will be chunked into *itemsize* pieces.

copy : bool, optional

If true (default), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (*itemsize*, unicode, *order*, etc.).

unicode : bool, optional

When true, the resulting *chararray* can contain Unicode characters, when false only 8-bit characters. If unicode is *None* and *obj* is one of the following:

- a *chararray*,
- an ndarray of type *str* or *unicode*
- a Python str or unicode object,

then the unicode setting of the output array will be automatically determined.

order : {'C', 'F', 'A'}, optional

Specify the order of the array. If order is 'C' (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is 'F', then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is 'A', then the returned array may be in any order (either C-, Fortran-contiguous, or even discontiguous).

Another difference with the standard ndarray of str data-type is that the chararray inherits the feature introduced by Numarray that white-space at the end of any element in the array will be ignored on item retrieval and comparison operations.

1.6.5 Record arrays (`numpy.rec`)

See also:

Creating record arrays (numpy.rec), *Data type routines*, *Data type objects (dtype)*.

Numpy provides the *recarray* class which allows accessing the fields of a structured array as attributes, and a corresponding scalar data type object *record*.

recarray	Construct an ndarray that allows field access using attributes.
record	A data-type scalar that allows field access as attribute lookup.

class numpy.**recarray**

Construct an ndarray that allows field access using attributes.

Arrays may have a data-types containing fields, analogous to columns in a spread sheet. An example is `[(x, int), (y, float)]`, where each entry in the array is a pair of `(int, float)`. Normally, these attributes are accessed using dictionary lookups such as `arr['x']` and `arr['y']`. Record arrays allow the fields to be accessed as members of the array, using `arr.x` and `arr.y`.

Parameters

shape : tuple

Shape of output array.

dtype : data-type, optional

The desired data-type. By default, the data-type is determined from *formats*, *names*, *titles*, *aligned* and *byteorder*.

formats : list of data-types, optional

A list containing the data-types for the different columns, e.g. `['i4', 'f8', 'i4']`. *formats* does *not* support the new convention of using types directly, i.e. `(int, float, int)`. Note that *formats* must be a list, not a tuple. Given that *formats* is somewhat limited, we recommend specifying *dtype* instead.

names : tuple of str, optional

The name of each column, e.g. `('x', 'y', 'z')`.

buf : buffer, optional

By default, a new array is created of the given shape and data-type. If *buf* is specified and is an object exposing the buffer interface, the array will use the memory from the existing buffer. In this case, the *offset* and *strides* keywords are available.

Returns

rec : recarray

Empty array of the given shape and type.

Other Parameters

titles : tuple of str, optional

Aliases for column names. For example, if *names* were `('x', 'y', 'z')` and *titles* is `('x_coordinate', 'y_coordinate', 'z_coordinate')`, then `arr['x']` is equivalent to both `arr.x` and `arr.x_coordinate`.

byteorder : {'<', '>', '='}, optional

Byte-order for all fields.

aligned : bool, optional

Align the fields in memory as the C-compiler would.

strides : tuple of ints, optional

Buffer (*buf*) is interpreted according to these strides (strides define how many bytes each array element, row, column, etc. occupy in memory).

offset : int, optional

Start reading buffer (*buf*) from this offset onwards.

order : {'C', 'F'}, optional

Row-major (C-style) or column-major (Fortran-style) order.

See also:

`rec.fromrecords`
Construct a record array from data.

record
fundamental data-type for *recarray*.

`format_parser`
determine a data-type from formats, names, titles.

Notes

This constructor can be compared to `empty`: it creates a new record array but does not fill it with data. To create a record array from data, use one of the following methods:

1. Create a standard ndarray and convert it to a record array, using `arr.view(np.recarray)`

2.Use the *buf* keyword.

3.Use *np.rec.fromrecords*.

Examples

Create an array with two fields, x and y:

```
>>> x = np.array([(1.0, 2), (3.0, 4)], dtype=[('x', float), ('y', int)])
>>> x
array([(1.0, 2), (3.0, 4)],
      dtype=[('x', '<f8'), ('y', '<i4')])
```

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

View the array as a record array:

```
>>> x = x.view(np.recarray)
```

```
>>> x.x
array([ 1.,  3.])
```

```
>>> x.y
array([2, 4])
```

Create a new, empty record array:

```
>>> np.recarray((2,),
... dtype=[('x', int), ('y', float), ('z', int)])
rec.array([(-1073741821, 1.2249118382103472e-301, 24547520),
       (3471280, 1.2134086255804012e-316, 0)],
      dtype=[('x', '<i4'), ('y', '<f8'), ('z', '<i4')])
```

Attributes

T	Same as self.transpose(), except that self is returned if self.ndim < 2.
base	Base object if memory is from some other object.
ctypes	An object to simplify the interaction of the array with the ctypes module.
data	Python buffer object pointing to the start of the array's data.
dtype	Data-type of the array's elements.
flags	Information about the memory layout of the array.
flat	A 1-D iterator over the array.
imag	The imaginary part of the array.
itemsize	Length of one array element in bytes.
nbytes	Total bytes consumed by the elements of the array.
ndim	Number of array dimensions.
real	The real part of the array.
shape	Tuple of array dimensions.
size	Number of elements in the array.
strides	Tuple of bytes to step in each dimension when traversing an array.

recarray.**T**
Same as self.transpose(), except that self is returned if self.ndim < 2.

Examples

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

recarray.**base**

Base object if memory is from some other object.

Examples

The base of an array that owns its memory is None:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```
>>> y = x[2:]
>>> y.base is x
True
```

recarray.**ctypes**

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters
None

Returns
c : Python object

Possessing attributes data, shape, strides, etc.

See also:

numpy.ctypeslib

Notes

Below are the public attributes of this object which were documented in "Guide to NumPy" (we have omitted undocumented public attributes, as well as documented private attributes):

•data: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].

- shape (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.
- strides (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.
- data_as(obj): Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).
- shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).
- strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

Examples

```
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

`recarray.`**`data`**
Python buffer object pointing to the start of the array's data.

`recarray.`**`dtype`**
Data-type of the array's elements.

Parameters
None

Returns
d : numpy dtype object

See also:

numpy.dtype

Examples

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

recarray.**flags**
Information about the memory layout of the array.

Notes

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling *ndarray.setflags*.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set `False`.
- ALIGNED can only be set `True` if the data is truly aligned.
- WRITEABLE can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

Attributes

C_CONTIGUOUS (C)	The data is in a single, C-style contiguous segment.
F_CONTIGUOUS (F)	The data is in a single, Fortran-style contiguous segment.
OWNDATA (O)	The array owns the memory it uses or borrows it from another object.
WRITEABLE (W)	The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.
ALIGNED (A)	The data and all elements are aligned appropriately for the hardware.
UPDATEIFCOPY (U)	This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.
FNC	F_CONTIGUOUS and not C_CONTIGUOUS.
FORC	F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).
BEHAVED (B)	ALIGNED and WRITEABLE.
CARRAY (CA)	BEHAVED and C_CONTIGUOUS.
FARRAY (FA)	BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

recarray.**flat**

A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See also:

flatten

Return a copy of the array collapsed into one dimension.

flatiter

Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```

```
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

recarray.**imag**
The imaginary part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.        ,  0.70710678])
>>> x.imag.dtype
dtype('float64')
```

recarray.**itemsize**
Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

recarray.**nbytes**
Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

recarray.**ndim**
Number of array dimensions.

Examples

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

```
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

recarray.**real**
The real part of the array.

See also:

numpy.real
equivalent function

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.        ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

recarray.**shape**
Tuple of array dimensions.

Notes

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

recarray.**size**
Number of elements in the array.

Equivalent to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

recarray.**strides**

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the "ndarray.rst" file in the NumPy reference guide.

See also:

`numpy.lib.stride_tricks.as_strided`

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array *x* will be `(20, 4)`.

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> 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]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

Methods

all([axis, out, keepdims])	Returns True if all elements evaluate to True.
any([axis, out, keepdims])	Returns True if any of the elements of *a* evaluate to True.

Table 1.44 – continued from previous page

argmax([axis, out])	Return indices of the maximum values along the given axis.
argmin([axis, out])	Return indices of the minimum values along the given axis of *a*.
argpartition(kth[, axis, kind, order])	Returns the indices that would partition this array.
argsort([axis, kind, order])	Returns the indices that would sort this array.
astype(dtype[, order, casting, subok, copy])	Copy of the array, cast to a specified type.
byteswap(inplace)	Swap the bytes of the array elements
choose(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
clip([min, max, out])	Return an array whose values are limited to `[min, max]`.
compress(condition[, axis, out])	Return selected slices of this array along given axis.
conj()	Complex-conjugate all elements.
conjugate()	Return the complex conjugate, element-wise.
copy([order])	Return a copy of the array.
cumprod([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
cumsum([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
diagonal([offset, axis1, axis2])	Return specified diagonals.
dot(b[, out])	Dot product of two arrays.
dump(file)	Dump a pickle of the array to the specified file.
dumps()	Returns the pickle of the array as a string.
fill(value)	Fill the array with a scalar value.
flatten([order])	Return a copy of the array collapsed into one dimension.
getfield(dtype[, offset])	Returns a field of the given array as a certain type.
item(*args)	Copy an element of an array to a standard Python scalar and return it.
itemset(*args)	Insert scalar into an array (scalar is cast to array's dtype, if possible)
max([axis, out])	Return the maximum along a given axis.
mean([axis, dtype, out, keepdims])	Returns the average of the array elements along given axis.
min([axis, out, keepdims])	Return the minimum along a given axis.
newbyteorder([new_order])	Return the array with the same data viewed with a different byte order.
nonzero()	Return the indices of the elements that are non-zero.
partition(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that value of the element in kth po
prod([axis, dtype, out, keepdims])	Return the product of the array elements over the given axis
ptp([axis, out])	Peak to peak (maximum - minimum) value along a given axis.
put(indices, values[, mode])	Set `a.flat[n] = values[n]` for all *n* in indices.
ravel([order])	Return a flattened array.
repeat(repeats[, axis])	Repeat elements of an array.
reshape(shape[, order])	Returns an array containing the same data with a new shape.
resize(new_shape[, refcheck])	Change shape and size of array in-place.
round([decimals, out])	Return *a* with each element rounded to the given number of decimals.
searchsorted(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
setfield(val, dtype[, offset])	Put a value into a specified place in a field defined by a data-type.
setflags([write, align, uic])	Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.
sort([axis, kind, order])	Sort an array, in-place.
squeeze([axis])	Remove single-dimensional entries from the shape of *a*.
std([axis, dtype, out, ddof, keepdims])	Returns the standard deviation of the array elements along given axis.
sum([axis, dtype, out, keepdims])	Return the sum of the array elements over the given axis.
swapaxes(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
take(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.
tobytes([order])	Construct Python bytes containing the raw data bytes in the array.
tofile(fid[, sep, format])	Write array to a file as text or binary (default).
tolist()	Return the array as a (possibly nested) list.
tostring([order])	Construct Python bytes containing the raw data bytes in the array.

Table 1.44 – continued from previous page

trace([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.
transpose(*axes)	Returns a view of the array with axes transposed.
var([axis, dtype, out, ddof, keepdims])	Returns the variance of the array elements, along given axis.
view([dtype, type])	New view of array with the same data.

recarray.**all**(*axis=None*, *out=None*, *keepdims=False*)
Returns True if all elements evaluate to True.

Refer to `numpy.all` for full documentation.

See also:

numpy.all
equivalent function

recarray.**any**(*axis=None*, *out=None*, *keepdims=False*)
Returns True if any of the elements of *a* evaluate to True.

Refer to `numpy.any` for full documentation.

See also:

numpy.any
equivalent function

recarray.**argmax**(*axis=None*, *out=None*)
Return indices of the maximum values along the given axis.

Refer to `numpy.argmax` for full documentation.

See also:

numpy.argmax
equivalent function

recarray.**argmin**(*axis=None*, *out=None*)
Return indices of the minimum values along the given axis of *a*.

Refer to `numpy.argmin` for detailed documentation.

See also:

numpy.argmin
equivalent function

recarray.**argpartition**(*kth*, *axis=-1*, *kind='introselect'*, *order=None*)
Returns the indices that would partition this array.

Refer to `numpy.argpartition` for full documentation.

New in version 1.8.0.

See also:

numpy.argpartition
equivalent function

recarray.**argsort**(*axis=-1*, *kind='quicksort'*, *order=None*)
Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

See also:

numpy.argsort
equivalent function

recarray.**astype**(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)
Copy of the array, cast to a specified type.

Parameters

dtype : str or dtype

Typecode or data-type to which the array is cast.

order : {'C', 'F', 'A', 'K'}, optional

Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional

Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

subok : bool, optional

If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy : bool, optional

By default, astype always returns a newly allocated array. If this is set to false, and the `dtype`, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

Returns

arr_t : ndarray

Unless `copy` is False and the other conditions for returning the input array are satisfied (see description for `copy` input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by `dtype`, *order*.

Raises

ComplexWarning

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Notes

Starting in NumPy 1.9, 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.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

recarray.**byteswap**(*inplace*)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

Parameters

inplace : bool, optional

If `True`, swap bytes in-place, default is `False`.

Returns

out : ndarray

The byteswapped array. If *inplace* is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([  256,     1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']
```

Arrays of strings are not swapped

```
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
      dtype='|S3')
```

recarray.**choose**(*choices*, *out=None*, *mode='raise'*)

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See also:

numpy.choose
equivalent function

recarray.**clip**(*min=None*, *max=None*, *out=None*)

Return an array whose values are limited to `[min, max]`. One of max or min must be given.

Refer to `numpy.clip` for full documentation.

See also:

numpy.clip
equivalent function

recarray.**compress**(*condition*, *axis=None*, *out=None*)
Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See also:

numpy.compress
equivalent function

recarray.**conj**()
Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

See also:

numpy.conjugate
equivalent function

recarray.**conjugate**()
Return the complex conjugate, element-wise.

Refer to `numpy.conjugate` for full documentation.

See also:

numpy.conjugate
equivalent function

recarray.**copy**(*order='C'*)
Return a copy of the array.

Parameters
order : {'C', 'F', 'A', 'K'}, optional

Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and :func:numpy.copy are very similar, but have different default values for their order= arguments.)

See also:

`numpy.copy`, `numpy.copyto`

Examples

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

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

recarray.**cumprod**(*axis=None*, *dtype=None*, *out=None*)

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

See also:

numpy.cumprod
equivalent function

recarray.**cumsum**(*axis=None*, *dtype=None*, *out=None*)

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See also:

numpy.cumsum
equivalent function

recarray.**diagonal**(*offset=0*, *axis1=0*, *axis2=1*)

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal` for full documentation.

See also:

numpy.diagonal
equivalent function

recarray.**dot**(*b*, *out=None*)

Dot product of two arrays.

Refer to `numpy.dot` for full documentation.

See also:

numpy.dot
equivalent function

Examples

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

recarray.**dump**(*file*)
Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters
file : str

A string naming the dump file.

recarray.**dumps**()
Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

Parameters
None

recarray.**fill**(*value*)
Fill the array with a scalar value.

Parameters
value : scalar

All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1.,  1.])
```

recarray.**flatten**(*order='C'*)
Return a copy of the array collapsed into one dimension.

Parameters
order : {'C', 'F', 'A', 'K'}, optional

'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

Returns
y : ndarray

A copy of the input array, flattened to one dimension.

See also:

ravel
Return a flattened array.

flat
A 1-D flat iterator over the array.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

recarray.**getfield**(*dtype*, *offset=0*)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype : str or dtype

The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset : int

Number of bytes to skip before beginning the element view.

Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[ 1.+1.j,  0.+0.j],
       [ 0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[ 1.,  0.],
       [ 0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[ 1.,  0.],
   [ 0.,  4.]])
```

recarray.**item**(**args*)

Copy an element of an array to a standard Python scalar and return it.

Parameters

***args** : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z : Standard Python scalar object

A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

recarray.**itemset**(**args*)

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

Parameters

***args** : Arguments

If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes

Compared to indexing syntax, *itemset* provides some speed increase for placing a scalar into a particular location in an *ndarray*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using *itemset* (and *item*) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

recarray.**max**(*axis=None*, *out=None*)
Return the maximum along a given axis.

Refer to `numpy.amax` for full documentation.

See also:

numpy.amax
equivalent function

recarray.**mean**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

See also:

numpy.mean
equivalent function

recarray.**min**(*axis=None*, *out=None*, *keepdims=False*)
Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

See also:

numpy.amin
equivalent function

recarray.**newbyteorder**(*new_order='S'*)
Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

Parameters
new_order : string, optional

Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns
new_arr : array

New array object with the dtype reflecting given change to the byte order.

recarray.**nonzero**()
Return the indices of the elements that are non-zero.

Refer to numpy.nonzero for full documentation.

See also:

numpy.nonzero
equivalent function

recarray.**partition**(*kth*, *axis=-1*, *kind='introselect'*, *order=None*)
Rearranges the elements in the array in such a way that value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

Parameters

kth : int or sequence of ints

Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order all elements in the partitions is undefined. If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.

axis : int, optional

Axis along which to sort. Default is -1, which means sort along the last axis.

kind : {'introselect'}, optional

Selection algorithm. Default is 'introselect'.

order : str or list of str, optional

When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See also:

numpy.partition
Return a parititioned copy of an array.

argpartition
Indirect partition.

sort
Full sort.

Notes

See np.partition for notes on the different algorithms.

Examples

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

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

recarray.**prod**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Return the product of the array elements over the given axis

Refer to `numpy.prod` for full documentation.

See also:

numpy.prod
equivalent function

recarray.**ptp**(*axis=None*, *out=None*)
Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See also:

numpy.ptp
equivalent function

recarray.**put**(*indices*, *values*, *mode='raise'*)
Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to `numpy.put` for full documentation.

See also:

numpy.put
equivalent function

recarray.**ravel**([*order*])
Return a flattened array.

Refer to `numpy.ravel` for full documentation.

See also:

numpy.ravel
equivalent function

ndarray.flat
a flat iterator on the array.

recarray.**repeat**(*repeats*, *axis=None*)
Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

See also:

numpy.repeat
equivalent function

recarray.**reshape**(*shape*, *order='C'*)
Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

See also:

numpy.reshape
equivalent function

recarray.**resize**(*new_shape*, *refcheck=True*)
Change shape and size of array in-place.

Parameters
new_shape : tuple of ints, or *n* ints

Shape of resized array.

refcheck : bool, optional

If False, reference count will not be checked. Default is True.

Returns
None

Raises
ValueError

If *a* does not own its own data or references or views to it exist, and the data memory must be changed.

SystemError

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

See also:

resize
Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

recarray.**round**(*decimals=0*, *out=None*)

Return *a* with each element rounded to the given number of decimals.

Refer to `numpy.around` for full documentation.

See also:

numpy.around
equivalent function

recarray.**searchsorted**(*v*, *side='left'*, *sorter=None*)

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see `numpy.searchsorted`

See also:

numpy.searchsorted
equivalent function

recarray.**setfield**(*val*, *dtype*, *offset=0*)

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *dtype* and beginning *offset* bytes into the field.

Parameters

val : object

Value to be placed in field.

dtype : dtype object

Data-type of the field in which to place *val*.

offset : int, optional

The number of bytes into the field at which to place *val*.

Returns

None

See also:

getfield

Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
>>> x
array([[  1.00000000e+000,   1.48219694e-323,   1.48219694e-323],
       [  1.48219694e-323,   1.00000000e+000,   1.48219694e-323],
       [  1.48219694e-323,   1.48219694e-323,   1.00000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

recarray.**setflags**(*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write : bool, optional

Describes whether or not *a* can be written to.

align : bool, optional

Describes whether or not *a* is aligned properly for its type.

uic : bool, optional

Describes whether or not *a* is a copy of another "base" array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

Examples

```
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

recarray.**sort**(*axis=-1*, *kind='quicksort'*, *order=None*)

Sort an array, in-place.

Parameters

axis : int, optional

Axis along which to sort. Default is -1, which means sort along the last axis.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional

Sorting algorithm. Default is 'quicksort'.

order : str or list of str, optional

When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See also:

numpy.sort
Return a sorted copy of an array.

argsort
Indirect sort.

lexsort
Indirect stable sort on multiple keys.

searchsorted
Find elements in sorted array.

partition
Partial sort.

Notes

See `sort` for notes on the different sorting algorithms.

Examples

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

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

recarray.**squeeze**(*axis=None*)
Remove single-dimensional entries from the shape of *a*.

Refer to `numpy.squeeze` for full documentation.

See also:

numpy.squeeze
equivalent function

recarray.**std**(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)
Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

See also:

numpy.std
equivalent function

recarray.**sum**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

See also:

numpy.sum
equivalent function

recarray.**swapaxes**(*axis1*, *axis2*)
Return a view of the array with *axis1* and *axis2* interchanged.

Refer to numpy.swapaxes for full documentation.

See also:

numpy.swapaxes
equivalent function

recarray.**take**(*indices*, *axis=None*, *out=None*, *mode='raise'*)
Return an array formed from the elements of *a* at the given indices.

Refer to numpy.take for full documentation.

See also:

numpy.take
equivalent function

recarray.**tobytes**(*order='C'*)
Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

New in version 1.9.0.

Parameters
order : {'C', 'F', None}, optional

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns
s : bytes

Python bytes exhibiting a copy of *a*'s raw data.

Examples

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

recarray.**tofile**(*fid*, *sep=""*, *format="%s"*)
Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function fromfile().

Parameters
fid : file or str

An open file object, or a string containing a filename.

sep : str

Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format : str

Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

`recarray.`**`tolist`**`()`

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

Parameters

none

Returns

y : list

The possibly nested list of array elements.

Notes

The array may be recreated, `a = np.array(a.tolist())`.

Examples

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

`recarray.`**`tostring`**`(`*`order='C'`*`)`

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

This function is a compatibility alias for tobytes. Despite its name it returns bytes not strings.

Parameters

order : {'C', 'F', None}, optional

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns

s : bytes

Python bytes exhibiting a copy of *a*'s raw data.

Examples

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

recarray.**trace**(*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See also:

numpy.trace

equivalent function

recarray.**transpose**(**axes*)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

Parameters

axes : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

Returns

out : ndarray

View of *a*, with axes suitably permuted.

See also:

ndarray.T

Array property returning the array transposed.

Examples

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

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

recarray.**var**(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)

Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

See also:

numpy.var
equivalent function

recarray.**view**(*dtype=None*, *type=None*)

New view of array with the same data.

Parameters

dtype : data-type or ndarray sub-class, optional

Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

type : Python type, optional

Type of the returned view, e.g., ndarray or matrix. Again, the default None results in type preservation.

Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrixlib.defmatrix.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([ 2.,  3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> print(x)
[(1, 20) (3, 4)]
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

class numpy.**record**

A data-type scalar that allows field access as attribute lookup.

Attributes

T	transpose
base	base object

Continued on next page

Table 1.45 – continued from previous page

data	pointer to start of data
dtype	dtype object
flags	integer value of flags
flat	a 1-d view of scalar
imag	imaginary part of scalar
itemsize	length of one element in bytes
nbytes	length of item in bytes
ndim	number of array dimensions
real	real part of scalar
shape	tuple of array dimensions
size	number of elements in the gentype
strides	tuple of bytes steps in each dimension

record.**T**
transpose

record.**base**
base object

record.**data**
pointer to start of data

record.**dtype**
dtype object

record.**flags**
integer value of flags

record.**flat**
a 1-d view of scalar

record.**imag**
imaginary part of scalar

record.**itemsize**
length of one element in bytes

record.**nbytes**
length of item in bytes

record.**ndim**
number of array dimensions

record.**real**
real part of scalar

record.**shape**
tuple of array dimensions

record.**size**
number of elements in the gentype

record.**strides**
tuple of bytes steps in each dimension

Methods

all	Not implemented (virtual attribute)

Continued on next page

Table 1.46 – continued from previous page

any	Not implemented (virtual attribute)
argmax	Not implemented (virtual attribute)
argmin	Not implemented (virtual attribute)
argsort	Not implemented (virtual attribute)
astype	Not implemented (virtual attribute)
byteswap	Not implemented (virtual attribute)
choose	Not implemented (virtual attribute)
clip	Not implemented (virtual attribute)
compress	Not implemented (virtual attribute)
conj	
conjugate	Not implemented (virtual attribute)
copy	Not implemented (virtual attribute)
cumprod	Not implemented (virtual attribute)
cumsum	Not implemented (virtual attribute)
diagonal	Not implemented (virtual attribute)
dump	Not implemented (virtual attribute)
dumps	Not implemented (virtual attribute)
fill	Not implemented (virtual attribute)
flatten	Not implemented (virtual attribute)
getfield	
item	Not implemented (virtual attribute)
itemset	Not implemented (virtual attribute)
max	Not implemented (virtual attribute)
mean	Not implemented (virtual attribute)
min	Not implemented (virtual attribute)
newbyteorder([new_order])	Return a new *dtype* with a different byte order.
nonzero	Not implemented (virtual attribute)
prod	Not implemented (virtual attribute)
ptp	Not implemented (virtual attribute)
put	Not implemented (virtual attribute)
ravel	Not implemented (virtual attribute)
repeat	Not implemented (virtual attribute)
reshape	Not implemented (virtual attribute)
resize	Not implemented (virtual attribute)
round	Not implemented (virtual attribute)
searchsorted	Not implemented (virtual attribute)
setfield	
setflags	Not implemented (virtual attribute)
sort	Not implemented (virtual attribute)
squeeze	Not implemented (virtual attribute)
std	Not implemented (virtual attribute)
sum	Not implemented (virtual attribute)
swapaxes	Not implemented (virtual attribute)
take	Not implemented (virtual attribute)
tobytes	
tofile	Not implemented (virtual attribute)
tolist	Not implemented (virtual attribute)
tostring	Not implemented (virtual attribute)
trace	Not implemented (virtual attribute)
transpose	Not implemented (virtual attribute)

Continued on next page

Table 1.46 – continued from previous page

var	Not implemented (virtual attribute)
view	Not implemented (virtual attribute)

record.**all**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**any**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**argmax**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**argmin**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**argsort**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**astype**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**byteswap**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**choose**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**clip**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**compress**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**conj**()

record.**conjugate**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**copy**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**cumprod**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**cumsum**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**diagonal**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**dump**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**dumps**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**fill**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**flatten**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**getfield**()

record.**item**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**itemset**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**max**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**mean**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**min**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**newbyteorder**(*new_order='S'*)
Return a new *dtype* with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The *new_order* code can be any from the following:

- 'S' - swap dtype from current to opposite endian
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order

•{'|', 'I'} - ignore (no change to byte order)

Parameters
new_order : str, optional

Byte order to force; a value from the byte order specifications above. The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns
new_dtype : dtype

New *dtype* object with the given change to the byte order.

record.**nonzero**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**prod**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**ptp**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**put**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**ravel**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**repeat**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**reshape**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**resize**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**round**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**searchsorted**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**setfield**()

record.**setflags**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**sort**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**squeeze**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**std**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**sum**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**swapaxes**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**take**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**tobytes**()

record.**tofile**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**tolist**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**tostring**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**trace**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**transpose**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**var**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

record.**view**()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also:

The

1.6.6 Masked arrays (`numpy.ma`)

See also:

Masked arrays

1.6.7 Standard container class

For backward compatibility and as a standard "container "class, the UserArray from Numeric has been brought over to NumPy and named *numpy.lib.user_array.container* The container class is a Python class whose self.array attribute is an ndarray. Multiple inheritance is probably easier with numpy.lib.user_array.container than with the ndarray itself and so it is included by default. It is not documented here beyond mentioning its existence because you are encouraged to use the ndarray class directly if you can.

numpy.lib.user_array.container(data[, ...])	Standard container-class for easy multiple-inheritance.

class numpy.lib.user_array.**container**(*data*, *dtype=None*, *copy=True*)

Standard container-class for easy multiple-inheritance.

Methods

copy	
tostring	
byteswap	
astype	

1.6.8 Array Iterators

Iterators are a powerful concept for array processing. Essentially, iterators implement a generalized for-loop. If *myiter* is an iterator object, then the Python code:

```
for val in myiter:
    ...
    some code involving val
    ...
```

calls `val = myiter.next()` repeatedly until `StopIteration` is raised by the iterator. There are several ways to iterate over an array that may be useful: default iteration, flat iteration, and N-dimensional enumeration.

Default iteration

The default iterator of an ndarray object is the default Python iterator of a sequence type. Thus, when the array object itself is used as an iterator. The default behavior is equivalent to:

```
for i in range(arr.shape[0]):
    val = arr[i]
```

This default iterator selects a sub-array of dimension $N-1$ from the array. This can be a useful construct for defining recursive algorithms. To loop over the entire array requires N for-loops.

```
>>> a = arange(24).reshape(3,2,4)+10
>>> for val in a:
...     print 'item:', val
item: [[10 11 12 13]
```

```
 [14 15 16 17]]
item: [[18 19 20 21]
 [22 23 24 25]]
item: [[26 27 28 29]
 [30 31 32 33]]
```

Flat iteration

ndarray.flat	A 1-D iterator over the array.

ndarray.**flat**
A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See also:

flatten
Return a copy of the array collapsed into one dimension.

flatiter

Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

As mentioned previously, the flat attribute of ndarray objects returns an iterator that will cycle over the entire array in C-style contiguous order.

```
>>> for i, val in enumerate(a.flat):
...     if i%5 == 0: print i, val
0 10
5 15
```

```
10 20
15 25
20 30
```

Here, I've used the built-in enumerate iterator to return the iterator index as well as the value.

N-dimensional enumeration

Sometimes it may be useful to get the N-dimensional index while iterating. The ndenumerate iterator can achieve this.

```
>>> for i, val in ndenumerate(a):
...     if sum(i)%5 == 0: print i, val
(0, 0, 0) 10
(1, 1, 3) 25
(2, 0, 3) 29
(2, 1, 2) 32
```

Iterator for broadcasting

broadcast	Produce an object that mimics broadcasting.

class numpy.**broadcast**

Produce an object that mimics broadcasting.

Parameters

in1, in2, ... : array_like

Input parameters.

Returns

b : broadcast object

Broadcast the input parameters against one another, and return an object that encapsulates the result. Amongst others, it has `shape` and `nd` properties, and may be used as an iterator.

Examples

Manually adding two vectors, using broadcasting:

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

```
>>> out = np.empty(b.shape)
>>> out.flat = [u+v for (u,v) in b]
>>> out
array([[ 5.,  6.,  7.],
       [ 6.,  7.,  8.],
       [ 7.,  8.,  9.]])
```

Compare against built-in broadcasting:

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

Attributes

index	current index in broadcasted result
iters	tuple of iterators along `self`'s "components."
shape	Shape of broadcasted result.
size	Total size of broadcasted result.

`broadcast.`**`index`**
current index in broadcasted result

Examples

```
>>> x = np.array([[1], [2], [3]])
>>> y = np.array([4, 5, 6])
>>> b = np.broadcast(x, y)
>>> b.index
0
>>> b.next(), b.next(), b.next()
((1, 4), (1, 5), (1, 6))
>>> b.index
3
```

`broadcast.`**`iters`**
tuple of iterators along `self`'s "components."

Returns a tuple of *numpy.flatiter* objects, one for each "component" of `self`.

See also:

numpy.flatiter

Examples

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> row, col = b.iters
>>> row.next(), col.next()
(1, 4)
```

`broadcast.`**`shape`**
Shape of broadcasted result.

Examples

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

`broadcast.`**`size`**
Total size of broadcasted result.

Examples

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> b.size
9
```

Methods

next	
reset()	Reset the broadcasted result's iterator(s).

broadcast.**next**

broadcast.**reset**()

Reset the broadcasted result's iterator(s).

Parameters
None

Returns
None

Examples

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]]
>>> b = np.broadcast(x, y)
>>> b.index
0
>>> b.next(), b.next(), b.next()
((1, 4), (2, 4), (3, 4))
>>> b.index
3
>>> b.reset()
>>> b.index
0
```

The general concept of broadcasting is also available from Python using the *broadcast* iterator. This object takes N objects as inputs and returns an iterator that returns tuples providing each of the input sequence elements in the broadcasted result.

```
>>> for val in broadcast([[1,0],[2,3]],[0,1]):
...     print val
(1, 0)
(0, 1)
(2, 0)
(3, 1)
```

1.7 Masked arrays

Masked arrays are arrays that may have missing or invalid entries. The numpy.ma module provides a nearly work-alike replacement for numpy that supports data arrays with masks.

1.7.1 The numpy.ma module

Rationale

Masked arrays are arrays that may have missing or invalid entries. The numpy.ma module provides a nearly work-alike replacement for numpy that supports data arrays with masks.

What is a masked array?

In many circumstances, datasets can be incomplete or tainted by the presence of invalid data. For example, a sensor may have failed to record a data, or recorded an invalid value. The numpy.ma module provides a convenient way to address this issue, by introducing masked arrays.

A masked array is the combination of a standard *numpy.ndarray* and a mask. A mask is either *nomask*, indicating that no value of the associated array is invalid, or an array of booleans that determines for each element of the associated array whether the value is valid or not. When an element of the mask is False, the corresponding element of the associated array is valid and is said to be unmasked. When an element of the mask is True, the corresponding element of the associated array is said to be masked (invalid).

The package ensures that masked entries are not used in computations.

As an illustration, let's consider the following dataset:

```
>>> import numpy as np
>>> import numpy.ma as ma
>>> x = np.array([1, 2, 3, -1, 5])
```

We wish to mark the fourth entry as invalid. The easiest is to create a masked array:

```
>>> mx = ma.masked_array(x, mask=[0, 0, 0, 1, 0])
```

We can now compute the mean of the dataset, without taking the invalid data into account:

```
>>> mx.mean()
2.75
```

The numpy.ma module

The main feature of the numpy.ma module is the *MaskedArray* class, which is a subclass of *numpy.ndarray*. The class, its attributes and methods are described in more details in the *MaskedArray class* section.

The numpy.ma module can be used as an addition to *numpy*:

```
>>> import numpy as np
>>> import numpy.ma as ma
```

To create an array with the second element invalid, we would do:

```
>>> y = ma.array([1, 2, 3], mask = [0, 1, 0])
```

To create a masked array where all values close to 1.e20 are invalid, we would do:

```
>>> z = masked_values([1.0, 1.e20, 3.0, 4.0], 1.e20)
```

For a complete discussion of creation methods for masked arrays please see section *Constructing masked arrays*.

1.7.2 Using numpy.ma

Constructing masked arrays

There are several ways to construct a masked array.

- A first possibility is to directly invoke the *MaskedArray* class.
- A second possibility is to use the two masked array constructors, `array` and `masked_array`.

- A third option is to take the view of an existing array. In that case, the mask of the view is set to *nomask* if the array has no named fields, or an array of boolean with the same structure as the array otherwise.

```
>>> x = np.array([1, 2, 3])
>>> x.view(ma.MaskedArray)
masked_array(data = [1 2 3],
             mask = False,
       fill_value = 999999)
>>> x = np.array([(1, 1.), (2, 2.)], dtype=[('a',int), ('b', float)])
>>> x.view(ma.MaskedArray)
masked_array(data = [(1, 1.0) (2, 2.0)],
             mask = [(False, False) (False, False)],
       fill_value = (999999, 1e+20),
            dtype = [('a', '<i4'), ('b', '<f8')])
```

- Yet another possibility is to use any of the following functions:

Accessing the data

The underlying data of a masked array can be accessed in several ways:

- through the *data* attribute. The output is a view of the array as a *numpy.ndarray* or one of its subclasses, depending on the type of the underlying data at the masked array creation.
- through the *__array__* method. The output is then a *numpy.ndarray*.
- by directly taking a view of the masked array as a *numpy.ndarray* or one of its subclass (which is actually what using the *data* attribute does).
- by using the `getdata` function.

None of these methods is completely satisfactory if some entries have been marked as invalid. As a general rule, where a representation of the array is required without any masked entries, it is recommended to fill the array with the `filled` method.

Accessing the mask

The mask of a masked array is accessible through its *mask* attribute. We must keep in mind that a `True` entry in the mask indicates an *invalid* data.

Another possibility is to use the `getmask` and `getmaskarray` functions. `getmask(x)` outputs the mask of `x` if `x` is a masked array, and the special value *nomask* otherwise. `getmaskarray(x)` outputs the mask of `x` if `x` is a masked array. If `x` has no invalid entry or is not a masked array, the function outputs a boolean array of `False` with as many elements as `x`.

Accessing only the valid entries

To retrieve only the valid entries, we can use the inverse of the mask as an index. The inverse of the mask can be calculated with the *numpy.logical_not* function or simply with the ~ operator:

```
>>> x = ma.array([[1, 2], [3, 4]], mask=[[0, 1], [1, 0]])
>>> x[~x.mask]
masked_array(data = [1 4],
             mask = [False False],
       fill_value = 999999)
```

Another way to retrieve the valid data is to use the `compressed` method, which returns a one-dimensional *ndarray* (or one of its subclasses, depending on the value of the *baseclass* attribute):

```
>>> x.compressed()
array([1, 4])
```

Note that the output of `compressed` is always 1D.

Modifying the mask

Masking an entry

The recommended way to mark one or several specific entries of a masked array as invalid is to assign the special value *masked* to them:

```
>>> x = ma.array([1, 2, 3])
>>> x[0] = ma.masked
>>> x
masked_array(data = [-- 2 3],
             mask = [ True False False],
       fill_value = 999999)
>>> y = ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> y[(0, 1, 2), (1, 2, 0)] = ma.masked
>>> y
masked_array(data =
 [[1 -- 3]
  [4 5 --]
  [-- 8 9]],
             mask =
 [[False  True False]
  [False False  True]
  [ True False False]],
       fill_value = 999999)
>>> z = ma.array([1, 2, 3, 4])
>>> z[:-2] = ma.masked
>>> z
masked_array(data = [-- -- 3 4],
             mask = [ True  True False False],
       fill_value = 999999)
```

A second possibility is to modify the *mask* directly, but this usage is discouraged.

Note: When creating a new masked array with a simple, non-structured datatype, the mask is initially set to the special value *nomask*, that corresponds roughly to the boolean `False`. Trying to set an element of *nomask* will fail with a `TypeError` exception, as a boolean does not support item assignment.

All the entries of an array can be masked at once by assigning `True` to the mask:

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x.mask = True
>>> x
masked_array(data = [-- -- --],
             mask = [ True  True  True],
       fill_value = 999999)
```

Finally, specific entries can be masked and/or unmasked by assigning to the mask a sequence of booleans:

```
>>> x = ma.array([1, 2, 3])
>>> x.mask = [0, 1, 0]
>>> x
masked_array(data = [1 -- 3],
             mask = [False  True False],
       fill_value = 999999)
```

Unmasking an entry

To unmask one or several specific entries, we can just assign one or several new valid values to them:

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x
masked_array(data = [1 2 --],
             mask = [False False  True],
       fill_value = 999999)
>>> x[-1] = 5
>>> x
masked_array(data = [1 2 5],
             mask = [False False False],
       fill_value = 999999)
```

Note: Unmasking an entry by direct assignment will silently fail if the masked array has a *hard* mask, as shown by the `hardmask` attribute. This feature was introduced to prevent overwriting the mask. To force the unmasking of an entry where the array has a hard mask, the mask must first to be softened using the `soften_mask` method before the allocation. It can be re-hardened with `harden_mask`:

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1], hard_mask=True)
>>> x
masked_array(data = [1 2 --],
             mask = [False False  True],
       fill_value = 999999)
>>> x[-1] = 5
>>> x
masked_array(data = [1 2 --],
             mask = [False False  True],
       fill_value = 999999)
>>> x.soften_mask()
>>> x[-1] = 5
>>> x
masked_array(data = [1 2 5],
             mask = [False False  False],
       fill_value = 999999)
>>> x.harden_mask()
```

To unmask all masked entries of a masked array (provided the mask isn't a hard mask), the simplest solution is to assign the constant *nomask* to the mask:

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x
masked_array(data = [1 2 --],
             mask = [False False  True],
       fill_value = 999999)
>>> x.mask = ma.nomask
>>> x
masked_array(data = [1 2 3],
             mask = [False False False],
       fill_value = 999999)
```

Indexing and slicing

As a *MaskedArray* is a subclass of *numpy.ndarray*, it inherits its mechanisms for indexing and slicing.

When accessing a single entry of a masked array with no named fields, the output is either a scalar (if the corresponding entry of the mask is `False`) or the special value *masked* (if the corresponding entry of the mask is `True`):

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x[0]
1
>>> x[-1]
masked_array(data = --,
             mask = True,
       fill_value = 1e+20)
>>> x[-1] is ma.masked
True
```

If the masked array has named fields, accessing a single entry returns a `numpy.void` object if none of the fields are masked, or a 0d masked array with the same dtype as the initial array if at least one of the fields is masked.

```
>>> y = ma.masked_array([(1,2), (3, 4)],
...                     mask=[(0, 0), (0, 1)],
...                    dtype=[('a', int), ('b', int)])
>>> y[0]
(1, 2)
>>> y[-1]
masked_array(data = (3, --),
             mask = (False, True),
       fill_value = (999999, 999999),
            dtype = [('a', '<i4'), ('b', '<i4')])
```

When accessing a slice, the output is a masked array whose *data* attribute is a view of the original data, and whose mask is either *nomask* (if there was no invalid entries in the original array) or a copy of the corresponding slice of the original mask. The copy is required to avoid propagation of any modification of the mask to the original.

```
>>> x = ma.array([1, 2, 3, 4, 5], mask=[0, 1, 0, 0, 1])
>>> mx = x[:3]
>>> mx
masked_array(data = [1 -- 3],
             mask = [False  True False],
       fill_value = 999999)
>>> mx[1] = -1
>>> mx
masked_array(data = [1 -1 3],
             mask = [False  True False],
       fill_value = 999999)
>>> x.mask
```

```
array([False,  True, False, False,  True], dtype=bool)
>>> x.data
array([ 1, -1,  3,  4,  5])
```

Accessing a field of a masked array with structured datatype returns a *MaskedArray*.

Operations on masked arrays

Arithmetic and comparison operations are supported by masked arrays. As much as possible, invalid entries of a masked array are not processed, meaning that the corresponding `data` entries *should* be the same before and after the operation.

Warning: We need to stress that this behavior may not be systematic, that masked data may be affected by the operation in some cases and therefore users should not rely on this data remaining unchanged.

The `numpy.ma` module comes with a specific implementation of most ufuncs. Unary and binary functions that have a validity domain (such as *log* or *divide*) return the *masked* constant whenever the input is masked or falls outside the validity domain:

```
>>> ma.log([-1, 0, 1, 2])
masked_array(data = [-- -- 0.0 0.69314718056],
             mask = [ True  True False False],
       fill_value = 1e+20)
```

Masked arrays also support standard numpy ufuncs. The output is then a masked array. The result of a unary ufunc is masked wherever the input is masked. The result of a binary ufunc is masked wherever any of the input is masked. If the ufunc also returns the optional context output (a 3-element tuple containing the name of the ufunc, its arguments and its domain), the context is processed and entries of the output masked array are masked wherever the corresponding input fall outside the validity domain:

```
>>> x = ma.array([-1, 1, 0, 2, 3], mask=[0, 0, 0, 0, 1])
>>> np.log(x)
masked_array(data = [-- -- 0.0 0.69314718056 --],
             mask = [ True  True False False  True],
       fill_value = 1e+20)
```

1.7.3 Examples

Data with a given value representing missing data

Let's consider a list of elements, `x`, where values of -9999. represent missing data. We wish to compute the average value of the data and the vector of anomalies (deviations from the average):

```
>>> import numpy.ma as ma
>>> x = [0.,1.,-9999.,3.,4.]
>>> mx = ma.masked_values (x, -9999.)
>>> print mx.mean()
2.0
>>> print mx - mx.mean()
[-2.0 -1.0 -- 1.0 2.0]
>>> print mx.anom()
[-2.0 -1.0 -- 1.0 2.0]
```

Filling in the missing data

Suppose now that we wish to print that same data, but with the missing values replaced by the average value.

```
>>> print mx.filled(mx.mean())
[ 0.  1.  2.  3.  4.]
```

Numerical operations

Numerical operations can be easily performed without worrying about missing values, dividing by zero, square roots of negative numbers, etc.:

```
>>> import numpy as np, numpy.ma as ma
>>> x = ma.array([1., -1., 3., 4., 5., 6.], mask=[0,0,0,0,1,0])
>>> y = ma.array([1., 2., 0., 4., 5., 6.], mask=[0,0,0,0,0,1])
>>> print np.sqrt(x/y)
[1.0 -- -- 1.0 -- --]
```

Four values of the output are invalid: the first one comes from taking the square root of a negative number, the second from the division by zero, and the last two where the inputs were masked.

Ignoring extreme values

Let's consider an array `d` of random floats between 0 and 1. We wish to compute the average of the values of `d` while ignoring any data outside the range `[0.1, 0.9]`:

```
>>> print ma.masked_outside(d, 0.1, 0.9).mean()
```

1.7.4 Constants of the `numpy.ma` module

In addition to the *MaskedArray* class, the `numpy.ma` module defines several constants.

`numpy.ma.`**`masked`**
: The *masked* constant is a special case of *MaskedArray*, with a float datatype and a null shape. It is used to test whether a specific entry of a masked array is masked, or to mask one or several entries of a masked array:

```
>>> x = ma.array([1, 2, 3], mask=[0, 1, 0])
>>> x[1] is ma.masked
True
>>> x[-1] = ma.masked
>>> x
masked_array(data = [1 -- --],
             mask = [False  True  True],
       fill_value = 999999)
```

`numpy.ma.`**`nomask`**
: Value indicating that a masked array has no invalid entry. *nomask* is used internally to speed up computations when the mask is not needed.

`numpy.ma.`**`masked_print_options`**
: String used in lieu of missing data when a masked array is printed. By default, this string is `'--'`.

1.7.5 The `MaskedArray` class

class numpy.ma.**MaskedArray**

A subclass of *ndarray* designed to manipulate numerical arrays with missing data.

An instance of *MaskedArray* can be thought as the combination of several elements:

- The *data*, as a regular *numpy.ndarray* of any shape or datatype (the data).
- A boolean *mask* with the same shape as the data, where a `True` value indicates that the corresponding element of the data is invalid. The special value *nomask* is also acceptable for arrays without named fields, and indicates that no data is invalid.
- A *fill_value*, a value that may be used to replace the invalid entries in order to return a standard *numpy.ndarray*.

Attributes and properties of masked arrays

See also:

Array Attributes

MaskedArray.**data**
Returns the underlying data, as a view of the masked array. If the underlying data is a subclass of *numpy.ndarray*, it is returned as such.

```
>>> x = ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.data
matrix([[1, 2],
        [3, 4]])
```

The type of the data can be accessed through the *baseclass* attribute.

MaskedArray.**mask**
Returns the underlying mask, as an array with the same shape and structure as the data, but where all fields are atomically booleans. A value of `True` indicates an invalid entry.

MaskedArray.**recordmask**
Returns the mask of the array if it has no named fields. For structured arrays, returns a ndarray of booleans where entries are `True` if **all** the fields are masked, `False` otherwise:

```
>>> x = ma.array([(1, 1), (2, 2), (3, 3), (4, 4), (5, 5)],
...              mask=[(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)],
...             dtype=[('a', int), ('b', int)])
>>> x.recordmask
array([False, False,  True, False, False], dtype=bool)
```

MaskedArray.**fill_value**
Returns the value used to fill the invalid entries of a masked array. The value is either a scalar (if the masked array has no named fields), or a 0-D ndarray with the same *dtype* as the masked array if it has named fields.

The default filling value depends on the datatype of the array:

datatype	default
bool	True
int	999999
float	1.e20
complex	1.e20+0j
object	'?'
string	'N/A'

MaskedArray.**baseclass**
Returns the class of the underlying data.

```
>>> x =  ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 0], [1, 0]])
>>> x.baseclass
<class 'numpy.matrixlib.defmatrix.matrix'>
```

MaskedArray.**sharedmask**
Returns whether the mask of the array is shared between several masked arrays. If this is the case, any modification to the mask of one array will be propagated to the others.

MaskedArray.**hardmask**
Returns whether the mask is hard (`True`) or soft (`False`). When the mask is hard, masked entries cannot be unmasked.

As *MaskedArray* is a subclass of *ndarray*, a masked array also inherits all the attributes and properties of a *ndarray* instance.

MaskedArray.base	Base object if memory is from some other object.
MaskedArray.ctypes	An object to simplify the interaction of the array with the ctypes module.
MaskedArray.dtype	Data-type of the array's elements.
MaskedArray.flags	Information about the memory layout of the array.
MaskedArray.itemsize	Length of one array element in bytes.
MaskedArray.nbytes	Total bytes consumed by the elements of the array.
MaskedArray.ndim	Number of array dimensions.
MaskedArray.shape	Tuple of array dimensions.
MaskedArray.size	Number of elements in the array.
MaskedArray.strides	Tuple of bytes to step in each dimension when traversing an array.
MaskedArray.imag	Imaginary part.
MaskedArray.real	Real part
MaskedArray.flat	Flat version of the array.
MaskedArray.__array_priority__	

MaskedArray.**base**
Base object if memory is from some other object.

Examples

The base of an array that owns its memory is None:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```
>>> y = x[2:]
>>> y.base is x
True
```

MaskedArray.**ctypes**

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters
None

Returns
c : Python object

Possessing attributes data, shape, strides, etc.

See also:

numpy.ctypeslib

Notes

Below are the public attributes of this object which were documented in "Guide to NumPy" (we have omitted undocumented public attributes, as well as documented private attributes):

- data: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].
- shape (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.
- strides (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.
- data_as(obj): Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).
- shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).
- strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

Examples

```
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

MaskedArray.**dtype**

Data-type of the array's elements.

Parameters
None

Returns
d : numpy dtype object

See also:

numpy.dtype

Examples

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

MaskedArray.**flags**

Information about the memory layout of the array.

Notes

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling *ndarray.setflags*.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set `False`.
- ALIGNED can only be set `True` if the data is truly aligned.

- WRITEABLE can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

Attributes

C_CONTIGUOUS (C)	The data is in a single, C-style contiguous segment.
F_CONTIGUOUS (F)	The data is in a single, Fortran-style contiguous segment.
OWNDATA (O)	The array owns the memory it uses or borrows it from another object.
WRITEABLE (W)	The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.
ALIGNED (A)	The data and all elements are aligned appropriately for the hardware.
UPDATEIFCOPY (U)	This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.
FNC	F_CONTIGUOUS and not C_CONTIGUOUS.
FORC	F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).
BEHAVED (B)	ALIGNED and WRITEABLE.
CARRAY (CA)	BEHAVED and C_CONTIGUOUS.
FARRAY (FA)	BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

MaskedArray.**itemsize**

Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

MaskedArray.**nbytes**

Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

MaskedArray.**ndim**

Number of array dimensions.

Examples

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

MaskedArray.**shape**

Tuple of array dimensions.

Notes

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

MaskedArray.**size**

Number of elements in the array.

Equivalent to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

MaskedArray.**strides**
Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the "ndarray.rst" file in the NumPy reference guide.

See also:

`numpy.lib.stride_tricks.as_strided`

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array *x* will be `(20, 4)`.

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> 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]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

MaskedArray.**imag**
Imaginary part.

MaskedArray.**real**
Real part

MaskedArray.**flat**
Flat version of the array.

MaskedArray.**__array_priority__** = 15

1.7.6 MaskedArray methods

See also:

Array methods

Conversion

MaskedArray.__hex__() <==> hex(x)	
MaskedArray.__long__() <==> long(x)	
MaskedArray.__oct__() <==> oct(x)	
MaskedArray.byteswap(inplace)	Swap the bytes of the array elements

MaskedArray.**__hex__**() <==> *hex(x)*

MaskedArray.**__long__**() <==> *long(x)*

MaskedArray.**__oct__**() <==> *oct(x)*

MaskedArray.**byteswap**(*inplace*)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

Parameters

inplace : bool, optional

If `True`, swap bytes in-place, default is `False`.

Returns

out : ndarray

The byteswapped array. If *inplace* is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([  256,     1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']
```

Arrays of strings are not swapped

```
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
      dtype='|S3')
```

Shape manipulation

For reshape, resize, and transpose, the single tuple argument may be replaced with `n` integers which will be interpreted as an n-tuple.

MaskedArray.T

MaskedArray.**T**

Item selection and manipulation

For array methods that take an *axis* keyword, it defaults to *None*. If axis is *None*, then the array is treated as a 1-D array. Any other value for *axis* represents the dimension along which the operation should proceed.

MaskedArray.choose(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
MaskedArray.fill(value)	Fill the array with a scalar value.
MaskedArray.item(*args)	Copy an element of an array to a standard Python scalar and return it.
MaskedArray.searchsorted(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.

MaskedArray.**choose**(*choices*, *out=None*, *mode='raise'*)

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See also:

numpy.choose
equivalent function

MaskedArray.**fill**(*value*)

Fill the array with a scalar value.

Parameters
value : scalar

All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1.,  1.])
```

MaskedArray.**item**(**args*)

Copy an element of an array to a standard Python scalar and return it.

Parameters
***args** : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z : Standard Python scalar object

A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

MaskedArray.**searchsorted**(*v*, *side='left'*, *sorter=None*)

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see `numpy.searchsorted`

See also:

numpy.searchsorted
equivalent function

Pickling and copy

MaskedArray.dump(file)	Dump a pickle of the array to the specified file.
MaskedArray.dumps()	Returns the pickle of the array as a string.

MaskedArray.**dump**(*file*)

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters

file : str

A string naming the dump file.

MaskedArray.**dumps**()
Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

Parameters
None

Calculations

MaskedArray.conj()	Complex-conjugate all elements.
MaskedArray.conjugate()	Return the complex conjugate, element-wise.

MaskedArray.**conj**()
Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

See also:

numpy.conjugate
equivalent function

MaskedArray.**conjugate**()
Return the complex conjugate, element-wise.

Refer to `numpy.conjugate` for full documentation.

See also:

numpy.conjugate
equivalent function

Arithmetic and comparison operations

Comparison operators:

MaskedArray.__lt__	x.__lt__(y) <==> x<y
MaskedArray.__le__	x.__le__(y) <==> x<=y
MaskedArray.__gt__	x.__gt__(y) <==> x>y
MaskedArray.__ge__	x.__ge__(y) <==> x>=y

MaskedArray.**__lt__**
x.__lt__(y) <==> x<y

MaskedArray.**__le__**
x.__le__(y) <==> x<=y

MaskedArray.**__gt__**
x.__gt__(y) <==> x>y

MaskedArray.**__ge__**
x.__ge__(y) <==> x>=y

Truth value of an array (`bool`):

MaskedArray.__nonzero__	x.__nonzero__() <==> x != 0

MaskedArray.**__nonzero__**
x.__nonzero__() <==> x != 0

Arithmetic:

MaskedArray.__abs__() <==> abs(x)	
MaskedArray.__rdiv__	x.__rdiv__(y) <==> y/x
MaskedArray.__mod__	x.__mod__(y) <==> x%y
MaskedArray.__rmod__	x.__rmod__(y) <==> y%x
MaskedArray.__divmod__(y) <==> divmod(x, y)	
MaskedArray.__rdivmod__(y) <==> divmod(y, x)	
MaskedArray.__lshift__	x.__lshift__(y) <==> x<<y
MaskedArray.__rlshift__	x.__rlshift__(y) <==> y<<x
MaskedArray.__rshift__	x.__rshift__(y) <==> x>>y
MaskedArray.__rrshift__	x.__rrshift__(y) <==> y>>x
MaskedArray.__and__	x.__and__(y) <==> x&y
MaskedArray.__rand__	x.__rand__(y) <==> y&x
MaskedArray.__or__	x.__or__(y) <==> x|y
MaskedArray.__ror__	x.__ror__(y) <==> y|x
MaskedArray.__xor__	x.__xor__(y) <==> x^y
MaskedArray.__rxor__	x.__rxor__(y) <==> y^x

MaskedArray.**__abs__**() <==> *abs(x)*

MaskedArray.**__rdiv__**
x.__rdiv__(y) <==> y/x

MaskedArray.**__mod__**
x.__mod__(y) <==> x%y

MaskedArray.**__rmod__**
x.__rmod__(y) <==> y%x

MaskedArray.**__divmod__**(*y*) <==> *divmod(x, y)*

MaskedArray.**__rdivmod__**(*y*) <==> *divmod(y, x)*

MaskedArray.**__lshift__**
x.__lshift__(y) <==> x<<y

MaskedArray.**__rlshift__**
x.__rlshift__(y) <==> y<<x

MaskedArray.**__rshift__**
x.__rshift__(y) <==> x>>y

MaskedArray.**__rrshift__**
x.__rrshift__(y) <==> y>>x

MaskedArray.__**and**__
x.__and__(y) <==> x&y

MaskedArray.__**rand**__
x.__rand__(y) <==> y&x

MaskedArray.__**or**__
x.__or__(y) <==> x|y

MaskedArray.__**ror**__
x.__ror__(y) <==> y|x

MaskedArray.__**xor**__
x.__xor__(y) <==> x^y

MaskedArray.__**rxor**__
x.__rxor__(y) <==> y^x

Arithmetic, in-place:

MaskedArray.__imod__	x.__imod__(y) <==> x%=y
MaskedArray.__ilshift__	x.__ilshift__(y) <==> x<<=y
MaskedArray.__irshift__	x.__irshift__(y) <==> x>>=y
MaskedArray.__iand__	x.__iand__(y) <==> x&=y
MaskedArray.__ior__	x.__ior__(y) <==> x|=y
MaskedArray.__ixor__	x.__ixor__(y) <==> x^=y

MaskedArray.__**imod**__
x.__imod__(y) <==> x%=y

MaskedArray.__**ilshift**__
x.__ilshift__(y) <==> x<<=y

MaskedArray.__**irshift**__
x.__irshift__(y) <==> x>>=y

MaskedArray.__**iand**__
x.__iand__(y) <==> x&=y

MaskedArray.__**ior**__
x.__ior__(y) <==> x|=y

MaskedArray.__**ixor**__
x.__ixor__(y) <==> x^=y

Representation

Special methods

For standard library functions:

MaskedArray.__copy__([order])	Return a copy of the array.

MaskedArray.__**copy**__(*[order]*)

Return a copy of the array.

Parameters

order : {'C', 'F', 'A'}, optional

If order is 'C' (False) then the result is contiguous (default). If order is 'Fortran' (True) then the result has fortran order. If order is 'Any' (None) then the result has fortran order only if the array already is in fortran order.

Basic customization:

MaskedArray.__array__(...)	Returns either a new reference to self if dtype is not given or a new array of provided data type i

MaskedArray.**__array__**(|*dtype*) → reference if type unchanged, copy otherwise.

Returns either a new reference to self if dtype is not given or a new array of provided data type if dtype is different from the current dtype of the array.

Container customization: (see *Indexing*)

MaskedArray.__len__() <==> len(x)	
MaskedArray.__delitem__	x.__delitem__(y) <==> del x[y]
MaskedArray.__contains__	x.__contains__(y) <==> y in x

MaskedArray.**__len__**(*x*) <==> *len(x)*

MaskedArray.**__delitem__**

x.__delitem__(y) <==> del x[y]

MaskedArray.**__contains__**

x.__contains__(y) <==> y in x

Specific methods

Handling the mask

The following methods can be used to access information about the mask or to manipulate the mask.

Handling the *fill_value*

Counting the missing elements

1.7.7 Masked array operations

Constants

Creation

From existing data

Ones and zeros

Inspecting the array

ma.MaskedArray.data	Return the current data, as a view of the original underlying data.
ma.MaskedArray.mask	Mask
ma.MaskedArray.recordmask	Return the mask of the records.

MaskedArray.**data**
Return the current data, as a view of the original underlying data.

MaskedArray.**mask**
Mask

MaskedArray.**recordmask**
Return the mask of the records.

A record is masked when all the fields are masked.

Manipulating a MaskedArray

Changing the shape

Modifying axes

Changing the number of dimensions

Joining arrays

Operations on masks

Creating a mask

Accessing a mask

ma.masked_array.mask	Mask

masked_array.**mask**
Mask

Finding masked data

Modifying a mask

Conversion operations

> to a masked array

> to a ndarray

> to another object

Pickling and unpickling

Filling a masked array

ma.MaskedArray.fill_value	Filling value.

MaskedArray.**fill_value**
Filling value.

Masked arrays arithmetics

Arithmetics

Minimum/maximum

Sorting

Algebra

Polynomial fit

Clipping and rounding

Miscellanea

1.8 The Array Interface

Note: This page describes the numpy-specific API for accessing the contents of a numpy array from other C extensions. **PEP 3118** – `The Revised Buffer Protocol` introduces similar, standardized API to Python 2.6 and 3.0 for any extension module to use. Cython's buffer array support uses the **PEP 3118** API; see the Cython numpy tutorial. Cython provides a way to write code that supports the buffer protocol with Python versions older than 2.6 because it has a backward-compatible implementation utilizing the array interface described here.

version
3

The array interface (sometimes called array protocol) was created in 2005 as a means for array-like Python objects to re-use each other's data buffers intelligently whenever possible. The homogeneous N-dimensional array interface is a default mechanism for objects to share N-dimensional array memory and information. The interface consists of a Python-side and a C-side using two attributes. Objects wishing to be considered an N-dimensional array in application code should support at least one of these attributes. Objects wishing to support an N-dimensional array in application code should look for at least one of these attributes and use the information provided appropriately.

This interface describes homogeneous arrays in the sense that each item of the array has the same "type". This type can be very simple or it can be a quite arbitrary and complicated C-like structure.

There are two ways to use the interface: A Python side and a C-side. Both are separate attributes.

1.8.1 Python side

This approach to the interface consists of the object having an *__array_interface__* attribute.

__array_interface__
A dictionary of items (3 required and 5 optional). The optional keys in the dictionary have implied defaults if they are not provided.

The keys are:

shape (required)

Tuple whose elements are the array size in each dimension. Each entry is an integer (a Python int or long). Note that these integers could be larger than the platform "int" or "long" could hold (a Python int is a C long). It is up to the code using this attribute to handle this appropriately; either by raising an error when overflow is possible, or by using `Py_LONG_LONG` as the C type for the shapes.

typestr (required)

A string providing the basic type of the homogenous array The basic string format consists of 3 parts: a character describing the byteorder of the data (<: little-endian, >: big-endian, |: not-relevant), a character code giving the basic type of the array, and an integer providing the number of bytes the type uses.

The basic type character codes are:

`t`	Bit field (following integer gives the number of bits in the bit field).
`b`	Boolean (integer type where all values are only True or False)
`i`	Integer
`u`	Unsigned integer
`f`	Floating point
`c`	Complex floating point
`m`	Timedelta
`M`	Datetime
`O`	Object (i.e. the memory contains a pointer to `PyObject`)
`S`	String (fixed-length sequence of char)
`U`	Unicode (fixed-length sequence of `Py_UNICODE`)
`V`	Other (void * – each item is a fixed-size chunk of memory)

descr (optional)

A list of tuples providing a more detailed description of the memory layout for each item in the homogeneous array. Each tuple in the list has two or three elements. Normally, this attribute would be used when *typestr* is `V[0-9]+`, but this is not a requirement. The only requirement is that the number of bytes represented in the *typestr* key is the same as the total number of bytes represented here. The idea is to support descriptions of C-like structs that make up array elements. The elements of each tuple in the list are

1. A string providing a name associated with this portion of the datatype. This could also be a tuple of `('full name', 'basic_name')` where basic name would be a valid Python variable name representing the full name of the field.
2. Either a basic-type description string as in *typestr* or another list (for nested structured types)
3. An optional shape tuple providing how many times this part of the structure should be repeated. No repeats are assumed if this is not given. Very complicated structures can be described using this generic interface. Notice, however, that each element of the array is still of the same data-type. Some examples of using this interface are given below.

Default: `[('', typestr)]`

data (optional)

A 2-tuple whose first argument is an integer (a long integer if necessary) that points to the data-area storing the array contents. This pointer must point to the first element of data (in other words any offset is always ignored in this case). The second entry in the tuple is a read-only flag (true means the data area is read-only).

This attribute can also be an object exposing the `buffer interface` which will be used to share the data. If this key is not present (or returns `None`), then memory sharing will be done through the buffer interface of the object itself. In this case, the offset key can be used to indicate the start of the

buffer. A reference to the object exposing the array interface must be stored by the new object if the memory area is to be secured.

Default: `None`

strides (optional)

Either `None` to indicate a C-style contiguous array or a Tuple of strides which provides the number of bytes needed to jump to the next array element in the corresponding dimension. Each entry must be an integer (a Python `int` or `long`). As with shape, the values may be larger than can be represented by a C "int" or "long"; the calling code should handle this appropiately, either by raising an error, or by using `Py_LONG_LONG` in C. The default is `None` which implies a C-style contiguous memory buffer. In this model, the last dimension of the array varies the fastest. For example, the default strides tuple for an object whose array entries are 8 bytes long and whose shape is (10,20,30) would be (4800, 240, 8)

Default: `None` (C-style contiguous)

mask (optional)

`None` or an object exposing the array interface. All elements of the mask array should be interpreted only as true or not true indicating which elements of this array are valid. The shape of this object should be *"broadcastable"* to the shape of the original array.

Default: `None` (All array values are valid)

offset (optional)

An integer offset into the array data region. This can only be used when data is `None` or returns a `buffer` object.

Default: 0.

version (required)

An integer showing the version of the interface (i.e. 3 for this version). Be careful not to use this to invalidate objects exposing future versions of the interface.

1.8.2 C-struct access

This approach to the array interface allows for faster access to an array using only one attribute lookup and a well-defined C-structure.

`__array_struct__`
A :c:type: *PyCObject* whose `voidptr` member contains a pointer to a filled *`PyArrayInterface`* structure. Memory for the structure is dynamically created and the `PyCObject` is also created with an appropriate destructor so the retriever of this attribute simply has to apply `Py_DECREF` to the object returned by this attribute when it is finished. Also, either the data needs to be copied out, or a reference to the object exposing this attribute must be held to ensure the data is not freed. Objects exposing the `__array_struct__` interface must also not reallocate their memory if other objects are referencing them.

The PyArrayInterface structure is defined in `numpy/ndarrayobject.h` as:

```
typedef struct {
  int two;              /* contains the integer 2 -- simple sanity check */
  int nd;               /* number of dimensions */
  char typekind;        /* kind in array --- character code of typestr */
  int itemsize;         /* size of each element */
  int flags;            /* flags indicating how the data should be interpreted */
                        /*   must set ARR_HAS_DESCR bit to validate descr */
  Py_intptr_t *shape;   /* A length-nd array of shape information */
```

```
  Py_intptr_t *strides; /* A length-nd array of stride information */
  void *data;           /* A pointer to the first element of the array */
  PyObject *descr;      /* NULL or data-description (same as descr key
                                of __array_interface__) -- must set ARR_HAS_DESCR
                                flag or this will be ignored. */
} PyArrayInterface;
```

The flags member may consist of 5 bits showing how the data should be interpreted and one bit showing how the Interface should be interpreted. The data-bits are `CONTIGUOUS` (0x1), `FORTRAN` (0x2), `ALIGNED` (0x100), `NOTSWAPPED` (0x200), and `WRITEABLE` (0x400). A final flag `ARR_HAS_DESCR` (0x800) indicates whether or not this structure has the arrdescr field. The field should not be accessed unless this flag is present.

New since June 16, 2006:

In the past most implementations used the "desc" member of the `PyCObject` itself (do not confuse this with the "descr" member of the *`PyArrayInterface`* structure above — they are two separate things) to hold the pointer to the object exposing the interface. This is now an explicit part of the interface. Be sure to own a reference to the object when the `PyCObject` is created using `PyCObject_FromVoidPtrAndDesc`.

1.8.3 Type description examples

For clarity it is useful to provide some examples of the type description and corresponding *`__array_interface__`* 'descr' entries. Thanks to Scott Gilbert for these examples:

In every case, the 'descr' key is optional, but of course provides more information which may be important for various applications:

```
* Float data
    typestr == '>f4'
    descr == [('','>f4')]

* Complex double
    typestr == '>c8'
    descr == [('real','>f4'), ('imag','>f4')]

* RGB Pixel data
    typestr == '|V3'
    descr == [('r','|u1'), ('g','|u1'), ('b','|u1')]

* Mixed endian (weird but could happen).
    typestr == '|V8' (or '>u8')
    descr == [('big','>i4'), ('little','<i4')]

* Nested structure
    struct {
        int ival;
        struct {
            unsigned short sval;
            unsigned char bval;
            unsigned char cval;
        } sub;
    }
    typestr == '|V8' (or '<u8' if you want)
    descr == [('ival','<i4'), ('sub', [('sval','<u2'), ('bval','|u1'), ('cval','|u1') ]) ]

* Nested array
```

```
    struct {
        int ival;
        double data[16*4];
    }
    typestr == '|V516'
    descr == [('ival','>i4'), ('data','>f8',(16,4))]

* Padded structure
    struct {
        int ival;
        double dval;
    }
    typestr == '|V16'
    descr == [('ival','>i4'),('','|V4'),('dval','>f8')]
```

It should be clear that any structured type could be described using this interface.

1.8.4 Differences with Array interface (Version 2)

The version 2 interface was very similar. The differences were largely aesthetic. In particular:

1. The PyArrayInterface structure had no descr member at the end (and therefore no flag ARR_HAS_DESCR)
2. The desc member of the PyCObject returned from __array_struct__ was not specified. Usually, it was the object exposing the array (so that a reference to it could be kept and destroyed when the C-object was destroyed). Now it must be a tuple whose first element is a string with "PyArrayInterface Version #" and whose second element is the object exposing the array.
3. The tuple returned from __array_interface__['data'] used to be a hex-string (now it is an integer or a long integer).
4. There was no __array_interface__ attribute instead all of the keys (except for version) in the __array_interface__ dictionary were their own attribute: Thus to obtain the Python-side information you had to access separately the attributes:
    - __array_data__
    - __array_shape__
    - __array_strides__
    - __array_typestr__
    - __array_descr__
    - __array_offset__
    - __array_mask__

1.9 Datetimes and Timedeltas

New in version 1.7.0.

Starting in NumPy 1.7, there are core array data types which natively support datetime functionality. The data type is called "datetime64", so named because "datetime" is already taken by the datetime library included in Python.

Note: The datetime API is *experimental* in 1.7.0, and may undergo changes in future versions of NumPy.

1.9.1 Basic Datetimes

The most basic way to create datetimes is from strings in ISO 8601 date or datetime format. The unit for internal storage is automatically selected from the form of the string, and can be either a *date unit* or a *time unit*. The date units are years ('Y'), months ('M'), weeks ('W'), and days ('D'), while the time units are hours ('h'), minutes ('m'), seconds ('s'), milliseconds ('ms'), and some additional SI-prefix seconds-based units.

Example

A simple ISO date:

```
>>> np.datetime64('2005-02-25')
numpy.datetime64('2005-02-25')
```

Using months for the unit:

```
>>> np.datetime64('2005-02')
numpy.datetime64('2005-02')
```

Specifying just the month, but forcing a 'days' unit:

```
>>> np.datetime64('2005-02', 'D')
numpy.datetime64('2005-02-01')
```

From a date and time:

```
>>> np.datetime64('2005-02-25T03:30')
numpy.datetime64('2005-02-25T03:30')
```

When creating an array of datetimes from a string, it is still possible to automatically select the unit from the inputs, by using the datetime type with generic units.

Example

```
>>> np.array(['2007-07-13', '2006-01-13', '2010-08-13'], dtype='datetime64')
array(['2007-07-13', '2006-01-13', '2010-08-13'], dtype='datetime64[D]')
```

```
>>> np.array(['2001-01-01T12:00', '2002-02-03T13:56:03.172'], dtype='datetime64')
array(['2001-01-01T12:00:00.000-0600', '2002-02-03T13:56:03.172-0600'], dtype='datetime64[ms]')
```

The datetime type works with many common NumPy functions, for example `arange` can be used to generate ranges of dates.

Example

All the dates for one month:

```
>>> np.arange('2005-02', '2005-03', dtype='datetime64[D]')
array(['2005-02-01', '2005-02-02', '2005-02-03', '2005-02-04',
       '2005-02-05', '2005-02-06', '2005-02-07', '2005-02-08',
       '2005-02-09', '2005-02-10', '2005-02-11', '2005-02-12',
       '2005-02-13', '2005-02-14', '2005-02-15', '2005-02-16',
       '2005-02-17', '2005-02-18', '2005-02-19', '2005-02-20',
       '2005-02-21', '2005-02-22', '2005-02-23', '2005-02-24',
       '2005-02-25', '2005-02-26', '2005-02-27', '2005-02-28'],
      dtype='datetime64[D]')
```

The datetime object represents a single moment in time. If two datetimes have different units, they may still be representing the same moment of time, and converting from a bigger unit like months to a smaller unit like days is considered a 'safe' cast because the moment of time is still being represented exactly.

Example

```
>>> np.datetime64('2005') == np.datetime64('2005-01-01')
True
```

```
>>> np.datetime64('2010-03-14T15Z') == np.datetime64('2010-03-14T15:00:00.00Z')
True
```

1.9.2 Datetime and Timedelta Arithmetic

NumPy allows the subtraction of two Datetime values, an operation which produces a number with a time unit. Because NumPy doesn't have a physical quantities system in its core, the timedelta64 data type was created to complement datetime64.

Datetimes and Timedeltas work together to provide ways for simple datetime calculations.

Example

```
>>> np.datetime64('2009-01-01') - np.datetime64('2008-01-01')
numpy.timedelta64(366,'D')
```

```
>>> np.datetime64('2009') + np.timedelta64(20, 'D')
numpy.datetime64('2009-01-21')
```

```
>>> np.datetime64('2011-06-15T00:00') + np.timedelta64(12, 'h')
numpy.datetime64('2011-06-15T12:00-0500')
```

```
>>> np.timedelta64(1,'W') / np.timedelta64(1,'D')
7.0
```

There are two Timedelta units ('Y', years and 'M', months) which are treated specially, because how much time they represent changes depending on when they are used. While a timedelta day unit is equivalent to 24 hours, there is no way to convert a month unit into days, because different months have different numbers of days.

Example

```
>>> a = np.timedelta64(1, 'Y')
```

```
>>> np.timedelta64(a, 'M')
numpy.timedelta64(12,'M')
```

```
>>> np.timedelta64(a, 'D')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Cannot cast NumPy timedelta64 scalar from metadata [Y] to [D] according to the rule 'same_
```

1.9.3 Datetime Units

The Datetime and Timedelta data types support a large number of time units, as well as generic units which can be coerced into any of the other units based on input data.

Datetimes are always stored based on POSIX time (though having a TAI mode which allows for accounting of leap-seconds is proposed), with a epoch of 1970-01-01T00:00Z. This means the supported dates are always a symmetric interval around the epoch, called "time span" in the table below.

The length of the span is the range of a 64-bit integer times the length of the date or unit. For example, the time span for 'W' (week) is exactly 7 times longer than the time span for 'D' (day), and the time span for 'D' (day) is exactly 24 times longer than the time span for 'h' (hour).

Here are the date units:

Code	Meaning	Time span (relative)	Time span (absolute)
Y	year	+/- 9.2e18 years	[9.2e18 BC, 9.2e18 AD]
M	month	+/- 7.6e17 years	[7.6e17 BC, 7.6e17 AD]
W	week	+/- 1.7e17 years	[1.7e17 BC, 1.7e17 AD]
D	day	+/- 2.5e16 years	[2.5e16 BC, 2.5e16 AD]

And here are the time units:

Code	Meaning	Time span (relative)	Time span (absolute)
h	hour	+/- 1.0e15 years	[1.0e15 BC, 1.0e15 AD]
m	minute	+/- 1.7e13 years	[1.7e13 BC, 1.7e13 AD]
s	second	+/- 2.9e12 years	[2.9e9 BC, 2.9e9 AD]
ms	millisecond	+/- 2.9e9 years	[2.9e6 BC, 2.9e6 AD]
us	microsecond	+/- 2.9e6 years	[290301 BC, 294241 AD]
ns	nanosecond	+/- 292 years	[1678 AD, 2262 AD]
ps	picosecond	+/- 106 days	[1969 AD, 1970 AD]
fs	femtosecond	+/- 2.6 hours	[1969 AD, 1970 AD]
as	attosecond	+/- 9.2 seconds	[1969 AD, 1970 AD]

1.9.4 Business Day Functionality

To allow the datetime to be used in contexts where only certain days of the week are valid, NumPy includes a set of "busday" (business day) functions.

The default for busday functions is that the only valid days are Monday through Friday (the usual business days). The implementation is based on a "weekmask" containing 7 Boolean flags to indicate valid days; custom weekmasks are possible that specify other sets of valid days.

The "busday" functions can additionally check a list of "holiday" dates, specific dates that are not valid days.

The function `busday_offset` allows you to apply offsets specified in business days to datetimes with a unit of 'D' (day).

Example

```
>>> np.busday_offset('2011-06-23', 1)
numpy.datetime64('2011-06-24')
```

```
>>> np.busday_offset('2011-06-23', 2)
numpy.datetime64('2011-06-27')
```

When an input date falls on the weekend or a holiday, `busday_offset` first applies a rule to roll the date to a valid business day, then applies the offset. The default rule is 'raise', which simply raises an exception. The rules most typically used are 'forward' and 'backward'.

Example

```
>>> np.busday_offset('2011-06-25', 2)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
ValueError: Non-business day date in busday_offset
```

```
>>> np.busday_offset('2011-06-25', 0, roll='forward')
numpy.datetime64('2011-06-27')
```

```
>>> np.busday_offset('2011-06-25', 2, roll='forward')
numpy.datetime64('2011-06-29')
```

```
>>> np.busday_offset('2011-06-25', 0, roll='backward')
numpy.datetime64('2011-06-24')
```

```
>>> np.busday_offset('2011-06-25', 2, roll='backward')
numpy.datetime64('2011-06-28')
```

In some cases, an appropriate use of the roll and the offset is necessary to get a desired answer.

Example

The first business day on or after a date:

```
>>> np.busday_offset('2011-03-20', 0, roll='forward')
numpy.datetime64('2011-03-21','D')
>>> np.busday_offset('2011-03-22', 0, roll='forward')
numpy.datetime64('2011-03-22','D')
```

The first business day strictly after a date:

```
>>> np.busday_offset('2011-03-20', 1, roll='backward')
numpy.datetime64('2011-03-21','D')
>>> np.busday_offset('2011-03-22', 1, roll='backward')
numpy.datetime64('2011-03-23','D')
```

The function is also useful for computing some kinds of days like holidays. In Canada and the U.S., Mother's day is on the second Sunday in May, which can be computed with a custom weekmask.

Example

```
>>> np.busday_offset('2012-05', 1, roll='forward', weekmask='Sun')
numpy.datetime64('2012-05-13','D')
```

When performance is important for manipulating many business dates with one particular choice of weekmask and holidays, there is an object *busdaycalendar* which stores the data necessary in an optimized form.

np.is_busday():

To test a datetime64 value to see if it is a valid day, use `is_busday`.

Example

```
>>> np.is_busday(np.datetime64('2011-07-15'))  # a Friday
True
>>> np.is_busday(np.datetime64('2011-07-16')) # a Saturday
False
>>> np.is_busday(np.datetime64('2011-07-16'), weekmask="Sat Sun")
True
```

```
>>> a = np.arange(np.datetime64('2011-07-11'), np.datetime64('2011-07-18'))
>>> np.is_busday(a)
array([ True,  True,  True,  True,  True, False, False], dtype='bool')
```

np.busday_count():

To find how many valid days there are in a specified range of datetime64 dates, use `busday_count`:

Example

```
>>> np.busday_count(np.datetime64('2011-07-11'), np.datetime64('2011-07-18'))
5
>>> np.busday_count(np.datetime64('2011-07-18'), np.datetime64('2011-07-11'))
-5
```

If you have an array of datetime64 day values, and you want a count of how many of them are valid dates, you can do this:

Example

```
>>> a = np.arange(np.datetime64('2011-07-11'), np.datetime64('2011-07-18'))
>>> np.count_nonzero(np.is_busday(a))
5
```

Custom Weekmasks

Here are several examples of custom weekmask values. These examples specify the "busday" default of Monday through Friday being valid days.

Some examples:

```
# Positional sequences; positions are Monday through Sunday.
# Length of the sequence must be exactly 7.
weekmask = [1, 1, 1, 1, 1, 0, 0]
# list or other sequence; 0 == invalid day, 1 == valid day
weekmask = "1111100"
# string '0' == invalid day, '1' == valid day

# string abbreviations from this list: Mon Tue Wed Thu Fri Sat Sun
weekmask = "Mon Tue Wed Thu Fri"
# any amount of whitespace is allowed; abbreviations are case-sensitive.
weekmask = "MonTue Wed  Thu\tFri"
```

1.9.5 Changes with NumPy 1.11

In prior versions of NumPy, the 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 concensus 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 f
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 timeunits. With timezone naive datetimes, the rule for casting from dates to times is no longer ambiguous.

pandas_: http://pandas.pydata.org

1.9.6 Differences Between 1.6 and 1.7 Datetimes

The NumPy 1.6 release includes a more primitive datetime data type than 1.7. This section documents many of the changes that have taken place.

String Parsing

The datetime string parser in NumPy 1.6 is very liberal in what it accepts, and silently allows invalid input without raising errors. The parser in NumPy 1.7 is quite strict about only accepting ISO 8601 dates, with a few convenience extensions. 1.6 always creates microsecond (us) units by default, whereas 1.7 detects a unit based on the format of the string. Here is a comparison.:

```
# NumPy 1.6.1
>>> np.datetime64('1979-03-22')
1979-03-22 00:00:00
# NumPy 1.7.0
>>> np.datetime64('1979-03-22')
numpy.datetime64('1979-03-22')

# NumPy 1.6.1, unit default microseconds
>>> np.datetime64('1979-03-22').dtype
dtype('datetime64[us]')
# NumPy 1.7.0, unit of days detected from string
>>> np.datetime64('1979-03-22').dtype
dtype('<M8[D]')

# NumPy 1.6.1, ignores invalid part of string
>>> np.datetime64('1979-03-2corruptedstring')
1979-03-02 00:00:00
# NumPy 1.7.0, raises error for invalid input
>>> np.datetime64('1979-03-2corruptedstring')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Error parsing datetime string "1979-03-2corruptedstring" at position 8

# NumPy 1.6.1, 'nat' produces today's date
>>> np.datetime64('nat')
2012-04-30 00:00:00
# NumPy 1.7.0, 'nat' produces not-a-time
```

```
>>> np.datetime64('nat')
numpy.datetime64('NaT')

# NumPy 1.6.1, 'garbage' produces today's date
>>> np.datetime64('garbage')
2012-04-30 00:00:00
# NumPy 1.7.0, 'garbage' raises an exception
>>> np.datetime64('garbage')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Error parsing datetime string "garbage" at position 0

# NumPy 1.6.1, can't specify unit in scalar constructor
>>> np.datetime64('1979-03-22T19:00', 'h')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function takes at most 1 argument (2 given)
# NumPy 1.7.0, unit in scalar constructor
>>> np.datetime64('1979-03-22T19:00', 'h')
numpy.datetime64('1979-03-22T19:00-0500','h')

# NumPy 1.6.1, reads ISO 8601 strings w/o TZ as UTC
>>> np.array(['1979-03-22T19:00'], dtype='M8[h]')
array([1979-03-22 19:00:00], dtype=datetime64[h])
# NumPy 1.7.0, reads ISO 8601 strings w/o TZ as local (ISO specifies this)
>>> np.array(['1979-03-22T19:00'], dtype='M8[h]')
array(['1979-03-22T19-0500'], dtype='datetime64[h]')

# NumPy 1.6.1, doesn't parse all ISO 8601 strings correctly
>>> np.array(['1979-03-22T12'], dtype='M8[h]')
array([1979-03-22 00:00:00], dtype=datetime64[h])
>>> np.array(['1979-03-22T12:00'], dtype='M8[h]')
array([1979-03-22 12:00:00], dtype=datetime64[h])
# NumPy 1.7.0, handles this case correctly
>>> np.array(['1979-03-22T12'], dtype='M8[h]')
array(['1979-03-22T12-0500'], dtype='datetime64[h]')
>>> np.array(['1979-03-22T12:00'], dtype='M8[h]')
array(['1979-03-22T12-0500'], dtype='datetime64[h]')
```

Unit Conversion

The 1.6 implementation of datetime does not convert between units correctly.:

```
# NumPy 1.6.1, the representation value is untouched
>>> np.array(['1979-03-22'], dtype='M8[D]')
array([1979-03-22 00:00:00], dtype=datetime64[D])
>>> np.array(['1979-03-22'], dtype='M8[D]').astype('M8[M]')
array([2250-08-01 00:00:00], dtype=datetime64[M])
# NumPy 1.7.0, the representation is scaled accordingly
>>> np.array(['1979-03-22'], dtype='M8[D]')
array(['1979-03-22'], dtype='datetime64[D]')
>>> np.array(['1979-03-22'], dtype='M8[D]').astype('M8[M]')
array(['1979-03'], dtype='datetime64[M]')
```

Datetime Arithmetic

The 1.6 implementation of datetime only works correctly for a small subset of arithmetic operations. Here we show some simple cases.:

```
# NumPy 1.6.1, produces invalid results if units are incompatible
>>> a = np.array(['1979-03-22T12'], dtype='M8[h]')
>>> b = np.array([3*60], dtype='m8[m]')
>>> a + b
array([1970-01-01 00:00:00.080988], dtype=datetime64[us])
# NumPy 1.7.0, promotes to higher-resolution unit
>>> a = np.array(['1979-03-22T12'], dtype='M8[h]')
>>> b = np.array([3*60], dtype='m8[m]')
>>> a + b
array(['1979-03-22T15:00-0500'], dtype='datetime64[m]')

# NumPy 1.6.1, arithmetic works if everything is microseconds
>>> a = np.array(['1979-03-22T12:00'], dtype='M8[us]')
>>> b = np.array([3*60*60*1000000], dtype='m8[us]')
>>> a + b
array([1979-03-22 15:00:00], dtype=datetime64[us])
# NumPy 1.7.0
>>> a = np.array(['1979-03-22T12:00'], dtype='M8[us]')
>>> b = np.array([3*60*60*1000000], dtype='m8[us]')
>>> a + b
array(['1979-03-22T15:00:00.000000-0500'], dtype='datetime64[us]')
```

CHAPTER

TWO

UNIVERSAL FUNCTIONS (UFUNC)

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 scalar inputs and produces a fixed number of scalar outputs.

In Numpy, universal functions are instances of the `numpy.ufunc` class. Many of the built-in functions are implemented in compiled C code, but `ufunc` instances can also be produced using the `frompyfunc` factory function.

2.1 Broadcasting

Each universal function takes array inputs and produces array outputs by performing the core function element-wise on the inputs. Standard broadcasting rules are applied so that inputs not sharing exactly the same shapes can still be usefully operated on. Broadcasting can be understood by four rules:

1. All input arrays with *ndim* smaller than the input array of largest *ndim*, have 1's prepended to their shapes.
2. The size in each dimension of the output shape is the maximum of all the input sizes in that dimension.
3. An input can be used in the calculation if its size in a particular dimension either matches the output size in that dimension, or has value exactly 1.
4. 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).

Broadcasting is used throughout NumPy to decide how to handle disparately shaped arrays; for example, all arithmetic operations (`+`, `-`, `*`, ...) between *ndarrays* broadcast the arrays before operation. A set of arrays is called "broadcastable" to the same shape if the above rules produce a valid result, *i.e.*, one of the following is true:

1. The arrays all have exactly the same shape.
2. The arrays all have the same number of dimensions and the length of each dimensions is either a common length or 1.
3. The arrays that have too few dimensions can have their shapes prepended with a dimension of length 1 to satisfy property 2.

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.

2.2 Output type determination

The output of the ufunc (and its methods) is not necessarily an *ndarray*, if all input arguments are not *ndarrays*.

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 1.0. 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.

2.3 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

2.4 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

2.5 Casting Rules

Note: In NumPy 1.6.0, a type promotion API was created to encapsulate the mechansim for determining output types. See the functions result_type, promote_types, and 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 `ldexp` 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 Figure 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 Figure.

Figure

Code segment showing the "can cast safely" table for a 32-bit system.

```
>>> def print_table(ntypes):
...     print 'X',
...     for char in ntypes: print char,
...     print
...     for row in ntypes:
...         print row,
...         for col in ntypes:
...             print int(np.can_cast(row, col)),
...         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
? 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
b 0 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0
h 0 0 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0
i 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1 0 0
l 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1 0 0
q 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1 0 0
p 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1 0 0
B 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
H 0 0 0 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 0 0
I 0 0 0 0 1 1 1 0 0 1 1 1 1 0 0 1 1 0 1 1 1 1 1 1 0 0
L 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 0 0
Q 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 0 0
P 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 0 0
e 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0
f 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0
d 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1 0 0
g 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 0
F 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0
D 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0
G 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0
S 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0
U 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0
V 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
O 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
M 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
m 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
```

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.

2.6 Overriding Ufunc behavior

Classes (including ndarray subclasses) can override how ufuncs act on them by defining certain special methods. For details, see *Standard array subclasses*.

2.7 `ufunc`

2.7.1 Optional keyword arguments

All ufuncs take optional keyword arguments. Most of these represent advanced usage and will not typically be used.

out

New in version 1.6.

The first output can be provided as either a positional or a keyword parameter. Keyword 'out' arguments are incompatible with positional ones.

..versionadded:: 1.10

The 'out' keyword argument is expected to be a tuple with one entry per output (which can be *None* for arrays to be allocated by the ufunc). For ufuncs with a single output, passing a single array (instead of a tuple holding a single array) is also valid.

Passing a single array in the 'out' keyword argument to a ufunc with multiple outputs is deprecated, and will raise a warning in numpy 1.10, and an error in a future release.

where

New in version 1.7.

Accepts a boolean array which is broadcast together with the operands. Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

casting

New in version 1.6.

May be 'no', 'equiv', 'safe', 'same_kind', or 'unsafe'. See `can_cast` for explanations of the parameter values.

Provides a policy for what kind of casting is permitted. For compatibility with previous versions of NumPy, this defaults to 'unsafe' for numpy < 1.7. In numpy 1.7 a transition to 'same_kind' was begun where ufuncs produce a DeprecationWarning for calls which are allowed under the 'unsafe' rules, but not under the 'same_kind' rules. From numpy 1.10 and onwards, the default is 'same_kind'.

order

New in version 1.6.

Specifies the calculation iteration order/memory layout of the output array. Defaults to 'K'. 'C' means the output should be C-contiguous, 'F' means F-contiguous, 'A' means F-contiguous if the inputs are F-contiguous and not also not C-contiguous, C-contiguous otherwise, and 'K' means to match the element ordering of the inputs as closely as possible.

dtype

New in version 1.6.

Overrides the dtype of the calculation and output arrays. Similar to *signature*.

subok

New in version 1.6.

Defaults to true. If set to false, the output will always be a strict array, not a subtype.

signature

Either a data-type, a tuple of data-types, or a special signature string indicating the input and output types of a ufunc. This argument allows you to provide a specific signature for the 1-d loop to use in the underlying calculation. If the loop specified does not exist for the ufunc, then a TypeError is raised. Normally, a suitable loop is found automatically by comparing the input types with what is available and searching for a loop with data-types to which all inputs can be cast safely. This keyword argument lets you bypass that search and choose a particular loop. A list of available signatures is provided by the **types** attribute of the ufunc object. For backwards compatibility this argument can also be provided as *sig*, although the long form is preferred.

extobj

a list of length 1, 2, or 3 specifying the ufunc buffer-size, the error mode integer, and the error call-back function. Normally, these values are looked up in a thread-specific dictionary. Passing them here circumvents that look up and uses the low-level specification provided for the error mode. This may be useful, for example, as an optimization for calculations requiring many ufunc calls on small arrays in a loop.

2.7.2 Attributes

There are some informational attributes that universal functions possess. None of the attributes can be set.

__doc__	A docstring for each ufunc. The first part of the docstring is dynamically generated from the number of outputs, the name, and the number of inputs. The second part of the docstring is provided at creation time and stored with the ufunc.
__name__	The name of the ufunc.

ufunc.nin	The number of inputs.
ufunc.nout	The number of outputs.
ufunc.nargs	The number of arguments.
ufunc.ntypes	The number of types.
ufunc.types	Returns a list with types grouped input->output.
ufunc.identity	The identity value.

ufunc.**nin**
The number of inputs.

Data attribute containing the number of arguments the ufunc treats as input.

Examples

```
>>> np.add.nin
2
>>> np.multiply.nin
2
>>> np.power.nin
2
>>> np.exp.nin
1
```

ufunc.**nout**
The number of outputs.

Data attribute containing the number of arguments the ufunc treats as output.

Notes

Since all ufuncs can take output arguments, this will always be (at least) 1.

Examples

```
>>> np.add.nout
1
>>> np.multiply.nout
1
>>> np.power.nout
1
>>> np.exp.nout
1
```

ufunc.**nargs**
The number of arguments.

Data attribute containing the number of arguments the ufunc takes, including optional ones.

Notes

Typically this value will be one more than what you might expect because all ufuncs take the optional "out" argument.

Examples

```
>>> np.add.nargs
3
>>> np.multiply.nargs
3
>>> np.power.nargs
3
>>> np.exp.nargs
2
```

ufunc.**ntypes**
The number of types.

The number of numerical NumPy types - of which there are 18 total - on which the ufunc can operate.

See also:

numpy.ufunc.types

Examples

```
>>> np.add.ntypes
18
>>> np.multiply.ntypes
18
>>> np.power.ntypes
17
>>> np.exp.ntypes
7
>>> np.remainder.ntypes
14
```

ufunc.**types**

Returns a list with types grouped input->output.

Data attribute listing the data-type "Domain-Range" groupings the ufunc can deliver. The data-types are given using the character codes.

See also:

numpy.ufunc.ntypes

Examples

```
>>> np.add.types
['??->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',
'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',
'GG->G', 'OO->O']
```

```
>>> np.multiply.types
['??->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',
'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',
'GG->G', 'OO->O']
```

```
>>> np.power.types
['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l', 'LL->L',
'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D', 'GG->G',
'OO->O']
```

```
>>> np.exp.types
['f->f', 'd->d', 'g->g', 'F->F', 'D->D', 'G->G', 'O->O']
```

```
>>> np.remainder.types
['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l', 'LL->L',
'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'OO->O']
```

ufunc.**identity**

The identity value.

Data attribute containing the identity element for the ufunc, if it has one. If it does not, the attribute value is None.

Examples

```
>>> np.add.identity
0
>>> np.multiply.identity
1
>>> np.power.identity
1
```

```
>>> print(np.exp.identity)
None
```

2.7.3 Methods

All ufuncs have four methods. However, these methods only make sense on 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 and a *dtype* 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 and may be negative, but must be an integer. The *dtype* keyword allows you to manage a very common problem that arises when naively using {op}.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 `int_` data type, it will be internally upcast to the `int_` (or `uint`) data-type.

Ufuncs also have a fifth method that allows in place operations to be performed using fancy indexing. No buffering is used on the dimensions where fancy indexing is used, so the fancy index can list an item more than once and the operation will be performed on the result of the previous operation for that item.

ufunc.reduce(a[, axis, dtype, out, keepdims])	Reduces *a*'s dimension by one, by applying ufunc along one axis.
ufunc.accumulate(array[, axis, dtype, out])	Accumulate the result of applying the operator to all elements.
ufunc.reduceat(a, indices[, axis, dtype, out])	Performs a (local) reduce with specified slices over a single axis.
ufunc.outer(A, B)	Apply the ufunc *op* to all pairs (a, b) with a in *A* and b in *B*.
ufunc.at(a, indices[, b])	Performs unbuffered in place operation on operand 'a' for elements specified by

ufunc.**reduce**(*a*, *axis=0*, *dtype=None*, *out=None*, *keepdims=False*)

Reduces *a*'s dimension by one, by applying ufunc along one axis.

Let $a.shape = (N_0, ..., N_i, ..., N_{M-1})$. Then $ufunc.reduce(a, axis = i)[k_0, .., k_{i-1}, k_{i+1}, .., k_{M-1}]$ = the result of iterating *j* over $range(N_i)$, cumulatively applying ufunc to each $a[k_0, .., k_{i-1}, j, k_{i+1}, .., k_{M-1}]$. For a one-dimensional array, reduce produces results equivalent to:

```
r = op.identity # op = ufunc
for i in range(len(A)):
  r = op(r, A[i])
return r
```

For example, add.reduce() is equivalent to sum().

Parameters

a : array_like

The array to act on.

axis : None or int or tuple of ints, optional

Axis or axes along which a reduction is performed. The default (*axis* = 0) is perform a reduction over the first dimension of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is *None*, a reduction is performed over all the axes. If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as

before.

For operations which are either not commutative or not associative, doing a reduction over multiple axes is not well-defined. The ufuncs do not currently raise an exception in this case, but will likely do so in the future.

dtype : data-type code, optional

The type used to represent the intermediate results. Defaults to the data-type of the output array if this is provided, or the data-type of the input array if no output array is provided.

out : ndarray, optional

A location into which the result is stored. If not provided, a freshly-allocated array is returned.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

New in version 1.7.0.

Returns

r : ndarray

The reduced array. If *out* was supplied, *r* is a reference to it.

Examples

```
>>> np.multiply.reduce([2,3,5])
30
```

A multi-dimensional array example:

```
>>> X = np.arange(8).reshape((2,2,2))
>>> X
array([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])
>>> np.add.reduce(X, 0)
array([[ 4,  6],
       [ 8, 10]])
>>> np.add.reduce(X) # confirm: default axis value is 0
array([[ 4,  6],
       [ 8, 10]])
>>> np.add.reduce(X, 1)
array([[ 2,  4],
       [10, 12]])
>>> np.add.reduce(X, 2)
array([[ 1,  5],
       [ 9, 13]])
```

ufunc.**accumulate**(*array*, *axis=0*, *dtype=None*, *out=None*)

Accumulate the result of applying the operator to all elements.

For a one-dimensional array, accumulate produces results equivalent to:

```
r = np.empty(len(A))
t = op.identity        # op = the ufunc being applied to A's  elements
for i in range(len(A)):
```

```
        t = op(t, A[i])
        r[i] = t
    return r
```

For example, add.accumulate() is equivalent to np.cumsum().

For a multi-dimensional array, accumulate is applied along only one axis (axis zero by default; see Examples below) so repeated use is necessary if one wants to accumulate over multiple axes.

Parameters

array : array_like

The array to act on.

axis : int, optional

The axis along which to apply the accumulation; default is zero.

dtype : data-type code, optional

The data-type used to represent the intermediate results. Defaults to the data-type of the output array if such is provided, or the the data-type of the input array if no output array is provided.

out : ndarray, optional

A location into which the result is stored. If not provided a freshly-allocated array is returned.

Returns

r : ndarray

The accumulated values. If *out* was supplied, *r* is a reference to *out*.

Examples

1-D array examples:

```
>>> np.add.accumulate([2, 3, 5])
array([ 2,  5, 10])
>>> np.multiply.accumulate([2, 3, 5])
array([ 2,  6, 30])
```

2-D array examples:

```
>>> I = np.eye(2)
>>> I
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Accumulate along axis 0 (rows), down columns:

```
>>> np.add.accumulate(I, 0)
array([[ 1.,  0.],
       [ 1.,  1.]])
>>> np.add.accumulate(I) # no axis specified = axis zero
array([[ 1.,  0.],
       [ 1.,  1.]])
```

Accumulate along axis 1 (columns), through rows:

```
>>> np.add.accumulate(I, 1)
array([[ 1.,  1.],
       [ 0.,  1.]])
```

ufunc.**reduceat**(*a*, *indices*, *axis=0*, *dtype=None*, *out=None*)

Performs a (local) reduce with specified slices over a single axis.

For i in `range(len(indices))`, *reduceat* computes `ufunc.reduce(a[indices[i]:indices[i+1]])`, which becomes the i-th generalized "row" parallel to *axis* in the final result (i.e., in a 2-D array, for example, if *axis = 0*, it becomes the i-th row, but if *axis = 1*, it becomes the i-th column). There are three exceptions to this:

- when `i = len(indices) - 1` (so for the last index), `indices[i+1] = a.shape[axis]`.
- if `indices[i] >= indices[i + 1]`, the i-th generalized "row" is simply `a[indices[i]]`.
- if `indices[i] >= len(a)` or `indices[i] < 0`, an error is raised.

The shape of the output depends on the size of `indices`, and may be larger than *a* (this happens if `len(indices) > a.shape[axis]`).

Parameters

a : array_like

The array to act on.

indices : array_like

Paired indices, comma separated (not colon), specifying slices to reduce.

axis : int, optional

The axis along which to apply the reduceat.

dtype : data-type code, optional

The type used to represent the intermediate results. Defaults to the data type of the output array if this is provided, or the data type of the input array if no output array is provided.

out : ndarray, optional

A location into which the result is stored. If not provided a freshly-allocated array is returned.

Returns

r : ndarray

The reduced values. If *out* was supplied, *r* is a reference to *out*.

Notes

A descriptive example:

If *a* is 1-D, the function *ufunc.accumulate(a)* is the same as `ufunc.reduceat(a, indices)[::2]` where `indices` is `range(len(array) - 1)` with a zero placed in every other element: `indices = zeros(2 * len(a) - 1), indices[1::2] = range(1, len(a))`.

Don't be fooled by this attribute's name: *reduceat(a)* is not necessarily smaller than *a*.

Examples

To take the running sum of four successive values:

```
>>> np.add.reduceat(np.arange(8),[0,4, 1,5, 2,6, 3,7])[::2]
array([ 6, 10, 14, 18])
```

A 2-D example:

```
>>> x = np.linspace(0, 15, 16).reshape(4,4)
>>> x
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.],
       [ 12.,  13.,  14.,  15.]])
```

```
# reduce such that the result has the following five rows:
# [row1 + row2 + row3]
# [row4]
# [row2]
# [row3]
# [row1 + row2 + row3 + row4]
```

```
>>> np.add.reduceat(x, [0, 3, 1, 2, 0])
array([[ 12.,  15.,  18.,  21.],
       [ 12.,  13.,  14.,  15.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.],
       [ 24.,  28.,  32.,  36.]])
```

```
# reduce such that result has the following two columns:
# [col1 * col2 * col3, col4]
```

```
>>> np.multiply.reduceat(x, [0, 3], 1)
array([[    0.,     3.],
       [  120.,     7.],
       [  720.,    11.],
       [ 2184.,    15.]])
```

`ufunc.outer`(*A*, *B*)

Apply the ufunc *op* to all pairs (a, b) with a in *A* and b in *B*.

Let `M = A.ndim`, `N = B.ndim`. Then the result, *C*, of `op.outer(A, B)` is an array of dimension M + N such that:

$$C[i_0, ..., i_{M-1}, j_0, ..., j_{N-1}] = op(A[i_0, ..., i_{M-1}], B[j_0, ..., j_{N-1}])$$

For *A* and *B* one-dimensional, this is equivalent to:

```
r = empty(len(A),len(B))
for i in range(len(A)):
    for j in range(len(B)):
        r[i,j] = op(A[i], B[j]) # op = ufunc in question
```

Parameters

A : array_like

First array

B : array_like

Second array

Returns

r : ndarray

Output array

See also:

`numpy.outer`

Examples

```
>>> np.multiply.outer([1, 2, 3], [4, 5, 6])
array([[ 4,  5,  6],
       [ 8, 10, 12],
       [12, 15, 18]])
```

A multi-dimensional example:

```
>>> A = np.array([[1, 2, 3], [4, 5, 6]])
>>> A.shape
(2, 3)
>>> B = np.array([[1, 2, 3, 4]])
>>> B.shape
(1, 4)
>>> C = np.multiply.outer(A, B)
>>> C.shape; C
(2, 3, 1, 4)
array([[[[ 1,  2,  3,  4]],
        [[ 2,  4,  6,  8]],
        [[ 3,  6,  9, 12]]],
       [[[ 4,  8, 12, 16]],
        [[ 5, 10, 15, 20]],
        [[ 6, 12, 18, 24]]]])
```

`ufunc.at`(*a*, *indices*, *b=None*)

Performs unbuffered in place operation on operand 'a' for elements specified by 'indices'. For addition ufunc, this method is equivalent to *a[indices] += b*, except that results are accumulated for elements that are indexed more than once. For example, *a[[0,0]] += 1* will only increment the first element once because of buffering, whereas *add.at(a, [0,0], 1)* will increment the first element twice.

New in version 1.8.0.

Parameters

a : array_like

The array to perform in place operation on.

indices : array_like or tuple

Array like index object or slice object for indexing into first operand. If first operand has multiple dimensions, indices can be a tuple of array like index objects or slice objects.

b : array_like

Second operand for ufuncs requiring two operands. Operand must be broadcastable over first operand after indexing or slicing.

Examples

Set items 0 and 1 to their negative values:

```
>>> a = np.array([1, 2, 3, 4])
>>> np.negative.at(a, [0, 1])
>>> print(a)
array([-1, -2, 3, 4])
```

Increment items 0 and 1, and increment item 2 twice:

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

Add items 0 and 1 in first array to second array, and store results in first array:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([1, 2])
>>> np.add.at(a, [0, 1], b)
>>> print(a)
array([2, 4, 3, 4])
```

Warning: A reduce-like operation on an array with a data-type that has a range "too small" to handle the result will silently wrap. One should use *dtype* to increase the size of the data-type over which reduction takes place.

2.8 Available ufuncs

There are currently more than 60 universal functions defined in *numpy* on one or more types, covering a wide variety of operations. Some of these ufuncs are called automatically on arrays when the relevant infix notation is used (*e.g.*, *add(a, b)* is called internally when `a + b` is written and *a* or *b* is an *ndarray*). Nevertheless, you may still want to use the ufunc call in order to use the optional output argument(s) to place the output(s) in an object (or objects) of your choice.

Recall that each ufunc operates element-by-element. Therefore, each ufunc will be described as if acting on a set of scalar inputs to return a set of scalar outputs.

Note: The ufunc still returns its output(s) even if you use the optional output argument(s).

2.8.1 Math operations

add(x1, x2[, out])	Add arguments element-wise.
subtract(x1, x2[, out])	Subtract arguments, element-wise.
multiply(x1, x2[, out])	Multiply arguments element-wise.
divide(x1, x2[, out])	Divide arguments element-wise.
logaddexp(x1, x2[, out])	Logarithm of the sum of exponentiations of the inputs.
logaddexp2(x1, x2[, out])	Logarithm of the sum of exponentiations of the inputs in base-2.
true_divide(x1, x2[, out])	Returns a true division of the inputs, element-wise.
floor_divide(x1, x2[, out])	Return the largest integer smaller or equal to the division of the inputs.
negative(x[, out])	Numerical negative, element-wise.
power(x1, x2[, out])	First array elements raised to powers from second array, element-wise.
remainder(x1, x2[, out])	Return element-wise remainder of division.
mod(x1, x2[, out])	Return element-wise remainder of division.
fmod(x1, x2[, out])	Return the element-wise remainder of division.
absolute(x[, out])	Calculate the absolute value element-wise.
rint(x[, out])	Round elements of the array to the nearest integer.
sign(x[, out])	Returns an element-wise indication of the sign of a number.
conj(x[, out])	Return the complex conjugate, element-wise.
exp(x[, out])	Calculate the exponential of all elements in the input array.

Continued on next page

Table 2.5 – continued from previous page

exp2(x[, out])	Calculate *2**p* for all *p* in the input array.
log(x[, out])	Natural logarithm, element-wise.
log2(x[, out])	Base-2 logarithm of *x*.
log10(x[, out])	Return the base 10 logarithm of the input array, element-wise.
expm1(x[, out])	Calculate `exp(x) - 1` for all elements in the array.
log1p(x[, out])	Return the natural logarithm of one plus the input array, element-wise.
sqrt(x[, out])	Return the positive square-root of an array, element-wise.
square(x[, out])	Return the element-wise square of the input.
reciprocal(x[, out])	Return the reciprocal of the argument, element-wise.

Tip: The optional output arguments can be used to help you save memory for large calculations. If your arrays are large, complicated expressions can take longer than absolutely necessary due to the creation and (later) destruction of temporary calculation spaces. For example, the expression `G = a * b + c` is equivalent to `t1 = A * B; G = T1 + C; del t1`. It will be more quickly executed as `G = A * B; add(G, C, G)` which is the same as `G = A * B; G += C`.

2.8.2 Trigonometric functions

All trigonometric functions use radians when an angle is called for. The ratio of degrees to radians is $180^\circ/\pi$.

sin(x[, out])	Trigonometric sine, element-wise.
cos(x[, out])	Cosine element-wise.
tan(x[, out])	Compute tangent element-wise.
arcsin(x[, out])	Inverse sine, element-wise.
arccos(x[, out])	Trigonometric inverse cosine, element-wise.
arctan(x[, out])	Trigonometric inverse tangent, element-wise.
arctan2(x1, x2[, out])	Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.
hypot(x1, x2[, out])	Given the "legs" of a right triangle, return its hypotenuse.
sinh(x[, out])	Hyperbolic sine, element-wise.
cosh(x[, out])	Hyperbolic cosine, element-wise.
tanh(x[, out])	Compute hyperbolic tangent element-wise.
arcsinh(x[, out])	Inverse hyperbolic sine element-wise.
arccosh(x[, out])	Inverse hyperbolic cosine, element-wise.
arctanh(x[, out])	Inverse hyperbolic tangent element-wise.
deg2rad(x[, out])	Convert angles from degrees to radians.
rad2deg(x[, out])	Convert angles from radians to degrees.

2.8.3 Bit-twiddling functions

These function all require integer arguments and they manipulate the bit-pattern of those arguments.

bitwise_and(x1, x2[, out])	Compute the bit-wise AND of two arrays element-wise.
bitwise_or(x1, x2[, out])	Compute the bit-wise OR of two arrays element-wise.
bitwise_xor(x1, x2[, out])	Compute the bit-wise XOR of two arrays element-wise.
invert(x[, out])	Compute bit-wise inversion, or bit-wise NOT, element-wise.
left_shift(x1, x2[, out])	Shift the bits of an integer to the left.
right_shift(x1, x2[, out])	Shift the bits of an integer to the right.

2.8.4 Comparison functions

greater(x1, x2[, out])	Return the truth value of (x1 > x2) element-wise.
greater_equal(x1, x2[, out])	Return the truth value of (x1 >= x2) element-wise.
less(x1, x2[, out])	Return the truth value of (x1 < x2) element-wise.
less_equal(x1, x2[, out])	Return the truth value of (x1 =< x2) element-wise.
not_equal(x1, x2[, out])	Return (x1 != x2) element-wise.
equal(x1, x2[, out])	Return (x1 == x2) element-wise.

Warning: Do not use the Python keywords `and` and `or` to combine logical array expressions. These keywords will test the truth value of the entire array (not element-by-element as you might expect). Use the bitwise operators & and | instead.

logical_and(x1, x2[, out])	Compute the truth value of x1 AND x2 element-wise.
logical_or(x1, x2[, out])	Compute the truth value of x1 OR x2 element-wise.
logical_xor(x1, x2[, out])	Compute the truth value of x1 XOR x2, element-wise.
logical_not(x[, out])	Compute the truth value of NOT x element-wise.

Warning: The bit-wise operators & and | are the proper way to perform element-by-element array comparisons. Be sure you understand the operator precedence: `(a > 2) & (a < 5)` is the proper syntax because `a > 2 & a < 5` will result in an error due to the fact that `2 & a` is evaluated first.

maximum(x1, x2[, out])	Element-wise maximum of array elements.

Tip: The Python function `max()` will find the maximum over a one-dimensional array, but it will do so using a slower sequence interface. The reduce method of the maximum ufunc is much faster. Also, the `max()` method will not give answers you might expect for arrays with greater than one dimension. The reduce method of minimum also allows you to compute a total minimum over an array.

minimum(x1, x2[, out])	Element-wise minimum of array elements.

Warning: the behavior of `maximum(a, b)` is different than that of `max(a, b)`. As a ufunc, `maximum(a, b)` performs an element-by-element comparison of *a* and *b* and chooses each element of the result according to which element in the two arrays is larger. In contrast, `max(a, b)` treats the objects *a* and *b* as a whole, looks at the (total) truth value of `a > b` and uses it to return either *a* or *b* (as a whole). A similar difference exists between `minimum(a, b)` and `min(a, b)`.

fmax(x1, x2[, out])	Element-wise maximum of array elements.
fmin(x1, x2[, out])	Element-wise minimum of array elements.

2.8.5 Floating functions

Recall that all of these functions work element-by-element over an array, returning an array output. The description details only a single operation.

isfinite(x[, out])	Test element-wise for finiteness (not infinity or not Not a Number).
isinf(x[, out])	Test element-wise for positive or negative infinity.
isnan(x[, out])	Test element-wise for NaN and return result as a boolean array.
signbit(x[, out])	Returns element-wise True where signbit is set (less than zero).
copysign(x1, x2[, out])	Change the sign of x1 to that of x2, element-wise.
nextafter(x1, x2[, out])	Return the next floating-point value after x1 towards x2, element-wise.
modf(x[, out1, out2])	Return the fractional and integral parts of an array, element-wise.
ldexp(x1, x2[, out])	Returns x1 * 2**x2, element-wise.
frexp(x[, out1, out2])	Decompose the elements of x into mantissa and twos exponent.
fmod(x1, x2[, out])	Return the element-wise remainder of division.
floor(x[, out])	Return the floor of the input, element-wise.
ceil(x[, out])	Return the ceiling of the input, element-wise.
trunc(x[, out])	Return the truncated value of the input, element-wise.

CHAPTER

THREE

ROUTINES

In this chapter routine docstrings are presented, grouped by functionality. Many docstrings contain example code, which demonstrates basic usage of the routine. The examples assume that NumPy is imported with:

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

A convenient way to execute examples is the `%doctest_mode` mode of IPython, which allows for pasting of multi-line examples and preserves indentation.

3.1 Array creation routines

See also:

Array creation

3.1.1 Ones and zeros

3.1.2 From existing data

3.1.3 Creating record arrays (`numpy.rec`)

Note: `numpy.rec` is the preferred alias for `numpy.core.records`.

core.records.array(obj[, dtype, shape, ...])	Construct a record array from a wide-variety of objects.
core.records.fromarrays(arrayList[, dtype, ...])	create a record array from a (flat) list of arrays
core.records.fromrecords(recList[, dtype, ...])	create a recarray from a list of records in text form
core.records.fromstring(datastring[, dtype, ...])	create a (read-only) record array from binary data contained in
core.records.fromfile(fd[, dtype, shape, ...])	Create an array from binary file data

numpy.core.records.**array**(*obj*, *dtype=None*, *shape=None*, *offset=0*, *strides=None*, *formats=None*, *names=None*, *titles=None*, *aligned=False*, *byteorder=None*, *copy=True*)

Construct a record array from a wide-variety of objects.

numpy.core.records.**fromarrays**(*arrayList*, *dtype=None*, *shape=None*, *formats=None*, *names=None*, *titles=None*, *aligned=False*, *byteorder=None*)

create a record array from a (flat) list of arrays

```
>>> x1=np.array([1,2,3,4])
>>> x2=np.array(['a','dd','xyz','12'])
>>> x3=np.array([1.1,2,3,4])
>>> r = np.core.records.fromarrays([x1,x2,x3],names='a,b,c')
>>> print(r[1])
(2, 'dd', 2.0)
>>> x1[1]=34
>>> r.a
array([1, 2, 3, 4])
```

numpy.core.records.**fromrecords**(*recList*, *dtype=None*, *shape=None*, *formats=None*, *names=None*, *titles=None*, *aligned=False*, *byteorder=None*)

create a recarray from a list of records in text form

The data in the same field can be heterogeneous, they will be promoted to the highest data type. This method is intended for creating smaller record arrays. If used to create large array without formats defined

r=fromrecords([(2,3.,'abc')]*100000)

it can be slow.

If formats is None, then this will auto-detect formats. Use list of tuples rather than list of lists for faster processing.

```
>>> r=np.core.records.fromrecords([(456,'dbe',1.2),(2,'de',1.3)],
... names='col1,col2,col3')
>>> print(r[0])
(456, 'dbe', 1.2)
>>> r.col1
array([456,   2])
>>> r.col2
array(['dbe', 'de'],
      dtype='|S3')
>>> import pickle
>>> print(pickle.loads(pickle.dumps(r)))
[(456, 'dbe', 1.2) (2, 'de', 1.3)]
```

numpy.core.records.**fromstring**(*datastring*, *dtype=None*, *shape=None*, *offset=0*, *formats=None*, *names=None*, *titles=None*, *aligned=False*, *byteorder=None*)

create a (read-only) record array from binary data contained in a string

numpy.core.records.**fromfile**(*fd*, *dtype=None*, *shape=None*, *offset=0*, *formats=None*, *names=None*, *titles=None*, *aligned=False*, *byteorder=None*)

Create an array from binary file data

If file is a string then that file is opened, else it is assumed to be a file object.

```
>>> from tempfile import TemporaryFile
>>> a = np.empty(10,dtype='f8,i4,a5')
>>> a[5] = (0.5,10,'abcde')
>>>
>>> fd=TemporaryFile()
>>> a = a.newbyteorder('<')
>>> a.tofile(fd)
>>>
>>> fd.seek(0)
```

```
>>> r=np.core.records.fromfile(fd, formats='f8,i4,a5', shape=10,
... byteorder='<')
>>> print(r[5])
(0.5, 10, 'abcde')
>>> r.shape
(10,)
```

3.1.4 Creating character arrays (`numpy.char`)

Note: `numpy.char` is the preferred alias for `numpy.core.defchararray`.

core.defchararray.array(obj[, itemsize, ...])	Create a `chararray`.
core.defchararray.asarray(obj[, itemsize, ...])	Convert the input to a `chararray`, copying the data only if necessary.

numpy.core.defchararray.**array**(*obj*, *itemsize=None*, *copy=True*, *unicode=None*, *order=None*)

Create a *chararray*.

Note: This class is provided for numarray backward-compatibility. New code (not concerned with numarray compatibility) should use arrays of type `string_` or `unicode_` and use the free functions in `numpy.char` for fast vectorized string operations instead.

Versus a regular Numpy array of type *str* or *unicode*, this class adds the following functionality:

1. values automatically have whitespace removed from the end when indexed
2. comparison operators automatically remove whitespace from the end when comparing values
3. vectorized string operations are provided as methods (e.g. *str.endswith*) and infix operators (e.g. `+, *, %`)

Parameters

obj : array of str or unicode-like

itemsize : int, optional

itemsize is the number of characters per scalar in the resulting array. If *itemsize* is None, and *obj* is an object array or a Python list, the *itemsize* will be automatically determined. If *itemsize* is provided and *obj* is of type str or unicode, then the *obj* string will be chunked into *itemsize* pieces.

copy : bool, optional

If true (default), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (*itemsize*, unicode, *order*, etc.).

unicode : bool, optional

When true, the resulting *chararray* can contain Unicode characters, when false only 8-bit characters. If unicode is *None* and *obj* is one of the following:

- a *chararray*,
- an ndarray of type *str* or *unicode*
- a Python str or unicode object,

then the unicode setting of the output array will be automatically determined.

order : {'C', 'F', 'A'}, optional

Specify the order of the array. If order is 'C' (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is 'F', then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is 'A', then the returned array may be in any order (either C-, Fortran-contiguous, or even discontiguous).

numpy.core.defchararray.**asarray**(*obj*, *itemsize=None*, *unicode=None*, *order=None*)

Convert the input to a *chararray*, copying the data only if necessary.

Versus a regular Numpy array of type *str* or *unicode*, this class adds the following functionality:

1. values automatically have whitespace removed from the end when indexed
2. comparison operators automatically remove whitespace from the end when comparing values
3. vectorized string operations are provided as methods (e.g. *str.endswith*) and infix operators (e.g. +, *, ''%'')

Parameters

obj : array of str or unicode-like

itemsize : int, optional

itemsize is the number of characters per scalar in the resulting array. If *itemsize* is None, and *obj* is an object array or a Python list, the *itemsize* will be automatically determined. If *itemsize* is provided and *obj* is of type str or unicode, then the *obj* string will be chunked into *itemsize* pieces.

unicode : bool, optional

When true, the resulting *chararray* can contain Unicode characters, when false only 8-bit characters. If unicode is *None* and *obj* is one of the following:

- a *chararray*,
- an ndarray of type *str* or 'unicode'
- a Python str or unicode object,

then the unicode setting of the output array will be automatically determined.

order : {'C', 'F'}, optional

Specify the order of the array. If order is 'C' (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is 'F', then the returned array will be in Fortran-contiguous order (first-index varies the fastest).

3.1.5 Numerical ranges

3.1.6 Building matrices

3.1.7 The Matrix class

3.2 Array manipulation routines

3.2.1 Basic operations

3.2.2 Changing array shape

ndarray.flat	A 1-D iterator over the array.
ndarray.flatten([order])	Return a copy of the array collapsed into one dimension.

ndarray.**flat**
A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See also:

flatten
Return a copy of the array collapsed into one dimension.

flatiter

Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

ndarray.**flatten**(*order='C'*)

Return a copy of the array collapsed into one dimension.

Parameters

order : {'C', 'F', 'A', 'K'}, optional

'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

Returns

y : ndarray

A copy of the input array, flattened to one dimension.

See also:

ravel

Return a flattened array.

flat

A 1-D flat iterator over the array.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

3.2.3 Transpose-like operations

ndarray.T	Same as self.transpose(), except that self is returned if self.ndim < 2.

ndarray.**T**

Same as self.transpose(), except that self is returned if self.ndim < 2.

Examples

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

3.2.4 Changing number of dimensions

broadcast	Produce an object that mimics broadcasting.

class numpy.**broadcast**

Produce an object that mimics broadcasting.

Parameters

in1, in2, ... : array_like

Input parameters.

Returns

b : broadcast object

Broadcast the input parameters against one another, and return an object that encapsulates the result. Amongst others, it has `shape` and `nd` properties, and may be used as an iterator.

Examples

Manually adding two vectors, using broadcasting:

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

```
>>> out = np.empty(b.shape)
>>> out.flat = [u+v for (u,v) in b]
>>> out
array([[ 5.,  6.,  7.],
       [ 6.,  7.,  8.],
       [ 7.,  8.,  9.]])
```

Compare against built-in broadcasting:

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

Attributes

index	current index in broadcasted result
iters	tuple of iterators along `self`'s "components."
shape	Shape of broadcasted result.
size	Total size of broadcasted result.

broadcast.**index**

current index in broadcasted result

Examples

```
>>> x = np.array([[1], [2], [3]])
>>> y = np.array([4, 5, 6])
>>> b = np.broadcast(x, y)
>>> b.index
0
>>> b.next(), b.next(), b.next()
```

```
((1, 4), (1, 5), (1, 6))
>>> b.index
3
```

broadcast.**iters**
tuple of iterators along `self`'s "components."

Returns a tuple of *numpy.flatiter* objects, one for each "component" of `self`.

See also:

numpy.flatiter

Examples

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> row, col = b.iters
>>> row.next(), col.next()
(1, 4)
```

broadcast.**shape**
Shape of broadcasted result.

Examples

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

broadcast.**size**
Total size of broadcasted result.

Examples

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> b.size
9
```

Methods

next	
reset()	Reset the broadcasted result's iterator(s).

broadcast.**next**

broadcast.**reset**()
Reset the broadcasted result's iterator(s).

Parameters
None

Returns

None

Examples

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]]
>>> b = np.broadcast(x, y)
>>> b.index
0
>>> b.next(), b.next(), b.next()
((1, 4), (2, 4), (3, 4))
>>> b.index
3
>>> b.reset()
>>> b.index
0
```

3.2.5 Changing kind of array

3.2.6 Joining arrays

3.2.7 Splitting arrays

3.2.8 Tiling arrays

3.2.9 Adding and removing elements

3.2.10 Rearranging elements

3.3 Binary operations

3.3.1 Elementwise bit operations

bitwise_and(x1, x2[, out])	Compute the bit-wise AND of two arrays element-wise.
bitwise_or(x1, x2[, out])	Compute the bit-wise OR of two arrays element-wise.
bitwise_xor(x1, x2[, out])	Compute the bit-wise XOR of two arrays element-wise.

Continued on next page

Table 3.20 – continued from previous page

invert(x[, out])	Compute bit-wise inversion, or bit-wise NOT, element-wise.
left_shift(x1, x2[, out])	Shift the bits of an integer to the left.
right_shift(x1, x2[, out])	Shift the bits of an integer to the right.

numpy.**bitwise_and**(*x1*, *x2*[, *out*]) = **<ufunc 'bitwise_and'>**

Compute the bit-wise AND of two arrays element-wise.

Computes the bit-wise AND of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `&`.

Parameters

x1, x2 : array_like

Only integer and boolean types are handled.

Returns

out : array_like

Result.

See also:

logical_and, *bitwise_or*, *bitwise_xor*

binary_repr

Return the binary representation of the input number as a string.

Examples

The number 13 is represented by `00001101`. Likewise, 17 is represented by `00010001`. The bit-wise AND of 13 and 17 is therefore `000000001`, or 1:

```
>>> np.bitwise_and(13, 17)
1
```

```
>>> np.bitwise_and(14, 13)
12
>>> np.binary_repr(12)
'1100'
>>> np.bitwise_and([14,3], 13)
array([12,  1])
```

```
>>> np.bitwise_and([11,7], [4,25])
array([0, 1])
>>> np.bitwise_and(np.array([2,5,255]), np.array([3,14,16]))
array([ 2,  4, 16])
>>> np.bitwise_and([True, True], [False, True])
array([False,  True], dtype=bool)
```

numpy.**bitwise_or**(*x1*, *x2*[, *out*]) = **<ufunc 'bitwise_or'>**

Compute the bit-wise OR of two arrays element-wise.

Computes the bit-wise OR of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `|`.

Parameters

x1, x2 : array_like

Only integer and boolean types are handled.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

out : array_like

Result.

See also:

logical_or, *bitwise_and*, *bitwise_xor*

binary_repr

Return the binary representation of the input number as a string.

Examples

The number 13 has the binaray representation `00001101`. Likewise, 16 is represented by `00010000`. The bit-wise OR of 13 and 16 is then `000111011`, or 29:

```
>>> np.bitwise_or(13, 16)
29
>>> np.binary_repr(29)
'11101'
```

```
>>> np.bitwise_or(32, 2)
34
>>> np.bitwise_or([33, 4], 1)
array([33,  5])
>>> np.bitwise_or([33, 4], [1, 2])
array([33,  6])
```

```
>>> np.bitwise_or(np.array([2, 5, 255]), np.array([4, 4, 4]))
array([  6,   5, 255])
>>> np.array([2, 5, 255]) | np.array([4, 4, 4])
array([  6,   5, 255])
>>> np.bitwise_or(np.array([2, 5, 255, 2147483647L], dtype=np.int32),
...               np.array([4, 4, 4, 2147483647L], dtype=np.int32))
array([         6,          5,        255, 2147483647])
>>> np.bitwise_or([True, True], [False, True])
array([ True,  True], dtype=bool)
```

numpy.**bitwise_xor**(*x1*, *x2*[, *out*]) **= <ufunc 'bitwise_xor'>**

Compute the bit-wise XOR of two arrays element-wise.

Computes the bit-wise XOR of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator ^.

Parameters

x1, x2 : array_like

Only integer and boolean types are handled.

Returns

out : array_like

Result.

See also:

logical_xor, *bitwise_and*, *bitwise_or*

binary_repr
Return the binary representation of the input number as a string.

Examples

The number 13 is represented by `00001101`. Likewise, 17 is represented by `00010001`. The bit-wise XOR of 13 and 17 is therefore `00011100`, or 28:

```
>>> np.bitwise_xor(13, 17)
28
>>> np.binary_repr(28)
'11100'
```

```
>>> np.bitwise_xor(31, 5)
26
>>> np.bitwise_xor([31,3], 5)
array([26,  6])
```

```
>>> np.bitwise_xor([31,3], [5,6])
array([26,  5])
>>> np.bitwise_xor([True, True], [False, True])
array([ True, False], dtype=bool)
```

numpy.**invert**(*x*[, *out*]) = <ufunc 'invert'>
Compute bit-wise inversion, or bit-wise NOT, element-wise.

Computes the bit-wise NOT of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator ~.

For signed integer inputs, the two's complement is returned. In a two's-complement system negative numbers are represented by the two's complement of the absolute value. This is the most common method of representing signed integers on computers *[R32]*. A N-bit two's-complement system can represent every integer in the range -2^{N-1} to $+2^{N-1}-1$.

Parameters
x1 : array_like

Only integer and boolean types are handled.

Returns
out : array_like

Result.

See also:

bitwise_and, *bitwise_or*, *bitwise_xor*, *logical_not*

binary_repr
Return the binary representation of the input number as a string.

Notes

`bitwise_not` is an alias for *invert*:

```
>>> np.bitwise_not is np.invert
True
```

References

[R32]

Examples

We've seen that 13 is represented by `00001101`. The invert or bit-wise NOT of 13 is then:

```
>>> np.invert(np.array([13], dtype=uint8))
array([242], dtype=uint8)
>>> np.binary_repr(x, width=8)
'00001101'
>>> np.binary_repr(242, width=8)
'11110010'
```

The result depends on the bit-width:

```
>>> np.invert(np.array([13], dtype=uint16))
array([65522], dtype=uint16)
>>> np.binary_repr(x, width=16)
'0000000000001101'
>>> np.binary_repr(65522, width=16)
'1111111111110010'
```

When using signed integer types the result is the two's complement of the result for the unsigned type:

```
>>> np.invert(np.array([13], dtype=int8))
array([-14], dtype=int8)
>>> np.binary_repr(-14, width=8)
'11110010'
```

Booleans are accepted as well:

```
>>> np.invert(array([True, False]))
array([False,  True], dtype=bool)
```

numpy.**left_shift**(*x1*, *x2*[, *out*]) **= <ufunc 'left_shift'>**

Shift the bits of an integer to the left.

Bits are shifted to the left by appending *x2* 0s at the right of *x1*. Since the internal representation of numbers is in binary format, this operation is equivalent to multiplying *x1* by `2**x2`.

Parameters

x1 : array_like of integer type

Input values.

x2 : array_like of integer type

Number of zeros to append to *x1*. Has to be non-negative.

Returns

out : array of integer type

Return *x1* with bits shifted *x2* times to the left.

See also:

right_shift

Shift the bits of an integer to the right.

binary_repr

Return the binary representation of the input number as a string.

Examples

```
>>> np.binary_repr(5)
'101'
>>> np.left_shift(5, 2)
20
>>> np.binary_repr(20)
'10100'
```

```
>>> np.left_shift(5, [1,2,3])
array([10, 20, 40])
```

numpy.**right_shift**(*x1*, *x2*[, *out*]) = <ufunc 'right_shift'>

Shift the bits of an integer to the right.

Bits are shifted to the right *x2*. Because the internal representation of numbers is in binary format, this operation is equivalent to dividing *x1* by `2**x2`.

Parameters

x1 : array_like, int

Input values.

x2 : array_like, int

Number of bits to remove at the right of *x1*.

Returns

out : ndarray, int

Return *x1* with bits shifted *x2* times to the right.

See also:

left_shift

Shift the bits of an integer to the left.

binary_repr

Return the binary representation of the input number as a string.

Examples

```
>>> np.binary_repr(10)
'1010'
>>> np.right_shift(10, 1)
5
>>> np.binary_repr(5)
'101'
```

```
>>> np.right_shift(10, [1,2,3])
array([5, 2, 1])
```

3.3.2 Bit packing

3.3.3 Output formatting

3.4 String operations

This module provides a set of vectorized string operations for arrays of type `numpy.string_` or `numpy.unicode_`. All of them are based on the string methods in the Python standard library.

3.4.1 String operations

add(x1, x2)	Return element-wise string concatenation for two arrays of str or unicode.
multiply(a, i)	Return (a * i), that is string multiple concatenation, element-wise.
mod(a, values)	Return (a % i), that is pre-Python 2.6 string formatting (iterpolation), element-wise for a pair o
capitalize(a)	Return a copy of *a* with only the first character of each element capitalized.
center(a, width[, fillchar])	Return a copy of *a* with its elements centered in a string of length *width*.
decode(a[, encoding, errors])	Calls *str.decode* element-wise.
encode(a[, encoding, errors])	Calls *str.encode* element-wise.
join(sep, seq)	Return a string which is the concatenation of the strings in the sequence *seq*.
ljust(a, width[, fillchar])	Return an array with the elements of *a* left-justified in a string of length *width*.
lower(a)	Return an array with the elements converted to lowercase.
lstrip(a[, chars])	For each element in *a*, return a copy with the leading characters removed.
partition(a, sep)	Partition each element in *a* around *sep*.
replace(a, old, new[, count])	For each element in *a*, return a copy of the string with all occurrences of substring *old* replaced
rjust(a, width[, fillchar])	Return an array with the elements of *a* right-justified in a string of length *width*.
rpartition(a, sep)	Partition (split) each element around the right-most separator.
rsplit(a[, sep, maxsplit])	For each element in *a*, return a list of the words in the string, using *sep* as the delimiter string.
rstrip(a[, chars])	For each element in *a*, return a copy with the trailing characters removed.
split(a[, sep, maxsplit])	For each element in *a*, return a list of the words in the string, using *sep* as the delimiter string.
splitlines(a[, keepends])	For each element in *a*, return a list of the lines in the element, breaking at line boundaries.
strip(a[, chars])	For each element in *a*, return a copy with the leading and trailing characters removed.
swapcase(a)	Return element-wise a copy of the string with uppercase characters converted to lowercase and
title(a)	Return element-wise title cased version of string or unicode.
translate(a, table[, deletechars])	For each element in *a*, return a copy of the string where all characters occurring in the optional
upper(a)	Return an array with the elements converted to uppercase.
zfill(a, width)	Return the numeric string left-filled with zeros

`numpy.core.defchararray.`**`add`**(*x1*, *x2*)

Return element-wise string concatenation for two arrays of str or unicode.

Arrays *x1* and *x2* must have the same shape.

Parameters

x1 : array_like of str or unicode

Input array.

x2 : array_like of str or unicode

Input array.

Returns

add : ndarray

Output array of `string_` or `unicode_`, depending on input types of the same shape as *x1* and *x2*.

`numpy.core.defchararray.`**`multiply`**(*a*, *i*)
Return (a * i), that is string multiple concatenation, element-wise.

Values in *i* of less than 0 are treated as 0 (which yields an empty string).

Parameters
a : array_like of str or unicode

i : array_like of ints

Returns
out : ndarray

Output array of str or unicode, depending on input types

`numpy.core.defchararray.`**`mod`**(*a*, *values*)
Return (a % i), that is pre-Python 2.6 string formatting (iterpolation), element-wise for a pair of array_likes of str or unicode.

Parameters
a : array_like of str or unicode

values : array_like of values

These values will be element-wise interpolated into the string.

Returns
out : ndarray

Output array of str or unicode, depending on input types

See also:

`str.__mod__`

`numpy.core.defchararray.`**`capitalize`**(*a*)
Return a copy of *a* with only the first character of each element capitalized.

Calls *str.capitalize* element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters
a : array_like of str or unicode

Input array of strings to capitalize.

Returns
out : ndarray

Output array of str or unicode, depending on input types

See also:

`str.capitalize`

Examples

```
>>> c = np.array(['a1b2','1b2a','b2a1','2a1b'],'S4'); c
array(['a1b2', '1b2a', 'b2a1', '2a1b'],
    dtype='|S4')
>>> np.char.capitalize(c)
```

```
    array(['A1b2', '1b2a', 'B2a1', '2a1b'],
        dtype='|S4')
```

numpy.core.defchararray.**center**(*a*, *width*, *fillchar=' '*)

Return a copy of *a* with its elements centered in a string of length *width*.

Calls *str.center* element-wise.

Parameters

a : array_like of str or unicode

width : int

The length of the resulting strings

fillchar : str or unicode, optional

The padding character to use (default is space).

Returns

out : ndarray

Output array of str or unicode, depending on input types

See also:

`str.center`

numpy.core.defchararray.**decode**(*a*, *encoding=None*, *errors=None*)

Calls *str.decode* element-wise.

The set of available codecs comes from the Python standard library, and may be extended at runtime. For more information, see the `codecs` module.

Parameters

a : array_like of str or unicode

encoding : str, optional

The name of an encoding

errors : str, optional

Specifies how to handle encoding errors

Returns

out : ndarray

See also:

`str.decode`

Notes

The type of the result will depend on the encoding specified.

Examples

```
    >>> c = np.array(['aAaAaA', '  aA  ', 'abBABba'])
    >>> c
    array(['aAaAaA', '  aA  ', 'abBABba'],
        dtype='|S7')
    >>> np.char.encode(c, encoding='cp037')
    array(['\x81\xc1\x81\xc1\x81\xc1', '@@\x81\xc1@@',
```

```
        '\x81\x82\xc2\xc1\xc2\x82\x81'],
        dtype='|S7')
```

numpy.core.defchararray.**encode**(*a*, *encoding=None*, *errors=None*)

Calls *str.encode* element-wise.

The set of available codecs comes from the Python standard library, and may be extended at runtime. For more information, see the codecs module.

Parameters

a : array_like of str or unicode

encoding : str, optional

The name of an encoding

errors : str, optional

Specifies how to handle encoding errors

Returns

out : ndarray

See also:

`str.encode`

Notes

The type of the result will depend on the encoding specified.

numpy.core.defchararray.**join**(*sep*, *seq*)

Return a string which is the concatenation of the strings in the sequence *seq*.

Calls *str.join* element-wise.

Parameters

sep : array_like of str or unicode

seq : array_like of str or unicode

Returns

out : ndarray

Output array of str or unicode, depending on input types

See also:

`str.join`

numpy.core.defchararray.**ljust**(*a*, *width*, *fillchar=' '*)

Return an array with the elements of *a* left-justified in a string of length *width*.

Calls *str.ljust* element-wise.

Parameters

a : array_like of str or unicode

width : int

The length of the resulting strings

fillchar : str or unicode, optional

The character to use for padding

Returns

out : ndarray

Output array of str or unicode, depending on input type

See also:

`str.ljust`

numpy.core.defchararray.**lower**(*a*)

Return an array with the elements converted to lowercase.

Call *str.lower* element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

a : array_like, {str, unicode}

Input array.

Returns

out : ndarray, {str, unicode}

Output array of str or unicode, depending on input type

See also:

`str.lower`

Examples

```
>>> c = np.array(['A1B C', '1BCA', 'BCA1']); c
array(['A1B C', '1BCA', 'BCA1'],
      dtype='|S5')
>>> np.char.lower(c)
array(['a1b c', '1bca', 'bca1'],
      dtype='|S5')
```

numpy.core.defchararray.**lstrip**(*a*, *chars=None*)

For each element in *a*, return a copy with the leading characters removed.

Calls *str.lstrip* element-wise.

Parameters

a : array-like, {str, unicode}

Input array.

chars : {str, unicode}, optional

The *chars* argument is a string specifying the set of characters to be removed. If omitted or None, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped.

Returns

out : ndarray, {str, unicode}

Output array of str or unicode, depending on input type

See also:

`str.lstrip`

Examples

```
>>> c = np.array(['aAaAaA', '  aA  ', 'abBABba'])
>>> c
array(['aAaAaA', '  aA  ', 'abBABba'],
    dtype='|S7')
```

The 'a' variable is unstripped from c[1] because whitespace leading.

```
>>> np.char.lstrip(c, 'a')
array(['AaAaA', '  aA  ', 'bBABba'],
    dtype='|S7')
```

```
>>> np.char.lstrip(c, 'A') # leaves c unchanged
array(['aAaAaA', '  aA  ', 'abBABba'],
    dtype='|S7')
>>> (np.char.lstrip(c, ' ') == np.char.lstrip(c, '')).all()
... # XXX: is this a regression? this line now returns False
... # np.char.lstrip(c,'') does not modify c at all.
True
>>> (np.char.lstrip(c, ' ') == np.char.lstrip(c, None)).all()
True
```

numpy.core.defchararray.**partition**(*a*, *sep*)

Partition each element in *a* around *sep*.

Calls *str.partition* element-wise.

For each element in *a*, split the element as the first occurrence of *sep*, and return 3 strings containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return 3 strings containing the string itself, followed by two empty strings.

Parameters

a : array_like, {str, unicode}

Input array

sep : {str, unicode}

Separator to split each string element in *a*.

Returns

out : ndarray, {str, unicode}

Output array of str or unicode, depending on input type. The output array will have an extra dimension with 3 elements per input element.

See also:

`str.partition`

numpy.core.defchararray.**replace**(*a*, *old*, *new*, *count=None*)

For each element in *a*, return a copy of the string with all occurrences of substring *old* replaced by `new`.

Calls *str.replace* element-wise.

Parameters

a : array-like of str or unicode

old, new : str or unicode

count : int, optional

If the optional argument `count` is given, only the first `count` occurrences are replaced.

Returns

out : ndarray

Output array of str or unicode, depending on input type

See also:

`str.replace`

numpy.core.defchararray.**rjust**(*a*, *width*, *fillchar=' '*)

Return an array with the elements of *a* right-justified in a string of length *width*.

Calls *str.rjust* element-wise.

Parameters

a : array_like of str or unicode

width : int

The length of the resulting strings

fillchar : str or unicode, optional

The character to use for padding

Returns

out : ndarray

Output array of str or unicode, depending on input type

See also:

`str.rjust`

numpy.core.defchararray.**rpartition**(*a*, *sep*)

Partition (split) each element around the right-most separator.

Calls *str.rpartition* element-wise.

For each element in *a*, split the element as the last occurrence of *sep*, and return 3 strings containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return 3 strings containing the string itself, followed by two empty strings.

Parameters

a : array_like of str or unicode

Input array

sep : str or unicode

Right-most separator to split each element in array.

Returns

out : ndarray

Output array of string or unicode, depending on input type. The output array will have an extra dimension with 3 elements per input element.

See also:

`str.rpartition`

numpy.core.defchararray.**rsplit**(*a*, *sep=None*, *maxsplit=None*)

For each element in *a*, return a list of the words in the string, using *sep* as the delimiter string.

Calls *str.rsplit* element-wise.

Except for splitting from the right, *rsplit* behaves like *split*.

Parameters

a : array_like of str or unicode

sep : str or unicode, optional

If *sep* is not specified or *None*, any whitespace string is a separator.

maxsplit : int, optional

If *maxsplit* is given, at most *maxsplit* splits are done, the rightmost ones.

Returns

out : ndarray

Array of list objects

See also:

`str.rsplit`, *split*

`numpy.core.defchararray.`**`rstrip`**(*a*, *chars=None*)

For each element in *a*, return a copy with the trailing characters removed.

Calls *str.rstrip* element-wise.

Parameters

a : array-like of str or unicode

chars : str or unicode, optional

The *chars* argument is a string specifying the set of characters to be removed. If omitted or None, the *chars* argument defaults to removing whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped.

Returns

out : ndarray

Output array of str or unicode, depending on input type

See also:

`str.rstrip`

Examples

```
>>> c = np.array(['aAaAaA', 'abBABba'], dtype='S7'); c
array(['aAaAaA', 'abBABba'],
    dtype='|S7')
>>> np.char.rstrip(c, 'a')
array(['aAaAaA', 'abBABb'],
    dtype='|S7')
>>> np.char.rstrip(c, 'A')
array(['aAaAa', 'abBABba'],
    dtype='|S7')
```

`numpy.core.defchararray.`**`split`**(*a*, *sep=None*, *maxsplit=None*)

For each element in *a*, return a list of the words in the string, using *sep* as the delimiter string.

Calls *str.rsplit* element-wise.

Parameters

a : array_like of str or unicode

sep : str or unicode, optional

If *sep* is not specified or *None*, any whitespace string is a separator.

maxsplit : int, optional

If *maxsplit* is given, at most *maxsplit* splits are done.

Returns

out : ndarray

Array of list objects

See also:

`str.split`, *rsplit*

`numpy.core.defchararray.`**`splitlines`**(*a*, *keepends=None*)

For each element in *a*, return a list of the lines in the element, breaking at line boundaries.

Calls *str.splitlines* element-wise.

Parameters

a : array_like of str or unicode

keepends : bool, optional

Line breaks are not included in the resulting list unless keepends is given and true.

Returns

out : ndarray

Array of list objects

See also:

`str.splitlines`

`numpy.core.defchararray.`**`strip`**(*a*, *chars=None*)

For each element in *a*, return a copy with the leading and trailing characters removed.

Calls *str.rstrip* element-wise.

Parameters

a : array-like of str or unicode

chars : str or unicode, optional

The *chars* argument is a string specifying the set of characters to be removed. If omitted or None, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix or suffix; rather, all combinations of its values are stripped.

Returns

out : ndarray

Output array of str or unicode, depending on input type

See also:

`str.strip`

Examples

```
>>> c = np.array(['aAaAaA', '  aA  ', 'abBABba'])
>>> c
array(['aAaAaA', '  aA  ', 'abBABba'],
    dtype='|S7')
>>> np.char.strip(c)
array(['aAaAaA', 'aA', 'abBABba'],
    dtype='|S7')
```

```
    >>> np.char.strip(c, 'a') # 'a' unstripped from c[1] because whitespace leads
    array(['AaAaA', '  aA  ', 'bBABb'],
        dtype='|S7')
    >>> np.char.strip(c, 'A') # 'A' unstripped from c[1] because (unprinted) ws trails
    array(['aAaAa', '  aA  ', 'abBABba'],
        dtype='|S7')
```

numpy.core.defchararray.**swapcase**(*a*)

Return element-wise a copy of the string with uppercase characters converted to lowercase and vice versa.

Calls *str.swapcase* element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

a : array_like, {str, unicode}

Input array.

Returns

out : ndarray, {str, unicode}

Output array of str or unicode, depending on input type

See also:

`str.swapcase`

Examples

```
    >>> c=np.array(['a1B c','1b Ca','b Ca1','cA1b'],'S5'); c
    array(['a1B c', '1b Ca', 'b Ca1', 'cA1b'],
        dtype='|S5')
    >>> np.char.swapcase(c)
    array(['A1b C', '1B cA', 'B cA1', 'Ca1B'],
        dtype='|S5')
```

numpy.core.defchararray.**title**(*a*)

Return element-wise title cased version of string or unicode.

Title case words start with uppercase characters, all remaining cased characters are lowercase.

Calls *str.title* element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

a : array_like, {str, unicode}

Input array.

Returns

out : ndarray

Output array of str or unicode, depending on input type

See also:

`str.title`

Examples

```
>>> c=np.array(['a1b c','1b ca','b ca1','ca1b'],'S5'); c
array(['a1b c', '1b ca', 'b ca1', 'ca1b'],
    dtype='|S5')
>>> np.char.title(c)
array(['A1B C', '1B Ca', 'B Ca1', 'Ca1B'],
    dtype='|S5')
```

numpy.core.defchararray.**translate**(*a*, *table*, *deletechars=None*)

For each element in *a*, return a copy of the string where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation table.

Calls *str.translate* element-wise.

Parameters

a : array-like of str or unicode

table : str of length 256

deletechars : str

Returns

out : ndarray

Output array of str or unicode, depending on input type

See also:

`str.translate`

numpy.core.defchararray.**upper**(*a*)

Return an array with the elements converted to uppercase.

Calls *str.upper* element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

a : array_like, {str, unicode}

Input array.

Returns

out : ndarray, {str, unicode}

Output array of str or unicode, depending on input type

See also:

`str.upper`

Examples

```
>>> c = np.array(['a1b c', '1bca', 'bca1']); c
array(['a1b c', '1bca', 'bca1'],
    dtype='|S5')
>>> np.char.upper(c)
array(['A1B C', '1BCA', 'BCA1'],
    dtype='|S5')
```

numpy.core.defchararray.**zfill**(*a*, *width*)

Return the numeric string left-filled with zeros

Calls *str.zfill* element-wise.

Parameters

a : array_like, {str, unicode}

Input array.

width : int

Width of string to left-fill elements in *a*.

Returns

out : ndarray, {str, unicode}

Output array of str or unicode, depending on input type

See also:

`str.zfill`

3.4.2 Comparison

Unlike the standard numpy comparison operators, the ones in the *char* module strip trailing whitespace characters before performing the comparison.

equal(x1, x2)	Return (x1 == x2) element-wise.
not_equal(x1, x2)	Return (x1 != x2) element-wise.
greater_equal(x1, x2)	Return (x1 >= x2) element-wise.
less_equal(x1, x2)	Return (x1 <= x2) element-wise.
greater(x1, x2)	Return (x1 > x2) element-wise.
less(x1, x2)	Return (x1 < x2) element-wise.

numpy.core.defchararray.**equal**(*x1*, *x2*)

Return (x1 == x2) element-wise.

Unlike *numpy.equal*, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters

x1, x2 : array_like of str or unicode

Input arrays of the same shape.

Returns

out : ndarray or bool

Output array of bools, or a single bool if x1 and x2 are scalars.

See also:

not_equal, *greater_equal*, *less_equal*, *greater*, *less*

numpy.core.defchararray.**not_equal**(*x1*, *x2*)

Return (x1 != x2) element-wise.

Unlike *numpy.not_equal*, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters

x1, x2 : array_like of str or unicode

Input arrays of the same shape.

Returns

out : ndarray or bool

Output array of bools, or a single bool if x1 and x2 are scalars.

See also:

equal, *greater_equal*, *less_equal*, *greater*, *less*

numpy.core.defchararray.**greater_equal**(*x1*, *x2*)

Return (x1 >= x2) element-wise.

Unlike *numpy.greater_equal*, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters

x1, x2 : array_like of str or unicode

Input arrays of the same shape.

Returns

out : ndarray or bool

Output array of bools, or a single bool if x1 and x2 are scalars.

See also:

equal, *not_equal*, *less_equal*, *greater*, *less*

numpy.core.defchararray.**less_equal**(*x1*, *x2*)

Return (x1 <= x2) element-wise.

Unlike *numpy.less_equal*, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters

x1, x2 : array_like of str or unicode

Input arrays of the same shape.

Returns

out : ndarray or bool

Output array of bools, or a single bool if x1 and x2 are scalars.

See also:

equal, *not_equal*, *greater_equal*, *greater*, *less*

numpy.core.defchararray.**greater**(*x1*, *x2*)

Return (x1 > x2) element-wise.

Unlike *numpy.greater*, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters

x1, x2 : array_like of str or unicode

Input arrays of the same shape.

Returns

out : ndarray or bool

Output array of bools, or a single bool if x1 and x2 are scalars.

See also:

equal, *not_equal*, *greater_equal*, *less_equal*, *less*

numpy.core.defchararray.**less**(*x1*, *x2*)

Return (x1 < x2) element-wise.

Unlike *numpy.greater*, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters

x1, x2 : array_like of str or unicode

Input arrays of the same shape.

Returns

out : ndarray or bool

Output array of bools, or a single bool if x1 and x2 are scalars.

See also:

equal, *not_equal*, *greater_equal*, *less_equal*, *greater*

3.4.3 String information

count(a, sub[, start, end])	Returns an array with the number of non-overlapping occurrences of substring *sub* in the rang
find(a, sub[, start, end])	For each element, return the lowest index in the string where substring *sub* is found.
index(a, sub[, start, end])	Like *find*, but raises *ValueError* when the substring is not found.
isalpha(a)	Returns true for each element if all characters in the string are alphabetic and there is at least o
isdecimal(a)	For each element, return True if there are only decimal characters in the element.
isdigit(a)	Returns true for each element if all characters in the string are digits and there is at least one ch
islower(a)	Returns true for each element if all cased characters in the string are lowercase and there is at l
isnumeric(a)	For each element, return True if there are only numeric characters in the element.
isspace(a)	Returns true for each element if there are only whitespace characters in the string and there is
istitle(a)	Returns true for each element if the element is a titlecased string and there is at least one chara
isupper(a)	Returns true for each element if all cased characters in the string are uppercase and there is at l
rfind(a, sub[, start, end])	For each element in *a*, return the highest index in the string where substring *sub* is found, such
rindex(a, sub[, start, end])	Like *rfind*, but raises *ValueError* when the substring *sub* is not found.
startswith(a, prefix[, start, end])	Returns a boolean array which is *True* where the string element in *a* starts with *prefix*, otherwi

numpy.core.defchararray.**count**(*a*, *sub*, *start=0*, *end=None*)

Returns an array with the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*].

Calls *str.count* element-wise.

Parameters

a : array_like of str or unicode

sub : str or unicode

The substring to search for.

start, end : int, optional

Optional arguments *start* and *end* are interpreted as slice notation to specify the range in which to count.

Returns

out : ndarray

Output array of ints.

See also:

`str.count`

Examples

```
>>> c = np.array(['aAaAaA', '  aA  ', 'abBABba'])
>>> c
array(['aAaAaA', '  aA  ', 'abBABba'],
    dtype='|S7')
>>> np.char.count(c, 'A')
array([3, 1, 1])
>>> np.char.count(c, 'aA')
array([3, 1, 0])
>>> np.char.count(c, 'A', start=1, end=4)
array([2, 1, 1])
>>> np.char.count(c, 'A', start=1, end=3)
array([1, 0, 0])
```

numpy.core.defchararray.**find**(*a*, *sub*, *start=0*, *end=None*)

For each element, return the lowest index in the string where substring *sub* is found.

Calls *str.find* element-wise.

For each element, return the lowest index in the string where substring *sub* is found, such that *sub* is contained in the range [*start*, *end*].

Parameters

a : array_like of str or unicode

sub : str or unicode

start, end : int, optional

Optional arguments *start* and *end* are interpreted as in slice notation.

Returns

out : ndarray or int

Output array of ints. Returns -1 if *sub* is not found.

See also:

`str.find`

numpy.core.defchararray.**index**(*a*, *sub*, *start=0*, *end=None*)

Like *find*, but raises *ValueError* when the substring is not found.

Calls *str.index* element-wise.

Parameters

a : array_like of str or unicode

sub : str or unicode

start, end : int, optional

Returns

out : ndarray

Output array of ints. Returns -1 if *sub* is not found.

See also:

find, str.find

numpy.core.defchararray.**isalpha**(*a*)

Returns true for each element if all characters in the string are alphabetic and there is at least one character, false otherwise.

Calls *str.isalpha* element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

a : array_like of str or unicode

Returns

out : ndarray

Output array of bools

See also:

str.isalpha

numpy.core.defchararray.**isdecimal**(*a*)

For each element, return True if there are only decimal characters in the element.

Calls *unicode.isdecimal* element-wise.

Decimal characters include digit characters, and all characters that that can be used to form decimal-radix numbers, e.g. U+0660, ARABIC-INDIC DIGIT ZERO.

Parameters

a : array_like, unicode

Input array.

Returns

out : ndarray, bool

Array of booleans identical in shape to *a*.

See also:

unicode.isdecimal

numpy.core.defchararray.**isdigit**(*a*)

Returns true for each element if all characters in the string are digits and there is at least one character, false otherwise.

Calls *str.isdigit* element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

a : array_like of str or unicode

Returns

out : ndarray

Output array of bools

See also:

str.isdigit

numpy.core.defchararray.**islower**(*a*)

Returns true for each element if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

Calls *str.islower* element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

a : array_like of str or unicode

Returns

out : ndarray

Output array of bools

See also:

`str.islower`

numpy.core.defchararray.**isnumeric**(*a*)

For each element, return True if there are only numeric characters in the element.

Calls *unicode.isnumeric* element-wise.

Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. `U+2155, VULGAR FRACTION ONE FIFTH`.

Parameters

a : array_like, unicode

Input array.

Returns

out : ndarray, bool

Array of booleans of same shape as *a*.

See also:

`unicode.isnumeric`

numpy.core.defchararray.**isspace**(*a*)

Returns true for each element if there are only whitespace characters in the string and there is at least one character, false otherwise.

Calls *str.isspace* element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

a : array_like of str or unicode

Returns

out : ndarray

Output array of bools

See also:

`str.isspace`

numpy.core.defchararray.**istitle**(*a*)

Returns true for each element if the element is a titlecased string and there is at least one character, false otherwise.

Call *str.istitle* element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters
a : array_like of str or unicode

Returns
out : ndarray

Output array of bools

See also:

`str.istitle`

`numpy.core.defchararray.`**`isupper`**`(a)`
Returns true for each element if all cased characters in the string are uppercase and there is at least one character, false otherwise.

Call *str.isupper* element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters
a : array_like of str or unicode

Returns
out : ndarray

Output array of bools

See also:

`str.isupper`

`numpy.core.defchararray.`**`rfind`**`(a, sub, start=0, end=None)`
For each element in *a*, return the highest index in the string where substring *sub* is found, such that *sub* is contained within [*start*, *end*].

Calls *str.rfind* element-wise.

Parameters
a : array-like of str or unicode

sub : str or unicode

start, end : int, optional

Optional arguments *start* and *end* are interpreted as in slice notation.

Returns
out : ndarray

Output array of ints. Return -1 on failure.

See also:

`str.rfind`

`numpy.core.defchararray.`**`rindex`**`(a, sub, start=0, end=None)`
Like *rfind*, but raises *ValueError* when the substring *sub* is not found.

Calls *str.rindex* element-wise.

Parameters
a : array-like of str or unicode

sub : str or unicode

start, end : int, optional

Returns

out : ndarray

Output array of ints.

See also:

rfind, `str.rindex`

`numpy.core.defchararray.`**`startswith`**(*a*, *prefix*, *start=0*, *end=None*)

Returns a boolean array which is *True* where the string element in *a* starts with *prefix*, otherwise *False*.

Calls *str.startswith* element-wise.

Parameters

a : array_like of str or unicode

prefix : str

start, end : int, optional

With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

Returns

out : ndarray

Array of booleans

See also:

`str.startswith`

3.4.4 Convenience class

chararray	Provides a convenient view on arrays of string and unicode values.

class `numpy.core.defchararray.`**`chararray`**

Provides a convenient view on arrays of string and unicode values.

Note: The *chararray* class exists for backwards compatibility with Numarray, it is not recommended for new development. Starting from numpy 1.4, if one needs arrays of strings, it is recommended to use arrays of *dtype* `object_`, `string_` or `unicode_`, and use the free functions in the `numpy.char` module for fast vectorized string operations.

Versus a regular Numpy array of type *str* or *unicode*, this class adds the following functionality:

1. values automatically have whitespace removed from the end when indexed
2. comparison operators automatically remove whitespace from the end when comparing values
3. vectorized string operations are provided as methods (e.g. *endswith*) and infix operators (e.g. `"+"`, `"*"`, `"%"`)

chararrays should be created using `numpy.char.array` or `numpy.char.asarray`, rather than this constructor directly.

This constructor creates the array, using *buffer* (with *offset* and *strides*) if it is not `None`. If *buffer* is `None`, then constructs a new array with *strides* in "C order", unless both `len(shape) >= 2` and

`order='Fortran'`, in which case *strides* is in "Fortran order".

Parameters

shape : tuple

Shape of the array.

itemsize : int, optional

Length of each array element, in number of characters. Default is 1.

unicode : bool, optional

Are the array elements of type unicode (True) or string (False). Default is False.

buffer : int, optional

Memory address of the start of the array data. Default is None, in which case a new array is created.

offset : int, optional

Fixed stride displacement from the beginning of an axis? Default is 0. Needs to be >=0.

strides : array_like of ints, optional

Strides for the array (see `ndarray.strides` for full description). Default is None.

order : {'C', 'F'}, optional

The order in which the array data is stored in memory: 'C' -> "row major" order (the default), 'F' -> "column major" (Fortran) order.

Examples

```
>>> charar = np.chararray((3, 3))
>>> charar[:] = 'a'
>>> charar
chararray([['a', 'a', 'a'],
       ['a', 'a', 'a'],
       ['a', 'a', 'a']],
      dtype='|S1')
```

```
>>> charar = np.chararray(charar.shape, itemsize=5)
>>> charar[:] = 'abc'
>>> charar
chararray([['abc', 'abc', 'abc'],
       ['abc', 'abc', 'abc'],
       ['abc', 'abc', 'abc']],
      dtype='|S5')
```

Attributes

T	Same as self.transpose(), except that self is returned if self.ndim < 2.
base	Base object if memory is from some other object.
ctypes	An object to simplify the interaction of the array with the ctypes module.
data	Python buffer object pointing to the start of the array's data.
dtype	Data-type of the array's elements.
flags	Information about the memory layout of the array.
flat	A 1-D iterator over the array.
imag	The imaginary part of the array.

Continued on next page

Table 3.27 – continued from previous page

itemsize	Length of one array element in bytes.
nbytes	Total bytes consumed by the elements of the array.
ndim	Number of array dimensions.
real	The real part of the array.
shape	Tuple of array dimensions.
size	Number of elements in the array.
strides	Tuple of bytes to step in each dimension when traversing an array.

chararray.**T**

Same as self.transpose(), except that self is returned if self.ndim < 2.

Examples

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

chararray.**base**

Base object if memory is from some other object.

Examples

The base of an array that owns its memory is None:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```
>>> y = x[2:]
>>> y.base is x
True
```

chararray.**ctypes**

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters
None

Returns
c : Python object

Possessing attributes data, shape, strides, etc.

See also:

`numpy.ctypeslib`

Notes

Below are the public attributes of this object which were documented in "Guide to NumPy" (we have omitted undocumented public attributes, as well as documented private attributes):

- data: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].
- shape (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.
- strides (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.
- data_as(obj): Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).
- shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).
- strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

Examples

```
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
```

```
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

chararray.**data**
Python buffer object pointing to the start of the array's data.

chararray.**dtype**
Data-type of the array's elements.

Parameters
None

Returns
d : numpy dtype object

See also:

numpy.dtype

Examples

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

chararray.**flags**
Information about the memory layout of the array.

Notes

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set `False`.
- ALIGNED can only be set `True` if the data is truly aligned.
- WRITEABLE can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

Attributes

C_CONTIGUOUS (C)	The data is in a single, C-style contiguous segment.
F_CONTIGUOUS (F)	The data is in a single, Fortran-style contiguous segment.
OWNDATA (O)	The array owns the memory it uses or borrows it from another object.
WRITEABLE (W)	The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.
ALIGNED (A)	The data and all elements are aligned appropriately for the hardware.
UPDATEIFCOPY (U)	This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.
FNC	F_CONTIGUOUS and not C_CONTIGUOUS.
FORC	F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).
BEHAVED (B)	ALIGNED and WRITEABLE.
CARRAY (CA)	BEHAVED and C_CONTIGUOUS.
FARRAY (FA)	BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

chararray.**flat**

A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See also:

flatten

Return a copy of the array collapsed into one dimension.

flatiter

Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```

```
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

chararray.**imag**
The imaginary part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.        ,  0.70710678])
>>> x.imag.dtype
dtype('float64')
```

chararray.**itemsize**
Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

chararray.**nbytes**
Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

chararray.**ndim**
Number of array dimensions.

Examples

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

```
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

chararray.**real**

The real part of the array.

See also:

numpy.real
equivalent function

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.        ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

chararray.**shape**

Tuple of array dimensions.

Notes

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

chararray.**size**

Number of elements in the array.

Equivalent to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

chararray.**strides**

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the "ndarray.rst" file in the NumPy reference guide.

See also:

`numpy.lib.stride_tricks.as_strided`

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array *x* will be `(20, 4)`.

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> 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]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

Methods

astype(dtype[, order, casting, subok, copy])	Copy of the array, cast to a specified type.
copy([order])	Return a copy of the array.

Table 3.28 – continued

count(sub[, start, end])	Returns an array with the number of non-overlapping occurrences of substring *sub* in
decode([encoding, errors])	Calls *str.decode* element-wise.
dump(file)	Dump a pickle of the array to the specified file.
dumps()	Returns the pickle of the array as a string.
encode([encoding, errors])	Calls *str.encode* element-wise.
endswith(suffix[, start, end])	Returns a boolean array which is *True* where the string element in *self* ends with *suff*
expandtabs([tabsize])	Return a copy of each string element where all tab characters are replaced by one or m
fill(value)	Fill the array with a scalar value.
find(sub[, start, end])	For each element, return the lowest index in the string where substring *sub* is found.
flatten([order])	Return a copy of the array collapsed into one dimension.
getfield(dtype[, offset])	Returns a field of the given array as a certain type.
index(sub[, start, end])	Like *find*, but raises *ValueError* when the substring is not found.
isalnum()	Returns true for each element if all characters in the string are alphanumeric and there
isalpha()	Returns true for each element if all characters in the string are alphabetic and there is
isdecimal()	For each element in *self*, return True if there are only decimal characters in the eleme
isdigit()	Returns true for each element if all characters in the string are digits and there is at le
islower()	Returns true for each element if all cased characters in the string are lowercase and th
isnumeric()	For each element in *self*, return True if there are only numeric characters in the eleme
isspace()	Returns true for each element if there are only whitespace characters in the string and
istitle()	Returns true for each element if the element is a titlecased string and there is at least
isupper()	Returns true for each element if all cased characters in the string are uppercase and th
item(*args)	Copy an element of an array to a standard Python scalar and return it.
join(seq)	Return a string which is the concatenation of the strings in the sequence *seq*.
ljust(width[, fillchar])	Return an array with the elements of *self* left-justified in a string of length *width*.
lower()	Return an array with the elements of *self* converted to lowercase.
lstrip([chars])	For each element in *self*, return a copy with the leading characters removed.
nonzero()	Return the indices of the elements that are non-zero.
put(indices, values[, mode])	Set `a.flat[n] = values[n]` for all *n* in indices.
ravel([order])	Return a flattened array.
repeat(repeats[, axis])	Repeat elements of an array.
replace(old, new[, count])	For each element in *self*, return a copy of the string with all occurrences of substring
reshape(shape[, order])	Returns an array containing the same data with a new shape.
resize(new_shape[, refcheck])	Change shape and size of array in-place.
rfind(sub[, start, end])	For each element in *self*, return the highest index in the string where substring *sub* is
rindex(sub[, start, end])	Like *rfind*, but raises *ValueError* when the substring *sub* is not found.
rjust(width[, fillchar])	Return an array with the elements of *self* right-justified in a string of length *width*.
rsplit([sep, maxsplit])	For each element in *self*, return a list of the words in the string, using *sep* as the delim
rstrip([chars])	For each element in *self*, return a copy with the trailing characters removed.
searchsorted(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
setfield(val, dtype[, offset])	Put a value into a specified place in a field defined by a data-type.
setflags([write, align, uic])	Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.
sort([axis, kind, order])	Sort an array, in-place.
split([sep, maxsplit])	For each element in *self*, return a list of the words in the string, using *sep* as the delim
splitlines([keepends])	For each element in *self*, return a list of the lines in the element, breaking at line boun
squeeze([axis])	Remove single-dimensional entries from the shape of *a*.
startswith(prefix[, start, end])	Returns a boolean array which is *True* where the string element in *self* starts with *pre*
strip([chars])	For each element in *self*, return a copy with the leading and trailing characters remove
swapaxes(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
swapcase()	For each element in *self*, return a copy of the string with uppercase characters convert
take(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.

Table 3.28 – continued

title()	For each element in *self*, return a titlecased version of the string: words start with upp
tofile(fid[, sep, format])	Write array to a file as text or binary (default).
tolist()	Return the array as a (possibly nested) list.
tostring([order])	Construct Python bytes containing the raw data bytes in the array.
translate(table[, deletechars])	For each element in *self*, return a copy of the string where all characters occurring in
transpose(*axes)	Returns a view of the array with axes transposed.
upper()	Return an array with the elements of *self* converted to uppercase.
view([dtype, type])	New view of array with the same data.
zfill(width)	Return the numeric string left-filled with zeros in a string of length *width*.

chararray.**astype**(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)

Copy of the array, cast to a specified type.

Parameters

dtype : str or dtype

Typecode or data-type to which the array is cast.

order : {'C', 'F', 'A', 'K'}, optional

Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional

Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

subok : bool, optional

If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy : bool, optional

By default, astype always returns a newly allocated array. If this is set to false, and the dtype, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

Returns

arr_t : ndarray

Unless copy is False and the other conditions for returning the input array are satisfied (see description for copy input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by dtype, *order*.

Raises

ComplexWarning

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Notes

Starting in NumPy 1.9, 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.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

`chararray.`**`copy`**(*order='C'*)

Return a copy of the array.

Parameters

order : {'C', 'F', 'A', 'K'}, optional

Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and :func:numpy.copy are very similar, but have different default values for their order= arguments.)

See also:

`numpy.copy`, `numpy.copyto`

Examples

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

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

`chararray.`**`count`**(*sub*, *start=0*, *end=None*)

Returns an array with the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*].

See also:

`char.count`

chararray.**decode**(*encoding=None*, *errors=None*)
Calls *str.decode* element-wise.

See also:

char.decode

chararray.**dump**(*file*)
Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters
file : str

A string naming the dump file.

chararray.**dumps**()
Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

Parameters
None

chararray.**encode**(*encoding=None*, *errors=None*)
Calls *str.encode* element-wise.

See also:

char.encode

chararray.**endswith**(*suffix*, *start=0*, *end=None*)
Returns a boolean array which is *True* where the string element in *self* ends with *suffix*, otherwise *False*.

See also:

char.endswith

chararray.**expandtabs**(*tabsize=8*)
Return a copy of each string element where all tab characters are replaced by one or more spaces.

See also:

char.expandtabs

chararray.**fill**(*value*)
Fill the array with a scalar value.

Parameters
value : scalar

All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1.,  1.])
```

chararray.**find**(*sub*, *start=0*, *end=None*)
For each element, return the lowest index in the string where substring *sub* is found.

See also:

`char.find`

chararray.**flatten**(*order='C'*)

Return a copy of the array collapsed into one dimension.

Parameters

order : {'C', 'F', 'A', 'K'}, optional

'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

Returns

y : ndarray

A copy of the input array, flattened to one dimension.

See also:

ravel

Return a flattened array.

flat

A 1-D flat iterator over the array.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

chararray.**getfield**(*dtype*, *offset=0*)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype : str or dtype

The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset : int

Number of bytes to skip before beginning the element view.

Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[ 1.+1.j,  0.+0.j],
       [ 0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
```

```
array([[ 1.,  0.],
       [ 0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[ 1.,  0.],
   [ 0.,  4.]])
```

chararray.**index**(*sub*, *start=0*, *end=None*)
Like *find*, but raises *ValueError* when the substring is not found.

See also:

char.index

chararray.**isalnum**()
Returns true for each element if all characters in the string are alphanumeric and there is at least one character, false otherwise.

See also:

char.isalnum

chararray.**isalpha**()
Returns true for each element if all characters in the string are alphabetic and there is at least one character, false otherwise.

See also:

char.isalpha

chararray.**isdecimal**()
For each element in *self*, return True if there are only decimal characters in the element.

See also:

char.isdecimal

chararray.**isdigit**()
Returns true for each element if all characters in the string are digits and there is at least one character, false otherwise.

See also:

char.isdigit

chararray.**islower**()
Returns true for each element if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

See also:

char.islower

chararray.**isnumeric**()
For each element in *self*, return True if there are only numeric characters in the element.

See also:

char.isnumeric

chararray.**isspace**()
Returns true for each element if there are only whitespace characters in the string and there is at least one character, false otherwise.

See also:

`char.isspace`

`chararray.`**`istitle`**`()`

Returns true for each element if the element is a titlecased string and there is at least one character, false otherwise.

See also:

`char.istitle`

`chararray.`**`isupper`**`()`

Returns true for each element if all cased characters in the string are uppercase and there is at least one character, false otherwise.

See also:

`char.isupper`

`chararray.`**`item`**`(*args)`

Copy an element of an array to a standard Python scalar and return it.

Parameters

***args** : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z : Standard Python scalar object

A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
```

```
>>> x.item((2, 2))
3
```

chararray.**join**(*seq*)
Return a string which is the concatenation of the strings in the sequence *seq*.

See also:

char.join

chararray.**ljust**(*width*, *fillchar=' '*)
Return an array with the elements of *self* left-justified in a string of length *width*.

See also:

char.ljust

chararray.**lower**()
Return an array with the elements of *self* converted to lowercase.

See also:

char.lower

chararray.**lstrip**(*chars=None*)
For each element in *self*, return a copy with the leading characters removed.

See also:

char.lstrip

chararray.**nonzero**()
Return the indices of the elements that are non-zero.

Refer to numpy.nonzero for full documentation.

See also:

numpy.nonzero
equivalent function

chararray.**put**(*indices*, *values*, *mode='raise'*)
Set a.flat[n] = values[n] for all *n* in indices.

Refer to numpy.put for full documentation.

See also:

numpy.put
equivalent function

chararray.**ravel**([*order*])
Return a flattened array.

Refer to numpy.ravel for full documentation.

See also:

numpy.ravel
equivalent function

ndarray.flat
a flat iterator on the array.

chararray.**repeat**(*repeats*, *axis=None*)
Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

See also:

numpy.repeat
equivalent function

chararray.**replace**(*old*, *new*, *count=None*)
For each element in *self*, return a copy of the string with all occurrences of substring *old* replaced by `new`.

See also:

`char.replace`

chararray.**reshape**(*shape*, *order='C'*)
Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

See also:

numpy.reshape
equivalent function

chararray.**resize**(*new_shape*, *refcheck=True*)
Change shape and size of array in-place.

Parameters
new_shape : tuple of ints, or *n* ints

Shape of resized array.

refcheck : bool, optional

If False, reference count will not be checked. Default is True.

Returns
None

Raises
ValueError

If *a* does not own its own data or references or views to it exist, and the data memory must be changed.

SystemError

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

See also:

resize
Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

`chararray.`**`rfind`**(*sub*, *start=0*, *end=None*)

For each element in *self*, return the highest index in the string where substring *sub* is found, such that *sub* is contained within [*start*, *end*].

See also:

`char.rfind`

`chararray.`**`rindex`**(*sub*, *start=0*, *end=None*)

Like *rfind*, but raises *ValueError* when the substring *sub* is not found.

See also:

`char.rindex`

chararray.**rjust**(*width*, *fillchar=' '*)
Return an array with the elements of *self* right-justified in a string of length *width*.

See also:

`char.rjust`

chararray.**rsplit**(*sep=None*, *maxsplit=None*)
For each element in *self*, return a list of the words in the string, using *sep* as the delimiter string.

See also:

`char.rsplit`

chararray.**rstrip**(*chars=None*)
For each element in *self*, return a copy with the trailing characters removed.

See also:

`char.rstrip`

chararray.**searchsorted**(*v*, *side='left'*, *sorter=None*)
Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see `numpy.searchsorted`

See also:

numpy.searchsorted
equivalent function

chararray.**setfield**(*val*, *dtype*, *offset=0*)
Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *dtype* and beginning *offset* bytes into the field.

Parameters
val : object

Value to be placed in field.

dtype : dtype object

Data-type of the field in which to place *val*.

offset : int, optional

The number of bytes into the field at which to place *val*.

Returns
None

See also:

getfield

Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
```

```
       array([[3, 3, 3],
              [3, 3, 3],
              [3, 3, 3]])
       >>> x
       array([[  1.00000000e+000,   1.48219694e-323,   1.48219694e-323],
              [  1.48219694e-323,   1.00000000e+000,   1.48219694e-323],
              [  1.48219694e-323,   1.48219694e-323,   1.00000000e+000]])
       >>> x.setfield(np.eye(3), np.int32)
       >>> x
       array([[ 1.,  0.,  0.],
              [ 0.,  1.,  0.],
              [ 0.,  0.,  1.]])
```

chararray.**setflags**(*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write : bool, optional

Describes whether or not *a* can be written to.

align : bool, optional

Describes whether or not *a* is aligned properly for its type.

uic : bool, optional

Describes whether or not *a* is a copy of another "base" array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

Examples

```
       >>> y
       array([[3, 1, 7],
              [2, 0, 0],
              [8, 5, 9]])
       >>> y.flags
         C_CONTIGUOUS : True
         F_CONTIGUOUS : False
         OWNDATA : True
         WRITEABLE : True
```

```
  ALIGNED : True
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

chararray.**sort**(*axis=-1*, *kind='quicksort'*, *order=None*)

Sort an array, in-place.

Parameters

axis : int, optional

Axis along which to sort. Default is -1, which means sort along the last axis.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional

Sorting algorithm. Default is 'quicksort'.

order : str or list of str, optional

When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See also:

numpy.sort
Return a sorted copy of an array.

argsort
Indirect sort.

lexsort
Indirect stable sort on multiple keys.

searchsorted
Find elements in sorted array.

partition
Partial sort.

Notes

See `sort` for notes on the different sorting algorithms.

Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
```

```
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

chararray.**split**(*sep=None*, *maxsplit=None*)
For each element in *self*, return a list of the words in the string, using *sep* as the delimiter string.

See also:

`char.split`

chararray.**splitlines**(*keepends=None*)
For each element in *self*, return a list of the lines in the element, breaking at line boundaries.

See also:

`char.splitlines`

chararray.**squeeze**(*axis=None*)
Remove single-dimensional entries from the shape of *a*.

Refer to `numpy.squeeze` for full documentation.

See also:

numpy.squeeze
equivalent function

chararray.**startswith**(*prefix*, *start=0*, *end=None*)
Returns a boolean array which is *True* where the string element in *self* starts with *prefix*, otherwise *False*.

See also:

`char.startswith`

chararray.**strip**(*chars=None*)
For each element in *self*, return a copy with the leading and trailing characters removed.

See also:

`char.strip`

chararray.**swapaxes**(*axis1*, *axis2*)
Return a view of the array with *axis1* and *axis2* interchanged.

Refer to `numpy.swapaxes` for full documentation.

See also:

numpy.swapaxes
equivalent function

chararray.**swapcase**()
For each element in *self*, return a copy of the string with uppercase characters converted to lowercase and vice versa.

See also:

`char.swapcase`

chararray.**take**(*indices*, *axis=None*, *out=None*, *mode='raise'*)
Return an array formed from the elements of *a* at the given indices.

Refer to `numpy.take` for full documentation.

See also:

numpy.take
equivalent function

chararray.**title**()
For each element in *self*, return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase.

See also:

`char.title`

chararray.**tofile**(*fid*, *sep=""*, *format="%s"*)
Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function fromfile().

Parameters
fid : file or str

An open file object, or a string containing a filename.

sep : str

Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format : str

Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

chararray.**tolist**()
Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

Parameters
none

Returns
y : list

The possibly nested list of array elements.

Notes

The array may be recreated, `a = np.array(a.tolist())`.

Examples

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

chararray.**tostring**(*order='C'*)
Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

This function is a compatibility alias for tobytes. Despite its name it returns bytes not strings.

Parameters
order : {'C', 'F', None}, optional

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns
s : bytes

Python bytes exhibiting a copy of *a*'s raw data.

Examples

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

chararray.**translate**(*table*, *deletechars=None*)
For each element in *self*, return a copy of the string where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation table.

See also:

`char.translate`

chararray.**transpose**(**axes*)
Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

Parameters

axes : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

Returns

out : ndarray

View of *a*, with axes suitably permuted.

See also:

`ndarray.T`
Array property returning the array transposed.

Examples

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

`chararray.`**`upper`**`()`
Return an array with the elements of *self* converted to uppercase.

See also:

`char.upper`

`chararray.`**`view`**`(`*dtype=None*, *type=None*`)`
New view of array with the same data.

Parameters

dtype : data-type or ndarray sub-class, optional

Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

type : Python type, optional

Type of the returned view, e.g., ndarray or matrix. Again, the default None results in type preservation.

Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrixlib.defmatrix.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([ 2.,  3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> print(x)
[(1, 20) (3, 4)]
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

chararray.**zfill**(*width*)
Return the numeric string left-filled with zeros in a string of length *width*.

See also:

char.zfill

argsort	

3.5 C-Types Foreign Function Interface (numpy.ctypeslib)

numpy.ctypeslib.**as_array**(*obj*, *shape=None*)
Create a numpy array from a ctypes array or a ctypes POINTER. The numpy array shares the memory with the ctypes object.

The size parameter must be given if converting from a ctypes POINTER. The size parameter is ignored if converting from a ctypes array

numpy.ctypeslib.**as_ctypes**(*obj*)
Create and return a ctypes object from a numpy array. Actually anything that exposes the __array_interface__ is accepted.

numpy.ctypeslib.**ctypes_load_library**(**args*, ***kwds*)
ctypes_load_library is deprecated, use *load_library* instead!

It is possible to load a library using >>> lib = ctypes.cdll[<full_path_name>]

But there are cross-platform considerations, such as library file extensions, plus the fact Windows will just load the first library it finds with that name. Numpy supplies the load_library function as a convenience.

Parameters
libname : str

Name of the library, which can have 'lib' as a prefix, but without an extension.

loader_path : str

Where the library can be found.

Returns
ctypes.cdll[libpath] : library object

A ctypes library object

Raises
OSError

If there is no library with the expected extension, or the library is defective and cannot be loaded.

numpy.ctypeslib.**load_library**(*libname*, *loader_path*)
It is possible to load a library using >>> lib = ctypes.cdll[<full_path_name>]

But there are cross-platform considerations, such as library file extensions, plus the fact Windows will just load the first library it finds with that name. Numpy supplies the load_library function as a convenience.

Parameters
libname : str

Name of the library, which can have 'lib' as a prefix, but without an extension.

loader_path : str

Where the library can be found.

Returns
ctypes.cdll[libpath] : library object

A ctypes library object

Raises
OSError

If there is no library with the expected extension, or the library is defective and cannot be loaded.

numpy.ctypeslib.**ndpointer**(*dtype=None*, *ndim=None*, *shape=None*, *flags=None*)
Array-checking restype/argtypes.

An ndpointer instance is used to describe an ndarray in restypes and argtypes specifications. This approach is more flexible than using, for example, `POINTER(c_double)`, since several restrictions can be specified, which are verified upon calling the ctypes function. These include data type, number of dimensions, shape and flags. If a given array does not satisfy the specified restrictions, a `TypeError` is raised.

Parameters
dtype : data-type, optional

Array data-type.

ndim : int, optional

Number of array dimensions.

shape : tuple of ints, optional

Array shape.

flags : str or tuple of str

Array flags; may be one or more of:

- C_CONTIGUOUS / C / CONTIGUOUS
- F_CONTIGUOUS / F / FORTRAN
- OWNDATA / O
- WRITEABLE / W
- ALIGNED / A

- UPDATEIFCOPY / U

Returns

klass : ndpointer type object

A type object, which is an `_ndtpr` instance containing dtype, ndim, shape and flags information.

Raises

TypeError

If a given array does not satisfy the specified restrictions.

Examples

```
>>> clib.somefunc.argtypes = [np.ctypeslib.ndpointer(dtype=np.float64,
...                                                  ndim=1,
...                                                  flags='C_CONTIGUOUS')]
...
>>> clib.somefunc(np.array([1, 2, 3], dtype=np.float64))
...
```

3.6 Datetime Support Functions

3.6.1 Business Day Functions

busdaycalendar	A business day calendar object that efficiently stores information defining valid days for the busday family of

class numpy.**busdaycalendar**

A business day calendar object that efficiently stores information defining valid days for the busday family of functions.

The default valid days are Monday through Friday ("business days"). A busdaycalendar object can be specified with any set of weekly valid days, plus an optional "holiday" dates that always will be invalid.

Once a busdaycalendar object is created, the weekmask and holidays cannot be modified.

New in version 1.7.0.

Parameters

weekmask : str or array_like of bool, optional

A seven-element array indicating which of Monday through Sunday are valid days. May be specified as a length-seven list or array, like [1,1,1,1,1,0,0]; a length-seven string, like '1111100'; or a string like "Mon Tue Wed Thu Fri", made up of 3-character abbreviations for weekdays, optionally separated by white space. Valid abbreviations are: Mon Tue Wed Thu Fri Sat Sun

holidays : array_like of datetime64[D], optional

An array of dates to consider as invalid dates, no matter which weekday they fall upon. Holiday dates may be specified in any order, and NaT (not-a-time) dates are ignored. This list is saved in a normalized form that is suited for fast calculations of valid days.

Returns

out : busdaycalendar

A business day calendar object containing the specified weekmask and holidays values.

See also:

is_busday
Returns a boolean array indicating valid days.

busday_offset
Applies an offset counted in valid days.

busday_count
Counts how many valid days are in a half-open date range.

Examples

```
>>> # Some important days in July
... bdd = np.busdaycalendar(
...             holidays=['2011-07-01', '2011-07-04', '2011-07-17'])
>>> # Default is Monday to Friday weekdays
... bdd.weekmask
array([ True,  True,  True,  True,  True, False, False], dtype='bool')
>>> # Any holidays already on the weekend are removed
... bdd.holidays
array(['2011-07-01', '2011-07-04'], dtype='datetime64[D]')
```

Attributes

weekmask	A copy of the seven-element boolean mask indicating valid days.
holidays	A copy of the holiday array indicating additional invalid days.

busdaycalendar.**weekmask**
A copy of the seven-element boolean mask indicating valid days.

busdaycalendar.**holidays**
A copy of the holiday array indicating additional invalid days.

Note: once a busdaycalendar object is created, you cannot modify the weekmask or holidays. The attributes return copies of internal data.

3.7 Data type routines

3.7.1 Creating data types

dtype	Create a data type object.

class numpy.**dtype**
Create a data type object.

A numpy array is homogeneous, and contains elements described by a dtype object. A dtype object can be constructed from different combinations of fundamental numeric types.

Parameters
obj

Object to be converted to a data type object.

align : bool, optional

Add padding to the fields to match what a C compiler would output for a similar C-struct. Can be `True` only if *obj* is a dictionary or a comma-separated string. If a struct dtype is being created, this also sets a sticky alignment flag `isalignedstruct`.

copy : bool, optional

Make a new copy of the data-type object. If `False`, the result may just be a reference to a built-in data-type object.

See also:

`result_type`

Examples

Using array-scalar type:

```
>>> np.dtype(np.int16)
dtype('int16')
```

Structured type, one field name 'f1', containing int16:

```
>>> np.dtype([('f1', np.int16)])
dtype([('f1', '<i2')])
```

Structured type, one field named 'f1', in itself containing a structured type with one field:

```
>>> np.dtype([('f1', [('f1', np.int16)])])
dtype([('f1', [('f1', '<i2')])])
```

Structured type, two fields: the first field contains an unsigned int, the second an int32:

```
>>> np.dtype([('f1', np.uint), ('f2', np.int32)])
dtype([('f1', '<u4'), ('f2', '<i4')])
```

Using array-protocol type strings:

```
>>> np.dtype([('a','f8'),('b','S10')])
dtype([('a', '<f8'), ('b', '|S10')])
```

Using comma-separated field formats. The shape is (2,3):

```
>>> np.dtype("i4, (2,3)f8")
dtype([('f0', '<i4'), ('f1', '<f8', (2, 3))])
```

Using tuples. `int` is a fixed type, 3 the field's shape. `void` is a flexible type, here of size 10:

```
>>> np.dtype([('hello',(np.int,3)),('world',np.void,10)])
dtype([('hello', '<i4', 3), ('world', '|V10')])
```

Subdivide `int16` into 2 `int8`'s, called x and y. 0 and 1 are the offsets in bytes:

```
>>> np.dtype((np.int16, {'x':(np.int8,0), 'y':(np.int8,1)}))
dtype(('<i2', [('x', '|i1'), ('y', '|i1')]))
```

Using dictionaries. Two fields named 'gender' and 'age':

```
>>> np.dtype({'names':['gender','age'], 'formats':['S1',np.uint8]})
dtype([('gender', '|S1'), ('age', '|u1')])
```

Offsets in bytes, here 0 and 25:

```
>>> np.dtype({'surname':('S25',0),'age':(np.uint8,25)})
dtype([('surname', '|S25'), ('age', '|u1')])
```

Attributes

base	
descr	Array-interface compliant full description of the data-type.
fields	Dictionary of named fields defined for this data type, or None.
hasobject	Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.
isalignedstruct	Boolean indicating whether the dtype is a struct which maintains field alignment.
isbuiltin	Integer indicating how this dtype relates to the built-in dtypes.
isnative	Boolean indicating whether the byte order of this dtype is native to the platform.
metadata	
name	A bit-width name for this data-type.
names	Ordered list of field names, or None if there are no fields.
shape	Shape tuple of the sub-array if this data type describes a sub-array, and () otherwise.
str	The array-protocol typestring of this data-type object.
subdtype	Tuple (item_dtype, shape) if this *dtype* describes a sub-array, and None otherwise.

dtype.**base**

dtype.**descr**

Array-interface compliant full description of the data-type.

The format is that required by the 'descr' key in the *__array_interface__* attribute.

dtype.**fields**

Dictionary of named fields defined for this data type, or None.

The dictionary is indexed by keys that are the names of the fields. Each entry in the dictionary is a tuple fully describing the field:

```
(dtype, offset[, title])
```

If present, the optional title can be any object (if it is a string or unicode then it will also be a key in the fields dictionary, otherwise it's meta-data). Notice also that the first two elements of the tuple can be passed directly as arguments to the ndarray.getfield and ndarray.setfield methods.

See also:

ndarray.getfield, *ndarray.setfield*

Examples

```
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> print(dt.fields)
{'grades': (dtype(('float64',(2,))), 16), 'name': (dtype('|S16'), 0)}
```

dtype.**hasobject**

Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.

Recall that what is actually in the ndarray memory representing the Python object is the memory address of that object (a pointer). Special handling may be required, and this attribute is useful for distinguishing data types that may contain arbitrary Python objects and data-types that won't.

dtype.**isalignedstruct**
: Boolean indicating whether the dtype is a struct which maintains field alignment. This flag is sticky, so when combining multiple structs together, it is preserved and produces new dtypes which are also aligned.

dtype.**isbuiltin**
: Integer indicating how this dtype relates to the built-in dtypes.

  Read-only.

0	if this is a structured array type, with fields
1	if this is a dtype compiled into numpy (such as ints, floats etc)
2	if the dtype is for a user-defined numpy type A user-defined type uses the numpy C-API machinery to extend numpy to handle a new array type. See user.user-defined-data-types in the Numpy manual.

Examples

```
>>> dt = np.dtype('i2')
>>> dt.isbuiltin
1
>>> dt = np.dtype('f8')
>>> dt.isbuiltin
1
>>> dt = np.dtype([('field1', 'f8')])
>>> dt.isbuiltin
0
```

dtype.**isnative**
: Boolean indicating whether the byte order of this dtype is native to the platform.

dtype.**metadata**

dtype.**name**
: A bit-width name for this data-type.

  Un-sized flexible data-type objects do not have this attribute.

dtype.**names**
: Ordered list of field names, or `None` if there are no fields.

  The names are ordered according to increasing byte offset. This can be used, for example, to walk through all of the named fields in offset order.

Examples

```
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> dt.names
('name', 'grades')
```

dtype.**shape**
: Shape tuple of the sub-array if this data type describes a sub-array, and `()` otherwise.

dtype.**str**
: The array-protocol typestring of this data-type object.

dtype.**subdtype**
: Tuple `(item_dtype, shape)` if this *dtype* describes a sub-array, and None otherwise.

  The *shape* is the fixed shape of the sub-array described by this data type, and *item_dtype* the data type of the array.

If a field whose dtype object has this attribute is retrieved, then the extra dimensions implied by *shape* are tacked on to the end of the retrieved array.

Methods

newbyteorder([new_order])	Return a new dtype with a different byte order.

dtype.**newbyteorder**(*new_order='S'*)

Return a new dtype with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

Parameters

new_order : string, optional

Byte order to force; a value from the byte order specifications below. The default value ('S') results in swapping the current byte order. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

The code does a case-insensitive check on the first letter of *new_order* for these alternatives. For example, any of '>' or 'B' or 'b' or 'brian' are valid to specify big-endian.

Returns

new_dtype : dtype

New dtype object with the given change to the byte order.

Notes

Changes are also made in all fields and sub-arrays of the data type.

Examples

```
>>> import sys
>>> sys_is_le = sys.byteorder == 'little'
>>> native_code = sys_is_le and '<' or '>'
>>> swapped_code = sys_is_le and '>' or '<'
>>> native_dt = np.dtype(native_code+'i2')
>>> swapped_dt = np.dtype(swapped_code+'i2')
>>> native_dt.newbyteorder('S') == swapped_dt
True
>>> native_dt.newbyteorder() == swapped_dt
True
>>> native_dt == swapped_dt.newbyteorder('S')
True
>>> native_dt == swapped_dt.newbyteorder('=')
True
>>> native_dt == swapped_dt.newbyteorder('N')
True
>>> native_dt == native_dt.newbyteorder('|')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('<')
```

```
        True
        >>> np.dtype('<i2') == native_dt.newbyteorder('L')
        True
        >>> np.dtype('>i2') == native_dt.newbyteorder('>')
        True
        >>> np.dtype('>i2') == native_dt.newbyteorder('B')
        True
```

3.7.2 Data type information

3.7.3 Data type testing

3.7.4 Miscellaneous

3.8 Optionally Scipy-accelerated routines (`numpy.dual`)

Aliases for functions which may be accelerated by Scipy.

Scipy can be built to use accelerated or otherwise improved libraries for FFTs, linear algebra, and special functions. This module allows developers to transparently support these accelerated functions when scipy is available but still support users who have only installed Numpy.

3.8.1 Linear algebra

3.8.2 FFT

3.8.3 Other

3.9 Mathematical functions with automatic domain (`numpy.emath`)

Note: `numpy.emath` is a preferred alias for *numpy.lib.scimath*, available after *numpy* is imported.

Wrapper functions to more user-friendly calling of certain math functions whose output data-type is different than the input data-type in certain domains of the input.

For example, for functions like `log` with branch cuts, the versions in this module provide the mathematically valid

answers in the complex plane:

```
>>> import math
>>> from numpy.lib import scimath
>>> scimath.log(-math.exp(1)) == (1+1j*math.pi)
True
```

Similarly, `sqrt`, other base logarithms, `power` and trig functions are correctly handled. See their respective docstrings for specific examples.

3.10 Floating point error handling

3.10.1 Setting and getting error handling

3.10.2 Internal functions

3.11 Discrete Fourier Transform (`numpy.fft`)

3.11.1 Standard FFTs

3.11.2 Real FFTs

3.11.3 Hermitian FFTs

3.11.4 Helper routines

3.11.5 Background information

Fourier analysis is fundamentally a method for expressing a function as a sum of periodic components, and for recovering the function from those components. When both the function and its Fourier transform are replaced with discretized counterparts, it is called the discrete Fourier transform (DFT). The DFT has become a mainstay of numerical computing in part because of a very fast algorithm for computing it, called the Fast Fourier Transform (FFT), which was known to Gauss (1805) and was brought to light in its current form by Cooley and Tukey *[CT]*. Press et al. *[NR]* provide an accessible introduction to Fourier analysis and its applications.

Because the discrete Fourier transform separates its input into components that contribute at discrete frequencies, it has a great number of applications in digital signal processing, e.g., for filtering, and in this context the discretized

input to the transform is customarily referred to as a *signal*, which exists in the *time domain*. The output is called a *spectrum* or *transform* and exists in the *frequency domain*.

3.11.6 Implementation details

There are many ways to define the DFT, varying in the sign of the exponent, normalization, etc. In this implementation, the DFT is defined as

$$A_k = \sum_{m=0}^{n-1} a_m \exp\left\{-2\pi i \frac{mk}{n}\right\} \qquad k = 0, \ldots, n-1.$$

The DFT is in general defined for complex inputs and outputs, and a single-frequency component at linear frequency f is represented by a complex exponential $a_m = \exp\{2\pi i\, fm\Delta t\}$, where Δt is the sampling interval.

The values in the result follow so-called "standard" order: If `A = fft(a, n)`, then `A[0]` contains the zero-frequency term (the sum of the signal), which is always purely real for real inputs. Then `A[1:n/2]` contains the positive-frequency terms, and `A[n/2+1:]` contains the negative-frequency terms, in order of decreasingly negative frequency. For an even number of input points, `A[n/2]` represents both positive and negative Nyquist frequency, and is also purely real for real input. For an odd number of input points, `A[(n-1)/2]` contains the largest positive frequency, while `A[(n+1)/2]` contains the largest negative frequency. The routine `np.fft.fftfreq(n)` returns an array giving the frequencies of corresponding elements in the output. The routine `np.fft.fftshift(A)` shifts transforms and their frequencies to put the zero-frequency components in the middle, and `np.fft.ifftshift(A)` undoes that shift.

When the input a is a time-domain signal and `A = fft(a)`, `np.abs(A)` is its amplitude spectrum and `np.abs(A)**2` is its power spectrum. The phase spectrum is obtained by `np.angle(A)`.

The inverse DFT is defined as

$$a_m = \frac{1}{n} \sum_{k=0}^{n-1} A_k \exp\left\{2\pi i \frac{mk}{n}\right\} \qquad m = 0, \ldots, n-1.$$

It differs from the forward transform by the sign of the exponential argument and the default normalization by $1/n$.

3.11.7 Normalization

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 $1/\sqrt{n}$.

3.11.8 Real and Hermitian transforms

When the input is purely real, its transform is Hermitian, i.e., the component at frequency f_k is the complex conjugate of the component at frequency $-f_k$, which means that for real inputs there is no information in the negative frequency components that is not already available from the positive frequency components. The family of `rfft` functions is designed to operate on real inputs, and exploits this symmetry by computing only the positive frequency components, up to and including the Nyquist frequency. Thus, `n` input points produce `n/2+1` complex output points. The inverses of this family assumes the same symmetry of its input, and for an output of `n` points uses `n/2+1` input points.

Correspondingly, when the spectrum is purely real, the signal is Hermitian. The `hfft` family of functions exploits this symmetry by using `n/2+1` complex points in the input (time) domain for `n` real points in the frequency domain.

In higher dimensions, FFTs are used, e.g., for image analysis and filtering. The computational efficiency of the FFT means that it can also be a faster way to compute large convolutions, using the property that a convolution in the time domain is equivalent to a point-by-point multiplication in the frequency domain.

3.11.9 Higher dimensions

In two dimensions, the DFT is defined as

$$A_{kl} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} a_{mn} \exp\left\{-2\pi i \left(\frac{mk}{M} + \frac{nl}{N}\right)\right\} \qquad k = 0, \ldots, M-1; \quad l = 0, \ldots, N-1,$$

which extends in the obvious way to higher dimensions, and the inverses in higher dimensions also extend in the same way.

3.11.10 References

3.11.11 Examples

For examples, see the various functions.

3.12 Financial functions

3.12.1 Simple financial functions

3.13 Functional programming

3.14 Numpy-specific help functions

3.14.1 Finding help

3.14.2 Reading help

3.15 Indexing routines

See also:

Indexing

3.15.1 Generating index arrays

3.15.2 Indexing-like operations

3.15.3 Inserting data into arrays

3.15.4 Iterating over arrays

nditer	Efficient multi-dimensional iterator object to iterate over arrays.
flatiter	Flat iterator object to iterate over arrays.

class numpy.**nditer**

Efficient multi-dimensional iterator object to iterate over arrays. To get started using this object, see the *introductory guide to array iteration*.

Parameters

op : ndarray or sequence of array_like

The array(s) to iterate over.

flags : sequence of str, optional

Flags to control the behavior of the iterator.

- "buffered" enables buffering when required.
- "c_index" causes a C-order index to be tracked.
- "f_index" causes a Fortran-order index to be tracked.
- "multi_index" causes a multi-index, or a tuple of indices with one per iteration dimension, to be tracked.
- "common_dtype" causes all the operands to be converted to a common data type, with copying or buffering as necessary.
- "delay_bufalloc" delays allocation of the buffers until a reset() call is made. Allows "allocate" operands to be initialized before their values are copied into the buffers.
- "external_loop" causes the *values* given to be one-dimensional arrays with multiple values instead of zero-dimensional arrays.
- "grow_inner" allows the `value` array sizes to be made larger than the buffer size when both "buffered" and "external_loop" is used.
- "ranged" allows the iterator to be restricted to a sub-range of the iterindex values.
- "refs_ok" enables iteration of reference types, such as object arrays.
- "reduce_ok" enables iteration of "readwrite" operands which are broadcasted, also known as reduction operands.
- "zerosize_ok" allows `itersize` to be zero.

op_flags : list of list of str, optional

This is a list of flags for each operand. At minimum, one of "readonly", "readwrite", or "writeonly" must be specified.

- "readonly" indicates the operand will only be read from.
- "readwrite" indicates the operand will be read from and written to.
- "writeonly" indicates the operand will only be written to.
- "no_broadcast" prevents the operand from being broadcasted.
- "contig" forces the operand data to be contiguous.
- "aligned" forces the operand data to be aligned.
- "nbo" forces the operand data to be in native byte order.
- "copy" allows a temporary read-only copy if required.
- "updateifcopy" allows a temporary read-write copy if required.
- "allocate" causes the array to be allocated if it is None in the *op* parameter.
- "no_subtype" prevents an "allocate" operand from using a subtype.
- "arraymask" indicates that this operand is the mask to use for selecting elements when writing to operands with the 'writemasked' flag set. The iterator does not enforce this, but when writing from a buffer back to the array, it only copies those elements indicated by this mask.
- 'writemasked' indicates that only elements where the chosen 'arraymask' operand is True will be written to.

op_dtypes : dtype or tuple of dtype(s), optional

The required data type(s) of the operands. If copying or buffering is enabled, the data will be converted to/from their original types.

order : {'C', 'F', 'A', 'K'}, optional

Controls the iteration order. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. This also affects the element memory order of "allocate" operands, as they are allocated to be compatible with iteration order. Default is 'K'.

casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional

Controls what kind of data casting may occur when making a copy or buffering. Setting this to 'unsafe' is not recommended, as it can adversely affect accumulations.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

op_axes : list of list of ints, optional

If provided, is a list of ints or None for each operands. The list of axes for an operand is a mapping from the dimensions of the iterator to the dimensions of the operand. A value of -1 can be placed for entries, causing that dimension to be treated as "newaxis".

itershape : tuple of ints, optional

The desired shape of the iterator. This allows "allocate" operands with a dimension mapped by op_axes not corresponding to a dimension of a different operand to get a value not equal to 1 for that dimension.

buffersize : int, optional

When buffering is enabled, controls the size of the temporary buffers. Set to 0 for the default value.

Notes

nditer supersedes *flatiter*. The iterator implementation behind *nditer* is also exposed by the Numpy C API.

The Python exposure supplies two iteration interfaces, one which follows the Python iterator protocol, and another which mirrors the C-style do-while pattern. The native Python approach is better in most cases, but if you need the iterator's coordinates or index, use the C-style pattern.

Examples

Here is how we might write an `iter_add` function, using the Python iterator protocol:

```
def iter_add_py(x, y, out=None):
    addop = np.add
    it = np.nditer([x, y, out], [],
                [['readonly'], ['readonly'], ['writeonly','allocate']])
    for (a, b, c) in it:
        addop(a, b, out=c)
    return it.operands[2]
```

Here is the same function, but following the C-style pattern:

```
def iter_add(x, y, out=None):
    addop = np.add

    it = np.nditer([x, y, out], [],
                [['readonly'], ['readonly'], ['writeonly','allocate']])

    while not it.finished:
        addop(it[0], it[1], out=it[2])
        it.iternext()

    return it.operands[2]
```

Here is an example outer product function:

```
def outer_it(x, y, out=None):
    mulop = np.multiply

    it = np.nditer([x, y, out], ['external_loop'],
            [['readonly'], ['readonly'], ['writeonly', 'allocate']],
            op_axes=[range(x.ndim)+[-1]*y.ndim,
                     [-1]*x.ndim+range(y.ndim),
                     None])

    for (a, b, c) in it:
        mulop(a, b, out=c)

    return it.operands[2]

>>> a = np.arange(2)+1
```

```
>>> b = np.arange(3)+1
>>> outer_it(a,b)
array([[1, 2, 3],
       [2, 4, 6]])
```

Here is an example function which operates like a "lambda" ufunc:

```
def luf(lamdaexpr, *args, **kwargs):
    "luf(lambdaexpr, op1, ..., opn, out=None, order='K', casting='safe', buffersize=0)"
    nargs = len(args)
    op = (kwargs.get('out',None),) + args
    it = np.nditer(op, ['buffered','external_loop'],
            [['writeonly','allocate','no_broadcast']] +
                            [['readonly','nbo','aligned']]*nargs,
            order=kwargs.get('order','K'),
            casting=kwargs.get('casting','safe'),
            buffersize=kwargs.get('buffersize',0))
    while not it.finished:
        it[0] = lamdaexpr(*it[1:])
        it.iternext()
    return it.operands[0]

>>> a = np.arange(5)
>>> b = np.ones(5)
>>> luf(lambda i,j:i*i + j/2, a, b)
array([  0.5,   1.5,   4.5,   9.5,  16.5])
```

Attributes

dtypes	(tuple of dtype(s)) The data types of the values provided in `value`. This may be different from the operand data types if buffering is enabled.
finished	(bool) Whether the iteration over the operands is finished or not.
has_delayed_bufalloc	(bool) If True, the iterator was created with the "delay_bufalloc" flag, and no reset() function was called on it yet.
has_index	(bool) If True, the iterator was created with either the "c_index" or the "f_index" flag, and the property `index` can be used to retrieve it.
has_multi_index	(bool) If True, the iterator was created with the "multi_index" flag, and the property `multi_index` can be used to retrieve it.
index :	When the "c_index" or "f_index" flag was used, this property provides access to the index. Raises a ValueError if accessed and `has_index` is False.
iterationneedsapi	(bool) Whether iteration requires access to the Python API, for example if one of the operands is an object array.
iterindex	(int) An index which matches the order of iteration.
itersize	(int) Size of the iterator.
itviews :	Structured view(s) of `operands` in memory, matching the reordered and optimized iterator access pattern.
multi_index :	When the "multi_index" flag was used, this property provides access to the index. Raises a ValueError if accessed accessed and `has_multi_index` is False.
ndim	(int) The iterator's dimension.
nop	(int) The number of iterator operands.
operands	(tuple of operand(s)) The array(s) to be iterated over.
shape	(tuple of ints) Shape tuple, the shape of the iterator.
value :	Value of `operands` at current iteration. Normally, this is a tuple of array scalars, but if the flag "external_loop" is used, it is a tuple of one dimensional arrays.

Methods

copy()	Get a copy of the iterator in its current state.
debug_print()	Print the current state of the *nditer* instance and debug info to stdout.
enable_external_loop()	When the "external_loop" was not used during construction, but is desired, this modifies the iterato
iternext()	Check whether iterations are left, and perform a single internal iteration without returning the result
next	
remove_axis(i)	Removes axis *i* from the iterator.
remove_multi_index()	When the "multi_index" flag was specified, this removes it, allowing the internal iteration structure
reset()	Reset the iterator to its initial state.

nditer.**copy**()
Get a copy of the iterator in its current state.

Examples

```
>>> x = np.arange(10)
>>> y = x + 1
>>> it = np.nditer([x, y])
>>> it.next()
(array(0), array(1))
>>> it2 = it.copy()
>>> it2.next()
(array(1), array(2))
```

nditer.**debug_print**()
Print the current state of the *nditer* instance and debug info to stdout.

nditer.**enable_external_loop**()
When the "external_loop" was not used during construction, but is desired, this modifies the iterator to behave as if the flag was specified.

nditer.**iternext**()
Check whether iterations are left, and perform a single internal iteration without returning the result. Used in the C-style pattern do-while pattern. For an example, see *nditer*.

Returns
iternext : bool

Whether or not there are iterations left.

nditer.**next**

nditer.**remove_axis**(*i*)
Removes axis *i* from the iterator. Requires that the flag "multi_index" be enabled.

nditer.**remove_multi_index**()
When the "multi_index" flag was specified, this removes it, allowing the internal iteration structure to be optimized further.

nditer.**reset**()
Reset the iterator to its initial state.

class numpy.**flatiter**
Flat iterator object to iterate over arrays.

A *flatiter* iterator is returned by `x.flat` for any array *x*. It allows iterating over the array as if it were a 1-D array, either in a for-loop or by calling its *next* method.

Iteration is done in row-major, C-style order (the last index varying the fastest). The iterator can also be indexed using basic slicing or advanced indexing.

See also:

ndarray.flat
Return a flat iterator over an array.

ndarray.flatten
Returns a flattened copy of an array.

Notes

A *flatiter* iterator can not be constructed directly from Python code by calling the *flatiter* constructor.

Examples

```
>>> x = np.arange(6).reshape(2, 3)
>>> fl = x.flat
>>> type(fl)
<type 'numpy.flatiter'>
>>> for item in fl:
...     print(item)
...
0
1
2
3
4
5
```

```
>>> fl[2:4]
array([2, 3])
```

Attributes

coords	An N-dimensional tuple of current coordinates.

flatiter.coords
An N-dimensional tuple of current coordinates.

Examples

```
>>> x = np.arange(6).reshape(2, 3)
>>> fl = x.flat
>>> fl.coords
(0, 0)
>>> fl.next()
0
>>> fl.coords
(0, 1)
```

Methods

copy()	Get a copy of the iterator as a 1-D array.
next	

flatiter.**copy**()
Get a copy of the iterator as a 1-D array.

Examples

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

flatiter.**next**

3.16 Input and output

3.16.1 Numpy binary files (NPY, NPZ)

The format of these binary file types is documented in http://docs.scipy.org/doc/numpy/neps/npy-format.html

3.16.2 Text files

ndarray.tofile(fid[, sep, format])	Write array to a file as text or binary (default).
ndarray.tolist()	Return the array as a (possibly nested) list.

ndarray.**tofile**(*fid*, *sep=""*, *format="%s"*)
Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function fromfile().

Parameters

fid : file or str

An open file object, or a string containing a filename.

sep : str

Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format : str

Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with

different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

ndarray.**tolist**()
Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

Parameters
none

Returns
y : list

The possibly nested list of array elements.

Notes

The array may be recreated, `a = np.array(a.tolist())`.

Examples

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

3.16.3 Raw binary files

ndarray.tofile(fid[, sep, format])	Write array to a file as text or binary (default).

3.16.4 String formatting

3.16.5 Memory mapping files

3.16.6 Text formatting options

3.16.7 Base-n representations

3.16.8 Data sources

3.17 Linear algebra (`numpy.linalg`)

3.17.1 Matrix and vector products

3.17.2 Decompositions

3.17.3 Matrix eigenvalues

3.17.4 Norms and other numbers

3.17.5 Solving equations and inverting matrices

3.17.6 Exceptions

3.17.7 Linear algebra on several matrices at once

New in version 1.8.0.

Several of the linear algebra routines listed above are able to compute results for several matrices at once, if they are stacked into the same array.

This is indicated in the documentation via input parameter specifications such as `a : (..., M, M) array_like`. This means that if for instance given an input array `a.shape == (N, M, M)`, it is interpreted as a "stack" of N matrices, each of size M-by-M. Similar specification applies to return values, for instance the determinant has `det : (...)` and will in this case return an array of shape `det(a).shape == (N,)`. This generalizes to linear algebra operations on higher-dimensional arrays: the last 1 or 2 dimensions of a multidimensional array are interpreted as vectors or matrices, as appropriate for each operation.

3.18 Logic functions

3.18.1 Truth value testing

3.18.2 Array contents

isfinite(x[, out])	Test element-wise for finiteness (not infinity or not Not a Number).
isinf(x[, out])	Test element-wise for positive or negative infinity.
isnan(x[, out])	Test element-wise for NaN and return result as a boolean array.

numpy.**isfinite**(*x*[, *out*]) = **<ufunc 'isfinite'>**

Test element-wise for finiteness (not infinity or not Not a Number).

The result is returned as a boolean array.

Parameters

x : array_like

Input values.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See `doc.ufuncs`.

Returns

y : ndarray, bool

For scalar input, the result is a new boolean with value True if the input is finite; otherwise the value is False (input is either positive infinity, negative infinity or Not a Number).

For array input, the result is a boolean array with the same dimensions as the input and the values are True if the corresponding element of the input is finite; otherwise the values are False (element is either positive infinity, negative infinity or Not a Number).

See also:

isinf, `isneginf`, `isposinf`, *isnan*

Notes

Not a Number, positive infinity and negative infinity are considered to be non-finite.

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Also that positive infinity is not equivalent to negative infinity. But infinity is equivalent to positive infinity. Errors result if the second argument is also supplied when *x* is a scalar input, or if first and second arguments have different shapes.

Examples

```
>>> np.isfinite(1)
True
>>> np.isfinite(0)
True
>>> np.isfinite(np.nan)
False
>>> np.isfinite(np.inf)
False
>>> np.isfinite(np.NINF)
False
```

```
>>> np.isfinite([np.log(-1.),1.,np.log(0)])
array([False,  True, False], dtype=bool)
```

```
>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isfinite(x, y)
array([0, 1, 0])
>>> y
array([0, 1, 0])
```

numpy.**isinf**(*x*[, *out*]) **= <ufunc 'isinf'>**

Test element-wise for positive or negative infinity.

Returns a boolean array of the same shape as *x*, True where `x == +/-inf`, otherwise False.

Parameters

x : array_like

Input values

out : array_like, optional

An array with the same shape as *x* to store the result.

Returns

y : bool (scalar) or boolean ndarray

For scalar input, the result is a new boolean with value True if the input is positive or negative infinity; otherwise the value is False.

For array input, the result is a boolean array with the same shape as the input and the values are True where the corresponding element of the input is positive or negative infinity; elsewhere the values are False. If a second argument was supplied the result is stored there. If the type of that array is a numeric type the result is represented as zeros and ones, if the type is boolean then as False and True, respectively. The return value *y* is then a reference to that array.

See also:

`isneginf`, `isposinf`, *isnan*, *isfinite*

Notes

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754).

Errors result if the second argument is supplied when the first argument is a scalar, or if the first and second arguments have different shapes.

Examples

```
>>> np.isinf(np.inf)
True
>>> np.isinf(np.nan)
False
>>> np.isinf(np.NINF)
True
>>> np.isinf([np.inf, -np.inf, 1.0, np.nan])
array([ True,  True, False, False], dtype=bool)
```

```
>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isinf(x, y)
```

```
    array([1, 0, 1])
    >>> y
    array([1, 0, 1])
```

numpy.**isnan**(*x*[, *out*]) = <ufunc 'isnan'>
Test element-wise for NaN and return result as a boolean array.

Parameters
x : array_like

Input array.

Returns
y : ndarray or bool

For scalar input, the result is a new boolean with value True if the input is NaN; otherwise the value is False.

For array input, the result is a boolean array of the same dimensions as the input and the values are True if the corresponding element of the input is NaN; otherwise the values are False.

See also:

isinf, isneginf, isposinf, *isfinite*

Notes

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

Examples

```
    >>> np.isnan(np.nan)
    True
    >>> np.isnan(np.inf)
    False
    >>> np.isnan([np.log(-1.),1.,np.log(0)])
    array([ True, False, False], dtype=bool)
```

3.18.3 Array type testing

3.18.4 Logical operations

logical_and(x1, x2[, out])	Compute the truth value of x1 AND x2 element-wise.
logical_or(x1, x2[, out])	Compute the truth value of x1 OR x2 element-wise.
logical_not(x[, out])	Compute the truth value of NOT x element-wise.
logical_xor(x1, x2[, out])	Compute the truth value of x1 XOR x2, element-wise.

numpy.**logical_and**(*x1*, *x2*[, *out*]) = <ufunc 'logical_and'>
Compute the truth value of x1 AND x2 element-wise.

Parameters
x1, x2 : array_like

Input arrays. *x1* and *x2* must be of the same shape.

Returns

y : ndarray or bool

Boolean result with the same shape as *x1* and *x2* of the logical AND operation on corresponding elements of *x1* and *x2*.

See also:

logical_or, *logical_not*, *logical_xor*, *bitwise_and*

Examples

```
>>> np.logical_and(True, False)
False
>>> np.logical_and([True, False], [False, False])
array([False, False], dtype=bool)
```

```
>>> x = np.arange(5)
>>> np.logical_and(x>1, x<4)
array([False, False,  True,  True, False], dtype=bool)
```

numpy.**logical_or**(*x1*, *x2*[, *out*]) = <ufunc 'logical_or'>

Compute the truth value of x1 OR x2 element-wise.

Parameters

x1, x2 : array_like

Logical OR is applied to the elements of *x1* and *x2*. They have to be of the same shape.

Returns

y : ndarray or bool

Boolean result with the same shape as *x1* and *x2* of the logical OR operation on elements of *x1* and *x2*.

See also:

logical_and, *logical_not*, *logical_xor*, *bitwise_or*

Examples

```
>>> np.logical_or(True, False)
True
>>> np.logical_or([True, False], [False, False])
array([ True, False], dtype=bool)
```

```
>>> x = np.arange(5)
>>> np.logical_or(x < 1, x > 3)
array([ True, False, False, False,  True], dtype=bool)
```

numpy.**logical_not**(*x*[, *out*]) = <ufunc 'logical_not'>

Compute the truth value of NOT x element-wise.

Parameters

x : array_like

Logical NOT is applied to the elements of *x*.

Returns

y : bool or ndarray of bool

Boolean result with the same shape as *x* of the NOT operation on elements of *x*.

See also:

logical_and, *logical_or*, *logical_xor*

Examples

```
>>> np.logical_not(3)
False
>>> np.logical_not([True, False, 0, 1])
array([False,  True,  True, False], dtype=bool)
```

```
>>> x = np.arange(5)
>>> np.logical_not(x<3)
array([False, False, False,  True,  True], dtype=bool)
```

numpy.**logical_xor**(*x1*, *x2*[, *out*]) **= <ufunc 'logical_xor'>**

Compute the truth value of x1 XOR x2, element-wise.

Parameters

x1, x2 : array_like

Logical XOR is applied to the elements of *x1* and *x2*. They must be broadcastable to the same shape.

Returns

y : bool or ndarray of bool

Boolean result of the logical XOR operation applied to the elements of *x1* and *x2*; the shape is determined by whether or not broadcasting of one or both arrays was required.

See also:

logical_and, *logical_or*, *logical_not*, *bitwise_xor*

Examples

```
>>> np.logical_xor(True, False)
True
>>> np.logical_xor([True, True, False, False], [True, False, True, False])
array([False,  True,  True, False], dtype=bool)
```

```
>>> x = np.arange(5)
>>> np.logical_xor(x < 1, x > 3)
array([ True, False, False, False,  True], dtype=bool)
```

Simple example showing support of broadcasting

```
>>> np.logical_xor(0, np.eye(2))
array([[ True, False],
       [False,  True]], dtype=bool)
```

3.18.5 Comparison

greater(x1, x2[, out])	Return the truth value of (x1 > x2) element-wise.

Continued on next page

Table 3.77 – continued from previous page

greater_equal(x1, x2[, out])	Return the truth value of (x1 >= x2) element-wise.
less(x1, x2[, out])	Return the truth value of (x1 < x2) element-wise.
less_equal(x1, x2[, out])	Return the truth value of (x1 =< x2) element-wise.
equal(x1, x2[, out])	Return (x1 == x2) element-wise.
not_equal(x1, x2[, out])	Return (x1 != x2) element-wise.

numpy.**greater**(*x1*, *x2*[, *out*]) = <ufunc 'greater'>

Return the truth value of (x1 > x2) element-wise.

Parameters

x1, x2 : array_like

Input arrays. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which may be the shape of one or the other).

Returns

out : bool or ndarray of bool

Array of bools, or a single bool if *x1* and *x2* are scalars.

See also:

greater_equal, *less*, *less_equal*, *equal*, *not_equal*

Examples

```
>>> np.greater([4,2],[2,2])
array([ True, False], dtype=bool)
```

If the inputs are ndarrays, then np.greater is equivalent to '>'.

```
>>> a = np.array([4,2])
>>> b = np.array([2,2])
>>> a > b
array([ True, False], dtype=bool)
```

numpy.**greater_equal**(*x1*, *x2*[, *out*]) = <ufunc 'greater_equal'>

Return the truth value of (x1 >= x2) element-wise.

Parameters

x1, x2 : array_like

Input arrays. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which may be the shape of one or the other).

Returns

out : bool or ndarray of bool

Array of bools, or a single bool if *x1* and *x2* are scalars.

See also:

greater, *less*, *less_equal*, *equal*, *not_equal*

Examples

```
>>> np.greater_equal([4, 2, 1], [2, 2, 2])
array([ True, True, False], dtype=bool)
```

numpy.**less**(*x1*, *x2*[, *out*]) = <ufunc 'less'>

Return the truth value of (x1 < x2) element-wise.

Parameters

x1, x2 : array_like

Input arrays. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which may be the shape of one or the other).

Returns

out : bool or ndarray of bool

Array of bools, or a single bool if *x1* and *x2* are scalars.

See also:

greater, *less_equal*, *greater_equal*, *equal*, *not_equal*

Examples

```
>>> np.less([1, 2], [2, 2])
array([ True, False], dtype=bool)
```

numpy.**less_equal**(*x1*, *x2*[, *out*]) **= <ufunc 'less_equal'>**

Return the truth value of (x1 =< x2) element-wise.

Parameters

x1, x2 : array_like

Input arrays. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which may be the shape of one or the other).

Returns

out : bool or ndarray of bool

Array of bools, or a single bool if *x1* and *x2* are scalars.

See also:

greater, *less*, *greater_equal*, *equal*, *not_equal*

Examples

```
>>> np.less_equal([4, 2, 1], [2, 2, 2])
array([False,  True,  True], dtype=bool)
```

numpy.**equal**(*x1*, *x2*[, *out*]) **= <ufunc 'equal'>**

Return (x1 == x2) element-wise.

Parameters

x1, x2 : array_like

Input arrays of the same shape.

Returns

out : ndarray or bool

Output array of bools, or a single bool if x1 and x2 are scalars.

See also:

not_equal, *greater_equal*, *less_equal*, *greater*, *less*

Examples

```
>>> np.equal([0, 1, 3], np.arange(3))
array([ True,  True, False], dtype=bool)
```

What is compared are values, not types. So an int (1) and an array of length one can evaluate as True:

```
>>> np.equal(1, np.ones(1))
array([ True], dtype=bool)
```

numpy.**not_equal**(*x1*, *x2*[, *out*]) = **<ufunc 'not_equal'>**

Return (x1 != x2) element-wise.

Parameters

x1, x2 : array_like

Input arrays.

out : ndarray, optional

A placeholder the same shape as *x1* to store the result. See `doc.ufuncs` (Section "Output arguments") for more details.

Returns

not_equal : ndarray bool, scalar bool

For each element in *x1, x2*, return True if *x1* is not equal to *x2* and False otherwise.

See also:

equal, *greater*, *greater_equal*, *less*, *less_equal*

Examples

```
>>> np.not_equal([1.,2.], [1., 3.])
array([False,  True], dtype=bool)
>>> np.not_equal([1, 2], [[1, 3],[1, 4]])
array([[False,  True],
       [False,  True]], dtype=bool)
```

3.19 Masked array operations

3.19.1 Constants

3.19.2 Creation

From existing data

Ones and zeros

3.19.3 Inspecting the array

ma.MaskedArray.data	Return the current data, as a view of the original underlying data.
ma.MaskedArray.mask	Mask
ma.MaskedArray.recordmask	Return the mask of the records.

MaskedArray.**data**
Return the current data, as a view of the original underlying data.

MaskedArray.**mask**
Mask

MaskedArray.**recordmask**
Return the mask of the records.

A record is masked when all the fields are masked.

3.19.4 Manipulating a MaskedArray

Changing the shape

Modifying axes

Changing the number of dimensions

Joining arrays

3.19.5 Operations on masks

Creating a mask

Accessing a mask

ma.masked_array.mask	Mask

masked_array.**mask**
Mask

Finding masked data

Modifying a mask

3.19.6 Conversion operations

> to a masked array

> to a ndarray

> to another object

Pickling and unpickling

Filling a masked array

ma.MaskedArray.fill_value	Filling value.

MaskedArray.**fill_value**
Filling value.

3.19.7 Masked arrays arithmetics

Arithmetics

Minimum/maximum

Sorting

Algebra

Polynomial fit

Clipping and rounding

Miscellanea

3.20 Mathematical functions

3.20.1 Trigonometric functions

sin(x[, out])	Trigonometric sine, element-wise.
cos(x[, out])	Cosine element-wise.
tan(x[, out])	Compute tangent element-wise.
arcsin(x[, out])	Inverse sine, element-wise.
arccos(x[, out])	Trigonometric inverse cosine, element-wise.
arctan(x[, out])	Trigonometric inverse tangent, element-wise.
hypot(x1, x2[, out])	Given the "legs" of a right triangle, return its hypotenuse.
arctan2(x1, x2[, out])	Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.
degrees(x[, out])	Convert angles from radians to degrees.
radians(x[, out])	Convert angles from degrees to radians.
deg2rad(x[, out])	Convert angles from degrees to radians.
rad2deg(x[, out])	Convert angles from radians to degrees.

numpy.**sin**(*x*[, *out*]) **= <ufunc 'sin'>**

Trigonometric sine, element-wise.

Parameters

x : array_like

Angle, in radians (2π rad equals 360 degrees).

Returns

y : array_like

The sine of each element of x.

See also:

arcsin, *sinh*, *cos*

Notes

The sine is one of the fundamental functions of trigonometry (the mathematical study of triangles). Consider a circle of radius 1 centered on the origin. A ray comes in from the $+x$ axis, makes an angle at the origin (measured counter-clockwise from that axis), and departs from the origin. The y coordinate of the outgoing ray's intersection with the unit circle is the sine of that angle. It ranges from -1 for $x = 3\pi/2$ to +1 for $\pi/2$. The function has zeroes where the angle is a multiple of π. Sines of angles between π and 2π are negative. The numerous properties of the sine and related functions are included in any standard trigonometry text.

Examples

Print sine of one angle:

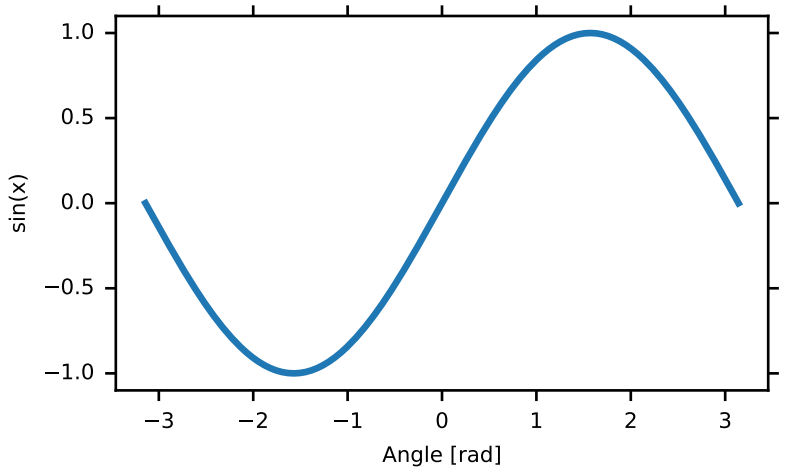
```
>>> np.sin(np.pi/2.)
1.0
```

Print sines of an array of angles given in degrees:

```
>>> np.sin(np.array((0., 30., 45., 60., 90.)) * np.pi / 180. )
array([ 0.        ,  0.5       ,  0.70710678,  0.8660254 ,  1.        ])
```

Plot the sine function:

```
>>> import matplotlib.pylab as plt
>>> x = np.linspace(-np.pi, np.pi, 201)
>>> plt.plot(x, np.sin(x))
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
>>> plt.axis('tight')
>>> plt.show()
```



numpy.**cos**(*x*[, *out*]) = <ufunc 'cos'>

Cosine element-wise.

Parameters

x : array_like

Input array in radians.

out : ndarray, optional

Output array of same shape as *x*.

Returns

y : ndarray

The corresponding cosine values.

Raises

ValueError: invalid return array shape

if *out* is provided and *out.shape* != *x.shape* (See Examples)

Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972.

Examples

```
>>> np.cos(np.array([0, np.pi/2, np.pi]))
array([  1.00000000e+00,   6.12303177e-17,  -1.00000000e+00])
>>>
>>> # Example of providing the optional output parameter
>>> out2 = np.cos([0.1], out1)
>>> out2 is out1
True
>>>
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.cos(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid return array shape
```

numpy.**tan**(*x*[, *out*]) = **<ufunc 'tan'>**

Compute tangent element-wise.

Equivalent to `np.sin(x)/np.cos(x)` element-wise.

Parameters

x : array_like

Input array.

out : ndarray, optional

Output array of same shape as *x*.

Returns

y : ndarray

The corresponding tangent values.

Raises

ValueError: invalid return array shape

if *out* is provided and *out.shape* != *x.shape* (See Examples)

Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972.

Examples

```
>>> from math import pi
>>> np.tan(np.array([-pi,pi/2,pi]))
array([  1.22460635e-16,   1.63317787e+16,  -1.22460635e-16])
>>>
>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out2 = np.cos([0.1], out1)
>>> out2 is out1
True
>>>
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.cos(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid return array shape
```

numpy.**arcsin**(*x*[, *out*]) = <ufunc 'arcsin'>

Inverse sine, element-wise.

Parameters

x : array_like

y-coordinate on the unit circle.

out : ndarray, optional

Array of the same shape as *x*, in which to store the results. See `doc.ufuncs` (Section "Output arguments") for more details.

Returns

angle : ndarray

The inverse sine of each element in *x*, in radians and in the closed interval `[-pi/2, pi/2]`. If *x* is a scalar, a scalar is returned, otherwise an array.

See also:

sin, *cos*, *arccos*, *tan*, *arctan*, *arctan2*, `emath.arcsin`

Notes

arcsin is a multivalued function: for each *x* there are infinitely many numbers *z* such that $sin(z) = x$. The convention is to return the angle *z* whose real part lies in [-pi/2, pi/2].

For real-valued input data types, *arcsin* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arcsin* is a complex analytic function that has, by convention, the branch cuts [-inf, -1] and [1, inf] and is continuous from above on the former and from below on the latter.

The inverse sine is also known as *asin* or sin^{-1}.

References

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79ff. http://www.math.sfu.ca/~cbm/aands/

Examples

```
>>> np.arcsin(1)     # pi/2
1.5707963267948966
>>> np.arcsin(-1)    # -pi/2
-1.5707963267948966
>>> np.arcsin(0)
0.0
```

numpy.**arccos**(*x*[, *out*]) = <ufunc 'arccos'>

Trigonometric inverse cosine, element-wise.

The inverse of *cos* so that, if `y = cos(x)`, then `x = arccos(y)`.

Parameters

x : array_like

x-coordinate on the unit circle. For real arguments, the domain is [-1, 1].

out : ndarray, optional

Array of the same shape as *a*, to store results in. See `doc.ufuncs` (Section "Output arguments") for more details.

Returns

angle : ndarray

The angle of the ray intersecting the unit circle at the given *x*-coordinate in radians [0, pi]. If *x* is a scalar then a scalar is returned, otherwise an array of the same shape as *x* is returned.

See also:

cos, *arctan*, *arcsin*, `emath.arccos`

Notes

arccos is a multivalued function: for each *x* there are infinitely many numbers *z* such that *cos(z) = x*. The convention is to return the angle *z* whose real part lies in *[0, pi]*.

For real-valued input data types, *arccos* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arccos* is a complex analytic function that has branch cuts *[-inf, -1]* and *[1, inf]* and is continuous from above on the former and from below on the latter.

The inverse *cos* is also known as *acos* or cos^-1.

References

M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 79. http://www.math.sfu.ca/~cbm/aands/

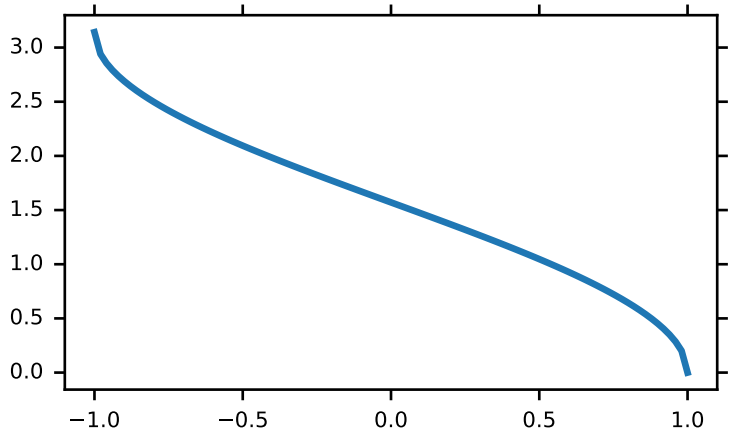
Examples

We expect the arccos of 1 to be 0, and of -1 to be pi:

```
>>> np.arccos([1, -1])
array([ 0.        ,  3.14159265])
```

Plot arccos:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-1, 1, num=100)
>>> plt.plot(x, np.arccos(x))
>>> plt.axis('tight')
>>> plt.show()
```



numpy.**arctan**(*x*[, *out*]) = <ufunc 'arctan'>

Trigonometric inverse tangent, element-wise.

The inverse of tan, so that if `y = tan(x)` then `x = arctan(y)`.

Parameters

x : array_like

Input values. *arctan* is applied to each element of *x*.

Returns

out : ndarray

Out has the same shape as *x*. Its real part is in `[-pi/2, pi/2]` (`arctan(+/-inf)` returns `+/-pi/2`). It is a scalar if *x* is a scalar.

See also:

arctan2

The "four quadrant" arctan of the angle formed by (*x*, *y*) and the positive *x*-axis.

angle

Argument of complex values.

Notes

arctan is a multi-valued function: for each *x* there are infinitely many numbers *z* such that tan(*z*) = *x*. The convention is to return the angle *z* whose real part lies in [-pi/2, pi/2].

For real-valued input data types, *arctan* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arctan* is a complex analytic function that has [*1j, infj*] and [*-1j, -infj*] as branch cuts, and is continuous from the left on the former and from the right on the latter.

The inverse tangent is also known as *atan* or tan^{-1}.

References

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79. http://www.math.sfu.ca/~cbm/aands/

Examples

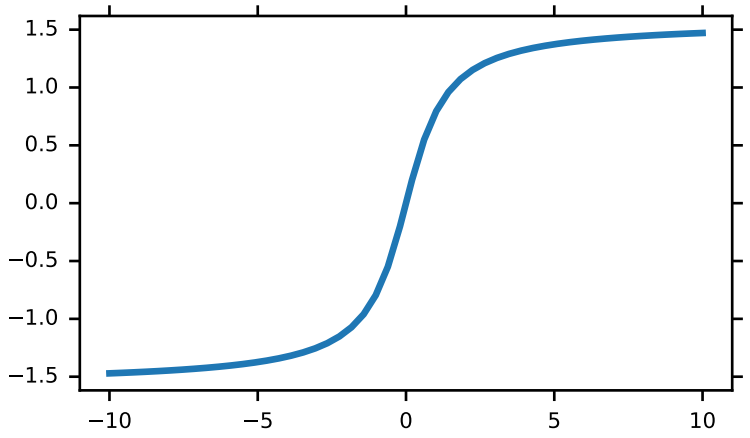
We expect the arctan of 0 to be 0, and of 1 to be pi/4:

```
>>> np.arctan([0, 1])
array([ 0.        ,  0.78539816])
```

```
>>> np.pi/4
0.7853981633974483
```

Plot arctan:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-10, 10)
>>> plt.plot(x, np.arctan(x))
>>> plt.axis('tight')
>>> plt.show()
```



numpy.**hypot**(*x1*, *x2*[, *out*]) = **<ufunc 'hypot'>**

Given the "legs" of a right triangle, return its hypotenuse.

Equivalent to `sqrt(x1**2 + x2**2)`, element-wise. If *x1* or *x2* is scalar_like (i.e., unambiguously castable to a scalar type), it is broadcast for use with each element of the other argument. (See Examples)

Parameters

x1, x2 : array_like

Leg of the triangle(s).

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

z : ndarray

The hypotenuse of the triangle(s).

Examples

```
>>> np.hypot(3*np.ones((3, 3)), 4*np.ones((3, 3)))
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

Example showing broadcast of scalar_like argument:

```
>>> np.hypot(3*np.ones((3, 3)), [4])
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

numpy.**arctan2**(*x1*, *x2*[, *out*]) = <ufunc 'arctan2'>

Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.

The quadrant (i.e., branch) is chosen so that `arctan2(x1, x2)` is the signed angle in radians between the ray ending at the origin and passing through the point (1,0), and the ray ending at the origin and passing through the point (*x2*, *x1*). (Note the role reversal: the "*y*-coordinate" is the first function parameter, the "*x*-coordinate" is the second.) By IEEE convention, this function is defined for *x2* = +/-0 and for either or both of *x1* and *x2* = +/-inf (see Notes for specific values).

This function is not defined for complex-valued arguments; for the so-called argument of complex values, use `angle`.

Parameters

x1 : array_like, real-valued

y-coordinates.

x2 : array_like, real-valued

x-coordinates. *x2* must be broadcastable to match the shape of *x1* or vice versa.

Returns

angle : ndarray

Array of angles in radians, in the range `[-pi, pi]`.

See also:

arctan, *tan*, `angle`

Notes

arctan2 is identical to the *atan2* function of the underlying C library. The following special values are defined in the C standard: *[R6]*

x1	*x2*	*arctan2(x1,x2)*
+/- 0	+0	+/- 0
+/- 0	-0	+/- pi
> 0	+/-inf	+0 / +pi
< 0	+/-inf	-0 / -pi
+/-inf	+inf	+/- (pi/4)
+/-inf	-inf	+/- (3*pi/4)

Note that +0 and -0 are distinct floating point numbers, as are +inf and -inf.

References

[R6]

Examples

Consider four points in different quadrants:

```
>>> x = np.array([-1, +1, +1, -1])
>>> y = np.array([-1, -1, +1, +1])
>>> np.arctan2(y, x) * 180 / np.pi
array([-135.,  -45.,   45.,  135.])
```

Note the order of the parameters. *arctan2* is defined also when *x2* = 0 and at several other special points, obtaining values in the range `[-pi, pi]`:

```
>>> np.arctan2([1., -1.], [0., 0.])
array([ 1.57079633, -1.57079633])
>>> np.arctan2([0., 0., np.inf], [+0., -0., np.inf])
array([ 0.        ,  3.14159265,  0.78539816])
```

numpy.**degrees**(*x*[, *out*]) = <ufunc 'degrees'>

Convert angles from radians to degrees.

Parameters

x : array_like

Input array in radians.

out : ndarray, optional

Output array of same shape as x.

Returns

y : ndarray of floats

The corresponding degree values; if *out* was supplied this is a reference to it.

See also:

rad2deg

equivalent function

Examples

Convert a radian array to degrees

```
>>> rad = np.arange(12.)*np.pi/6
>>> np.degrees(rad)
array([   0.,   30.,   60.,   90.,  120.,  150.,  180.,  210.,  240.,
        270.,  300.,  330.])
```

```
>>> out = np.zeros((rad.shape))
>>> r = degrees(rad, out)
>>> np.all(r == out)
True
```

numpy.**radians**(*x*[, *out*]) = <ufunc 'radians'>

Convert angles from degrees to radians.

Parameters

x : array_like

Input array in degrees.

out : ndarray, optional

Output array of same shape as *x*.

Returns

y : ndarray

The corresponding radian values.

See also:

deg2rad

equivalent function

Examples

Convert a degree array to radians

```
>>> deg = np.arange(12.) * 30.
>>> np.radians(deg)
array([ 0.        ,  0.52359878,  1.04719755,  1.57079633,  2.0943951 ,
        2.61799388,  3.14159265,  3.66519143,  4.1887902 ,  4.71238898,
        5.23598776,  5.75958653])
```

```
>>> out = np.zeros((deg.shape))
>>> ret = np.radians(deg, out)
>>> ret is out
True
```

numpy.**deg2rad**(*x*[, *out*]) = <ufunc 'deg2rad'>

Convert angles from degrees to radians.

Parameters

x : array_like

Angles in degrees.

Returns

y : ndarray

The corresponding angle in radians.

See also:

rad2deg

Convert angles from radians to degrees.

unwrap

Remove large jumps in angle by wrapping.

Notes

New in version 1.3.0.

`deg2rad(x)` is `x * pi / 180.`

Examples

```
>>> np.deg2rad(180)
3.1415926535897931
```

numpy.**rad2deg**(*x*[, *out*]) = <ufunc 'rad2deg'>
Convert angles from radians to degrees.

Parameters
x : array_like

Angle in radians.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns
y : ndarray

The corresponding angle in degrees.

See also:

deg2rad
Convert angles from degrees to radians.

unwrap
Remove large jumps in angle by wrapping.

Notes

New in version 1.3.0.

rad2deg(x) is `180 * x / pi`.

Examples

```
>>> np.rad2deg(np.pi/2)
90.0
```

3.20.2 Hyperbolic functions

sinh(x[, out])	Hyperbolic sine, element-wise.
cosh(x[, out])	Hyperbolic cosine, element-wise.
tanh(x[, out])	Compute hyperbolic tangent element-wise.
arcsinh(x[, out])	Inverse hyperbolic sine element-wise.
arccosh(x[, out])	Inverse hyperbolic cosine, element-wise.
arctanh(x[, out])	Inverse hyperbolic tangent element-wise.

numpy.**sinh**(*x*[, *out*]) = <ufunc 'sinh'>
Hyperbolic sine, element-wise.

Equivalent to `1/2 * (np.exp(x) - np.exp(-x))` or `-1j * np.sin(1j*x)`.

Parameters
x : array_like

Input array.

out : ndarray, optional

Output array of same shape as *x*.

Returns

y : ndarray

The corresponding hyperbolic sine values.

Raises

ValueError: invalid return array shape

if *out* is provided and *out.shape* != *x.shape* (See Examples)

Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83.

Examples

```
>>> np.sinh(0)
0.0
>>> np.sinh(np.pi*1j/2)
1j
>>> np.sinh(np.pi*1j) # (exact value is 0)
1.2246063538223773e-016j
>>> # Discrepancy due to vagaries of floating point arithmetic.
```

```
>>> # Example of providing the optional output parameter
>>> out2 = np.sinh([0.1], out1)
>>> out2 is out1
True
```

```
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.sinh(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid return array shape
```

numpy.**cosh**(*x*[, *out*]) = <ufunc 'cosh'>

Hyperbolic cosine, element-wise.

Equivalent to `1/2 * (np.exp(x) + np.exp(-x))` and `np.cos(1j*x)`.

Parameters

x : array_like

Input array.

Returns

out : ndarray

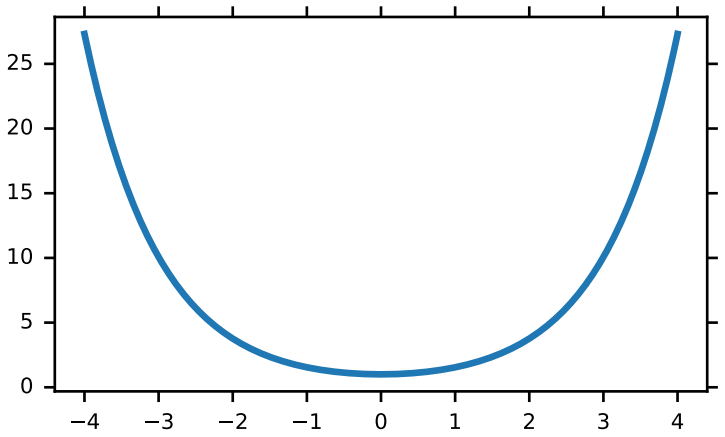
Output array of same shape as *x*.

Examples

```
>>> np.cosh(0)
1.0
```

The hyperbolic cosine describes the shape of a hanging cable:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-4, 4, 1000)
>>> plt.plot(x, np.cosh(x))
>>> plt.show()
```



numpy.**tanh**(*x*[, *out*]) = **<ufunc 'tanh'>**

Compute hyperbolic tangent element-wise.

Equivalent to `np.sinh(x)/np.cosh(x)` or `-1j * np.tan(1j*x)`.

Parameters

x : array_like

Input array.

out : ndarray, optional

Output array of same shape as *x*.

Returns

y : ndarray

The corresponding hyperbolic tangent values.

Raises

ValueError: invalid return array shape

if *out* is provided and *out.shape* != *x.shape* (See Examples)

Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

References

[R284], *[R285]*

Examples

```
>>> np.tanh((0, np.pi*1j, np.pi*1j/2))
array([ 0. +0.00000000e+00j,  0. -1.22460635e-16j,  0. +1.63317787e+16j])
```

```
>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out2 = np.tanh([0.1], out1)
>>> out2 is out1
True
```

```
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.tanh(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid return array shape
```

numpy.**arcsinh**(*x*[, *out*]) = <ufunc 'arcsinh'>

Inverse hyperbolic sine element-wise.

Parameters

x : array_like

Input array.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See `doc.ufuncs`.

Returns

out : ndarray

Array of of the same shape as *x*.

Notes

arcsinh is a multivalued function: for each *x* there are infinitely many numbers *z* such that *sinh(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi/2, pi/2]*.

For real-valued input data types, *arcsinh* always returns real output. For each value that cannot be expressed as a real number or infinity, it returns `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arccos* is a complex analytical function that has branch cuts *[1j, infj]* and *[-1j, -infj]* and is continuous from the right on the former and from the left on the latter.

The inverse hyperbolic sine is also known as *asinh* or `sinh^-1`.

References

[R4], *[R5]*

Examples

```
>>> np.arcsinh(np.array([np.e, 10.0]))
array([ 1.72538256,  2.99822295])
```

numpy.**arccosh**(*x*[, *out*]) = <ufunc 'arccosh'>

Inverse hyperbolic cosine, element-wise.

Parameters

x : array_like

Input array.

out : ndarray, optional

Array of the same shape as *x*, to store results in. See `doc.ufuncs` (Section "Output arguments") for details.

Returns

arccosh : ndarray

Array of the same shape as *x*.

See also:

cosh, *arcsinh*, *sinh*, *arctanh*, *tanh*

Notes

arccosh is a multivalued function: for each *x* there are infinitely many numbers *z* such that *cosh(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]* and the real part in `[0, inf]`.

For real-valued input data types, *arccosh* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arccosh* is a complex analytical function that has a branch cut *[-inf, 1]* and is continuous from above on it.

References

[R2], *[R3]*

Examples

```
>>> np.arccosh([np.e, 10.0])
array([ 1.65745445,  2.99322285])
>>> np.arccosh(1)
0.0
```

numpy.**arctanh**(*x*[, *out*]) = **<ufunc 'arctanh'>**

Inverse hyperbolic tangent element-wise.

Parameters

x : array_like

Input array.

Returns

out : ndarray

Array of the same shape as *x*.

See also:

`emath.arctanh`

Notes

arctanh is a multivalued function: for each *x* there are infinitely many numbers *z* such that *tanh(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi/2, pi/2]*.

For real-valued input data types, *arctanh* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arctanh* is a complex analytical function that has branch cuts *[-1, -inf]* and *[1, inf]* and is continuous from above on the former and from below on the latter.

The inverse hyperbolic tangent is also known as *atanh* or `tanh^-1`.

References

[R7], *[R8]*

Examples

```
>>> np.arctanh([0, -0.5])
array([ 0.        , -0.54930614])
```

3.20.3 Rounding

rint(x[, out])	Round elements of the array to the nearest integer.
floor(x[, out])	Return the floor of the input, element-wise.
ceil(x[, out])	Return the ceiling of the input, element-wise.
trunc(x[, out])	Return the truncated value of the input, element-wise.

numpy.**rint**(*x*[, *out*]) = <ufunc 'rint'>

Round elements of the array to the nearest integer.

Parameters

x : array_like

Input array.

Returns

out : ndarray or scalar

Output array is same shape and type as *x*.

See also:

ceil, *floor*, *trunc*

Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.rint(a)
array([-2., -2., -0.,  0.,  2.,  2.,  2.])
```

numpy.**floor**(*x*[, *out*]) = <ufunc 'floor'>

Return the floor of the input, element-wise.

The floor of the scalar *x* is the largest integer *i*, such that *i* <= *x*. It is often denoted as $\lfloor x \rfloor$.

Parameters

x : array_like

Input data.

Returns

y : ndarray or scalar

The floor of each element in *x*.

See also:

ceil, *trunc*, *rint*

Notes

Some spreadsheet programs calculate the "floor-towards-zero", in other words `floor(-2.5) == -2`. NumPy instead uses the definition of *floor* where *floor(-2.5) == -3*.

Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.floor(a)
array([-2., -2., -1.,  0.,  1.,  1.,  2.])
```

numpy.**ceil**(*x*[, *out*]) = <ufunc 'ceil'>

Return the ceiling of the input, element-wise.

The ceil of the scalar *x* is the smallest integer *i*, such that *i* >= *x*. It is often denoted as $\lceil x \rceil$.

Parameters

x : array_like

Input data.

Returns

y : ndarray or scalar

The ceiling of each element in *x*, with `float` dtype.

See also:

floor, *trunc*, *rint*

Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.ceil(a)
array([-1., -1., -0.,  1.,  2.,  2.,  2.])
```

numpy.**trunc**(*x*[, *out*]) = <ufunc 'trunc'>

Return the truncated value of the input, element-wise.

The truncated value of the scalar *x* is the nearest integer *i* which is closer to zero than *x* is. In short, the fractional part of the signed number *x* is discarded.

Parameters

x : array_like

Input data.

Returns

y : ndarray or scalar

The truncated value of each element in *x*.

See also:

ceil, *floor*, *rint*

Notes

New in version 1.3.0.

Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.trunc(a)
array([-1., -1., -0.,  0.,  1.,  1.,  2.])
```

3.20.4 Sums, products, differences

3.20.5 Exponents and logarithms

exp(x[, out])	Calculate the exponential of all elements in the input array.
expm1(x[, out])	Calculate `exp(x) - 1` for all elements in the array.
exp2(x[, out])	Calculate *2**p* for all *p* in the input array.
log(x[, out])	Natural logarithm, element-wise.
log10(x[, out])	Return the base 10 logarithm of the input array, element-wise.
log2(x[, out])	Base-2 logarithm of *x*.
log1p(x[, out])	Return the natural logarithm of one plus the input array, element-wise.
logaddexp(x1, x2[, out])	Logarithm of the sum of exponentiations of the inputs.
logaddexp2(x1, x2[, out])	Logarithm of the sum of exponentiations of the inputs in base-2.

numpy.**exp**(*x*[, *out*]) **= <ufunc 'exp'>**

Calculate the exponential of all elements in the input array.

Parameters

x : array_like

Input values.

Returns

out : ndarray

Output array, element-wise exponential of *x*.

See also:

expm1

Calculate `exp(x) - 1` for all elements in the array.

exp2

Calculate `2**x` for all elements in the array.

Notes

The irrational number `e` is also known as Euler's number. It is approximately 2.718281, and is the base of the natural logarithm, `ln` (this means that, if $x = \ln y = \log_e y$, then $e^x = y$. For real input, `exp(x)` is always positive.

For complex arguments, `x = a + ib`, we can write $e^x = e^a e^{ib}$. The first term, e^a, is already known (it is the real argument, described above). The second term, e^{ib}, is $\cos b + i \sin b$, a function with magnitude 1 and a periodic phase.

References

[R18], *[R19]*

Examples

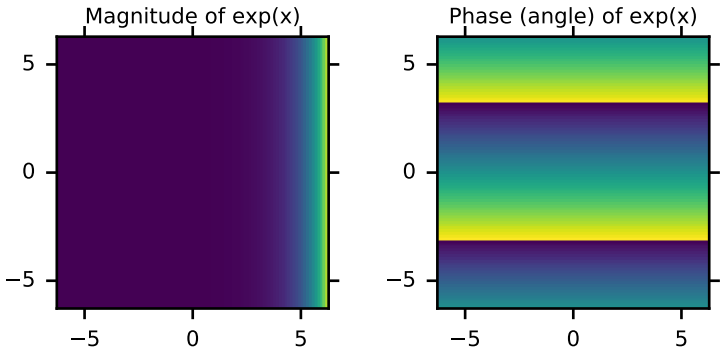
Plot the magnitude and phase of `exp(x)` in the complex plane:

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

```
>>> x = np.linspace(-2*np.pi, 2*np.pi, 100)
>>> xx = x + 1j * x[:, np.newaxis] # a + ib over complex plane
>>> out = np.exp(xx)
```

```
>>> plt.subplot(121)
>>> plt.imshow(np.abs(out),
...            extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi])
>>> plt.title('Magnitude of exp(x)')
```

```
>>> plt.subplot(122)
>>> plt.imshow(np.angle(out),
...            extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi])
>>> plt.title('Phase (angle) of exp(x)')
>>> plt.show()
```



numpy.**expm1**(*x*[, *out*]) **= <ufunc 'expm1'>**

Calculate `exp(x) - 1` for all elements in the array.

Parameters

x : array_like

Input values.

Returns

out : ndarray

Element-wise exponential minus one: `out = exp(x) - 1`.

See also:

log1p

`log(1 + x)`, the inverse of expm1.

Notes

This function provides greater precision than `exp(x) - 1` for small values of `x`.

Examples

The true value of `exp(1e-10) - 1` is `1.00000000005e-10` to about 32 significant digits. This example shows the superiority of expm1 in this case.

```
>>> np.expm1(1e-10)
1.00000000005e-10
>>> np.exp(1e-10) - 1
1.000000082740371e-10
```

numpy.**exp2**(*x*[, *out*]) = <ufunc 'exp2'>

Calculate *2**p* for all *p* in the input array.

Parameters

x : array_like

Input values.

out : ndarray, optional

Array to insert results into.

Returns

out : ndarray

Element-wise 2 to the power *x*.

See also:

power

Notes

New in version 1.3.0.

Examples

```
>>> np.exp2([2, 3])
array([ 4.,  8.])
```

numpy.**log**(*x*[, *out*]) = <ufunc 'log'>

Natural logarithm, element-wise.

The natural logarithm *log* is the inverse of the exponential function, so that *log(exp(x)) = x*. The natural logarithm is logarithm in base `e`.

Parameters

x : array_like

Input value.

Returns

y : ndarray

The natural logarithm of *x*, element-wise.

See also:

log10, *log2*, *log1p*, `emath.log`

Notes

Logarithm is a multivalued function: for each x there is an infinite number of z such that $exp(z) = x$. The convention is to return the z whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, `log` always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, `log` is a complex analytical function that has a branch cut *[-inf, 0]* and is continuous from above on it. `log` handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

[R44], *[R45]*

Examples

```
>>> np.log([1, np.e, np.e**2, 0])
array([  0.,   1.,   2., -Inf])
```

`numpy.log10`(*x*[, *out*]) = <ufunc 'log10'>

Return the base 10 logarithm of the input array, element-wise.

Parameters

x : array_like

Input values.

Returns

y : ndarray

The logarithm to the base 10 of *x*, element-wise. NaNs are returned where x is negative.

See also:

`emath.log10`

Notes

Logarithm is a multivalued function: for each x there is an infinite number of z such that $10^{**}z = x$. The convention is to return the z whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, `log10` always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, `log10` is a complex analytical function that has a branch cut *[-inf, 0]* and is continuous from above on it. `log10` handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

[R46], *[R47]*

Examples

```
>>> np.log10([1e-15, -3.])
array([-15.,  NaN])
```

`numpy.log2`(*x*[, *out*]) = <ufunc 'log2'>

Base-2 logarithm of *x*.

Parameters
x : array_like

Input values.

Returns
y : ndarray

Base-2 logarithm of *x*.

See also:

log, *log10*, *log1p*, `emath.log2`

Notes

New in version 1.3.0.

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *2**z = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log2* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log2* is a complex analytical function that has a branch cut *[-inf, 0]* and is continuous from above on it. *log2* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

Examples

```
>>> x = np.array([0, 1, 2, 2**4])
>>> np.log2(x)
array([-Inf,   0.,   1.,   4.])
```

```
>>> xi = np.array([0+1.j, 1, 2+0.j, 4.j])
>>> np.log2(xi)
array([ 0.+2.26618007j,  0.+0.j        ,  1.+0.j        ,  2.+2.26618007j])
```

`numpy.`**`log1p`**(*x*[, *out*]) **= <ufunc 'log1p'>**
Return the natural logarithm of one plus the input array, element-wise.

Calculates `log(1 + x)`.

Parameters
x : array_like

Input values.

Returns
y : ndarray

Natural logarithm of *1 + x*, element-wise.

See also:

expm1
`exp(x) - 1`, the inverse of *log1p*.

Notes

For real-valued input, *log1p* is accurate also for *x* so small that *1 + x == 1* in floating-point accuracy.

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *exp(z) = 1 + x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log1p* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log1p* is a complex analytical function that has a branch cut *[-inf, -1]* and is continuous from above on it. *log1p* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

[R48], *[R49]*

Examples

```
>>> np.log1p(1e-99)
1e-99
>>> np.log(1 + 1e-99)
0.0
```

numpy.**logaddexp**(*x1*, *x2*[, *out*]) = <ufunc 'logaddexp'>

Logarithm of the sum of exponentiations of the inputs.

Calculates `log(exp(x1) + exp(x2))`. This function is useful in statistics where the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the logarithm of the calculated probability is stored. This function allows adding probabilities stored in such a fashion.

Parameters

x1, x2 : array_like

Input values.

Returns

result : ndarray

Logarithm of `exp(x1) + exp(x2)`.

See also:

logaddexp2

Logarithm of the sum of exponentiations of inputs in base 2.

Notes

New in version 1.3.0.

Examples

```
>>> prob1 = np.log(1e-50)
>>> prob2 = np.log(2.5e-50)
>>> prob12 = np.logaddexp(prob1, prob2)
>>> prob12
-113.87649168120691
>>> np.exp(prob12)
3.5000000000000057e-50
```

numpy.**logaddexp2**(*x1*, *x2*[, *out*]) = <ufunc 'logaddexp2'>

Logarithm of the sum of exponentiations of the inputs in base-2.

Calculates `log2(2**x1 + 2**x2)`. This function is useful in machine learning when the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the base-2 logarithm of the calculated probability can be used instead. This function allows adding probabilities stored in such a fashion.

Parameters

x1, x2 : array_like

Input values.

out : ndarray, optional

Array to store results in.

Returns

result : ndarray

Base-2 logarithm of `2**x1 + 2**x2`.

See also:

logaddexp
Logarithm of the sum of exponentiations of the inputs.

Notes

New in version 1.3.0.

Examples

```
>>> prob1 = np.log2(1e-50)
>>> prob2 = np.log2(2.5e-50)
>>> prob12 = np.logaddexp2(prob1, prob2)
>>> prob1, prob2, prob12
(-166.09640474436813, -164.77447664948076, -164.28904982231052)
>>> 2**prob12
3.4999999999999914e-50
```

3.20.6 Other special functions

3.20.7 Floating point routines

signbit(x[, out])	Returns element-wise True where signbit is set (less than zero).
copysign(x1, x2[, out])	Change the sign of x1 to that of x2, element-wise.
frexp(x[, out1, out2])	Decompose the elements of x into mantissa and twos exponent.
ldexp(x1, x2[, out])	Returns x1 * 2**x2, element-wise.

numpy.**signbit**(*x*[, *out*]) **= <ufunc 'signbit'>**
Returns element-wise True where signbit is set (less than zero).

Parameters

x : array_like

The input value(s).

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See `doc.ufuncs`.

Returns

result : ndarray of bool

Output array, or reference to *out* if that was supplied.

Examples

```
>>> np.signbit(-1.2)
True
>>> np.signbit(np.array([1, -2.3, 2.1]))
array([False,  True, False], dtype=bool)
```

numpy.**copysign**(*x1*, *x2*[, *out*]) = <ufunc 'copysign'>

Change the sign of x1 to that of x2, element-wise.

If both arguments are arrays or sequences, they have to be of the same length. If *x2* is a scalar, its sign will be copied to all elements of *x1*.

Parameters

x1 : array_like

Values to change the sign of.

x2 : array_like

The sign of *x2* is copied to *x1*.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

out : array_like

The values of *x1* with the sign of *x2*.

Examples

```
>>> np.copysign(1.3, -1)
-1.3
>>> 1/np.copysign(0, 1)
inf
>>> 1/np.copysign(0, -1)
-inf
```

```
>>> np.copysign([-1, 0, 1], -1.1)
array([-1., -0., -1.])
>>> np.copysign([-1, 0, 1], np.arange(3)-1)
array([-1.,  0.,  1.])
```

numpy.**frexp**(*x*[, *out1*, *out2*]) = <ufunc 'frexp'>

Decompose the elements of x into mantissa and twos exponent.

Returns (*mantissa*, *exponent*), where *x = mantissa * 2**exponent*'. The mantissa is lies in the open interval(-1, 1), while the twos exponent is a signed integer.

Parameters

x : array_like

Array of numbers to be decomposed.

out1 : ndarray, optional

Output array for the mantissa. Must have the same shape as *x*.

out2 : ndarray, optional

Output array for the exponent. Must have the same shape as *x*.

Returns

(mantissa, exponent) : tuple of ndarrays, (float, int)

mantissa is a float array with values between -1 and 1. *exponent* is an int array which represents the exponent of 2.

See also:

ldexp

Compute `y = x1 * 2**x2`, the inverse of *frexp*.

Notes

Complex dtypes are not supported, they will raise a TypeError.

Examples

```
>>> x = np.arange(9)
>>> y1, y2 = np.frexp(x)
>>> y1
array([ 0.   ,  0.5  ,  0.5  ,  0.75 ,  0.5  ,  0.625,  0.75 ,  0.875,
        0.5  ])
>>> y2
array([0, 1, 2, 2, 3, 3, 3, 3, 4])
>>> y1 * 2**y2
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
```

numpy.**ldexp**(*x1*, *x2*[, *out*]) = <ufunc 'ldexp'>

Returns x1 * 2**x2, element-wise.

The mantissas *x1* and twos exponents *x2* are used to construct floating point numbers `x1 * 2**x2`.

Parameters

x1 : array_like

Array of multipliers.

x2 : array_like, int

Array of twos exponents.

out : ndarray, optional

Output array for the result.

Returns

y : ndarray or scalar

The result of `x1 * 2**x2`.

See also:

frexp

Return (y1, y2) from `x = y1 * 2**y2`, inverse to *ldexp*.

Notes

Complex dtypes are not supported, they will raise a TypeError.

ldexp is useful as the inverse of *frexp*, if used by itself it is more clear to simply use the expression `x1 * 2**x2`.

Examples

```
>>> np.ldexp(5, np.arange(4))
array([  5.,  10.,  20.,  40.], dtype=float32)
```

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

3.20.8 Arithmetic operations

add(x1, x2[, out])	Add arguments element-wise.
reciprocal(x[, out])	Return the reciprocal of the argument, element-wise.
negative(x[, out])	Numerical negative, element-wise.
multiply(x1, x2[, out])	Multiply arguments element-wise.
divide(x1, x2[, out])	Divide arguments element-wise.
power(x1, x2[, out])	First array elements raised to powers from second array, element-wise.
subtract(x1, x2[, out])	Subtract arguments, element-wise.
true_divide(x1, x2[, out])	Returns a true division of the inputs, element-wise.
floor_divide(x1, x2[, out])	Return the largest integer smaller or equal to the division of the inputs.
fmod(x1, x2[, out])	Return the element-wise remainder of division.
mod(x1, x2[, out])	Return element-wise remainder of division.
modf(x[, out1, out2])	Return the fractional and integral parts of an array, element-wise.
remainder(x1, x2[, out])	Return element-wise remainder of division.

numpy.**add**(*x1*, *x2*[, *out*]) **= <ufunc 'add'>**

Add arguments element-wise.

Parameters

x1, x2 : array_like

The arrays to be added. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which may be the shape of one or the other).

Returns

add : ndarray or scalar

The sum of *x1* and *x2*, element-wise. Returns a scalar if both *x1* and *x2* are scalars.

Notes

Equivalent to *x1* + *x2* in terms of array broadcasting.

Examples

```
>>> np.add(1.0, 4.0)
5.0
>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
```

```
>>> np.add(x1, x2)
array([[  0.,   2.,   4.],
       [  3.,   5.,   7.],
       [  6.,   8.,  10.]])
```

numpy.**reciprocal**(*x*[, *out*]) = <ufunc 'reciprocal'>

Return the reciprocal of the argument, element-wise.

Calculates `1/x`.

Parameters

x : array_like

Input array.

Returns

y : ndarray

Return array.

Notes

Note: This function is not designed to work with integers.

For integer arguments with absolute value larger than 1 the result is always zero because of the way Python handles integer division. For integer zero the result is an overflow.

Examples

```
>>> np.reciprocal(2.)
0.5
>>> np.reciprocal([1, 2., 3.33])
array([ 1.       ,  0.5      ,  0.3003003])
```

numpy.**negative**(*x*[, *out*]) = <ufunc 'negative'>

Numerical negative, element-wise.

Parameters

x : array_like or scalar

Input array.

Returns

y : ndarray or scalar

Returned array or scalar: *y* = *-x*.

Examples

```
>>> np.negative([1.,-1.])
array([-1.,  1.])
```

numpy.**multiply**(*x1*, *x2*[, *out*]) = <ufunc 'multiply'>

Multiply arguments element-wise.

Parameters

x1, x2 : array_like

Input arrays to be multiplied.

Returns

y : ndarray

The product of *x1* and *x2*, element-wise. Returns a scalar if both *x1* and *x2* are scalars.

Notes

Equivalent to *x1* * *x2* in terms of array broadcasting.

Examples

```
>>> np.multiply(2.0, 4.0)
8.0
```

```
>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> np.multiply(x1, x2)
array([[  0.,   1.,   4.],
       [  0.,   4.,  10.],
       [  0.,   7.,  16.]])
```

numpy.**divide**(*x1*, *x2*[, *out*]) **= <ufunc 'divide'>**

Divide arguments element-wise.

Parameters

x1 : array_like

Dividend array.

x2 : array_like

Divisor array.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

y : ndarray or scalar

The quotient `x1/x2`, element-wise. Returns a scalar if both `x1` and `x2` are scalars.

See also:

seterr

Set whether to raise or warn on overflow, underflow and division by zero.

Notes

Equivalent to `x1` / `x2` in terms of array-broadcasting.

Behavior on division by zero can be changed using `seterr`.

In Python 2, when both `x1` and `x2` are of an integer type, `divide` will behave like `floor_divide`. In Python 3, it behaves like `true_divide`.

Examples

```
>>> np.divide(2.0, 4.0)
0.5
>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> np.divide(x1, x2)
array([[ NaN,  1. ,  1. ],
```

```
           [ Inf,  4. ,  2.5],
           [ Inf,  7. ,  4. ]])
```

Note the behavior with integer types (Python 2 only):

```
>>> np.divide(2, 4)
0
>>> np.divide(2, 4.)
0.5
```

Division by zero always yields zero in integer arithmetic (again, Python 2 only), and does not raise an exception or a warning:

```
>>> np.divide(np.array([0, 1], dtype=int), np.array([0, 0], dtype=int))
array([0, 0])
```

Division by zero can, however, be caught using `seterr`:

```
>>> old_err_state = np.seterr(divide='raise')
>>> np.divide(1, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FloatingPointError: divide by zero encountered in divide
```

```
>>> ignored_states = np.seterr(**old_err_state)
>>> np.divide(1, 0)
0
```

numpy.**power**(*x1*, *x2*[, *out*]) = **<ufunc 'power'>**

First array elements raised to powers from second array, element-wise.

Raise each base in *x1* to the positionally-corresponding power in *x2*. *x1* and *x2* must be broadcastable to the same shape.

Parameters

x1 : array_like

The bases.

x2 : array_like

The exponents.

Returns

y : ndarray

The bases in *x1* raised to the exponents in *x2*.

Examples

Cube each element in a list.

```
>>> x1 = range(6)
>>> x1
[0, 1, 2, 3, 4, 5]
>>> np.power(x1, 3)
array([  0,   1,   8,  27,  64, 125])
```

Raise the bases to different exponents.

```
>>> x2 = [1.0, 2.0, 3.0, 3.0, 2.0, 1.0]
>>> np.power(x1, x2)
array([  0.,   1.,   8.,  27.,  16.,   5.])
```

The effect of broadcasting.

```
>>> x2 = np.array([[1, 2, 3, 3, 2, 1], [1, 2, 3, 3, 2, 1]])
>>> x2
array([[1, 2, 3, 3, 2, 1],
       [1, 2, 3, 3, 2, 1]])
>>> np.power(x1, x2)
array([[ 0,  1,  8, 27, 16,  5],
       [ 0,  1,  8, 27, 16,  5]])
```

numpy.**subtract**(*x1*, *x2*[, *out*]) = **<ufunc 'subtract'>**

Subtract arguments, element-wise.

Parameters

x1, x2 : array_like

The arrays to be subtracted from each other.

Returns

y : ndarray

The difference of *x1* and *x2*, element-wise. Returns a scalar if both *x1* and *x2* are scalars.

Notes

Equivalent to `x1 - x2` in terms of array broadcasting.

Examples

```
>>> np.subtract(1.0, 4.0)
-3.0
```

```
>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> np.subtract(x1, x2)
array([[ 0.,  0.,  0.],
       [ 3.,  3.,  3.],
       [ 6.,  6.,  6.]])
```

numpy.**true_divide**(*x1*, *x2*[, *out*]) = **<ufunc 'true_divide'>**

Returns a true division of the inputs, element-wise.

Instead of the Python traditional 'floor division', this returns a true division. True division adjusts the output type to present the best answer, regardless of input types.

Parameters

x1 : array_like

Dividend array.

x2 : array_like

Divisor array.

Returns

out : ndarray

Result is scalar if both inputs are scalar, ndarray otherwise.

Notes

The floor division operator `//` was added in Python 2.2 making `//` and `/` equivalent operators. The default floor division operation of `/` can be replaced by true division with `from __future__ import division`.

In Python 3.0, `//` is the floor division operator and `/` the true division operator. The `true_divide(x1, x2)` function is equivalent to true division in Python.

Examples

```
>>> x = np.arange(5)
>>> np.true_divide(x, 4)
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

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

```
>>> from __future__ import division
>>> x/4
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
>>> x//4
array([0, 0, 0, 0, 1])
```

numpy.**floor_divide**(*x1*, *x2*[, *out*]) **= <ufunc 'floor_divide'>**

Return the largest integer smaller or equal to the division of the inputs. It is equivalent to the Python `//` operator and pairs with the Python `%` (*remainder*), function so that `b = a % b + b * (a // b)` up to roundoff.

Parameters

x1 : array_like

Numerator.

x2 : array_like

Denominator.

Returns

y : ndarray

y = floor(*x1*/*x2*)

See also:

remainder

Remainder complementary to floor_divide.

divide

Standard division.

floor

Round a number to the nearest integer toward minus infinity.

ceil

Round a number to the nearest integer toward infinity.

Examples

```
>>> np.floor_divide(7,3)
2
>>> np.floor_divide([1., 2., 3., 4.], 2.5)
array([ 0.,  0.,  1.,  1.])
```

numpy.**fmod**(*x1*, *x2*[, *out*]) = <ufunc 'fmod'>

Return the element-wise remainder of division.

This is the NumPy implementation of the C library function fmod, the remainder has the same sign as the dividend *x1*. It is equivalent to the Matlab(TM) `rem` function and should not be confused with the Python modulus operator `x1 % x2`.

Parameters

x1 : array_like

Dividend.

x2 : array_like

Divisor.

Returns

y : array_like

The remainder of the division of *x1* by *x2*.

See also:

remainder

Equivalent to the Python `%` operator.

divide

Notes

The result of the modulo operation for negative dividend and divisors is bound by conventions. For *fmod*, the sign of result is the sign of the dividend, while for *remainder* the sign of the result is the sign of the divisor. The *fmod* function is equivalent to the Matlab(TM) `rem` function.

Examples

```
>>> np.fmod([-3, -2, -1, 1, 2, 3], 2)
array([-1,  0, -1,  1,  0,  1])
>>> np.remainder([-3, -2, -1, 1, 2, 3], 2)
array([1, 0, 1, 1, 0, 1])
```

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

numpy.**mod**(*x1*, *x2*[, *out*]) = <ufunc 'remainder'>

Return element-wise remainder of division.

Computes the remainder complementary to the *floor_divide* function. It is equivalent to the Python modulus operator``x1 % x2`` and has the same sign as the divisor *x2*. It should not be confused with the Matlab(TM) `rem` function.

Parameters

x1 : array_like

Dividend array.

x2 : array_like

Divisor array.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

y : ndarray

The element-wise remainder of the quotient `floor_divide(x1, x2)`. Returns a scalar if both *x1* and *x2* are scalars.

See also:

floor_divide
Equivalent of Python `//` operator.

fmod
Equivalent of the Matlab(TM) `rem` function.

divide, *floor*

Notes

Returns 0 when *x2* is 0 and both *x1* and *x2* are (arrays of) integers.

Examples

```
>>> np.remainder([4, 7], [2, 3])
array([0, 1])
>>> np.remainder(np.arange(7), 5)
array([0, 1, 2, 3, 4, 0, 1])
```

numpy.**modf**(*x*[, *out1*, *out2*]) **= <ufunc 'modf'>**
Return the fractional and integral parts of an array, element-wise.

The fractional and integral parts are negative if the given number is negative.

Parameters

x : array_like

Input array.

Returns

y1 : ndarray

Fractional part of *x*.

y2 : ndarray

Integral part of *x*.

Notes

For integer input the return values are floats.

Examples

```
>>> np.modf([0, 3.5])
(array([ 0. ,  0.5]), array([ 0.,  3.]))
>>> np.modf(-0.5)
(-0.5, -0)
```

numpy.**remainder**(*x1*, *x2*[, *out*]) = <ufunc 'remainder'>

Return element-wise remainder of division.

Computes the remainder complementary to the *floor_divide* function. It is equivalent to the Python modulus operator``x1 % x2`` and has the same sign as the divisor *x2*. It should not be confused with the Matlab(TM) `rem` function.

Parameters

x1 : array_like

Dividend array.

x2 : array_like

Divisor array.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

y : ndarray

The element-wise remainder of the quotient `floor_divide(x1, x2)`. Returns a scalar if both *x1* and *x2* are scalars.

See also:

floor_divide

Equivalent of Python `//` operator.

fmod

Equivalent of the Matlab(TM) `rem` function.

divide, *floor*

Notes

Returns 0 when *x2* is 0 and both *x1* and *x2* are (arrays of) integers.

Examples

```
>>> np.remainder([4, 7], [2, 3])
array([0, 1])
>>> np.remainder(np.arange(7), 5)
array([0, 1, 2, 3, 4, 0, 1])
```

3.20.9 Handling complex numbers

conj(x[, out])	Return the complex conjugate, element-wise.

numpy.**conj**(*x*[, *out*]) = <ufunc 'conjugate'>
Return the complex conjugate, element-wise.

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

Parameters
x : array_like

Input value.

Returns
y : ndarray

The complex conjugate of *x*, with same dtype as *y*.

Examples

```
>>> np.conjugate(1+2j)
(1-2j)
```

```
>>> x = np.eye(2) + 1j * np.eye(2)
>>> np.conjugate(x)
array([[ 1.-1.j,  0.-0.j],
       [ 0.-0.j,  1.-1.j]])
```

3.20.10 Miscellaneous

sqrt(x[, out])	Return the positive square-root of an array, element-wise.
square(x[, out])	Return the element-wise square of the input.
absolute(x[, out])	Calculate the absolute value element-wise.
fabs(x[, out])	Compute the absolute values element-wise.
sign(x[, out])	Returns an element-wise indication of the sign of a number.
maximum(x1, x2[, out])	Element-wise maximum of array elements.
minimum(x1, x2[, out])	Element-wise minimum of array elements.
fmax(x1, x2[, out])	Element-wise maximum of array elements.
fmin(x1, x2[, out])	Element-wise minimum of array elements.

numpy.**sqrt**(*x*[, *out*]) = <ufunc 'sqrt'>
Return the positive square-root of an array, element-wise.

Parameters
x : array_like

The values whose square-roots are required.

out : ndarray, optional

Alternate array object in which to put the result; if provided, it must have the same shape as *x*

Returns
y : ndarray

An array of the same shape as *x*, containing the positive square-root of each element in

x. If any element in *x* is complex, a complex array is returned (and the square-roots of negative reals are calculated). If all of the elements in *x* are real, so is *y*, with negative elements returning `nan`. If *out* was provided, *y* is a reference to it.

See also:

`lib.scimath.sqrt`
A version which returns complex numbers when given negative reals.

Notes

sqrt has–consistent with common convention–as its branch cut the real "interval" [-*inf*, 0), and is continuous from above on it. A branch cut is a curve in the complex plane across which a given complex function fails to be continuous.

Examples

```
>>> np.sqrt([1,4,9])
array([ 1.,  2.,  3.])
```

```
>>> np.sqrt([4, -1, -3+4J])
array([ 2.+0.j,  0.+1.j,  1.+2.j])
```

```
>>> np.sqrt([4, -1, numpy.inf])
array([  2.,  NaN,  Inf])
```

`numpy.`**`square`**`(`*x*`[, `*out*`])` **= <ufunc 'square'>**
Return the element-wise square of the input.

Parameters
x : array_like

Input data.

Returns
out : ndarray

Element-wise *x***x*, of the same shape and dtype as *x*. Returns scalar if *x* is a scalar.

See also:

`numpy.linalg.matrix_power`, *`sqrt`*, *`power`*

Examples

```
>>> np.square([-1j, 1])
array([-1.-0.j,  1.+0.j])
```

`numpy.`**`absolute`**`(`*x*`[, `*out*`])` **= <ufunc 'absolute'>**
Calculate the absolute value element-wise.

Parameters
x : array_like

Input array.

Returns
absolute : ndarray

An ndarray containing the absolute value of each element in *x*. For complex input, `a + ib`, the absolute value is $\sqrt{a^2 + b^2}$.

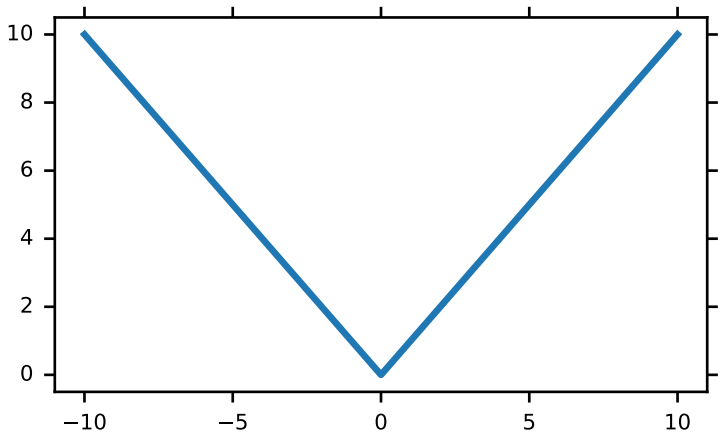
Examples

```
>>> x = np.array([-1.2, 1.2])
>>> np.absolute(x)
array([ 1.2,  1.2])
>>> np.absolute(1.2 + 1j)
1.5620499351813308
```

Plot the function over `[-10, 10]`:

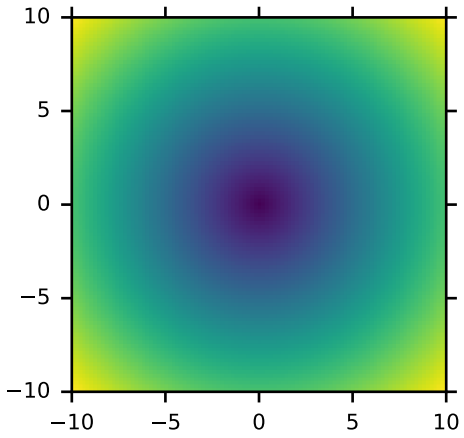
```
>>> import matplotlib.pyplot as plt
```

```
>>> x = np.linspace(start=-10, stop=10, num=101)
>>> plt.plot(x, np.absolute(x))
>>> plt.show()
```



Plot the function over the complex plane:

```
>>> xx = x + 1j * x[:, np.newaxis]
>>> plt.imshow(np.abs(xx), extent=[-10, 10, -10, 10])
>>> plt.show()
```



numpy.**fabs**(*x*[, *out*]) = <ufunc 'fabs'>

Compute the absolute values element-wise.

This function returns the absolute values (positive magnitude) of the data in *x*. Complex values are not handled, use *absolute* to find the absolute values of complex data.

Parameters

x : array_like

The array of numbers for which the absolute values are required. If *x* is a scalar, the result *y* will also be a scalar.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

y : ndarray or scalar

The absolute values of *x*, the returned values are always floats.

See also:

absolute

Absolute values including `complex` types.

Examples

```
>>> np.fabs(-1)
1.0
>>> np.fabs([-1.2, 1.2])
array([ 1.2,  1.2])
```

numpy.**sign**(*x*[, *out*]) = <ufunc 'sign'>

Returns an element-wise indication of the sign of a number.

The *sign* function returns `-1 if x < 0, 0 if x==0, 1 if x > 0`. nan is returned for nan inputs.

For complex inputs, the *sign* function returns `sign(x.real) + 0j if x.real != 0 else sign(x.imag) + 0j`.

complex(nan, 0) is returned for complex nan inputs.

Parameters

x : array_like

Input values.

Returns

y : ndarray

The sign of *x*.

Notes

There is more than one definition of sign in common use for complex numbers. The definition used here is equivalent to $x/\sqrt{x*x}$ which is different from a common alternative, $x/|x|$.

Examples

```
>>> np.sign([-5., 4.5])
array([-1.,  1.])
>>> np.sign(0)
0
>>> np.sign(5-2j)
(1+0j)
```

numpy.**maximum**(*x1*, *x2*[, *out*]) = <ufunc 'maximum'>

Element-wise maximum of array elements.

Compare two arrays and returns a new array containing the element-wise maxima. If one of the elements being compared is a NaN, then that element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are propagated.

Parameters

x1, x2 : array_like

The arrays holding the elements to be compared. They must have the same shape, or shapes that can be broadcast to a single shape.

Returns

y : ndarray or scalar

The maximum of *x1* and *x2*, element-wise. Returns scalar if both *x1* and *x2* are scalars.

See also:

minimum

Element-wise minimum of two arrays, propagates NaNs.

fmax

Element-wise maximum of two arrays, ignores NaNs.

amax

The maximum value of an array along a given axis, propagates NaNs.

nanmax

The maximum value of an array along a given axis, ignores NaNs.

fmin, amin, nanmin

Notes

The maximum is equivalent to `np.where(x1 >= x2, x1, x2)` when neither x1 nor x2 are nans, but it is faster and does proper broadcasting.

Examples

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

```
>>> np.maximum(np.eye(2), [0.5, 2]) # broadcasting
array([[ 1. ,  2. ],
       [ 0.5,  2. ]])
```

```
>>> np.maximum([np.nan, 0, np.nan], [0, np.nan, np.nan])
array([ NaN,  NaN,  NaN])
>>> np.maximum(np.Inf, 1)
inf
```

numpy.**minimum**(*x1*, *x2*[, *out*]) **= <ufunc 'minimum'>**

Element-wise minimum of array elements.

Compare two arrays and returns a new array containing the element-wise minima. If one of the elements being compared is a NaN, then that element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are propagated.

Parameters

x1, x2 : array_like

The arrays holding the elements to be compared. They must have the same shape, or shapes that can be broadcast to a single shape.

Returns

y : ndarray or scalar

The minimum of *x1* and *x2*, element-wise. Returns scalar if both *x1* and *x2* are scalars.

See also:

maximum

Element-wise maximum of two arrays, propagates NaNs.

fmin

Element-wise minimum of two arrays, ignores NaNs.

amin

The minimum value of an array along a given axis, propagates NaNs.

nanmin

The minimum value of an array along a given axis, ignores NaNs.

fmax, amax, nanmax

Notes

The minimum is equivalent to `np.where(x1 <= x2, x1, x2)` when neither x1 nor x2 are NaNs, but it is faster and does proper broadcasting.

Examples

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

```
>>> np.minimum(np.eye(2), [0.5, 2]) # broadcasting
array([[ 0.5,  0. ],
       [ 0. ,  1. ]])
```

```
>>> np.minimum([np.nan, 0, np.nan],[0, np.nan, np.nan])
array([ NaN,  NaN,  NaN])
>>> np.minimum(-np.Inf, 1)
-inf
```

numpy.**fmax**(*x1*, *x2*[, *out*]) **= <ufunc 'fmax'>**

Element-wise maximum of array elements.

Compare two arrays and returns a new array containing the element-wise maxima. If one of the elements being compared is a NaN, then the non-nan element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are ignored when possible.

Parameters

x1, x2 : array_like

The arrays holding the elements to be compared. They must have the same shape.

Returns

y : ndarray or scalar

The maximum of *x1* and *x2*, element-wise. Returns scalar if both *x1* and *x2* are scalars.

See also:

fmin

Element-wise minimum of two arrays, ignores NaNs.

maximum

Element-wise maximum of two arrays, propagates NaNs.

amax

The maximum value of an array along a given axis, propagates NaNs.

nanmax

The maximum value of an array along a given axis, ignores NaNs.

minimum, amin, nanmin

Notes

New in version 1.3.0.

The fmax is equivalent to `np.where(x1 >= x2, x1, x2)` when neither x1 nor x2 are NaNs, but it is faster and does proper broadcasting.

Examples

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

```
>>> np.fmax(np.eye(2), [0.5, 2])
array([[ 1. ,  2. ],
       [ 0.5,  2. ]])
```

```
>>> np.fmax([np.nan, 0, np.nan],[0, np.nan, np.nan])
array([  0.,   0.,  NaN])
```

numpy.**fmin**(*x1*, *x2*[, *out*]) **= <ufunc 'fmin'>**

Element-wise minimum of array elements.

Compare two arrays and returns a new array containing the element-wise minima. If one of the elements being compared is a NaN, then the non-nan element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are ignored when possible.

Parameters

x1, x2 : array_like

The arrays holding the elements to be compared. They must have the same shape.

Returns

y : ndarray or scalar

The minimum of *x1* and *x2*, element-wise. Returns scalar if both *x1* and *x2* are scalars.

See also:

fmax

Element-wise maximum of two arrays, ignores NaNs.

minimum

Element-wise minimum of two arrays, propagates NaNs.

amin

The minimum value of an array along a given axis, propagates NaNs.

nanmin

The minimum value of an array along a given axis, ignores NaNs.

maximum, amax, nanmax

Notes

New in version 1.3.0.

The fmin is equivalent to `np.where(x1 <= x2, x1, x2)` when neither x1 nor x2 are NaNs, but it is faster and does proper broadcasting.

Examples

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

```
>>> np.fmin(np.eye(2), [0.5, 2])
array([[ 1. ,  2. ],
       [ 0.5,  2. ]])
```

```
>>> np.fmin([np.nan, 0, np.nan],[0, np.nan, np.nan])
array([  0.,   0.,  NaN])
```

3.21 Matrix library (numpy.matlib)

This module contains all functions in the *numpy* namespace, with the following replacement functions that return matrices instead of *ndarrays*.

Functions that are also in the numpy namespace and return matrices

---

Replacement functions in matlib

empty(shape[, dtype, order])	Return a new matrix of given shape and type, without initializing entries.
zeros(shape[, dtype, order])	Return a matrix of given shape and type, filled with zeros.
ones(shape[, dtype, order])	Matrix of ones.
eye(n[, M, k, dtype])	Return a matrix with ones on the diagonal and zeros elsewhere.
identity(n[, dtype])	Returns the square identity matrix of given size.
repmat(a, m, n)	Repeat a 0-D to 2-D array or matrix MxN times.
rand(*args)	Return a matrix of random values with given shape.
randn(*args)	Return a random matrix with data from the "standard normal" distribution.

numpy.matlib.**empty**(*shape*, *dtype=None*, *order='C'*)

Return a new matrix of given shape and type, without initializing entries.

Parameters

shape : int or tuple of int

Shape of the empty matrix.

dtype : data-type, optional

Desired output data-type.

order : {'C', 'F'}, optional

Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

See also:

empty_like, *zeros*

Notes

empty, unlike *zeros*, does not set the matrix values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

Examples

```
>>> import numpy.matlib
>>> np.matlib.empty((2, 2))    # filled with random data
matrix([[  6.76425276e-320,   9.79033856e-307],
        [  7.39337286e-309,   3.22135945e-309]])         #random
>>> np.matlib.empty((2, 2), dtype=int)
matrix([[ 6600475,        0],
        [ 6586976, 22740995]])                           #random
```

numpy.matlib.**zeros**(*shape*, *dtype=None*, *order='C'*)

Return a matrix of given shape and type, filled with zeros.

Parameters

shape : int or sequence of ints

Shape of the matrix

dtype : data-type, optional

The desired data-type for the matrix, default is float.

order : {'C', 'F'}, optional

Whether to store the result in C- or Fortran-contiguous order, default is 'C'.

Returns

out : matrix

Zero matrix of given shape, dtype, and order.

See also:

numpy.zeros
Equivalent array function.

matlib.ones
Return a matrix of ones.

Notes

If `shape` has length one i.e. `(N,)`, or is a scalar `N`, *out* becomes a single row matrix of shape `(1,N)`.

Examples

```
>>> import numpy.matlib
>>> np.matlib.zeros((2, 3))
matrix([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])
```

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

numpy.matlib.**ones**(*shape*, *dtype=None*, *order='C'*)
Matrix of ones.

Return a matrix of given shape and type, filled with ones.

Parameters

shape : {sequence of ints, int}

Shape of the matrix

dtype : data-type, optional

The desired data-type for the matrix, default is np.float64.

order : {'C', 'F'}, optional

Whether to store matrix in C- or Fortran-contiguous order, default is 'C'.

Returns

out : matrix

Matrix of ones of given shape, dtype, and order.

See also:

ones
Array of ones.

matlib.zeros
Zero matrix.

Notes

If `shape` has length one i.e. `(N,)`, or is a scalar `N`, *out* becomes a single row matrix of shape `(1,N)`.

Examples

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

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

numpy.matlib.**eye**(*n*, *M=None*, *k=0*, *dtype=<type 'float'>*)
Return a matrix with ones on the diagonal and zeros elsewhere.

Parameters
n : int

Number of rows in the output.

M : int, optional

Number of columns in the output, defaults to *n*.

k : int, optional

Index of the diagonal: 0 refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.

dtype : dtype, optional

Data-type of the returned matrix.

Returns
I : matrix

A *n* x *M* matrix where all elements are equal to zero, except for the *k*-th diagonal, whose values are equal to one.

See also:

numpy.eye
Equivalent array function.

identity
Square identity matrix.

Examples

```
>>> import numpy.matlib
>>> np.matlib.eye(3, k=1, dtype=float)
matrix([[ 0.,  1.,  0.],
        [ 0.,  0.,  1.],
        [ 0.,  0.,  0.]])
```

numpy.matlib.**identity**(*n*, *dtype=None*)

Returns the square identity matrix of given size.

Parameters

n : int

Size of the returned identity matrix.

dtype : data-type, optional

Data-type of the output. Defaults to `float`.

Returns

out : matrix

n x *n* matrix with its main diagonal set to one, and all other elements zero.

See also:

numpy.identity

Equivalent array function.

matlib.eye

More general matrix identity function.

Examples

```
>>> import numpy.matlib
>>> np.matlib.identity(3, dtype=int)
matrix([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

numpy.matlib.**repmat**(*a*, *m*, *n*)

Repeat a 0-D to 2-D array or matrix MxN times.

Parameters

a : array_like

The array or matrix to be repeated.

m, n : int

The number of times *a* is repeated along the first and second axes.

Returns

out : ndarray

The result of repeating *a*.

Examples

```
>>> import numpy.matlib
>>> a0 = np.array(1)
>>> np.matlib.repmat(a0, 2, 3)
array([[1, 1, 1],
       [1, 1, 1]])
```

```
>>> a1 = np.arange(4)
>>> np.matlib.repmat(a1, 2, 2)
array([[0, 1, 2, 3, 0, 1, 2, 3],
       [0, 1, 2, 3, 0, 1, 2, 3]])
```

```
>>> a2 = np.asmatrix(np.arange(6).reshape(2, 3))
>>> np.matlib.repmat(a2, 2, 3)
matrix([[0, 1, 2, 0, 1, 2, 0, 1, 2],
        [3, 4, 5, 3, 4, 5, 3, 4, 5],
        [0, 1, 2, 0, 1, 2, 0, 1, 2],
        [3, 4, 5, 3, 4, 5, 3, 4, 5]])
```

numpy.matlib.**rand**(**args*)

Return a matrix of random values with given shape.

Create a matrix of the given shape and propagate it with random samples from a uniform distribution over `[0, 1)`.

Parameters

***args** : Arguments

Shape of the output. If given as N integers, each integer specifies the size of one dimension. If given as a tuple, this tuple gives the complete shape.

Returns

out : ndarray

The matrix of random values with shape given by **args*.

See also:

randn, *numpy.random.rand*

Examples

```
>>> import numpy.matlib
>>> np.matlib.rand(2, 3)
matrix([[ 0.68340382,  0.67926887,  0.83271405],
        [ 0.00793551,  0.20468222,  0.95253525]])       #random
>>> np.matlib.rand((2, 3))
matrix([[ 0.84682055,  0.73626594,  0.11308016],
        [ 0.85429008,  0.3294825 ,  0.89139555]])       #random
```

If the first argument is a tuple, other arguments are ignored:

```
>>> np.matlib.rand((2, 3), 4)
matrix([[ 0.46898646,  0.15163588,  0.95188261],
        [ 0.59208621,  0.09561818,  0.00583606]])       #random
```

numpy.matlib.**randn**(**args*)

Return a random matrix with data from the "standard normal" distribution.

randn generates a matrix filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1.

Parameters

***args** : Arguments

Shape of the output. If given as N integers, each integer specifies the size of one dimension. If given as a tuple, this tuple gives the complete shape.

Returns

Z : matrix of floats

A matrix of floating-point samples drawn from the standard normal distribution.

See also:

rand, random.randn

Notes

For random samples from $N(\mu, \sigma^2)$, use:

```
sigma * np.matlib.randn(...)  + mu
```

Examples

```
>>> import numpy.matlib
>>> np.matlib.randn(1)
matrix([[-0.09542833]])                                  #random
>>> np.matlib.randn(1, 2, 3)
matrix([[ 0.16198284,  0.0194571 ,  0.18312985],
        [-0.7509172 ,  1.61055   ,  0.45298599]])        #random
```

Two-by-four matrix of samples from $N(3, 6.25)$:

```
>>> 2.5 * np.matlib.randn((2, 4)) + 3
matrix([[ 4.74085004,  8.89381862,  4.09042411,  4.83721922],
        [ 7.52373709,  5.07933944, -2.64043543,  0.45610557]])  #random
```

3.22 Miscellaneous routines

3.22.1 Buffer objects

3.22.2 Performance tuning

3.22.3 Memory ranges

3.22.4 Numpy version comparison

3.23 Padding Arrays

3.24 Polynomials

Polynomials in NumPy can be *created*, *manipulated*, and even *fitted* using the Using the Convenience Classes of the numpy.polynomial package, introduced in NumPy 1.4.

Prior to NumPy 1.4, numpy.poly1d was the class of choice and it is still available in order to maintain backward

compatibility. However, the newer Polynomial package is more complete than `numpy.poly1d` and its convenience classes are better behaved in the numpy environment. Therefore Polynomial is recommended for new coding.

3.24.1 Transition notice

The various routines in the Polynomial package all deal with series whose coefficients go from degree zero upward, which is the *reverse order* of the Poly1d convention. The easy way to remember this is that indexes correspond to degree, i.e., coef[i] is the coefficient of the term of degree i.

Polynomial Package

New in version 1.4.0.

Using the Convenience Classes

The convenience classes provided by the polynomial package are:

Name	Provides
Polynomial	Power series
Chebyshev	Chebyshev series
Legendre	Legendre series
Laguerre	Laguerre series
Hermite	Hermite series
HermiteE	HermiteE series

The series in this context are finite sums of the corresponding polynomial basis functions multiplied by coefficients. For instance, a power series looks like

$$p(x) = 1 + 2x + 3x^2$$

and has coefficients $[1, 2, 3]$. The Chebyshev series with the same coefficients looks like

$$p(x) = 1T_0(x) + 2T_1(x) + 3T_2(x)$$

and more generally

$$p(x) = \sum_{i=0}^{n} c_i T_i(x)$$

where in this case the T_n are the Chebyshev functions of degree n, but could just as easily be the basis functions of any of the other classes. The convention for all the classes is that the coefficient $c[i]$ goes with the basis function of degree i.

All of the classes have the same methods, and especially they implement the Python numeric operators +, -, *, //, %, divmod, **, ==, and !=. The last two can be a bit problematic due to floating point roundoff errors. We now give a quick demonstration of the various operations using Numpy version 1.7.0.

Basics First we need a polynomial class and a polynomial instance to play with. The classes can be imported directly from the polynomial package or from the module of the relevant type. Here we import from the package and use the conventional Polynomial class because of its familiarity:

```
>>> from numpy.polynomial import Polynomial as P
>>> p = P([1,2,3])
>>> p
Polynomial([ 1.,  2.,  3.], [-1.,  1.], [-1.,  1.])
```

Note that there are three parts to the long version of the printout. The first is the coefficients, the second is the domain, and the third is the window:

```
>>> p.coef
array([ 1.,  2.,  3.])
>>> p.domain
array([-1.,  1.])
>>> p.window
array([-1.,  1.])
```

Printing a polynomial yields a shorter form without the domain and window:

```
>>> print p
poly([ 1.  2.  3.])
```

We will deal with the domain and window when we get to fitting, for the moment we ignore them and run through the basic algebraic and arithmetic operations.

Addition and Subtraction:

```
>>> p + p
Polynomial([ 2.,  4.,  6.], [-1.,  1.], [-1.,  1.])
>>> p - p
Polynomial([ 0.], [-1.,  1.], [-1.,  1.])
```

Multiplication:

```
>>> p * p
Polynomial([  1.,   4.,  10.,  12.,   9.], [-1.,  1.], [-1.,  1.])
```

Powers:

```
>>> p**2
Polynomial([  1.,   4.,  10.,  12.,   9.], [-1.,  1.], [-1.,  1.])
```

Division:

Floor division, '//', is the division operator for the polynomial classes, polynomials are treated like integers in this regard. For Python versions < 3.x the '/' operator maps to '//', as it does for Python, for later versions the '/' will only work for division by scalars. At some point it will be deprecated:

```
>>> p // P([-1, 1])
Polynomial([ 5.,  3.], [-1.,  1.], [-1.,  1.])
```

Remainder:

```
>>> p % P([-1, 1])
Polynomial([ 6.], [-1.,  1.], [-1.,  1.])
```

Divmod:

```
>>> quo, rem = divmod(p, P([-1, 1]))
>>> quo
Polynomial([ 5.,  3.], [-1.,  1.], [-1.,  1.])
>>> rem
Polynomial([ 6.], [-1.,  1.], [-1.,  1.])
```

Evaluation:

```
>>> x = np.arange(5)
>>> p(x)
array([  1.,   6.,  17.,  34.,  57.])
```

```
>>> x = np.arange(6).reshape(3,2)
>>> p(x)
array([[  1.,   6.],
       [ 17.,  34.],
       [ 57.,  86.]])
```

Substitution:

Substitute a polynomial for x and expand the result. Here we substitute p in itself leading to a new polynomial of degree 4 after expansion. If the polynomials are regarded as functions this is composition of functions:

```
>>> p(p)
Polynomial([  6.,  16.,  36.,  36.,  27.], [-1.,  1.], [-1.,  1.])
```

Roots:

```
>>> p.roots()
array([-0.33333333-0.47140452j, -0.33333333+0.47140452j])
```

It isn't always convenient to explicitly use Polynomial instances, so tuples, lists, arrays, and scalars are automatically cast in the arithmetic operations:

```
>>> p + [1, 2, 3]
Polynomial([ 2.,  4.,  6.], [-1.,  1.], [-1.,  1.])
>>> [1, 2, 3] * p
Polynomial([  1.,   4.,  10.,  12.,   9.], [-1.,  1.], [-1.,  1.])
>>> p / 2
Polynomial([ 0.5,  1. ,  1.5], [-1.,  1.], [-1.,  1.])
```

Polynomials that differ in domain, window, or class can't be mixed in arithmetic:

```
>>> from numpy.polynomial import Chebyshev as T
>>> p + P([1], domain=[0,1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 213, in __add__
TypeError: Domains differ
>>> p + P([1], window=[0,1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 215, in __add__
TypeError: Windows differ
>>> p + T([1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 211, in __add__
TypeError: Polynomial types differ
```

But different types can be used for substitution. In fact, this is how conversion of Polynomial classes among themselves is done for type, domain, and window casting:

```
>>> p(T([0, 1]))
Chebyshev([ 2.5,  2. ,  1.5], [-1.,  1.], [-1.,  1.])
```

Which gives the polynomial p in Chebyshev form. This works because $T_1(x) = x$ and substituting x for x doesn't change the original polynomial. However, all the multiplications and divisions will be done using Chebyshev series, hence the type of the result.

Calculus Polynomial instances can be integrated and differentiated.:

```
>>> from numpy.polynomial import Polynomial as P
>>> p = P([2, 6])
>>> p.integ()
Polynomial([ 0.,  2.,  3.], [-1.,  1.], [-1.,  1.])
>>> p.integ(2)
Polynomial([ 0.,  0.,  1.,  1.], [-1.,  1.], [-1.,  1.])
```

The first example integrates *p* once, the second example integrates it twice. By default, the lower bound of the integration and the integration constant are 0, but both can be specified.:

```
>>> p.integ(lbnd=-1)
Polynomial([-1.,  2.,  3.], [-1.,  1.], [-1.,  1.])
>>> p.integ(lbnd=-1, k=1)
Polynomial([ 0.,  2.,  3.], [-1.,  1.], [-1.,  1.])
```

In the first case the lower bound of the integration is set to -1 and the integration constant is 0. In the second the constant of integration is set to 1 as well. Differentiation is simpler since the only option is the number of times the polynomial is differentiated:

```
>>> p = P([1, 2, 3])
>>> p.deriv(1)
Polynomial([ 2.,  6.], [-1.,  1.], [-1.,  1.])
>>> p.deriv(2)
Polynomial([ 6.], [-1.,  1.], [-1.,  1.])
```

Other Polynomial Constructors Constructing polynomials by specifying coefficients is just one way of obtaining a polynomial instance, they may also be created by specifying their roots, by conversion from other polynomial types, and by least squares fits. Fitting is discussed in its own section, the other methods are demonstrated below:

```
>>> from numpy.polynomial import Polynomial as P
>>> from numpy.polynomial import Chebyshev as T
>>> p = P.fromroots([1, 2, 3])
>>> p
Polynomial([ -6.,  11.,  -6.,   1.], [-1.,  1.], [-1.,  1.])
>>> p.convert(kind=T)
Chebyshev([ -9.  ,  11.75,  -3.  ,   0.25], [-1.,  1.], [-1.,  1.])
```

The convert method can also convert domain and window:

```
>>> p.convert(kind=T, domain=[0, 1])
Chebyshev([-2.4375 ,  2.96875, -0.5625 ,  0.03125], [ 0.,  1.], [-1.,  1.])
>>> p.convert(kind=P, domain=[0, 1])
Polynomial([-1.875,  2.875, -1.125,  0.125], [ 0.,  1.], [-1.,  1.])
```

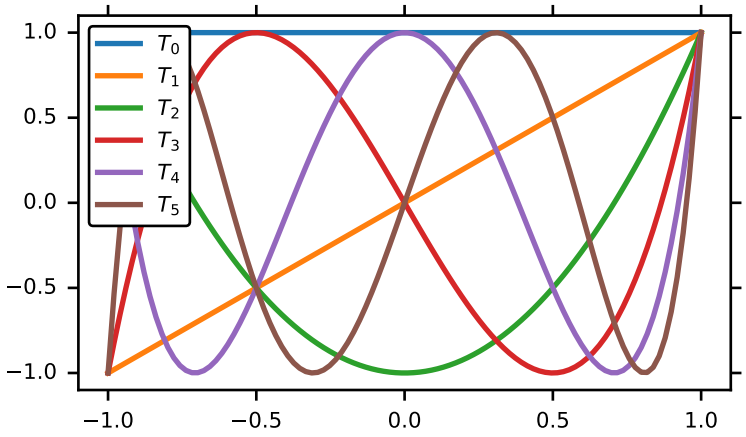
In numpy versions >= 1.7.0 the *basis* and *cast* class methods are also available. The cast method works like the convert method while the basis method returns the basis polynomial of given degree:

```
>>> P.basis(3)
Polynomial([ 0.,  0.,  0.,  1.], [-1.,  1.], [-1.,  1.])
>>> T.cast(p)
Chebyshev([ -9.  ,  11.75,  -3.  ,   0.25], [-1.,  1.], [-1.,  1.])
```

Conversions between types can be useful, but it is *not* recommended for routine use. The loss of numerical precision in passing from a Chebyshev series of degree 50 to a Polynomial series of the same degree can make the results of numerical evaluation essentially random.

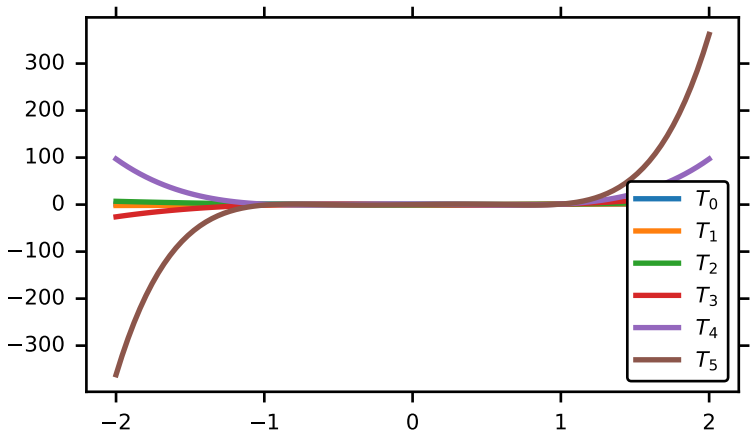
Fitting Fitting is the reason that the *domain* and *window* attributes are part of the convenience classes. To illustrate the problem, the values of the Chebyshev polynomials up to degree 5 are plotted below.

```
>>> import matplotlib.pyplot as plt
>>> from numpy.polynomial import Chebyshev as T
>>> x = np.linspace(-1, 1, 100)
>>> for i in range(6): ax = plt.plot(x, T.basis(i)(x), lw=2, label="$T_%d$"%i)
...
>>> plt.legend(loc="upper left")
<matplotlib.legend.Legend object at 0x3b3ee10>
>>> plt.show()
```



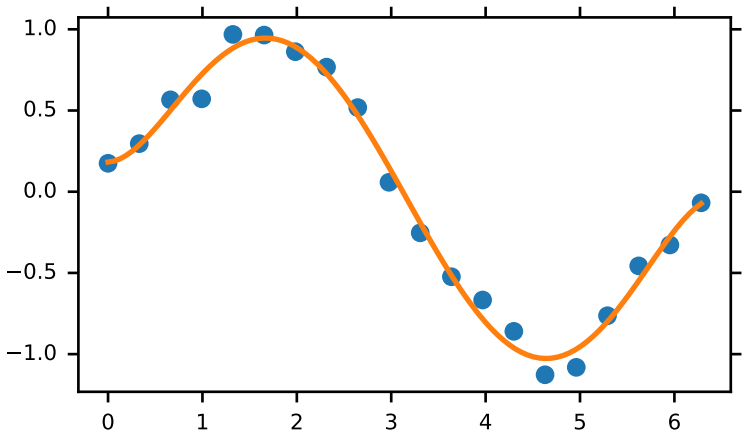
In the range -1 <= *x* <= 1 they are nice, equiripple functions lying between +/- 1. The same plots over the range -2 <= *x* <= 2 look very different:

```
>>> import matplotlib.pyplot as plt
>>> from numpy.polynomial import Chebyshev as T
>>> x = np.linspace(-2, 2, 100)
>>> for i in range(6): ax = plt.plot(x, T.basis(i)(x), lw=2, label="$T_%d$"%i)
...
>>> plt.legend(loc="lower right")
<matplotlib.legend.Legend object at 0x3b3ee10>
>>> plt.show()
```



As can be seen, the "good" parts have shrunk to insignificance. In using Chebyshev polynomials for fitting we want to use the region where *x* is between -1 and 1 and that is what the *window* specifies. However, it is unlikely that the data to be fit has all its data points in that interval, so we use *domain* to specify the interval where the data points lie. When the fit is done, the domain is first mapped to the window by a linear transformation and the usual least squares fit is done using the mapped data points. The window and domain of the fit are part of the returned series and are automatically used when computing values, derivatives, and such. If they aren't specified in the call the fitting routine will use the default window and the smallest domain that holds all the data points. This is illustrated below for a fit to a noisy sine curve.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from numpy.polynomial import Chebyshev as T
>>> np.random.seed(11)
>>> x = np.linspace(0, 2*np.pi, 20)
>>> y = np.sin(x) + np.random.normal(scale=.1, size=x.shape)
>>> p = T.fit(x, y, 5)
>>> plt.plot(x, y, 'o')
[<matplotlib.lines.Line2D object at 0x2136c10>]
>>> xx, yy = p.linspace()
>>> plt.plot(xx, yy, lw=2)
[<matplotlib.lines.Line2D object at 0x1cf2890>]
>>> p.domain
array([ 0.        ,  6.28318531])
>>> p.window
array([-1.,  1.])
>>> plt.show()
```



Polynomial Module (`numpy.polynomial.polynomial`)

New in version 1.4.0.

This module provides a number of objects (mostly functions) useful for dealing with Polynomial series, including a *Polynomial* class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its "parent" sub-package, `numpy.polynomial`).

Polynomial(coef[, domain, window])	A power series class.

Polynomial Class

class `numpy.polynomial.polynomial.`**`Polynomial`**(*coef*, *domain=None*, *window=None*)

A power series class.

The Polynomial class provides the standard Python numerical methods '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the attributes and methods listed in the `ABCPolyBase` documentation.

Parameters

coef : array_like

Polynomial coefficients in order of increasing degree, i.e., `(1, 2, 3)` give `1 + 2*x + 3*x**2`.

domain : (2,) array_like, optional

Domain to use. The interval `[domain[0], domain[1]]` is mapped to the interval `[window[0], window[1]]` by shifting and scaling. The default value is [-1, 1].

window : (2,) array_like, optional

Window, see `domain` for its use. The default value is [-1, 1].

New in version 1.6.0.

Methods

polyval(x, c[, tensor])	Evaluate a polynomial at points x.
polyval2d(x, y, c)	Evaluate a 2-D polynomial at points (x, y).
polyval3d(x, y, z, c)	Evaluate a 3-D polynomial at points (x, y, z).
polygrid2d(x, y, c)	Evaluate a 2-D polynomial on the Cartesian product of x and y.
polygrid3d(x, y, z, c)	Evaluate a 3-D polynomial on the Cartesian product of x, y and z.
polyroots(c)	Compute the roots of a polynomial.
polyfromroots(roots)	Generate a monic polynomial with given roots.

Basics

numpy.polynomial.polynomial.**polyval**(*x*, *c*, *tensor=True*)

Evaluate a polynomial at points x.

If *c* is of length *n + 1*, this function returns the value

$$p(x) = c_0 + c_1 * x + ... + c_n * x^n$$

The parameter *x* is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either *x* or its elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array, then *p(x)* will have the same shape as *x*. If *c* is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor* is true the shape will be c.shape[1:] + x.shape. If *tensor* is false the shape will be c.shape[1:]. Note that scalars have shape (,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

Parameters

x : array_like, compatible object

If *x* is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, *x* or its elements must support addition and multiplication with with themselves and with the elements of *c*.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree n are contained in c[n]. If *c* is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of *c*.

tensor : boolean, optional

If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of *x*. Scalars have dimension 0 for this action. The result is that every column of coefficients in *c* is evaluated for every element of *x*. If False, *x* is broadcast over the columns of *c* for the evaluation. This keyword is useful when *c* is multidimensional. The default value is True.

New in version 1.7.0.

Returns

values : ndarray, compatible object

The shape of the returned array is described above.

See also:

polyval2d, *polygrid2d*, *polyval3d*, *polygrid3d*

Notes

The evaluation uses Horner's method.

Examples

```
>>> from numpy.polynomial.polynomial import polyval
>>> polyval(1, [1,2,3])
6.0
>>> a = np.arange(4).reshape(2,2)
>>> a
array([[0, 1],
       [2, 3]])
>>> polyval(a, [1,2,3])
array([[  1.,   6.],
       [ 17.,  34.]])
>>> coef = np.arange(4).reshape(2,2) # multidimensional coefficients
>>> coef
array([[0, 1],
       [2, 3]])
>>> polyval([1,2], coef, tensor=True)
array([[ 2.,  4.],
       [ 4.,  7.]])
>>> polyval([1,2], coef, tensor=False)
array([ 2.,  7.])
```

numpy.polynomial.polynomial.**polyval2d**(*x*, *y*, *c*)

Evaluate a 2-D polynomial at points (x, y).

This function returns the value

$$p(x,y) = \sum_{i,j} c_{i,j} * x^i * y^j$$

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

Parameters

x, y : array_like, compatible objects

The two dimensional series is evaluated at the points *(x, y)*, where *x* and *y* must have the same shape. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in *c[i,j]*. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional polynomial at points formed with pairs of corresponding values from *x* and *y*.

See also:

polyval, *polygrid2d*, *polyval3d*, *polygrid3d*

Notes

New in version 1.7.0.

numpy.polynomial.polynomial.**polyval3d**(*x*, *y*, *z*, *c*)

Evaluate a 3-D polynomial at points (x, y, z).

This function returns the values:

$$p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * x^i * y^j * z^k$$

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape.

Parameters

x, y, z : array_like, compatible object

The three dimensional series is evaluated at the points *(x, y, z)*, where *x*, *y*, and *z* must have the same shape. If any of *x*, *y*, or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j,k is contained in `c[i,j,k]`. If *c* has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the multidimensional polynomial on points formed with triples of corresponding values from *x*, *y*, and *z*.

See also:

polyval, *polyval2d*, *polygrid2d*, *polygrid3d*

Notes

New in version 1.7.0.

numpy.polynomial.polynomial.**polygrid2d**(*x*, *y*, *c*)

Evaluate a 2-D polynomial on the Cartesian product of x and y.

This function returns the values:

$$p(a,b) = \sum_{i,j} c_{i,j} * a^i * b^j$$

where the points *(a, b)* consist of all pairs formed by taking *a* from *x* and *b* from *y*. The resulting points form a grid with *x* in the first dimension and *y* in the second.

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape + y.shape.

Parameters

x, y : array_like, compatible objects

The two dimensional series is evaluated at the points in the Cartesian product of *x* and *y*. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

See also:

polyval, *polyval2d*, *polyval3d*, *polygrid3d*

Notes

New in version 1.7.0.

numpy.polynomial.polynomial.**polygrid3d**(*x*, *y*, *z*, *c*)

Evaluate a 3-D polynomial on the Cartesian product of x, y and z.

This function returns the values:

$$p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * a^i * b^j * c^k$$

where the points *(a, b, c)* consist of all triples formed by taking *a* from *x*, *b* from *y*, and *c* from *z*. The resulting points form a grid with *x* in the first dimension, *y* in the second, and *z* in the third.

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape + y.shape + z.shape.

Parameters

x, y, z : array_like, compatible objects

The three dimensional series is evaluated at the points in the Cartesian product of *x*, *y*, and *z*. If *x*,'y', or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

See also:

polyval, *polyval2d*, *polygrid2d*, *polyval3d*

Notes

New in version 1.7.0.

numpy.polynomial.polynomial.**polyroots**(*c*)

Compute the roots of a polynomial.

Return the roots (a.k.a. "zeros") of the polynomial

$$p(x) = \sum_i c[i] * x^i.$$

Parameters

c : 1-D array_like

1-D array of polynomial coefficients.

Returns

out : ndarray

Array of the roots of the polynomial. If all the roots are real, then *out* is also real, otherwise it is complex.

See also:

`chebroots`

Notes

The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the complex plane may have large errors due to the numerical instability of the power series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton's method.

Examples

```
>>> import numpy.polynomial.polynomial as poly
>>> poly.polyroots(poly.polyfromroots((-1,0,1)))
array([-1.,  0.,  1.])
>>> poly.polyroots(poly.polyfromroots((-1,0,1))).dtype
dtype('float64')
>>> j = complex(0,1)
>>> poly.polyroots(poly.polyfromroots((-j,0,j)))
array([  0.00000000e+00+0.j,   0.00000000e+00+1.j,   2.77555756e-17-1.j])
```

numpy.polynomial.polynomial.**polyfromroots**(*roots*)

Generate a monic polynomial with given roots.

Return the coefficients of the polynomial

$$p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),$$

where the *r_n* are the roots specified in *roots*. If a zero has multiplicity n, then it must appear in *roots* n times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then *roots* looks something like [2, 2, 2, 3, 3]. The roots can appear in any order.

If the returned coefficients are *c*, then

$$p(x) = c_0 + c_1 * x + ... + x^n$$

The coefficient of the last term is 1 for monic polynomials in this form.

Parameters

roots : array_like

Sequence containing the roots.

Returns

out : ndarray

1-D array of the polynomial's coefficients If all the roots are real, then *out* is also real, otherwise it is complex. (see Examples below).

See also:

`chebfromroots`, `legfromroots`, `lagfromroots`, `hermfromroots`, `hermefromroots`

Notes

The coefficients are determined by multiplying together linear factors of the form *(x - r_i)*, i.e.

$$p(x) = (x - r_0)(x - r_1)...(x - r_n)$$

where `n == len(roots) - 1`; note that this implies that *1* is always returned for a_n.

Examples

```
>>> from numpy.polynomial import polynomial as P
>>> P.polyfromroots((-1,0,1)) # x(x - 1)(x + 1) = x^3 - x
array([ 0., -1.,  0.,  1.])
>>> j = complex(0,1)
>>> P.polyfromroots((-j,j)) # complex returned, though values are real
array([ 1.+0.j,  0.+0.j,  1.+0.j])
```

polyfit(x, y, deg[, rcond, full, w])	Least-squares fit of a polynomial to data.
polyvander(x, deg)	Vandermonde matrix of given degree.
polyvander2d(x, y, deg)	Pseudo-Vandermonde matrix of given degrees.
polyvander3d(x, y, z, deg)	Pseudo-Vandermonde matrix of given degrees.

Fitting

`numpy.polynomial.polynomial.`**`polyfit`**(*x*, *y*, *deg*, *rcond=None*, *full=False*, *w=None*)

Least-squares fit of a polynomial to data.

Return the coefficients of a polynomial of degree *deg* that is the least squares fit to the data values *y* given at points *x*. If *y* is 1-D the returned coefficients will also be 1-D. If *y* is 2-D multiple fits are done, one for each column of *y*, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

$$p(x) = c_0 + c_1 * x + ... + c_n * x^n,$$

where *n* is *deg*.

Parameters

x : array_like, shape (*M*,)

x-coordinates of the *M* sample (data) points `(x[i], y[i])`.

y : array_like, shape (*M*,) or (*M*, *K*)

y-coordinates of the sample points. Several sets of sample points sharing the same x-coordinates can be (independently) fit with one call to *polyfit* by passing in for *y* a 2-D array that contains one data set per column.

deg : int or 1-D array_like

Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For Numpy versions >= 1.11 a list of integers specifying the degrees of the terms to include may be used instead.

rcond : float, optional

Relative condition number of the fit. Singular values smaller than *rcond*, relative to the largest singular value, will be ignored. The default value is `len(x)*eps`, where *eps* is the relative precision of the platform's float type, about 2e-16 in most cases.

full : bool, optional

Switch determining the nature of the return value. When `False` (the default) just the coefficients are returned; when `True`, diagnostic information from the singular value decomposition (used to solve the fit's matrix equation) is also returned.

w : array_like, shape (*M*,), optional

Weights. If not None, the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

New in version 1.5.0.

Returns

coef : ndarray, shape (*deg* + 1,) or (*deg* + 1, *K*)

Polynomial coefficients ordered from low to high. If *y* was 2-D, the coefficients in column *k* of *coef* represent the polynomial fit to the data in *y*'s *k*-th column.

[residuals, rank, singular_values, rcond] : list

These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

Raises

RankWarning

Raised if the matrix in the least-squares fit is rank deficient. The warning is only raised if *full* == False. The warnings can be turned off by:

```
>>> import warnings
>>> warnings.simplefilter('ignore', RankWarning)
```

See also:

`chebfit`, `legfit`, `lagfit`, `hermfit`, `hermefit`

polyval
Evaluates a polynomial.

polyvander
Vandermonde matrix for powers.

linalg.lstsq
Computes a least-squares fit from the matrix.

scipy.interpolate.UnivariateSpline
Computes spline fits.

Notes

The solution is the coefficients of the polynomial *p* that minimizes the sum of the weighted squared errors

$$E = \sum_j w_j^2 * |y_j - p(x_j)|^2,$$

where the w_j are the weights. This problem is solved by setting up the (typically) over-determined matrix equation:

$$V(x) * c = w * y,$$

where *V* is the weighted pseudo Vandermonde matrix of *x*, *c* are the coefficients to be solved for, *w* are the weights, and *y* are the observed values. This equation is then solved using the singular value decomposition of *V*.

If some of the singular values of *V* are so small that they are neglected (and *full* == `False`), a *RankWarning* will be raised. This means that the coefficient values may be poorly determined. Fitting to a lower order polynomial will usually get rid of the warning (but may not be what you want, of course; if you have independent reason(s) for choosing the degree which isn't working, you may have to: a) reconsider those reasons, and/or b) reconsider the quality of your data). The *rcond* parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Polynomial fits using double precision tend to "fail" at about (polynomial) degree 20. Fits using Chebyshev or Legendre series are generally better conditioned, but much can still depend on the distribution of the sample points and the smoothness of the data. If the quality of the fit is inadequate, splines may be a good alternative.

Examples

```
>>> from numpy.polynomial import polynomial as P
>>> x = np.linspace(-1,1,51) # x "data": [-1, -0.96, ..., 0.96, 1]
>>> y = x**3 - x + np.random.randn(len(x)) # x^3 - x + N(0,1) "noise"
>>> c, stats = P.polyfit(x,y,3,full=True)
>>> c # c[0], c[2] should be approx. 0, c[1] approx. -1, c[3] approx. 1
array([ 0.01909725, -1.30598256, -0.00577963,  1.02644286])
>>> stats # note the large SSR, explaining the rather poor results
[array([ 38.06116253]), 4, array([ 1.38446749,  1.32119158,  0.50443316,
0.28853036]), 1.1324274851176597e-014]
```

Same thing without the added noise

```
>>> y = x**3 - x
>>> c, stats = P.polyfit(x,y,3,full=True)
>>> c # c[0], c[2] should be "very close to 0", c[1] ~= -1, c[3] ~= 1
array([ -1.73362882e-17,  -1.00000000e+00,  -2.67471909e-16,
         1.00000000e+00])
>>> stats # note the minuscule SSR
[array([  7.46346754e-31]), 4, array([ 1.38446749,  1.32119158,
0.50443316,  0.28853036]), 1.1324274851176597e-014]
```

numpy.polynomial.polynomial.**polyvander**(*x*, *deg*)
Vandermonde matrix of given degree.

Returns the Vandermonde matrix of degree *deg* and sample points *x*. The Vandermonde matrix is defined by

$$V[..., i] = x^i,$$

where *0 <= i <= deg*. The leading indices of *V* index the elements of *x* and the last index is the power of *x*.

If *c* is a 1-D array of coefficients of length *n + 1* and *V* is the matrix `V = polyvander(x, n)`, then `np.dot(V, c)` and `polyval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of polynomials of the same degree and sample points.

Parameters

x : array_like

Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If *x* is scalar it is converted to a 1-D array.

deg : int

Degree of the resulting matrix.

Returns

vander : ndarray.

The Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where the last index is the power of *x*. The dtype will be the same as the converted *x*.

See also:

polyvander2d, *polyvander3d*

numpy.polynomial.polynomial.**polyvander2d**(*x*, *y*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y)*. The pseudo-Vandermonde matrix is defined by

$$V[..., deg[1] * i + j] = x^i * y^j,$$

where *0 <= i <= deg[0]* and *0 <= j <= deg[1]*. The leading indices of *V* index the points *(x, y)* and the last index encodes the powers of *x* and *y*.

If `V = polyvander2d(x, y, [xdeg, ydeg])`, then the columns of *V* correspond to the elements of a 2-D coefficient array *c* of shape (xdeg + 1, ydeg + 1) in the order

$$c_{00}, c_{01}, c_{02}..., c_{10}, c_{11}, c_{12}...$$

and `np.dot(V, c.flat)` and `polyval2d(x, y, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D polynomials of the same degrees and sample points.

Parameters

x, y : array_like

Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

deg : list of ints

List of maximum degrees of the form [x_deg, y_deg].

Returns

vander2d : ndarray

The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0] + 1) * (deg([1] + 1)$. The dtype will be the same as the converted *x* and *y*.

See also:

polyvander, polyvander3d., *polyval3d*

numpy.polynomial.polynomial.**polyvander3d**(*x*, *y*, *z*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y, z)*. If *l, m, n* are the given degrees in *x, y, z*, then The pseudo-Vandermonde matrix is defined by

$$V[..., (m+1)(n+1)i + (n+1)j + k] = x^i * y^j * z^k,$$

where *0 <= i <= l*, *0 <= j <= m*, and *0 <= j <= n*. The leading indices of *V* index the points *(x, y, z)* and the last index encodes the powers of *x*, *y*, and *z*.

If `V = polyvander3d(x, y, z, [xdeg, ydeg, zdeg])`, then the columns of *V* correspond to the elements of a 3-D coefficient array *c* of shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order

$$c_{000}, c_{001}, c_{002}, ..., c_{010}, c_{011}, c_{012}, ...$$

and `np.dot(V, c.flat)` and `polyval3d(x, y, z, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D polynomials of the same degrees and sample points.

Parameters

x, y, z : array_like

Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

deg : list of ints

List of maximum degrees of the form [x_deg, y_deg, z_deg].

Returns

vander3d : ndarray

The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0]+1)*(deg([1]+1)*(deg[2]+1)$. The dtype will be the same as the converted *x*, *y*, and *z*.

See also:

polyvander, polyvander3d., *polyval3d*

Notes

New in version 1.7.0.

polyder(c[, m, scl, axis])	Differentiate a polynomial.
polyint(c[, m, k, lbnd, scl, axis])	Integrate a polynomial.

Calculus

numpy.polynomial.polynomial.**polyder**(*c*, *m=1*, *scl=1*, *axis=0*)

Differentiate a polynomial.

Returns the polynomial coefficients *c* differentiated *m* times along *axis*. At each iteration the result is multiplied by *scl* (the scaling factor is for use in a linear change of variable). The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the polynomial `1 + 2*x + 3*x**2` while

[[1,2],[1,2]] represents `1 + 1*x + 2*y + 2*x*y` if axis=0 is `x` and axis=1 is `y`.

Parameters

c : array_like

Array of polynomial coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

m : int, optional

Number of derivatives taken, must be non-negative. (Default: 1)

scl : scalar, optional

Each differentiation is multiplied by *scl*. The end result is multiplication by `scl**m`. This is for use in a linear change of variable. (Default: 1)

axis : int, optional

Axis over which the derivative is taken. (Default: 0).

New in version 1.7.0.

Returns

der : ndarray

Polynomial coefficients of the derivative.

See also:

polyint

Examples

```
>>> from numpy.polynomial import polynomial as P
>>> c = (1,2,3,4) # 1 + 2x + 3x**2 + 4x**3
>>> P.polyder(c) # (d/dx)(c) = 2 + 6x + 12x**2
array([  2.,   6.,  12.])
>>> P.polyder(c,3) # (d**3/dx**3)(c) = 24
array([ 24.])
>>> P.polyder(c,scl=-1) # (d/d(-x))(c) = -2 - 6x - 12x**2
array([ -2.,  -6., -12.])
>>> P.polyder(c,2,-1) # (d**2/d(-x)**2)(c) = 6 + 24x
array([  6.,  24.])
```

numpy.polynomial.polynomial.**polyint**(*c*, *m=1*, *k=[]*, *lbnd=0*, *scl=1*, *axis=0*)

Integrate a polynomial.

Returns the polynomial coefficients *c* integrated *m* times from *lbnd* along *axis*. At each iteration the resulting series is **multiplied** by *scl* and an integration constant, *k*, is added. The scaling factor is for use in a linear change of variable. ("Buyer beware": note that, depending on what one is doing, one may want *scl* to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument *c* is an array of coefficients, from low to high degree along each axis, e.g., [1,2,3] represents the polynomial `1 + 2*x + 3*x**2` while [[1,2],[1,2]] represents `1 + 1*x + 2*y + 2*x*y` if axis=0 is `x` and axis=1 is `y`.

Parameters

c : array_like

1-D array of polynomial coefficients, ordered from low to high.

m : int, optional

Order of integration, must be positive. (Default: 1)

k : {[], list, scalar}, optional

Integration constant(s). The value of the first integral at zero is the first value in the list, the value of the second integral at zero is the second value, etc. If `k == []` (the default), all constants are set to zero. If `m == 1`, a single scalar can be given instead of a list.

lbnd : scalar, optional

The lower bound of the integral. (Default: 0)

scl : scalar, optional

Following each integration the result is *multiplied* by *scl* before the integration constant is added. (Default: 1)

axis : int, optional

Axis over which the integral is taken. (Default: 0).

New in version 1.7.0.

Returns

S : ndarray

Coefficient array of the integral.

Raises

ValueError

If `m < 1`, `len(k) > m`.

See also:

polyder

Notes

Note that the result of each integration is *multiplied* by *scl*. Why is this important to note? Say one is making a linear change of variable $u = ax + b$ in an integral relative to x. Then .. math::dx = du/a, so one will need to set *scl* equal to $1/a$ - perhaps not what one would have first thought.

Examples

```
>>> from numpy.polynomial import polynomial as P
>>> c = (1,2,3)
>>> P.polyint(c) # should return array([0, 1, 1, 1])
array([ 0.,  1.,  1.,  1.])
>>> P.polyint(c,3) # should return array([0, 0, 0, 1/6, 1/12, 1/20])
array([ 0.        ,  0.        ,  0.        ,  0.16666667,  0.08333333,
        0.05      ])
>>> P.polyint(c,k=3) # should return array([3, 1, 1, 1])
array([ 3.,  1.,  1.,  1.])
>>> P.polyint(c,lbnd=-2) # should return array([6, 1, 1, 1])
array([ 6.,  1.,  1.,  1.])
>>> P.polyint(c,scl=-2) # should return array([0, -2, -2, -2])
array([ 0., -2., -2., -2.])
```

polyadd(c1, c2)	Add one polynomial to another.
polysub(c1, c2)	Subtract one polynomial from another.
polymul(c1, c2)	Multiply one polynomial by another.
polymulx(c)	Multiply a polynomial by x.

Continued on next page

Table 3.124 – continued from previous page

polydiv(c1, c2)	Divide one polynomial by another.
polypow(c, pow[, maxpower])	Raise a polynomial to a power.

Algebra

numpy.polynomial.polynomial.**polyadd**(*c1*, *c2*)

Add one polynomial to another.

Returns the sum of two polynomials *c1* + *c2*. The arguments are sequences of coefficients from lowest order term to highest, i.e., [1,2,3] represents the polynomial `1 + 2*x + 3*x**2`.

Parameters

c1, c2 : array_like

1-D arrays of polynomial coefficients ordered from low to high.

Returns

out : ndarray

The coefficient array representing their sum.

See also:

polysub, *polymul*, *polydiv*, *polypow*

Examples

```
>>> from numpy.polynomial import polynomial as P
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> sum = P.polyadd(c1,c2); sum
array([ 4.,  4.,  4.])
>>> P.polyval(2, sum) # 4 + 4(2) + 4(2**2)
28.0
```

numpy.polynomial.polynomial.**polysub**(*c1*, *c2*)

Subtract one polynomial from another.

Returns the difference of two polynomials *c1* - *c2*. The arguments are sequences of coefficients from lowest order term to highest, i.e., [1,2,3] represents the polynomial `1 + 2*x + 3*x**2`.

Parameters

c1, c2 : array_like

1-D arrays of polynomial coefficients ordered from low to high.

Returns

out : ndarray

Of coefficients representing their difference.

See also:

polyadd, *polymul*, *polydiv*, *polypow*

Examples

```
>>> from numpy.polynomial import polynomial as P
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> P.polysub(c1,c2)
array([-2.,  0.,  2.])
```

```
>>> P.polysub(c2,c1) # -P.polysub(c1,c2)
array([ 2.,  0., -2.])
```

numpy.polynomial.polynomial.**polymul**(*c1*, *c2*)

Multiply one polynomial by another.

Returns the product of two polynomials *c1* * *c2*. The arguments are sequences of coefficients, from lowest order term to highest, e.g., [1,2,3] represents the polynomial `1 + 2*x + 3*x**2`.

Parameters

c1, c2 : array_like

1-D arrays of coefficients representing a polynomial, relative to the "standard" basis, and ordered from lowest order term to highest.

Returns

out : ndarray

Of the coefficients of their product.

See also:

polyadd, *polysub*, *polydiv*, *polypow*

Examples

```
>>> from numpy.polynomial import polynomial as P
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> P.polymul(c1,c2)
array([  3.,   8.,  14.,   8.,   3.])
```

numpy.polynomial.polynomial.**polymulx**(*c*)

Multiply a polynomial by x.

Multiply the polynomial *c* by x, where x is the independent variable.

Parameters

c : array_like

1-D array of polynomial coefficients ordered from low to high.

Returns

out : ndarray

Array representing the result of the multiplication.

Notes

New in version 1.5.0.

numpy.polynomial.polynomial.**polydiv**(*c1*, *c2*)

Divide one polynomial by another.

Returns the quotient-with-remainder of two polynomials *c1* / *c2*. The arguments are sequences of coefficients, from lowest order term to highest, e.g., [1,2,3] represents `1 + 2*x + 3*x**2`.

Parameters

c1, c2 : array_like

1-D arrays of polynomial coefficients ordered from low to high.

Returns

[quo, rem] : ndarrays

Of coefficient series representing the quotient and remainder.

See also:

polyadd, *polysub*, *polymul*, *polypow*

Examples

```
>>> from numpy.polynomial import polynomial as P
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> P.polydiv(c1,c2)
(array([ 3.]), array([-8., -4.]))
>>> P.polydiv(c2,c1)
(array([ 0.33333333]), array([ 2.66666667,  1.33333333]))
```

numpy.polynomial.polynomial.**polypow**(*c*, *pow*, *maxpower=None*)

Raise a polynomial to a power.

Returns the polynomial *c* raised to the power *pow*. The argument *c* is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series `1 + 2*x + 3*x**2.`

Parameters

c : array_like

1-D array of array of series coefficients ordered from low to high degree.

pow : integer

Power to which the series will be raised

maxpower : integer, optional

Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

Returns

coef : ndarray

Power series of power.

See also:

polyadd, *polysub*, *polymul*, *polydiv*

polycompanion(c)	Return the companion matrix of c.
polydomain	
polyzero	
polyone	
polyx	
polyline(off, scl)	Returns an array representing a linear polynomial.

Miscellaneous

numpy.polynomial.polynomial.**polycompanion**(*c*)

Return the companion matrix of c.

The companion matrix for power series cannot be made symmetric by scaling the basis, so this function differs from those for the orthogonal polynomials.

Parameters

c : array_like

1-D array of polynomial coefficients ordered from low to high degree.

Returns

mat : ndarray

Companion matrix of dimensions (deg, deg).

Notes

New in version 1.7.0.

numpy.polynomial.polynomial.**polydomain** = **array([-1, 1])**

numpy.polynomial.polynomial.**polyzero** = **array([0])**

numpy.polynomial.polynomial.**polyone** = **array([1])**

numpy.polynomial.polynomial.**polyx** = **array([0, 1])**

numpy.polynomial.polynomial.**polyline**(*off*, *scl*)

Returns an array representing a linear polynomial.

Parameters

off, scl : scalars

The "y-intercept" and "slope" of the line, respectively.

Returns

y : ndarray

This module's representation of the linear polynomial `off + scl*x`.

See also:

`chebline`

Examples

```
>>> from numpy.polynomial import polynomial as P
>>> P.polyline(1,-1)
array([ 1, -1])
>>> P.polyval(1, P.polyline(1,-1)) # should be 0
0.0
```

Chebyshev Module (`numpy.polynomial.chebyshev`)

New in version 1.4.0.

This module provides a number of objects (mostly functions) useful for dealing with Chebyshev series, including a *Chebyshev* class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its "parent" sub-package, `numpy.polynomial`).

Chebyshev(coef[, domain, window])	A Chebyshev series class.

Chebyshev Class

class numpy.polynomial.chebyshev.**Chebyshev**(*coef*, *domain=None*, *window=None*)

A Chebyshev series class.

The Chebyshev class provides the standard Python numerical methods '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the methods listed below.

Parameters

coef : array_like

Chebyshev coefficients in order of increasing degree, i.e., `(1, 2, 3)` gives `1*T_0(x) + 2*T_1(x) + 3*T_2(x)`.

domain : (2,) array_like, optional

Domain to use. The interval `[domain[0], domain[1]]` is mapped to the interval `[window[0], window[1]]` by shifting and scaling. The default value is [-1, 1].

window : (2,) array_like, optional

Window, see `domain` for its use. The default value is [-1, 1].

New in version 1.6.0.

Methods

chebval(x, c[, tensor])	Evaluate a Chebyshev series at points x.
chebval2d(x, y, c)	Evaluate a 2-D Chebyshev series at points (x, y).
chebval3d(x, y, z, c)	Evaluate a 3-D Chebyshev series at points (x, y, z).
chebgrid2d(x, y, c)	Evaluate a 2-D Chebyshev series on the Cartesian product of x and y.
chebgrid3d(x, y, z, c)	Evaluate a 3-D Chebyshev series on the Cartesian product of x, y, and z.
chebroots(c)	Compute the roots of a Chebyshev series.
chebfromroots(roots)	Generate a Chebyshev series with given roots.

Basics

`numpy.polynomial.chebyshev.`**`chebval`**(*x*, *c*, *tensor=True*)

Evaluate a Chebyshev series at points x.

If *c* is of length *n + 1*, this function returns the value:

$$p(x) = c_0 * T_0(x) + c_1 * T_1(x) + ... + c_n * T_n(x)$$

The parameter *x* is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either *x* or its elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array, then *p(x)* will have the same shape as *x*. If *c* is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor* is true the shape will be c.shape[1:] + x.shape. If *tensor* is false the shape will be c.shape[1:]. Note that scalars have shape (,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

Parameters

x : array_like, compatible object

If *x* is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, *x* or its elements must support addition and multiplication with with themselves and with the elements of *c*.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree n are contained in c[n]. If *c* is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of *c*.

tensor : boolean, optional

If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of *x*. Scalars have dimension 0 for this action. The result is that every column of coefficients in *c* is evaluated for every element of *x*. If False, *x* is broadcast over the columns of *c* for the evaluation. This keyword is useful when *c* is multidimensional. The default value is True.

New in version 1.7.0.

Returns

values : ndarray, algebra_like

The shape of the return value is described above.

See also:

chebval2d, *chebgrid2d*, *chebval3d*, *chebgrid3d*

Notes

The evaluation uses Clenshaw recursion, aka synthetic division.

numpy.polynomial.chebyshev.**chebval2d**(*x*, *y*, *c*)

Evaluate a 2-D Chebyshev series at points (x, y).

This function returns the values:

$$p(x,y) = \sum_{i,j} c_{i,j} * T_i(x) * T_j(y)$$

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array a one is implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

Parameters

x, y : array_like, compatible objects

The two dimensional series is evaluated at the points *(x, y)*, where *x* and *y* must have the same shape. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in `c[i,j]`. If *c* has dimension greater than 2 the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional Chebyshev series at points formed from pairs of corresponding values from *x* and *y*.

See also:

chebval, *chebgrid2d*, *chebval3d*, *chebgrid3d*

Notes

numpy.polynomial.chebyshev.**chebval3d**(*x*, *y*, *z*, *c*)

Evaluate a 3-D Chebyshev series at points (x, y, z).

This function returns the values:

$$p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * T_i(x) * T_j(y) * T_k(z)$$

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape.

Parameters

x, y, z : array_like, compatible object

The three dimensional series is evaluated at the points *(x, y, z)*, where *x*, *y*, and *z* must have the same shape. If any of *x*, *y*, or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j,k is contained in `c[i,j,k]`. If *c* has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the multidimensional polynomial on points formed with triples of corresponding values from *x*, *y*, and *z*.

See also:

chebval, *chebval2d*, *chebgrid2d*, *chebgrid3d*

Notes

numpy.polynomial.chebyshev.**chebgrid2d**(*x*, *y*, *c*)

Evaluate a 2-D Chebyshev series on the Cartesian product of x and y.

This function returns the values:

$$p(a,b) = \sum_{i,j} c_{i,j} * T_i(a) * T_j(b),$$

where the points *(a, b)* consist of all pairs formed by taking *a* from *x* and *b* from *y*. The resulting points form a grid with *x* in the first dimension and *y* in the second.

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape + y.shape.

Parameters

x, y : array_like, compatible objects

The two dimensional series is evaluated at the points in the Cartesian product of *x* and *y*. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in *c[i,j]*. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional Chebyshev series at points in the Cartesian product of *x* and *y*.

See also:

chebval, *chebval2d*, *chebval3d*, *chebgrid3d*

Notes

numpy.polynomial.chebyshev.**chebgrid3d**(*x*, *y*, *z*, *c*)

Evaluate a 3-D Chebyshev series on the Cartesian product of x, y, and z.

This function returns the values:

$$p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * T_i(a) * T_j(b) * T_k(c)$$

where the points *(a, b, c)* consist of all triples formed by taking *a* from *x*, *b* from *y*, and *c* from *z*. The resulting points form a grid with *x* in the first dimension, *y* in the second, and *z* in the third.

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape + y.shape + z.shape.

Parameters

x, y, z : array_like, compatible objects

The three dimensional series is evaluated at the points in the Cartesian product of *x*, *y*, and *z*. If *x*,'y', or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

See also:

chebval, *chebval2d*, *chebgrid2d*, *chebval3d*

Notes

numpy.polynomial.chebyshev.**chebroots**(*c*)

Compute the roots of a Chebyshev series.

Return the roots (a.k.a. "zeros") of the polynomial

$$p(x) = \sum_i c[i] * T_i(x).$$

Parameters

c : 1-D array_like

1-D array of coefficients.

Returns

out : ndarray

Array of the roots of the series. If all the roots are real, then *out* is also real, otherwise it is complex.

See also:

`polyroots`, `legroots`, `lagroots`, `hermroots`, `hermeroots`

Notes

The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the complex plane may have large errors due to the numerical instability of the series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton's method.

The Chebyshev series basis polynomials aren't powers of *x* so the results of this function may seem unintuitive.

Examples

```
>>> import numpy.polynomial.chebyshev as cheb
>>> cheb.chebroots((-1, 1,-1, 1)) # T3 - T2 + T1 - T0 has real roots
array([ -5.00000000e-01,   2.60860684e-17,   1.00000000e+00])
```

numpy.polynomial.chebyshev.**chebfromroots**(*roots*)

Generate a Chebyshev series with given roots.

The function returns the coefficients of the polynomial

$$p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),$$

in Chebyshev form, where the *r_n* are the roots specified in *roots*. If a zero has multiplicity n, then it must appear in *roots* n times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then *roots* looks something like [2, 2, 2, 3, 3]. The roots can appear in any order.

If the returned coefficients are *c*, then

$$p(x) = c_0 + c_1 * T_1(x) + ... + c_n * T_n(x)$$

The coefficient of the last term is not generally 1 for monic polynomials in Chebyshev form.

Parameters

roots : array_like

Sequence containing the roots.

Returns

out : ndarray

1-D array of coefficients. If all roots are real then *out* is a real array, if some of the roots are complex, then *out* is complex even if all the coefficients in the result are real (see Examples below).

See also:

`polyfromroots`, `legfromroots`, `lagfromroots`, `hermfromroots`, `hermefromroots`.

Examples

```
>>> import numpy.polynomial.chebyshev as C
>>> C.chebfromroots((-1,0,1)) # x^3 - x relative to the standard basis
array([ 0.  , -0.25,  0.  ,  0.25])
>>> j = complex(0,1)
>>> C.chebfromroots((-j,j)) # x^2 + 1 relative to the standard basis
array([ 1.5+0.j,  0.0+0.j,  0.5+0.j])
```

chebfit(x, y, deg[, rcond, full, w])	Least squares fit of Chebyshev series to data.
chebvander(x, deg)	Pseudo-Vandermonde matrix of given degree.
chebvander2d(x, y, deg)	Pseudo-Vandermonde matrix of given degrees.
chebvander3d(x, y, z, deg)	Pseudo-Vandermonde matrix of given degrees.

Fitting

`numpy.polynomial.chebyshev.`**`chebfit`**(*x*, *y*, *deg*, *rcond=None*, *full=False*, *w=None*)

Least squares fit of Chebyshev series to data.

Return the coefficients of a Legendre series of degree *deg* that is the least squares fit to the data values *y* given at points *x*. If *y* is 1-D the returned coefficients will also be 1-D. If *y* is 2-D multiple fits are done, one for each column of *y*, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

$$p(x) = c_0 + c_1 * T_1(x) + ... + c_n * T_n(x),$$

where *n* is *deg*.

Parameters

x : array_like, shape (M,)

x-coordinates of the M sample points `(x[i], y[i])`.

y : array_like, shape (M,) or (M, K)

y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

deg : int or 1-D array_like

Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For Numpy versions >= 1.11 a list of integers specifying the degrees of the terms to include may be used instead.

rcond : float, optional

Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

full : bool, optional

Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

w : array_like, shape (*M*,), optional

Weights. If not None, the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

New in version 1.5.0.

Returns

coef : ndarray, shape (M,) or (M, K)

Chebyshev coefficients ordered from low to high. If *y* was 2-D, the coefficients for the data in column k of *y* are in column *k*.

[residuals, rank, singular_values, rcond] : list

These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

Warns

RankWarning

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if *full* = False. The warnings can be turned off by

```
>>> import warnings
>>> warnings.simplefilter('ignore', RankWarning)
```

See also:

`polyfit`, `legfit`, `lagfit`, `hermfit`, `hermefit`

chebval
Evaluates a Chebyshev series.

chebvander
Vandermonde matrix of Chebyshev series.

chebweight
Chebyshev weight function.

`linalg.lstsq`
Computes a least-squares fit from the matrix.

`scipy.interpolate.UnivariateSpline`
Computes spline fits.

Notes

The solution is the coefficients of the Chebyshev series *p* that minimizes the sum of the weighted squared errors

$$E = \sum_j w_j^2 * |y_j - p(x_j)|^2,$$

where w_j are the weights. This problem is solved by setting up as the (typically) overdetermined matrix equation

$$V(x) * c = w * y,$$

where *V* is the weighted pseudo Vandermonde matrix of *x*, *c* are the coefficients to be solved for, *w* are the weights, and *y* are the observed values. This equation is then solved using the singular value decomposition of *V*.

If some of the singular values of *V* are so small that they are neglected, then a *RankWarning* will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The *rcond* parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using Chebyshev series are usually better conditioned than fits using power series, but much can depend on the distribution of the sample points and the smoothness of the data. If the quality of the fit is inadequate splines may be a good alternative.

References

[R60]

numpy.polynomial.chebyshev.**chebvander**(*x*, *deg*)

Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree *deg* and sample points *x*. The pseudo-Vandermonde matrix is defined by

$$V[..., i] = T_i(x),$$

where *0 <= i <= deg*. The leading indices of *V* index the elements of *x* and the last index is the degree of the Chebyshev polynomial.

If *c* is a 1-D array of coefficients of length *n + 1* and *V* is the matrix `V = chebvander(x, n)`, then `np.dot(V, c)` and `chebval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of Chebyshev series of the same degree and sample points.

Parameters

x : array_like

Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If *x* is scalar it is converted to a 1-D array.

deg : int

Degree of the resulting matrix.

Returns

vander : ndarray

The pseudo Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where The last index is the degree of the corresponding Chebyshev polynomial. The dtype will be the same as the converted *x*.

numpy.polynomial.chebyshev.**chebvander2d**(*x*, *y*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y)*. The pseudo-Vandermonde matrix is defined by

$$V[..., deg[1] * i + j] = T_i(x) * T_j(y),$$

where *0 <= i <= deg[0]* and *0 <= j <= deg[1]*. The leading indices of *V* index the points *(x, y)* and the last index encodes the degrees of the Chebyshev polynomials.

If `V = chebvander2d(x, y, [xdeg, ydeg])`, then the columns of *V* correspond to the elements of a 2-D coefficient array *c* of shape (xdeg + 1, ydeg + 1) in the order

$$c_{00}, c_{01}, c_{02}..., c_{10}, c_{11}, c_{12}...$$

and `np.dot(V, c.flat)` and `chebval2d(x, y, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D Chebyshev series of the same degrees and sample points.

Parameters

x, y : array_like

Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

deg : list of ints

List of maximum degrees of the form [x_deg, y_deg].

Returns

vander2d : ndarray

The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0]+1)*(deg([1]+1)$. The dtype will be the same as the converted *x* and *y*.

See also:

chebvander, chebvander3d., *chebval3d*

Notes

numpy.polynomial.chebyshev.**chebvander3d**(*x*, *y*, *z*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y, z)*. If *l, m, n* are the given degrees in *x, y, z*, then The pseudo-Vandermonde matrix is defined by

$$V[..., (m+1)(n+1)i + (n+1)j + k] = T_i(x) * T_j(y) * T_k(z),$$

where *0 <= i <= l*, *0 <= j <= m*, and *0 <= j <= n*. The leading indices of *V* index the points *(x, y, z)* and the last index encodes the degrees of the Chebyshev polynomials.

If `V = chebvander3d(x, y, z, [xdeg, ydeg, zdeg])`, then the columns of *V* correspond to the elements of a 3-D coefficient array *c* of shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order

$$c_{000}, c_{001}, c_{002}, ..., c_{010}, c_{011}, c_{012}, ...$$

and `np.dot(V, c.flat)` and `chebval3d(x, y, z, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D Chebyshev series of the same degrees and sample points.

Parameters

x, y, z : array_like

Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

deg : list of ints

List of maximum degrees of the form [x_deg, y_deg, z_deg].

Returns

vander3d : ndarray

The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0]+1)*(deg([1]+1)*(deg[2]+1)$. The dtype will be the same as the converted *x*, *y*, and *z*.

See also:

chebvander, chebvander3d., *chebval3d*

Notes

chebder(c[, m, scl, axis])	Differentiate a Chebyshev series.
chebint(c[, m, k, lbnd, scl, axis])	Integrate a Chebyshev series.

Calculus

numpy.polynomial.chebyshev.**chebder**(*c*, *m=1*, *scl=1*, *axis=0*)

Differentiate a Chebyshev series.

Returns the Chebyshev series coefficients *c* differentiated *m* times along *axis*. At each iteration the result is multiplied by *scl* (the scaling factor is for use in a linear change of variable). The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `1*T_0 + 2*T_1 + 3*T_2` while [[1,2],[1,2]] represents `1*T_0(x)*T_0(y) + 1*T_1(x)*T_0(y) + 2*T_0(x)*T_1(y) + 2*T_1(x)*T_1(y)` if axis=0 is `x` and axis=1 is `y`.

Parameters

c : array_like

Array of Chebyshev series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

m : int, optional

Number of derivatives taken, must be non-negative. (Default: 1)

scl : scalar, optional

Each differentiation is multiplied by *scl*. The end result is multiplication by `scl**m`. This is for use in a linear change of variable. (Default: 1)

axis : int, optional

Axis over which the derivative is taken. (Default: 0).

New in version 1.7.0.

Returns

der : ndarray

Chebyshev series of the derivative.

See also:

chebint

Notes

In general, the result of differentiating a C-series needs to be "reprojected" onto the C-series basis set. Thus, typically, the result of this function is "unintuitive," albeit correct; see Examples section below.

Examples

```
>>> from numpy.polynomial import chebyshev as C
>>> c = (1,2,3,4)
>>> C.chebder(c)
array([ 14.,  12.,  24.])
>>> C.chebder(c,3)
array([ 96.])
>>> C.chebder(c,scl=-1)
array([-14., -12., -24.])
>>> C.chebder(c,2,-1)
array([ 12.,  96.])
```

numpy.polynomial.chebyshev.**chebint**(*c*, *m=1*, *k=[]*, *lbnd=0*, *scl=1*, *axis=0*)

Integrate a Chebyshev series.

Returns the Chebyshev series coefficients *c* integrated *m* times from *lbnd* along *axis*. At each iteration the resulting series is **multiplied** by *scl* and an integration constant, *k*, is added. The scaling factor is for use in a linear change of variable. ("Buyer beware": note that, depending on what one is doing, one may want *scl* to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `T_0 + 2*T_1 + 3*T_2` while [[1,2],[1,2]] represents `1*T_0(x)*T_0(y) + 1*T_1(x)*T_0(y) + 2*T_0(x)*T_1(y) + 2*T_1(x)*T_1(y)` if axis=0 is `x` and axis=1 is `y`.

Parameters

c : array_like

Array of Chebyshev series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

m : int, optional

Order of integration, must be positive. (Default: 1)

k : {[], list, scalar}, optional

Integration constant(s). The value of the first integral at zero is the first value in the list, the value of the second integral at zero is the second value, etc. If `k == []` (the default), all constants are set to zero. If `m == 1`, a single scalar can be given instead of a list.

lbnd : scalar, optional

The lower bound of the integral. (Default: 0)

scl : scalar, optional

Following each integration the result is *multiplied* by *scl* before the integration constant is added. (Default: 1)

axis : int, optional

Axis over which the integral is taken. (Default: 0).

New in version 1.7.0.

Returns

S : ndarray

C-series coefficients of the integral.

Raises

ValueError

If `m < 1`, `len(k) > m`, `np.isscalar(lbnd) == False`, or `np.isscalar(scl) == False`.

See also:

chebder

Notes

Note that the result of each integration is *multiplied* by *scl*. Why is this important to note? Say one is making a linear change of variable $u = ax + b$ in an integral relative to x. Then .. math::dx = du/a, so one will need to set *scl* equal to $1/a$- perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be "reprojected" onto the C-series basis set. Thus, typically, the result of this function is "unintuitive," albeit correct; see Examples section below.

Examples

```
>>> from numpy.polynomial import chebyshev as C
>>> c = (1,2,3)
>>> C.chebint(c)
array([ 0.5, -0.5,  0.5,  0.5])
>>> C.chebint(c,3)
array([ 0.03125   , -0.1875    ,  0.04166667, -0.05208333,  0.01041667,
        0.00625   ])
>>> C.chebint(c, k=3)
array([ 3.5, -0.5,  0.5,  0.5])
>>> C.chebint(c,lbnd=-2)
array([ 8.5, -0.5,  0.5,  0.5])
>>> C.chebint(c,scl=-2)
array([-1.,  1., -1., -1.])
```

chebadd(c1, c2)	Add one Chebyshev series to another.
chebsub(c1, c2)	Subtract one Chebyshev series from another.
chebmul(c1, c2)	Multiply one Chebyshev series by another.
chebmulx(c)	Multiply a Chebyshev series by x.
chebdiv(c1, c2)	Divide one Chebyshev series by another.
chebpow(c, pow[, maxpower])	Raise a Chebyshev series to a power.

Algebra

numpy.polynomial.chebyshev.**chebadd**(*c1*, *c2*)

Add one Chebyshev series to another.

Returns the sum of two Chebyshev series *c1* + *c2*. The arguments are sequences of coefficients ordered from lowest order term to highest, i.e., [1,2,3] represents the series `T_0 + 2*T_1 + 3*T_2`.

Parameters

c1, c2 : array_like

1-D arrays of Chebyshev series coefficients ordered from low to high.

Returns

out : ndarray

Array representing the Chebyshev series of their sum.

See also:

chebsub, *chebmul*, *chebdiv*, *chebpow*

Notes

Unlike multiplication, division, etc., the sum of two Chebyshev series is a Chebyshev series (without having to "reproject" the result onto the basis set) so addition, just like that of "standard" polynomials, is simply "component-wise."

Examples

```
>>> from numpy.polynomial import chebyshev as C
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> C.chebadd(c1,c2)
array([ 4.,  4.,  4.])
```

numpy.polynomial.chebyshev.**chebsub**(*c1*, *c2*)

Subtract one Chebyshev series from another.

Returns the difference of two Chebyshev series *c1* - *c2*. The sequences of coefficients are from lowest order term to highest, i.e., [1,2,3] represents the series `T_0 + 2*T_1 + 3*T_2`.

Parameters

c1, c2 : array_like

1-D arrays of Chebyshev series coefficients ordered from low to high.

Returns

out : ndarray

Of Chebyshev series coefficients representing their difference.

See also:

chebadd, *chebmul*, *chebdiv*, *chebpow*

Notes

Unlike multiplication, division, etc., the difference of two Chebyshev series is a Chebyshev series (without having to "reproject" the result onto the basis set) so subtraction, just like that of "standard" polynomials, is simply "component-wise."

Examples

```
>>> from numpy.polynomial import chebyshev as C
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> C.chebsub(c1,c2)
array([-2.,  0.,  2.])
>>> C.chebsub(c2,c1) # -C.chebsub(c1,c2)
array([ 2.,  0., -2.])
```

numpy.polynomial.chebyshev.**chebmul**(*c1*, *c2*)

Multiply one Chebyshev series by another.

Returns the product of two Chebyshev series *c1* * *c2*. The arguments are sequences of coefficients, from lowest order "term" to highest, e.g., [1,2,3] represents the series `T_0 + 2*T_1 + 3*T_2`.

Parameters

c1, c2 : array_like

1-D arrays of Chebyshev series coefficients ordered from low to high.

Returns

out : ndarray

Of Chebyshev series coefficients representing their product.

See also:

chebadd, *chebsub*, *chebdiv*, *chebpow*

Notes

In general, the (polynomial) product of two C-series results in terms that are not in the Chebyshev polynomial basis set. Thus, to express the product as a C-series, it is typically necessary to "reproject" the product onto said basis set, which typically produces "unintuitive live" (but correct) results; see Examples section below.

Examples

```
>>> from numpy.polynomial import chebyshev as C
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> C.chebmul(c1,c2) # multiplication requires "reprojection"
array([  6.5,  12. ,  12. ,   4. ,   1.5])
```

numpy.polynomial.chebyshev.**chebmulx**(*c*)

Multiply a Chebyshev series by x.

Multiply the polynomial *c* by x, where x is the independent variable.

Parameters

c : array_like

1-D array of Chebyshev series coefficients ordered from low to high.

Returns

out : ndarray

Array representing the result of the multiplication.

Notes

New in version 1.5.0.

numpy.polynomial.chebyshev.**chebdiv**(*c1*, *c2*)

Divide one Chebyshev series by another.

Returns the quotient-with-remainder of two Chebyshev series *c1* / *c2*. The arguments are sequences of coefficients from lowest order "term" to highest, e.g., [1,2,3] represents the series `T_0 + 2*T_1 + 3*T_2`.

Parameters

c1, c2 : array_like

1-D arrays of Chebyshev series coefficients ordered from low to high.

Returns

[quo, rem] : ndarrays

Of Chebyshev series coefficients representing the quotient and remainder.

See also:

chebadd, *chebsub*, *chebmul*, *chebpow*

Notes

In general, the (polynomial) division of one C-series by another results in quotient and remainder terms that are not in the Chebyshev polynomial basis set. Thus, to express these results as C-series, it is typically necessary

to "reproject" the results onto said basis set, which typically produces "unintuitive" (but correct) results; see Examples section below.

Examples

```
>>> from numpy.polynomial import chebyshev as C
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> C.chebdiv(c1,c2) # quotient "intuitive," remainder not
(array([ 3.]), array([-8., -4.]))
>>> c2 = (0,1,2,3)
>>> C.chebdiv(c2,c1) # neither "intuitive"
(array([ 0.,  2.]), array([-2., -4.]))
```

numpy.polynomial.chebyshev.**chebpow**(*c*, *pow*, *maxpower=16*)

Raise a Chebyshev series to a power.

Returns the Chebyshev series *c* raised to the power *pow*. The argument *c* is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series `T_0 + 2*T_1 + 3*T_2`.

Parameters

c : array_like

1-D array of Chebyshev series coefficients ordered from low to high.

pow : integer

Power to which the series will be raised

maxpower : integer, optional

Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

Returns

coef : ndarray

Chebyshev series of power.

See also:

chebadd, *chebsub*, *chebmul*, *chebdiv*

chebgauss(deg)	Gauss-Chebyshev quadrature.
chebweight(x)	The weight function of the Chebyshev polynomials.

Quadrature

numpy.polynomial.chebyshev.**chebgauss**(*deg*)

Gauss-Chebyshev quadrature.

Computes the sample points and weights for Gauss-Chebyshev quadrature. These sample points and weights will correctly integrate polynomials of degree $2*deg-1$ or less over the interval $[-1, 1]$ with the weight function $f(x) = 1/\sqrt{1-x^2}$.

Parameters

deg : int

Number of sample points and weights. It must be >= 1.

Returns

x : ndarray

1-D ndarray containing the sample points.

y : ndarray

1-D ndarray containing the weights.

Notes

New in version 1.7.0.

The results have only been tested up to degree 100, higher degrees may be problematic. For Gauss-Chebyshev there are closed form solutions for the sample points and weights. If n = *deg*, then

$$x_i = \cos(\pi(2i - 1)/(2n))$$

$$w_i = \pi/n$$

numpy.polynomial.chebyshev.**chebweight**(*x*)

The weight function of the Chebyshev polynomials.

The weight function is $1/\sqrt{1 - x^2}$ and the interval of integration is $[-1, 1]$. The Chebyshev polynomials are orthogonal, but not normalized, with respect to this weight function.

Parameters

x : array_like

Values at which the weight function will be computed.

Returns

w : ndarray

The weight function at *x*.

Notes

New in version 1.7.0.

chebcompanion(c)	Return the scaled companion matrix of c.
chebdomain	
chebzero	
chebone	
chebx	
chebline(off, scl)	Chebyshev series whose graph is a straight line.
cheb2poly(c)	Convert a Chebyshev series to a polynomial.
poly2cheb(pol)	Convert a polynomial to a Chebyshev series.

Miscellaneous

numpy.polynomial.chebyshev.**chebcompanion**(*c*)

Return the scaled companion matrix of c.

The basis polynomials are scaled so that the companion matrix is symmetric when *c* is a Chebyshev basis polynomial. This provides better eigenvalue estimates than the unscaled case and for basis polynomials the eigenvalues are guaranteed to be real if `numpy.linalg.eigvalsh` is used to obtain them.

Parameters

c : array_like

1-D array of Chebyshev series coefficients ordered from low to high degree.

Returns

mat : ndarray

Scaled companion matrix of dimensions (deg, deg).

Notes

numpy.polynomial.chebyshev.**chebdomain = array([-1, 1])**

numpy.polynomial.chebyshev.**chebzero = array([0])**

numpy.polynomial.chebyshev.**chebone = array([1])**

numpy.polynomial.chebyshev.**chebx = array([0, 1])**

numpy.polynomial.chebyshev.**chebline**(*off*, *scl*)

Chebyshev series whose graph is a straight line.

Parameters

off, scl : scalars

The specified line is given by `off + scl*x`.

Returns

y : ndarray

This module's representation of the Chebyshev series for `off + scl*x`.

See also:

`polyline`

Examples

```
>>> import numpy.polynomial.chebyshev as C
>>> C.chebline(3,2)
array([3, 2])
>>> C.chebval(-3, C.chebline(3,2)) # should be -3
-3.0
```

numpy.polynomial.chebyshev.**cheb2poly**(*c*)

Convert a Chebyshev series to a polynomial.

Convert an array representing the coefficients of a Chebyshev series, ordered from lowest degree to highest, to an array of the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest to highest degree.

Parameters

c : array_like

1-D array containing the Chebyshev series coefficients, ordered from lowest order term to highest.

Returns

pol : ndarray

1-D array containing the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest order term to highest.

See also:

poly2cheb

Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```
>>> from numpy import polynomial as P
>>> c = P.Chebyshev(range(4))
>>> c
Chebyshev([ 0.,  1.,  2.,  3.], [-1.,  1.])
>>> p = c.convert(kind=P.Polynomial)
>>> p
Polynomial([ -2.,  -8.,   4.,  12.], [-1.,  1.])
>>> P.cheb2poly(range(4))
array([ -2.,  -8.,   4.,  12.])
```

numpy.polynomial.chebyshev.**poly2cheb**(*pol*)

Convert a polynomial to a Chebyshev series.

Convert an array representing the coefficients of a polynomial (relative to the "standard" basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Chebyshev series, ordered from lowest to highest degree.

Parameters

pol : array_like

1-D array containing the polynomial coefficients

Returns

c : ndarray

1-D array containing the coefficients of the equivalent Chebyshev series.

See also:

cheb2poly

Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```
>>> from numpy import polynomial as P
>>> p = P.Polynomial(range(4))
>>> p
Polynomial([ 0.,  1.,  2.,  3.], [-1.,  1.])
>>> c = p.convert(kind=P.Chebyshev)
>>> c
Chebyshev([ 1.  ,  3.25,  1.  ,  0.75], [-1.,  1.])
>>> P.poly2cheb(range(4))
array([ 1.  ,  3.25,  1.  ,  0.75])
```

Legendre Module (`numpy.polynomial.legendre`)

New in version 1.6.0.

This module provides a number of objects (mostly functions) useful for dealing with Legendre series, including a *Legendre* class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its "parent" sub-package, `numpy.polynomial`).

Legendre(coef[, domain, window])	A Legendre series class.

Legendre Class

class `numpy.polynomial.legendre.`**Legendre**(*coef*, *domain=None*, *window=None*)

A Legendre series class.

The Legendre class provides the standard Python numerical methods '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the attributes and methods listed in the `ABCPolyBase` documentation.

Parameters

coef : array_like

Legendre coefficients in order of increasing degree, i.e., `(1, 2, 3)` gives `1*P_0(x) + 2*P_1(x) + 3*P_2(x)`.

domain : (2,) array_like, optional

Domain to use. The interval `[domain[0], domain[1]]` is mapped to the interval `[window[0], window[1]]` by shifting and scaling. The default value is [-1, 1].

window : (2,) array_like, optional

Window, see `domain` for its use. The default value is [-1, 1].

New in version 1.6.0.

Methods

legval(x, c[, tensor])	Evaluate a Legendre series at points x.
legval2d(x, y, c)	Evaluate a 2-D Legendre series at points (x, y).
legval3d(x, y, z, c)	Evaluate a 3-D Legendre series at points (x, y, z).
leggrid2d(x, y, c)	Evaluate a 2-D Legendre series on the Cartesian product of x and y.
leggrid3d(x, y, z, c)	Evaluate a 3-D Legendre series on the Cartesian product of x, y, and z.
legroots(c)	Compute the roots of a Legendre series.
legfromroots(roots)	Generate a Legendre series with given roots.

Basics

`numpy.polynomial.legendre.`**legval**(*x*, *c*, *tensor=True*)

Evaluate a Legendre series at points x.

If *c* is of length *n + 1*, this function returns the value:

$$p(x) = c_0 * L_0(x) + c_1 * L_1(x) + ... + c_n * L_n(x)$$

The parameter *x* is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either *x* or its elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array, then *p(x)* will have the same shape as *x*. If *c* is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor* is true the shape will be c.shape[1:] + x.shape. If *tensor* is false the shape will be c.shape[1:]. Note that scalars have shape (,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

Parameters

x : array_like, compatible object

If *x* is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, *x* or its elements must support addition and multiplication with with themselves and with the elements of *c*.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree n are contained in c[n]. If *c* is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of *c*.

tensor : boolean, optional

If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of *x*. Scalars have dimension 0 for this action. The result is that every column of coefficients in *c* is evaluated for every element of *x*. If False, *x* is broadcast over the columns of *c* for the evaluation. This keyword is useful when *c* is multidimensional. The default value is True.

New in version 1.7.0.

Returns

values : ndarray, algebra_like

The shape of the return value is described above.

See also:

legval2d, *leggrid2d*, *legval3d*, *leggrid3d*

Notes

The evaluation uses Clenshaw recursion, aka synthetic division.

numpy.polynomial.legendre.**legval2d**(*x*, *y*, *c*)

Evaluate a 2-D Legendre series at points (x, y).

This function returns the values:

$$p(x,y) = \sum_{i,j} c_{i,j} * L_i(x) * L_j(y)$$

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array a one is implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

Parameters

x, y : array_like, compatible objects

The two dimensional series is evaluated at the points *(x, y)*, where *x* and *y* must have the same shape. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional Legendre series at points formed from pairs of corresponding values from *x* and *y*.

See also:

legval, *leggrid2d*, *legval3d*, *leggrid3d*

Notes

numpy.polynomial.legendre.**legval3d**(*x*, *y*, *z*, *c*)

Evaluate a 3-D Legendre series at points (x, y, z).

This function returns the values:

$$p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * L_i(x) * L_j(y) * L_k(z)$$

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape.

Parameters

x, y, z : array_like, compatible object

The three dimensional series is evaluated at the points *(x, y, z)*, where *x*, *y*, and *z* must have the same shape. If any of *x*, *y*, or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j,k is contained in `c[i,j,k]`. If *c* has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the multidimensional polynomial on points formed with triples of corresponding values from *x*, *y*, and *z*.

See also:

legval, *legval2d*, *leggrid2d*, *leggrid3d*

Notes

numpy.polynomial.legendre.**leggrid2d**(*x*, *y*, *c*)

Evaluate a 2-D Legendre series on the Cartesian product of x and y.

This function returns the values:

$$p(a,b) = \sum_{i,j} c_{i,j} * L_i(a) * L_j(b)$$

where the points *(a, b)* consist of all pairs formed by taking *a* from *x* and *b* from *y*. The resulting points form a grid with *x* in the first dimension and *y* in the second.

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape + y.shape.

Parameters

x, y : array_like, compatible objects

The two dimensional series is evaluated at the points in the Cartesian product of *x* and *y*. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in *c[i,j]*. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional Chebyshev series at points in the Cartesian product of *x* and *y*.

See also:

legval, *legval2d*, *legval3d*, *leggrid3d*

Notes

numpy.polynomial.legendre.**leggrid3d**(*x*, *y*, *z*, *c*)

Evaluate a 3-D Legendre series on the Cartesian product of x, y, and z.

This function returns the values:

$$p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * L_i(a) * L_j(b) * L_k(c)$$

where the points *(a, b, c)* consist of all triples formed by taking *a* from *x*, *b* from *y*, and *c* from *z*. The resulting points form a grid with *x* in the first dimension, *y* in the second, and *z* in the third.

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape + y.shape + z.shape.

Parameters

x, y, z : array_like, compatible objects

The three dimensional series is evaluated at the points in the Cartesian product of *x*, *y*, and *z*. If *x*,'y', or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional polynomial at points in the Cartesian product of x and y.

See also:

legval, *legval2d*, *leggrid2d*, *legval3d*

Notes

numpy.polynomial.legendre.**legroots**(*c*)

Compute the roots of a Legendre series.

Return the roots (a.k.a. "zeros") of the polynomial

$$p(x) = \sum_i c[i] * L_i(x).$$

Parameters

c : 1-D array_like

1-D array of coefficients.

Returns

out : ndarray

Array of the roots of the series. If all the roots are real, then *out* is also real, otherwise it is complex.

See also:

`polyroots`, `chebroots`, `lagroots`, `hermroots`, `hermeroots`

Notes

The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the complex plane may have large errors due to the numerical instability of the series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton's method.

The Legendre series basis polynomials aren't powers of `x` so the results of this function may seem unintuitive.

Examples

```
>>> import numpy.polynomial.legendre as leg
>>> leg.legroots((1, 2, 3, 4)) # 4L_3 + 3L_2 + 2L_1 + 1L_0, all real roots
array([-0.85099543, -0.11407192,  0.51506735])
```

numpy.polynomial.legendre.**legfromroots**(*roots*)

Generate a Legendre series with given roots.

The function returns the coefficients of the polynomial

$$p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),$$

in Legendre form, where the *r_n* are the roots specified in *roots*. If a zero has multiplicity n, then it must appear in *roots* n times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then *roots* looks something like [2, 2, 2, 3, 3]. The roots can appear in any order.

If the returned coefficients are *c*, then

$$p(x) = c_0 + c_1 * L_1(x) + ... + c_n * L_n(x)$$

The coefficient of the last term is not generally 1 for monic polynomials in Legendre form.

Parameters

roots : array_like

Sequence containing the roots.

Returns

out : ndarray

1-D array of coefficients. If all roots are real then *out* is a real array, if some of the roots are complex, then *out* is complex even if all the coefficients in the result are real (see Examples below).

See also:

`polyfromroots`, `chebfromroots`, `lagfromroots`, `hermfromroots`, `hermefromroots`.

Examples

```
>>> import numpy.polynomial.legendre as L
>>> L.legfromroots((-1,0,1)) # x^3 - x relative to the standard basis
array([ 0. , -0.4,  0. ,  0.4])
>>> j = complex(0,1)
>>> L.legfromroots((-j,j)) # x^2 + 1 relative to the standard basis
array([ 1.33333333+0.j,  0.00000000+0.j,  0.66666667+0.j])
```

legfit(x, y, deg[, rcond, full, w])	Least squares fit of Legendre series to data.
legvander(x, deg)	Pseudo-Vandermonde matrix of given degree.
legvander2d(x, y, deg)	Pseudo-Vandermonde matrix of given degrees.
legvander3d(x, y, z, deg)	Pseudo-Vandermonde matrix of given degrees.

Fitting

`numpy.polynomial.legendre.`**`legfit`**(*x*, *y*, *deg*, *rcond=None*, *full=False*, *w=None*)

Least squares fit of Legendre series to data.

Return the coefficients of a Legendre series of degree *deg* that is the least squares fit to the data values *y* given at points *x*. If *y* is 1-D the returned coefficients will also be 1-D. If *y* is 2-D multiple fits are done, one for each column of *y*, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

$$p(x) = c_0 + c_1 * L_1(x) + ... + c_n * L_n(x),$$

where *n* is *deg*.

Parameters

x : array_like, shape (M,)

x-coordinates of the M sample points `(x[i], y[i])`.

y : array_like, shape (M,) or (M, K)

y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

deg : int or 1-D array_like

Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For Numpy versions >= 1.11 a list of integers specifying the degrees of the terms to include may be used instead.

rcond : float, optional

Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

full : bool, optional

Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

w : array_like, shape (*M*,), optional

Weights. If not None, the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

New in version 1.5.0.

Returns

coef : ndarray, shape (M,) or (M, K)

Legendre coefficients ordered from low to high. If *y* was 2-D, the coefficients for the data in column k of *y* are in column *k*. If *deg* is specified as a list, coefficients for terms not included in the fit are set equal to zero in the returned *coef*.

[residuals, rank, singular_values, rcond] : list

These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

Warns

RankWarning

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if *full* = False. The warnings can be turned off by

```
>>> import warnings
>>> warnings.simplefilter('ignore', RankWarning)
```

See also:

`chebfit`, `polyfit`, `lagfit`, `hermfit`, `hermefit`

legval

Evaluates a Legendre series.

legvander
Vandermonde matrix of Legendre series.

legweight
Legendre weight function (= 1).

linalg.lstsq
Computes a least-squares fit from the matrix.

scipy.interpolate.UnivariateSpline
Computes spline fits.

Notes

The solution is the coefficients of the Legendre series *p* that minimizes the sum of the weighted squared errors

$$E = \sum_j w_j^2 * |y_j - p(x_j)|^2,$$

where w_j are the weights. This problem is solved by setting up as the (typically) overdetermined matrix equation

$$V(x) * c = w * y,$$

where *V* is the weighted pseudo Vandermonde matrix of *x*, *c* are the coefficients to be solved for, *w* are the weights, and *y* are the observed values. This equation is then solved using the singular value decomposition of *V*.

If some of the singular values of *V* are so small that they are neglected, then a *RankWarning* will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The *rcond* parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using Legendre series are usually better conditioned than fits using power series, but much can depend on the distribution of the sample points and the smoothness of the data. If the quality of the fit is inadequate splines may be a good alternative.

References

[R64]

numpy.polynomial.legendre.**legvander**(*x*, *deg*)
Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree *deg* and sample points *x*. The pseudo-Vandermonde matrix is defined by

$$V[..., i] = L_i(x)$$

where *0 <= i <= deg*. The leading indices of *V* index the elements of *x* and the last index is the degree of the Legendre polynomial.

If *c* is a 1-D array of coefficients of length *n + 1* and *V* is the array `V = legvander(x, n)`, then `np.dot(V, c)` and `legval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of Legendre series of the same degree and sample points.

Parameters

x : array_like

Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If *x* is scalar it is converted to a 1-D array.

deg : int

Degree of the resulting matrix.

Returns

vander : ndarray

The pseudo-Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where The last index is the degree of the corresponding Legendre polynomial. The dtype will be the same as the converted *x*.

`numpy.polynomial.legendre.`**`legvander2d`**(*x*, *y*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y)*. The pseudo-Vandermonde matrix is defined by

$$V[..., deg[1] * i + j] = L_i(x) * L_j(y),$$

where *0 <= i <= deg[0]* and *0 <= j <= deg[1]*. The leading indices of *V* index the points *(x, y)* and the last index encodes the degrees of the Legendre polynomials.

If `V = legvander2d(x, y, [xdeg, ydeg])`, then the columns of *V* correspond to the elements of a 2-D coefficient array *c* of shape (xdeg + 1, ydeg + 1) in the order

$$c_{00}, c_{01}, c_{02}..., c_{10}, c_{11}, c_{12}...$$

and `np.dot(V, c.flat)` and `legval2d(x, y, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D Legendre series of the same degrees and sample points.

Parameters

x, y : array_like

Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

deg : list of ints

List of maximum degrees of the form [x_deg, y_deg].

Returns

vander2d : ndarray

The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0]+1)*(deg([1]+1)$. The dtype will be the same as the converted *x* and *y*.

See also:

legvander, `legvander3d.`, *legval3d*

Notes

`numpy.polynomial.legendre.`**`legvander3d`**(*x*, *y*, *z*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y, z)*. If *l, m, n* are the given degrees in *x, y, z*, then The pseudo-Vandermonde matrix is defined by

$$V[..., (m+1)(n+1)i + (n+1)j + k] = L_i(x) * L_j(y) * L_k(z),$$

where *0 <= i <= l*, *0 <= j <= m*, and *0 <= j <= n*. The leading indices of *V* index the points *(x, y, z)* and the last index encodes the degrees of the Legendre polynomials.

If `V = legvander3d(x, y, z, [xdeg, ydeg, zdeg])`, then the columns of *V* correspond to the elements of a 3-D coefficient array *c* of shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order

$$c_{000}, c_{001}, c_{002}, ..., c_{010}, c_{011}, c_{012}, ...$$

and `np.dot(V, c.flat)` and `legval3d(x, y, z, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D Legendre series of the same degrees and sample points.

Parameters

x, y, z : array_like

Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

deg : list of ints

List of maximum degrees of the form [x_deg, y_deg, z_deg].

Returns

vander3d : ndarray

The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0] + 1) * (deg([1] + 1) * (deg[2] + 1)$. The dtype will be the same as the converted *x*, *y*, and *z*.

See also:

legvander, `legvander3d.`, *legval3d*

Notes

legder(c[, m, scl, axis])	Differentiate a Legendre series.
legint(c[, m, k, lbnd, scl, axis])	Integrate a Legendre series.

Calculus

`numpy.polynomial.legendre.`**`legder`**(*c*, *m=1*, *scl=1*, *axis=0*)

Differentiate a Legendre series.

Returns the Legendre series coefficients *c* differentiated *m* times along *axis*. At each iteration the result is multiplied by *scl* (the scaling factor is for use in a linear change of variable). The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `1*L_0 + 2*L_1 + 3*L_2` while [[1,2],[1,2]] represents `1*L_0(x)*L_0(y) + 1*L_1(x)*L_0(y) + 2*L_0(x)*L_1(y) + 2*L_1(x)*L_1(y)` if axis=0 is `x` and axis=1 is `y`.

Parameters

c : array_like

Array of Legendre series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

m : int, optional

Number of derivatives taken, must be non-negative. (Default: 1)

scl : scalar, optional

Each differentiation is multiplied by *scl*. The end result is multiplication by `scl**m`. This is for use in a linear change of variable. (Default: 1)

axis : int, optional

Axis over which the derivative is taken. (Default: 0).

New in version 1.7.0.

Returns

der : ndarray

Legendre series of the derivative.

See also:

legint

Notes

In general, the result of differentiating a Legendre series does not resemble the same operation on a power series. Thus the result of this function may be "unintuitive," albeit correct; see Examples section below.

Examples

```
>>> from numpy.polynomial import legendre as L
>>> c = (1,2,3,4)
>>> L.legder(c)
array([  6.,   9.,  20.])
>>> L.legder(c, 3)
array([ 60.])
>>> L.legder(c, scl=-1)
array([ -6.,  -9., -20.])
>>> L.legder(c, 2,-1)
array([  9.,  60.])
```

numpy.polynomial.legendre.**legint**(*c*, *m=1*, *k=[]*, *lbnd=0*, *scl=1*, *axis=0*)

Integrate a Legendre series.

Returns the Legendre series coefficients *c* integrated *m* times from *lbnd* along *axis*. At each iteration the resulting series is **multiplied** by *scl* and an integration constant, *k*, is added. The scaling factor is for use in a linear change of variable. ("Buyer beware": note that, depending on what one is doing, one may want *scl* to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `L_0 + 2*L_1 + 3*L_2` while [[1,2],[1,2]] represents `1*L_0(x)*L_0(y) + 1*L_1(x)*L_0(y) + 2*L_0(x)*L_1(y) + 2*L_1(x)*L_1(y)` if axis=0 is `x` and axis=1 is `y`.

Parameters

c : array_like

Array of Legendre series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

m : int, optional

Order of integration, must be positive. (Default: 1)

k : {[], list, scalar}, optional

Integration constant(s). The value of the first integral at `lbnd` is the first value in the list, the value of the second integral at `lbnd` is the second value, etc. If `k == []` (the default), all constants are set to zero. If `m == 1`, a single scalar can be given instead of a list.

lbnd : scalar, optional

The lower bound of the integral. (Default: 0)

scl : scalar, optional

Following each integration the result is *multiplied* by *scl* before the integration constant is added. (Default: 1)

axis : int, optional

Axis over which the integral is taken. (Default: 0).

New in version 1.7.0.

Returns

S : ndarray

Legendre series coefficient array of the integral.

Raises

ValueError

If `m < 0`, `len(k) > m`, `np.isscalar(lbnd) == False`, or `np.isscalar(scl) == False`.

See also:

legder

Notes

Note that the result of each integration is *multiplied* by *scl*. Why is this important to note? Say one is making a linear change of variable $u = ax + b$ in an integral relative to *x*. Then .. math::*dx* = *du*/*a*, so one will need to set *scl* equal to $1/a$ - perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be "reprojected" onto the C-series basis set. Thus, typically, the result of this function is "unintuitive," albeit correct; see Examples section below.

Examples

```
>>> from numpy.polynomial import legendre as L
>>> c = (1,2,3)
>>> L.legint(c)
array([ 0.33333333,  0.4       ,  0.66666667,  0.6       ])
>>> L.legint(c, 3)
array([  1.66666667e-02,  -1.78571429e-02,   4.76190476e-02,
        -1.73472348e-18,   1.90476190e-02,   9.52380952e-03])
>>> L.legint(c, k=3)
array([ 3.33333333,  0.4       ,  0.66666667,  0.6       ])
>>> L.legint(c, lbnd=-2)
array([ 7.33333333,  0.4       ,  0.66666667,  0.6       ])
>>> L.legint(c, scl=2)
array([ 0.66666667,  0.8       ,  1.33333333,  1.2       ])
```

legadd(c1, c2)	Add one Legendre series to another.
legsub(c1, c2)	Subtract one Legendre series from another.
legmul(c1, c2)	Multiply one Legendre series by another.
legmulx(c)	Multiply a Legendre series by x.
legdiv(c1, c2)	Divide one Legendre series by another.
legpow(c, pow[, maxpower])	Raise a Legendre series to a power.

Algebra

numpy.polynomial.legendre.**legadd**(*c1*, *c2*)

Add one Legendre series to another.

Returns the sum of two Legendre series *c1* + *c2*. The arguments are sequences of coefficients ordered from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Legendre series coefficients ordered from low to high.

Returns

out : ndarray

Array representing the Legendre series of their sum.

See also:

legsub, *legmul*, *legdiv*, *legpow*

Notes

Unlike multiplication, division, etc., the sum of two Legendre series is a Legendre series (without having to "reproject" the result onto the basis set) so addition, just like that of "standard" polynomials, is simply "component-wise."

Examples

```
>>> from numpy.polynomial import legendre as L
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> L.legadd(c1,c2)
array([ 4.,  4.,  4.])
```

numpy.polynomial.legendre.**legsub**(*c1*, *c2*)

Subtract one Legendre series from another.

Returns the difference of two Legendre series *c1* - *c2*. The sequences of coefficients are from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Legendre series coefficients ordered from low to high.

Returns

out : ndarray

Of Legendre series coefficients representing their difference.

See also:

legadd, *legmul*, *legdiv*, *legpow*

Notes

Unlike multiplication, division, etc., the difference of two Legendre series is a Legendre series (without having to "reproject" the result onto the basis set) so subtraction, just like that of "standard" polynomials, is simply "component-wise."

Examples

```
>>> from numpy.polynomial import legendre as L
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> L.legsub(c1,c2)
array([-2.,  0.,  2.])
>>> L.legsub(c2,c1) # -C.legsub(c1,c2)
array([ 2.,  0., -2.])
```

numpy.polynomial.legendre.**legmul**(*c1*, *c2*)

Multiply one Legendre series by another.

Returns the product of two Legendre series *c1* * *c2*. The arguments are sequences of coefficients, from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Legendre series coefficients ordered from low to high.

Returns

out : ndarray

Of Legendre series coefficients representing their product.

See also:

legadd, *legsub*, *legdiv*, *legpow*

Notes

In general, the (polynomial) product of two C-series results in terms that are not in the Legendre polynomial basis set. Thus, to express the product as a Legendre series, it is necessary to "reproject" the product onto said basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

Examples

```
>>> from numpy.polynomial import legendre as L
>>> c1 = (1,2,3)
>>> c2 = (3,2)
>>> P.legmul(c1,c2) # multiplication requires "reprojection"
array([  4.33333333,  10.4       ,  11.66666667,   3.6       ])
```

numpy.polynomial.legendre.**legmulx**(*c*)

Multiply a Legendre series by x.

Multiply the Legendre series *c* by x, where x is the independent variable.

Parameters

c : array_like

1-D array of Legendre series coefficients ordered from low to high.

Returns

out : ndarray

Array representing the result of the multiplication.

Notes

The multiplication uses the recursion relationship for Legendre polynomials in the form

$$xP_i(x) = ((i+1)*P_{i+1}(x) + i*P_{i-1}(x))/(2i+1)$$

numpy.polynomial.legendre.**legdiv**(*c1*, *c2*)

Divide one Legendre series by another.

Returns the quotient-with-remainder of two Legendre series *c1* / *c2*. The arguments are sequences of coefficients from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Legendre series coefficients ordered from low to high.

Returns

quo, rem : ndarrays

Of Legendre series coefficients representing the quotient and remainder.

See also:

legadd, *legsub*, *legmul*, *legpow*

Notes

In general, the (polynomial) division of one Legendre series by another results in quotient and remainder terms that are not in the Legendre polynomial basis set. Thus, to express these results as a Legendre series, it is necessary to "reproject" the results onto the Legendre basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

Examples

```
>>> from numpy.polynomial import legendre as L
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> L.legdiv(c1,c2) # quotient "intuitive," remainder not
(array([ 3.]), array([-8., -4.]))
>>> c2 = (0,1,2,3)
>>> L.legdiv(c2,c1) # neither "intuitive"
(array([-0.07407407,  1.66666667]), array([-1.03703704, -2.51851852]))
```

numpy.polynomial.legendre.**legpow**(*c*, *pow*, *maxpower=16*)

Raise a Legendre series to a power.

Returns the Legendre series *c* raised to the power *pow*. The arguement *c* is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series `P_0 + 2*P_1 + 3*P_2.`

Parameters

c : array_like

1-D array of Legendre series coefficients ordered from low to high.

pow : integer

Power to which the series will be raised

maxpower : integer, optional

Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

Returns

coef : ndarray

Legendre series of power.

See also:

legadd, *legsub*, *legmul*, *legdiv*

leggauss(deg)	Gauss-Legendre quadrature.
legweight(x)	Weight function of the Legendre polynomials.

Quadrature

numpy.polynomial.legendre.**leggauss**(*deg*)

Gauss-Legendre quadrature.

Computes the sample points and weights for Gauss-Legendre quadrature. These sample points and weights will correctly integrate polynomials of degree $2 * deg - 1$ or less over the interval $[-1, 1]$ with the weight function $f(x) = 1$.

Parameters

deg : int

Number of sample points and weights. It must be >= 1.

Returns

x : ndarray

1-D ndarray containing the sample points.

y : ndarray

1-D ndarray containing the weights.

Notes

The results have only been tested up to degree 100, higher degrees may be problematic. The weights are determined by using the fact that

$$w_k = c/(L'_n(x_k) * L_{n-1}(x_k))$$

where c is a constant independent of k and x_k is the k'th root of L_n, and then scaling the results to get the right value when integrating 1.

numpy.polynomial.legendre.**legweight**(*x*)

Weight function of the Legendre polynomials.

The weight function is 1 and the interval of integration is $[-1, 1]$. The Legendre polynomials are orthogonal, but not normalized, with respect to this weight function.

Parameters

x : array_like

Values at which the weight function will be computed.

Returns

w : ndarray

The weight function at *x*.

Notes

legcompanion(c)	Return the scaled companion matrix of c.
legdomain	

Continued on next page

Table 3.141 – continued from previous page

legzero	
legone	
legx	
legline(off, scl)	Legendre series whose graph is a straight line.
leg2poly(c)	Convert a Legendre series to a polynomial.
poly2leg(pol)	Convert a polynomial to a Legendre series.

Miscellaneous

numpy.polynomial.legendre.**legcompanion**(*c*)

Return the scaled companion matrix of c.

The basis polynomials are scaled so that the companion matrix is symmetric when *c* is an Legendre basis polynomial. This provides better eigenvalue estimates than the unscaled case and for basis polynomials the eigenvalues are guaranteed to be real if `numpy.linalg.eigvalsh` is used to obtain them.

Parameters

c : array_like

1-D array of Legendre series coefficients ordered from low to high degree.

Returns

mat : ndarray

Scaled companion matrix of dimensions (deg, deg).

Notes

numpy.polynomial.legendre.**legdomain = array([-1, 1])**

numpy.polynomial.legendre.**legzero = array([0])**

numpy.polynomial.legendre.**legone = array([1])**

numpy.polynomial.legendre.**legx = array([0, 1])**

numpy.polynomial.legendre.**legline**(*off*, *scl*)

Legendre series whose graph is a straight line.

Parameters

off, scl : scalars

The specified line is given by `off + scl*x`.

Returns

y : ndarray

This module's representation of the Legendre series for `off + scl*x`.

See also:

`polyline`, `chebline`

Examples

```
>>> import numpy.polynomial.legendre as L
>>> L.legline(3,2)
array([3, 2])
```

```
    >>> L.legval(-3, L.legline(3,2)) # should be -3
    -3.0
```

numpy.polynomial.legendre.**leg2poly**(*c*)

Convert a Legendre series to a polynomial.

Convert an array representing the coefficients of a Legendre series, ordered from lowest degree to highest, to an array of the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest to highest degree.

Parameters

c : array_like

1-D array containing the Legendre series coefficients, ordered from lowest order term to highest.

Returns

pol : ndarray

1-D array containing the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest order term to highest.

See also:

poly2leg

Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```
    >>> c = P.Legendre(range(4))
    >>> c
    Legendre([ 0.,  1.,  2.,  3.], [-1.,  1.])
    >>> p = c.convert(kind=P.Polynomial)
    >>> p
    Polynomial([-1. , -3.5,  3. ,  7.5], [-1.,  1.])
    >>> P.leg2poly(range(4))
    array([-1. , -3.5,  3. ,  7.5])
```

numpy.polynomial.legendre.**poly2leg**(*pol*)

Convert a polynomial to a Legendre series.

Convert an array representing the coefficients of a polynomial (relative to the "standard" basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Legendre series, ordered from lowest to highest degree.

Parameters

pol : array_like

1-D array containing the polynomial coefficients

Returns

c : ndarray

1-D array containing the coefficients of the equivalent Legendre series.

See also:

leg2poly

Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```
>>> from numpy import polynomial as P
>>> p = P.Polynomial(np.arange(4))
>>> p
Polynomial([ 0.,  1.,  2.,  3.], [-1.,  1.])
>>> c = P.Legendre(P.poly2leg(p.coef))
>>> c
Legendre([ 1.  ,  3.25,  1.  ,  0.75], [-1.,  1.])
```

Laguerre Module (`numpy.polynomial.laguerre`)

New in version 1.6.0.

This module provides a number of objects (mostly functions) useful for dealing with Laguerre series, including a *Laguerre* class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its "parent" sub-package, `numpy.polynomial`).

Laguerre(coef[, domain, window])	A Laguerre series class.

Laguerre Class

class `numpy.polynomial.laguerre.`**Laguerre**(*coef*, *domain=None*, *window=None*)

A Laguerre series class.

The Laguerre class provides the standard Python numerical methods '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the attributes and methods listed in the `ABCPolyBase` documentation.

Parameters

coef : array_like

Laguerre coefficients in order of increasing degree, i.e, `(1, 2, 3)` gives `1*L_0(x) + 2*L_1(X) + 3*L_2(x)`.

domain : (2,) array_like, optional

Domain to use. The interval `[domain[0], domain[1]]` is mapped to the interval `[window[0], window[1]]` by shifting and scaling. The default value is [0, 1].

window : (2,) array_like, optional

Window, see `domain` for its use. The default value is [0, 1].

New in version 1.6.0.

Methods

lagval(x, c[, tensor])	Evaluate a Laguerre series at points x.
lagval2d(x, y, c)	Evaluate a 2-D Laguerre series at points (x, y).
lagval3d(x, y, z, c)	Evaluate a 3-D Laguerre series at points (x, y, z).
laggrid2d(x, y, c)	Evaluate a 2-D Laguerre series on the Cartesian product of x and y.
laggrid3d(x, y, z, c)	Evaluate a 3-D Laguerre series on the Cartesian product of x, y, and z.
lagroots(c)	Compute the roots of a Laguerre series.

Continued on next page

Table 3.144 – continued from previous page

lagfromroots(roots)	Generate a Laguerre series with given roots.

Basics

numpy.polynomial.laguerre.**lagval**(*x*, *c*, *tensor=True*)

Evaluate a Laguerre series at points x.

If *c* is of length *n + 1*, this function returns the value:

$$p(x) = c_0 * L_0(x) + c_1 * L_1(x) + ... + c_n * L_n(x)$$

The parameter *x* is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either *x* or its elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array, then *p(x)* will have the same shape as *x*. If *c* is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor* is true the shape will be c.shape[1:] + x.shape. If *tensor* is false the shape will be c.shape[1:]. Note that scalars have shape (,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

Parameters

x : array_like, compatible object

If *x* is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, *x* or its elements must support addition and multiplication with with themselves and with the elements of *c*.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree n are contained in c[n]. If *c* is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of *c*.

tensor : boolean, optional

If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of *x*. Scalars have dimension 0 for this action. The result is that every column of coefficients in *c* is evaluated for every element of *x*. If False, *x* is broadcast over the columns of *c* for the evaluation. This keyword is useful when *c* is multidimensional. The default value is True.

New in version 1.7.0.

Returns

values : ndarray, algebra_like

The shape of the return value is described above.

See also:

lagval2d, laggrid2d, lagval3d, laggrid3d

Notes

The evaluation uses Clenshaw recursion, aka synthetic division.

Examples

```
>>> from numpy.polynomial.laguerre import lagval
>>> coef = [1,2,3]
>>> lagval(1, coef)
-0.5
>>> lagval([[1,2],[3,4]], coef)
array([[-0.5, -4. ],
       [-4.5, -2. ]])
```

numpy.polynomial.laguerre.**lagval2d**(*x*, *y*, *c*)

Evaluate a 2-D Laguerre series at points (x, y).

This function returns the values:

$$p(x,y) = \sum_{i,j} c_{i,j} * L_i(x) * L_j(y)$$

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array a one is implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

Parameters

x, y : array_like, compatible objects

The two dimensional series is evaluated at the points *(x, y)*, where *x* and *y* must have the same shape. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional polynomial at points formed with pairs of corresponding values from *x* and *y*.

See also:

lagval, *laggrid2d*, *lagval3d*, *laggrid3d*

Notes

numpy.polynomial.laguerre.**lagval3d**(*x*, *y*, *z*, *c*)

Evaluate a 3-D Laguerre series at points (x, y, z).

This function returns the values:

$$p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * L_i(x) * L_j(y) * L_k(z)$$

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape.

Parameters

x, y, z : array_like, compatible object

The three dimensional series is evaluated at the points *(x, y, z)*, where *x*, *y*, and *z* must have the same shape. If any of *x*, *y*, or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j,k is contained in `c[i,j,k]`. If *c* has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the multidimension polynomial on points formed with triples of corresponding values from *x*, *y*, and *z*.

See also:

lagval, *lagval2d*, *laggrid2d*, *laggrid3d*

Notes

numpy.polynomial.laguerre.**laggrid2d**(*x*, *y*, *c*)

Evaluate a 2-D Laguerre series on the Cartesian product of x and y.

This function returns the values:

$$p(a,b) = \sum_{i,j} c_{i,j} * L_i(a) * L_j(b)$$

where the points *(a, b)* consist of all pairs formed by taking *a* from *x* and *b* from *y*. The resulting points form a grid with *x* in the first dimension and *y* in the second.

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape + y.shape.

Parameters

x, y : array_like, compatible objects

The two dimensional series is evaluated at the points in the Cartesian product of *x* and *y*. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in *c[i,j]*. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional Chebyshev series at points in the Cartesian product of *x* and *y*.

See also:

lagval, *lagval2d*, *lagval3d*, *laggrid3d*

Notes

numpy.polynomial.laguerre.**laggrid3d**(*x*, *y*, *z*, *c*)

Evaluate a 3-D Laguerre series on the Cartesian product of x, y, and z.

This function returns the values:

$$p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * L_i(a) * L_j(b) * L_k(c)$$

where the points *(a, b, c)* consist of all triples formed by taking *a* from *x*, *b* from *y*, and *c* from *z*. The resulting points form a grid with *x* in the first dimension, *y* in the second, and *z* in the third.

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape + y.shape + z.shape.

Parameters

x, y, z : array_like, compatible objects

The three dimensional series is evaluated at the points in the Cartesian product of *x*, *y*, and *z*. If *x*,'y', or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

See also:

lagval, *lagval2d*, *laggrid2d*, *lagval3d*

Notes

numpy.polynomial.laguerre.**lagroots**(*c*)

Compute the roots of a Laguerre series.

Return the roots (a.k.a. "zeros") of the polynomial

$$p(x) = \sum_i c[i] * L_i(x).$$

Parameters

c : 1-D array_like

1-D array of coefficients.

Returns

out : ndarray

Array of the roots of the series. If all the roots are real, then *out* is also real, otherwise it is complex.

See also:

`polyroots`, `legroots`, `chebroots`, `hermroots`, `hermeroots`

Notes

The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the complex plane may have large errors due to the numerical instability of the series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton's method.

The Laguerre series basis polynomials aren't powers of *x* so the results of this function may seem unintuitive.

Examples

```
>>> from numpy.polynomial.laguerre import lagroots, lagfromroots
>>> coef = lagfromroots([0, 1, 2])
>>> coef
array([  2.,  -8.,  12.,  -6.])
>>> lagroots(coef)
array([ -4.44089210e-16,   1.00000000e+00,   2.00000000e+00])
```

`numpy.polynomial.laguerre.`**`lagfromroots`**(*roots*)

Generate a Laguerre series with given roots.

The function returns the coefficients of the polynomial

$$p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),$$

in Laguerre form, where the *r_n* are the roots specified in *roots*. If a zero has multiplicity n, then it must appear in *roots* n times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then *roots* looks something like [2, 2, 2, 3, 3]. The roots can appear in any order.

If the returned coefficients are c, then

$$p(x) = c_0 + c_1 * L_1(x) + ... + c_n * L_n(x)$$

The coefficient of the last term is not generally 1 for monic polynomials in Laguerre form.

Parameters

roots : array_like

Sequence containing the roots.

Returns

out : ndarray

1-D array of coefficients. If all roots are real then *out* is a real array, if some of the roots are complex, then *out* is complex even if all the coefficients in the result are real (see Examples below).

See also:

`polyfromroots`, `legfromroots`, `chebfromroots`, `hermfromroots`, `hermefromroots`.

Examples

```
>>> from numpy.polynomial.laguerre import lagfromroots, lagval
>>> coef = lagfromroots((-1, 0, 1))
>>> lagval((-1, 0, 1), coef)
array([ 0.,  0.,  0.])
```

```
>>> coef = lagfromroots((-1j, 1j))
>>> lagval((-1j, 1j), coef)
array([ 0.+0.j,  0.+0.j])
```

lagfit(x, y, deg[, rcond, full, w])	Least squares fit of Laguerre series to data.
lagvander(x, deg)	Pseudo-Vandermonde matrix of given degree.
lagvander2d(x, y, deg)	Pseudo-Vandermonde matrix of given degrees.
lagvander3d(x, y, z, deg)	Pseudo-Vandermonde matrix of given degrees.

Fitting

numpy.polynomial.laguerre.**lagfit**(*x*, *y*, *deg*, *rcond=None*, *full=False*, *w=None*)

Least squares fit of Laguerre series to data.

Return the coefficients of a Laguerre series of degree *deg* that is the least squares fit to the data values *y* given at points *x*. If *y* is 1-D the returned coefficients will also be 1-D. If *y* is 2-D multiple fits are done, one for each column of *y*, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

$$p(x) = c_0 + c_1 * L_1(x) + ... + c_n * L_n(x),$$

where *n* is *deg*.

Parameters

x : array_like, shape (M,)

x-coordinates of the M sample points `(x[i], y[i])`.

y : array_like, shape (M,) or (M, K)

y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

deg : int or 1-D array_like

Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For Numpy versions >= 1.11 a list of integers specifying the degrees of the terms to include may be used instead.

rcond : float, optional

Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

full : bool, optional

Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

w : array_like, shape (*M*,), optional

Weights. If not None, the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

Returns

coef : ndarray, shape (M,) or (M, K)

Laguerre coefficients ordered from low to high. If *y* was 2-D, the coefficients for the data in column k of *y* are in column *k*.

[residuals, rank, singular_values, rcond] : list

These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

Warns

RankWarning

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if *full* = False. The warnings can be turned off by

```
>>> import warnings
>>> warnings.simplefilter('ignore', RankWarning)
```

See also:

`chebfit`, `legfit`, `polyfit`, `hermfit`, `hermefit`

lagval
Evaluates a Laguerre series.

lagvander
pseudo Vandermonde matrix of Laguerre series.

lagweight
Laguerre weight function.

linalg.lstsq
Computes a least-squares fit from the matrix.

scipy.interpolate.UnivariateSpline
Computes spline fits.

Notes

The solution is the coefficients of the Laguerre series *p* that minimizes the sum of the weighted squared errors

$$E = \sum_j w_j^2 * |y_j - p(x_j)|^2,$$

where the w_j are the weights. This problem is solved by setting up as the (typically) overdetermined matrix equation

$$V(x) * c = w * y,$$

where *V* is the weighted pseudo Vandermonde matrix of *x*, *c* are the coefficients to be solved for, *w* are the weights, and *y* are the observed values. This equation is then solved using the singular value decomposition of *V*.

If some of the singular values of *V* are so small that they are neglected, then a *RankWarning* will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The *rcond* parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using Laguerre series are probably most useful when the data can be approximated by `sqrt(w(x)) * p(x)`, where *w(x)* is the Laguerre weight. In that case the weight `sqrt(w(x[i])` should be used together with data values `y[i]/sqrt(w(x[i]))`. The weight function is available as *lagweight*.

References

[R63]

Examples

```
>>> from numpy.polynomial.laguerre import lagfit, lagval
>>> x = np.linspace(0, 10)
>>> err = np.random.randn(len(x))/10
>>> y = lagval(x, [1, 2, 3]) + err
>>> lagfit(x, y, 2)
array([ 0.96971004,  2.00193749,  3.00288744])
```

numpy.polynomial.laguerre.**lagvander**(*x*, *deg*)

Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree *deg* and sample points *x*. The pseudo-Vandermonde matrix is defined by

$$V[..., i] = L_i(x)$$

where *0 <= i <= deg*. The leading indices of *V* index the elements of *x* and the last index is the degree of the Laguerre polynomial.

If *c* is a 1-D array of coefficients of length *n + 1* and *V* is the array `V = lagvander(x, n)`, then `np.dot(V, c)` and `lagval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of Laguerre series of the same degree and sample points.

Parameters

x : array_like

Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If *x* is scalar it is converted to a 1-D array.

deg : int

Degree of the resulting matrix.

Returns

vander : ndarray

The pseudo-Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where The last index is the degree of the corresponding Laguerre polynomial. The dtype will be the same as the converted *x*.

Examples

```
>>> from numpy.polynomial.laguerre import lagvander
>>> x = np.array([0, 1, 2])
>>> lagvander(x, 3)
array([[ 1.        ,  1.        ,  1.        ,  1.        ],
       [ 1.        ,  0.        , -0.5       , -0.66666667],
       [ 1.        , -1.        , -1.        , -0.33333333]])
```

numpy.polynomial.laguerre.**lagvander2d**(*x*, *y*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y)*. The pseudo-Vandermonde matrix is defined by

$$V[..., deg[1] * i + j] = L_i(x) * L_j(y),$$

where *0 <= i <= deg[0]* and *0 <= j <= deg[1]*. The leading indices of *V* index the points *(x, y)* and the last index encodes the degrees of the Laguerre polynomials.

If `V = lagvander2d(x, y, [xdeg, ydeg])`, then the columns of *V* correspond to the elements of a 2-D coefficient array *c* of shape (xdeg + 1, ydeg + 1) in the order

$$c_{00}, c_{01}, c_{02}..., c_{10}, c_{11}, c_{12}...$$

and `np.dot(V, c.flat)` and `lagval2d(x, y, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D Laguerre series of the same degrees and sample points.

Parameters

x, y : array_like

Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

deg : list of ints

List of maximum degrees of the form [x_deg, y_deg].

Returns

vander2d : ndarray

The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0] + 1) * (deg([1] + 1)$. The dtype will be the same as the converted *x* and *y*.

See also:

lagvander, `lagvander3d.`, *lagval3d*

Notes

`numpy.polynomial.laguerre.`**`lagvander3d`**(*x*, *y*, *z*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y, z)*. If *l, m, n* are the given degrees in *x, y, z*, then The pseudo-Vandermonde matrix is defined by

$$V[..., (m + 1)(n + 1)i + (n + 1)j + k] = L_i(x) * L_j(y) * L_k(z),$$

where *0 <= i <= l*, *0 <= j <= m*, and *0 <= j <= n*. The leading indices of *V* index the points *(x, y, z)* and the last index encodes the degrees of the Laguerre polynomials.

If `V = lagvander3d(x, y, z, [xdeg, ydeg, zdeg])`, then the columns of *V* correspond to the elements of a 3-D coefficient array *c* of shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order

$$c_{000}, c_{001}, c_{002}, ..., c_{010}, c_{011}, c_{012}, ...$$

and `np.dot(V, c.flat)` and `lagval3d(x, y, z, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D Laguerre series of the same degrees and sample points.

Parameters

x, y, z : array_like

Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

deg : list of ints

List of maximum degrees of the form [x_deg, y_deg, z_deg].

Returns

vander3d : ndarray

The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0]+1)*(deg([1]+1)*(deg[2]+1)$. The dtype will be the same as the converted *x*, *y*, and *z*.

See also:

lagvander, lagvander3d., *lagval3d*

Notes

lagder(c[, m, scl, axis])	Differentiate a Laguerre series.
lagint(c[, m, k, lbnd, scl, axis])	Integrate a Laguerre series.

Calculus

numpy.polynomial.laguerre.**lagder**(*c*, *m=1*, *scl=1*, *axis=0*)

Differentiate a Laguerre series.

Returns the Laguerre series coefficients *c* differentiated *m* times along *axis*. At each iteration the result is multiplied by *scl* (the scaling factor is for use in a linear change of variable). The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `1*L_0 + 2*L_1 + 3*L_2` while [[1,2],[1,2]] represents `1*L_0(x)*L_0(y) + 1*L_1(x)*L_0(y) + 2*L_0(x)*L_1(y) + 2*L_1(x)*L_1(y)` if axis=0 is `x` and axis=1 is `y`.

Parameters

c : array_like

Array of Laguerre series coefficients. If *c* is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

m : int, optional

Number of derivatives taken, must be non-negative. (Default: 1)

scl : scalar, optional

Each differentiation is multiplied by *scl*. The end result is multiplication by `scl**m`. This is for use in a linear change of variable. (Default: 1)

axis : int, optional

Axis over which the derivative is taken. (Default: 0).

New in version 1.7.0.

Returns

der : ndarray

Laguerre series of the derivative.

See also:

lagint

Notes

In general, the result of differentiating a Laguerre series does not resemble the same operation on a power series. Thus the result of this function may be "unintuitive," albeit correct; see Examples section below.

Examples

```
>>> from numpy.polynomial.laguerre import lagder
>>> lagder([ 1.,  1.,  1., -3.])
array([ 1.,  2.,  3.])
>>> lagder([ 1.,  0.,  0., -4.,  3.], m=2)
array([ 1.,  2.,  3.])
```

numpy.polynomial.laguerre.**lagint**(*c*, *m=1*, *k=[]*, *lbnd=0*, *scl=1*, *axis=0*)

Integrate a Laguerre series.

Returns the Laguerre series coefficients *c* integrated *m* times from *lbnd* along *axis*. At each iteration the resulting series is **multiplied** by *scl* and an integration constant, *k*, is added. The scaling factor is for use in a linear change of variable. ("Buyer beware": note that, depending on what one is doing, one may want *scl* to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `L_0 + 2*L_1 + 3*L_2` while [[1,2],[1,2]] represents `1*L_0(x)*L_0(y) + 1*L_1(x)*L_0(y) + 2*L_0(x)*L_1(y) + 2*L_1(x)*L_1(y)` if axis=0 is `x` and axis=1 is `y`.

Parameters

c : array_like

Array of Laguerre series coefficients. If *c* is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

m : int, optional

Order of integration, must be positive. (Default: 1)

k : {[], list, scalar}, optional

Integration constant(s). The value of the first integral at `lbnd` is the first value in the list, the value of the second integral at `lbnd` is the second value, etc. If `k == []` (the default), all constants are set to zero. If `m == 1`, a single scalar can be given instead of a list.

lbnd : scalar, optional

The lower bound of the integral. (Default: 0)

scl : scalar, optional

Following each integration the result is *multiplied* by *scl* before the integration constant is added. (Default: 1)

axis : int, optional

Axis over which the integral is taken. (Default: 0).

New in version 1.7.0.

Returns

S : ndarray

Laguerre series coefficients of the integral.

Raises

ValueError

If `m < 0`, `len(k) > m`, `np.isscalar(lbnd) == False`, or `np.isscalar(scl) == False`.

See also:

lagder

Notes

Note that the result of each integration is *multiplied* by *scl*. Why is this important to note? Say one is making a linear change of variable $u = ax + b$ in an integral relative to *x*. Then .. math::*dx* = *du*/*a*, so one will need to set *scl* equal to $1/a$ - perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be "reprojected" onto the C-series basis set. Thus, typically, the result of this function is "unintuitive," albeit correct; see Examples section below.

Examples

```
>>> from numpy.polynomial.laguerre import lagint
>>> lagint([1,2,3])
array([ 1.,  1.,  1., -3.])
>>> lagint([1,2,3], m=2)
array([ 1.,  0.,  0., -4.,  3.])
>>> lagint([1,2,3], k=1)
array([ 2.,  1.,  1., -3.])
>>> lagint([1,2,3], lbnd=-1)
array([ 11.5,   1. ,   1. ,  -3. ])
>>> lagint([1,2], m=2, k=[1,2], lbnd=-1)
array([ 11.16666667,  -5.        ,  -3.        ,   2.        ])
```

lagadd(c1, c2)	Add one Laguerre series to another.
lagsub(c1, c2)	Subtract one Laguerre series from another.
lagmul(c1, c2)	Multiply one Laguerre series by another.
lagmulx(c)	Multiply a Laguerre series by x.
lagdiv(c1, c2)	Divide one Laguerre series by another.
lagpow(c, pow[, maxpower])	Raise a Laguerre series to a power.

Algebra

numpy.polynomial.laguerre.**lagadd**(*c1*, *c2*)

Add one Laguerre series to another.

Returns the sum of two Laguerre series *c1* + *c2*. The arguments are sequences of coefficients ordered from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Laguerre series coefficients ordered from low to high.

Returns

out : ndarray

Array representing the Laguerre series of their sum.

See also:

lagsub, *lagmul*, *lagdiv*, *lagpow*

Notes

Unlike multiplication, division, etc., the sum of two Laguerre series is a Laguerre series (without having to "reproject" the result onto the basis set) so addition, just like that of "standard" polynomials, is simply "component-wise."

Examples

```
>>> from numpy.polynomial.laguerre import lagadd
>>> lagadd([1, 2, 3], [1, 2, 3, 4])
array([ 2.,  4.,  6.,  4.])
```

numpy.polynomial.laguerre.**lagsub**(*c1*, *c2*)

Subtract one Laguerre series from another.

Returns the difference of two Laguerre series *c1* - *c2*. The sequences of coefficients are from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Laguerre series coefficients ordered from low to high.

Returns

out : ndarray

Of Laguerre series coefficients representing their difference.

See also:

lagadd, *lagmul*, *lagdiv*, *lagpow*

Notes

Unlike multiplication, division, etc., the difference of two Laguerre series is a Laguerre series (without having to "reproject" the result onto the basis set) so subtraction, just like that of "standard" polynomials, is simply "component-wise."

Examples

```
>>> from numpy.polynomial.laguerre import lagsub
>>> lagsub([1, 2, 3, 4], [1, 2, 3])
array([ 0.,  0.,  0.,  4.])
```

numpy.polynomial.laguerre.**lagmul**(*c1*, *c2*)

Multiply one Laguerre series by another.

Returns the product of two Laguerre series *c1* * *c2*. The arguments are sequences of coefficients, from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Laguerre series coefficients ordered from low to high.

Returns

out : ndarray

Of Laguerre series coefficients representing their product.

See also:

lagadd, *lagsub*, *lagdiv*, *lagpow*

Notes

In general, the (polynomial) product of two C-series results in terms that are not in the Laguerre polynomial basis set. Thus, to express the product as a Laguerre series, it is necessary to "reproject" the product onto said basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

Examples

```
>>> from numpy.polynomial.laguerre import lagmul
>>> lagmul([1, 2, 3], [0, 1, 2])
array([  8., -13.,  38., -51.,  36.])
```

numpy.polynomial.laguerre.**lagmulx**(*c*)

Multiply a Laguerre series by x.

Multiply the Laguerre series *c* by x, where x is the independent variable.

Parameters

c : array_like

1-D array of Laguerre series coefficients ordered from low to high.

Returns

out : ndarray

Array representing the result of the multiplication.

Notes

The multiplication uses the recursion relationship for Laguerre polynomials in the form

xP_i(x) = (-(i + 1)*P_{i + 1}(x) + (2i + 1)P_{i}(x) - iP_{i - 1}(x))

Examples

```
>>> from numpy.polynomial.laguerre import lagmulx
>>> lagmulx([1, 2, 3])
array([ -1.,  -1.,  11.,  -9.])
```

numpy.polynomial.laguerre.**lagdiv**(*c1*, *c2*)

Divide one Laguerre series by another.

Returns the quotient-with-remainder of two Laguerre series *c1* / *c2*. The arguments are sequences of coefficients from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Laguerre series coefficients ordered from low to high.

Returns

[quo, rem] : ndarrays

Of Laguerre series coefficients representing the quotient and remainder.

See also:

lagadd, *lagsub*, *lagmul*, *lagpow*

Notes

In general, the (polynomial) division of one Laguerre series by another results in quotient and remainder terms that are not in the Laguerre polynomial basis set. Thus, to express these results as a Laguerre series, it is

necessary to "reproject" the results onto the Laguerre basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

Examples

```
>>> from numpy.polynomial.laguerre import lagdiv
>>> lagdiv([  8., -13.,  38., -51.,  36.], [0, 1, 2])
(array([ 1.,  2.,  3.]), array([ 0.]))
>>> lagdiv([  9., -12.,  38., -51.,  36.], [0, 1, 2])
(array([ 1.,  2.,  3.]), array([ 1.,  1.]))
```

numpy.polynomial.laguerre.**lagpow**(*c*, *pow*, *maxpower=16*)

Raise a Laguerre series to a power.

Returns the Laguerre series *c* raised to the power *pow*. The argument *c* is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series `P_0 + 2*P_1 + 3*P_2.`

Parameters

c : array_like

1-D array of Laguerre series coefficients ordered from low to high.

pow : integer

Power to which the series will be raised

maxpower : integer, optional

Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

Returns

coef : ndarray

Laguerre series of power.

See also:

lagadd, *lagsub*, *lagmul*, *lagdiv*

Examples

```
>>> from numpy.polynomial.laguerre import lagpow
>>> lagpow([1, 2, 3], 2)
array([ 14., -16.,  56., -72.,  54.])
```

laggauss(deg)	Gauss-Laguerre quadrature.
lagweight(x)	Weight function of the Laguerre polynomials.

Quadrature

numpy.polynomial.laguerre.**laggauss**(*deg*)

Gauss-Laguerre quadrature.

Computes the sample points and weights for Gauss-Laguerre quadrature. These sample points and weights will correctly integrate polynomials of degree $2 * deg - 1$ or less over the interval $[0, \inf]$ with the weight function $f(x) = \exp(-x)$.

Parameters

deg : int

Number of sample points and weights. It must be >= 1.

Returns

x : ndarray

1-D ndarray containing the sample points.

y : ndarray

1-D ndarray containing the weights.

Notes

The results have only been tested up to degree 100 higher degrees may be problematic. The weights are determined by using the fact that

$$w_k = c/(L'_n(x_k) * L_{n-1}(x_k))$$

where c is a constant independent of k and x_k is the k'th root of L_n, and then scaling the results to get the right value when integrating 1.

numpy.polynomial.laguerre.**lagweight**(x)

Weight function of the Laguerre polynomials.

The weight function is $exp(-x)$ and the interval of integration is $[0, \inf]$. The Laguerre polynomials are orthogonal, but not normalized, with respect to this weight function.

Parameters

x : array_like

Values at which the weight function will be computed.

Returns

w : ndarray

The weight function at *x*.

Notes

lagcompanion(c)	Return the companion matrix of c.
lagdomain	
lagzero	
lagone	
lagx	
lagline(off, scl)	Laguerre series whose graph is a straight line.
lag2poly(c)	Convert a Laguerre series to a polynomial.
poly2lag(pol)	Convert a polynomial to a Laguerre series.

Miscellaneous

numpy.polynomial.laguerre.**lagcompanion**(c)

Return the companion matrix of c.

The usual companion matrix of the Laguerre polynomials is already symmetric when c is a basis Laguerre polynomial, so no scaling is applied.

Parameters

c : array_like

1-D array of Laguerre series coefficients ordered from low to high degree.

Returns

mat : ndarray

Companion matrix of dimensions (deg, deg).

Notes

numpy.polynomial.laguerre.**lagdomain** = array([0, 1])

numpy.polynomial.laguerre.**lagzero** = array([0])

numpy.polynomial.laguerre.**lagone** = array([1])

numpy.polynomial.laguerre.**lagx** = array([1, -1])

numpy.polynomial.laguerre.**lagline**(*off*, *scl*)

Laguerre series whose graph is a straight line.

Parameters

off, scl : scalars

The specified line is given by `off + scl*x`.

Returns

y : ndarray

This module's representation of the Laguerre series for `off + scl*x`.

See also:

`polyline`, `chebline`

Examples

```
>>> from numpy.polynomial.laguerre import lagline, lagval
>>> lagval(0,lagline(3, 2))
3.0
>>> lagval(1,lagline(3, 2))
5.0
```

numpy.polynomial.laguerre.**lag2poly**(*c*)

Convert a Laguerre series to a polynomial.

Convert an array representing the coefficients of a Laguerre series, ordered from lowest degree to highest, to an array of the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest to highest degree.

Parameters

c : array_like

1-D array containing the Laguerre series coefficients, ordered from lowest order term to highest.

Returns

pol : ndarray

1-D array containing the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest order term to highest.

See also:

poly2lag

Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```
>>> from numpy.polynomial.laguerre import lag2poly
>>> lag2poly([ 23., -63.,  58., -18.])
array([ 0.,  1.,  2.,  3.])
```

numpy.polynomial.laguerre.**poly2lag**(*pol*)

Convert a polynomial to a Laguerre series.

Convert an array representing the coefficients of a polynomial (relative to the "standard" basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Laguerre series, ordered from lowest to highest degree.

Parameters

pol : array_like

1-D array containing the polynomial coefficients

Returns

c : ndarray

1-D array containing the coefficients of the equivalent Laguerre series.

See also:

lag2poly

Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```
>>> from numpy.polynomial.laguerre import poly2lag
>>> poly2lag(np.arange(4))
array([ 23., -63.,  58., -18.])
```

Hermite Module, "Physicists"' (numpy.polynomial.hermite)

New in version 1.6.0.

This module provides a number of objects (mostly functions) useful for dealing with Hermite series, including a *Hermite* class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its "parent" sub-package, numpy.polynomial).

Hermite(coef[, domain, window])	An Hermite series class.

Hermite Class

class numpy.polynomial.hermite.**Hermite**(*coef*, *domain=None*, *window=None*)

An Hermite series class.

The Hermite class provides the standard Python numerical methods '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the attributes and methods listed in the ABCPolyBase documentation.

Parameters

coef : array_like

Hermite coefficients in order of increasing degree, i.e, (1, 2, 3) gives 1*H_0(x) + 2*H_1(X) + 3*H_2(x).

domain : (2,) array_like, optional

Domain to use. The interval `[domain[0], domain[1]]` is mapped to the interval `[window[0], window[1]]` by shifting and scaling. The default value is [-1, 1].

window : (2,) array_like, optional

Window, see `domain` for its use. The default value is [-1, 1].

New in version 1.6.0.

Methods

hermval(x, c[, tensor])	Evaluate an Hermite series at points x.
hermval2d(x, y, c)	Evaluate a 2-D Hermite series at points (x, y).
hermval3d(x, y, z, c)	Evaluate a 3-D Hermite series at points (x, y, z).
hermgrid2d(x, y, c)	Evaluate a 2-D Hermite series on the Cartesian product of x and y.
hermgrid3d(x, y, z, c)	Evaluate a 3-D Hermite series on the Cartesian product of x, y, and z.
hermroots(c)	Compute the roots of a Hermite series.
hermfromroots(roots)	Generate a Hermite series with given roots.

Basics

numpy.polynomial.hermite.**hermval**(*x*, *c*, *tensor=True*)

Evaluate an Hermite series at points x.

If *c* is of length *n + 1*, this function returns the value:

$$p(x) = c_0 * H_0(x) + c_1 * H_1(x) + ... + c_n * H_n(x)$$

The parameter *x* is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either *x* or its elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array, then *p(x)* will have the same shape as *x*. If *c* is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor* is true the shape will be c.shape[1:] + x.shape. If *tensor* is false the shape will be c.shape[1:]. Note that scalars have shape (,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

Parameters

x : array_like, compatible object

If *x* is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, *x* or its elements must support addition and multiplication with with themselves and with the elements of *c*.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree n are contained in c[n]. If *c* is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of *c*.

tensor : boolean, optional

If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of *x*. Scalars have dimension 0 for this action. The result is that every column

of coefficients in *c* is evaluated for every element of *x*. If False, *x* is broadcast over the columns of *c* for the evaluation. This keyword is useful when *c* is multidimensional. The default value is True.

New in version 1.7.0.

Returns

values : ndarray, algebra_like

The shape of the return value is described above.

See also:

hermval2d, *hermgrid2d*, *hermval3d*, *hermgrid3d*

Notes

The evaluation uses Clenshaw recursion, aka synthetic division.

Examples

```
>>> from numpy.polynomial.hermite import hermval
>>> coef = [1,2,3]
>>> hermval(1, coef)
11.0
>>> hermval([[1,2],[3,4]], coef)
array([[  11.,   51.],
       [ 115.,  203.]])
```

numpy.polynomial.hermite.**hermval2d**(*x*, *y*, *c*)

Evaluate a 2-D Hermite series at points (x, y).

This function returns the values:

$$p(x,y) = \sum_{i,j} c_{i,j} * H_i(x) * H_j(y)$$

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array a one is implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

Parameters

x, y : array_like, compatible objects

The two dimensional series is evaluated at the points *(x, y)*, where *x* and *y* must have the same shape. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional polynomial at points formed with pairs of corresponding values from *x* and *y*.

See also:

hermval, *hermgrid2d*, *hermval3d*, *hermgrid3d*

Notes

numpy.polynomial.hermite.**hermval3d**(*x*, *y*, *z*, *c*)

Evaluate a 3-D Hermite series at points (x, y, z).

This function returns the values:

$$p(x, y, z) = \sum_{i,j,k} c_{i,j,k} * H_i(x) * H_j(y) * H_k(z)$$

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape.

Parameters

x, y, z : array_like, compatible object

The three dimensional series is evaluated at the points *(x, y, z)*, where *x*, *y*, and *z* must have the same shape. If any of *x*, *y*, or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j,k is contained in `c[i,j,k]`. If *c* has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the multidimensional polynomial on points formed with triples of corresponding values from *x*, *y*, and *z*.

See also:

hermval, *hermval2d*, *hermgrid2d*, *hermgrid3d*

Notes

numpy.polynomial.hermite.**hermgrid2d**(*x*, *y*, *c*)

Evaluate a 2-D Hermite series on the Cartesian product of x and y.

This function returns the values:

$$p(a, b) = \sum_{i,j} c_{i,j} * H_i(a) * H_j(b)$$

where the points *(a, b)* consist of all pairs formed by taking *a* from *x* and *b* from *y*. The resulting points form a grid with *x* in the first dimension and *y* in the second.

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

Parameters

x, y : array_like, compatible objects

The two dimensional series is evaluated at the points in the Cartesian product of *x* and *y*. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

See also:

hermval, *hermval2d*, *hermval3d*, *hermgrid3d*

Notes

numpy.polynomial.hermite.**hermgrid3d**(*x*, *y*, *z*, *c*)

Evaluate a 3-D Hermite series on the Cartesian product of x, y, and z.

This function returns the values:

$$p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * H_i(a) * H_j(b) * H_k(c)$$

where the points *(a, b, c)* consist of all triples formed by taking *a* from *x*, *b* from *y*, and *c* from *z*. The resulting points form a grid with *x* in the first dimension, *y* in the second, and *z* in the third.

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape + y.shape + z.shape.

Parameters

x, y, z : array_like, compatible objects

The three dimensional series is evaluated at the points in the Cartesian product of *x*, *y*, and *z*. If *x*,'y', or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

See also:

hermval, *hermval2d*, *hermgrid2d*, *hermval3d*

Notes

numpy.polynomial.hermite.**hermroots**(*c*)

Compute the roots of a Hermite series.

Return the roots (a.k.a. "zeros") of the polynomial

$$p(x) = \sum_i c[i] * H_i(x).$$

Parameters

c : 1-D array_like

1-D array of coefficients.

Returns

out : ndarray

Array of the roots of the series. If all the roots are real, then *out* is also real, otherwise it is complex.

See also:

`polyroots`, `legroots`, `lagroots`, `chebroots`, `hermeroots`

Notes

The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the complex plane may have large errors due to the numerical instability of the series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton's method.

The Hermite series basis polynomials aren't powers of *x* so the results of this function may seem unintuitive.

Examples

```
>>> from numpy.polynomial.hermite import hermroots, hermfromroots
>>> coef = hermfromroots([-1, 0, 1])
>>> coef
array([ 0.   ,  0.25 ,  0.   ,  0.125])
>>> hermroots(coef)
array([ -1.00000000e+00,  -1.38777878e-17,   1.00000000e+00])
```

numpy.polynomial.hermite.**hermfromroots**(*roots*)

Generate a Hermite series with given roots.

The function returns the coefficients of the polynomial

$$p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),$$

in Hermite form, where the *r_n* are the roots specified in *roots*. If a zero has multiplicity n, then it must appear in *roots* n times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then *roots* looks something like [2, 2, 2, 3, 3]. The roots can appear in any order.

If the returned coefficients are *c*, then

$$p(x) = c_0 + c_1 * H_1(x) + ... + c_n * H_n(x)$$

The coefficient of the last term is not generally 1 for monic polynomials in Hermite form.

Parameters

roots : array_like

Sequence containing the roots.

Returns

out : ndarray

1-D array of coefficients. If all roots are real then *out* is a real array, if some of the roots are complex, then *out* is complex even if all the coefficients in the result are real (see Examples below).

See also:

`polyfromroots`, `legfromroots`, `lagfromroots`, `chebfromroots`, `hermefromroots`.

Examples

```
>>> from numpy.polynomial.hermite import hermfromroots, hermval
>>> coef = hermfromroots((-1, 0, 1))
>>> hermval((-1, 0, 1), coef)
array([ 0.,  0.,  0.])
>>> coef = hermfromroots((-1j, 1j))
>>> hermval((-1j, 1j), coef)
array([ 0.+0.j,  0.+0.j])
```

hermfit(x, y, deg[, rcond, full, w])	Least squares fit of Hermite series to data.
hermvander(x, deg)	Pseudo-Vandermonde matrix of given degree.
hermvander2d(x, y, deg)	Pseudo-Vandermonde matrix of given degrees.
hermvander3d(x, y, z, deg)	Pseudo-Vandermonde matrix of given degrees.

Fitting

`numpy.polynomial.hermite.`**`hermfit`**(*x*, *y*, *deg*, *rcond=None*, *full=False*, *w=None*)

Least squares fit of Hermite series to data.

Return the coefficients of a Hermite series of degree *deg* that is the least squares fit to the data values *y* given at points *x*. If *y* is 1-D the returned coefficients will also be 1-D. If *y* is 2-D multiple fits are done, one for each column of *y*, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

$$p(x) = c_0 + c_1 * H_1(x) + ... + c_n * H_n(x),$$

where *n* is *deg*.

Parameters

x : array_like, shape (M,)

x-coordinates of the M sample points `(x[i], y[i])`.

y : array_like, shape (M,) or (M, K)

y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

deg : int or 1-D array_like

Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For Numpy versions >= 1.11 a list of integers specifying the degrees of the terms to include may be used instead.

rcond : float, optional

Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

full : bool, optional

Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

w : array_like, shape (*M*,), optional

Weights. If not None, the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

Returns

coef : ndarray, shape (M,) or (M, K)

Hermite coefficients ordered from low to high. If *y* was 2-D, the coefficients for the data in column k of *y* are in column *k*.

[residuals, rank, singular_values, rcond] : list

These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

Warns

RankWarning

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if *full* = False. The warnings can be turned off by

```
>>> import warnings
>>> warnings.simplefilter('ignore', RankWarning)
```

See also:

`chebfit`, `legfit`, `lagfit`, `polyfit`, `hermefit`

hermval
Evaluates a Hermite series.

hermvander
Vandermonde matrix of Hermite series.

hermweight
Hermite weight function

linalg.lstsq
Computes a least-squares fit from the matrix.

scipy.interpolate.UnivariateSpline
Computes spline fits.

Notes

The solution is the coefficients of the Hermite series *p* that minimizes the sum of the weighted squared errors

$$E = \sum_j w_j^2 * |y_j - p(x_j)|^2,$$

where the w_j are the weights. This problem is solved by setting up the (typically) overdetermined matrix equation

$$V(x) * c = w * y,$$

where *V* is the weighted pseudo Vandermonde matrix of *x*, *c* are the coefficients to be solved for, *w* are the weights, *y* are the observed values. This equation is then solved using the singular value decomposition of *V*.

If some of the singular values of *V* are so small that they are neglected, then a *RankWarning* will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The *rcond* parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using Hermite series are probably most useful when the data can be approximated by `sqrt(w(x)) * p(x)`, where *w(x)* is the Hermite weight. In that case the weight `sqrt(w(x[i])` should be used together with data values `y[i]/sqrt(w(x[i])`. The weight function is available as *hermweight*.

References

[R61]

Examples

```
>>> from numpy.polynomial.hermite import hermfit, hermval
>>> x = np.linspace(-10, 10)
>>> err = np.random.randn(len(x))/10
>>> y = hermval(x, [1, 2, 3]) + err
>>> hermfit(x, y, 2)
array([ 0.97902637,  1.99849131,  3.00006   ])
```

numpy.polynomial.hermite.**hermvander**(*x*, *deg*)

Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree *deg* and sample points *x*. The pseudo-Vandermonde matrix is defined by

$$V[..., i] = H_i(x),$$

where *0 <= i <= deg*. The leading indices of *V* index the elements of *x* and the last index is the degree of the Hermite polynomial.

If *c* is a 1-D array of coefficients of length *n + 1* and *V* is the array `V = hermvander(x, n)`, then `np.dot(V, c)` and `hermval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of Hermite series of the same degree and sample points.

Parameters

x : array_like

Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If *x* is scalar it is converted to a 1-D array.

deg : int

Degree of the resulting matrix.

Returns

vander : ndarray

The pseudo-Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where The last index is the degree of the corresponding Hermite polynomial. The dtype will be the same as the converted *x*.

Examples

```
>>> from numpy.polynomial.hermite import hermvander
>>> x = np.array([-1, 0, 1])
>>> hermvander(x, 3)
array([[ 1., -2.,  2.,  4.],
       [ 1.,  0., -2., -0.],
       [ 1.,  2.,  2., -4.]])
```

numpy.polynomial.hermite.**hermvander2d**(*x*, *y*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y)*. The pseudo-Vandermonde matrix is defined by

$$V[..., deg[1] * i + j] = H_i(x) * H_j(y),$$

where *0 <= i <= deg[0]* and *0 <= j <= deg[1]*. The leading indices of *V* index the points *(x, y)* and the last index encodes the degrees of the Hermite polynomials.

If `V = hermvander2d(x, y, [xdeg, ydeg])`, then the columns of *V* correspond to the elements of a 2-D coefficient array *c* of shape (xdeg + 1, ydeg + 1) in the order

$$c_{00}, c_{01}, c_{02}..., c_{10}, c_{11}, c_{12}...$$

and `np.dot(V, c.flat)` and `hermval2d(x, y, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D Hermite series of the same degrees and sample points.

Parameters

x, y : array_like

Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

deg : list of ints

List of maximum degrees of the form [x_deg, y_deg].

Returns

vander2d : ndarray

The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0] + 1) * (deg([1] + 1)$. The dtype will be the same as the converted *x* and *y*.

See also:

hermvander, `hermvander3d.`, *hermval3d*

Notes

numpy.polynomial.hermite.**hermvander3d**(*x*, *y*, *z*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y, z)*. If *l, m, n* are the given degrees in *x, y, z*, then The pseudo-Vandermonde matrix is defined by

$$V[..., (m+1)(n+1)i + (n+1)j + k] = H_i(x) * H_j(y) * H_k(z),$$

where *0 <= i <= l*, *0 <= j <= m*, and *0 <= j <= n*. The leading indices of *V* index the points *(x, y, z)* and the last index encodes the degrees of the Hermite polynomials.

If `V = hermvander3d(x, y, z, [xdeg, ydeg, zdeg])`, then the columns of *V* correspond to the elements of a 3-D coefficient array *c* of shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order

$$c_{000}, c_{001}, c_{002}, ..., c_{010}, c_{011}, c_{012}, ...$$

and `np.dot(V, c.flat)` and `hermval3d(x, y, z, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D Hermite series of the same degrees and sample points.

Parameters

x, y, z : array_like

Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

deg : list of ints

List of maximum degrees of the form [x_deg, y_deg, z_deg].

Returns

vander3d : ndarray

The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0]+1) * (deg([1]+1) * (deg[2]+1)$. The dtype will be the same as the converted *x*, *y*, and *z*.

See also:

hermvander, `hermvander3d.`, *hermval3d*

Notes

hermder(c[, m, scl, axis])	Differentiate a Hermite series.
hermint(c[, m, k, lbnd, scl, axis])	Integrate a Hermite series.

Calculus

`numpy.polynomial.hermite.`**`hermder`**(*c*, *m=1*, *scl=1*, *axis=0*)

Differentiate a Hermite series.

Returns the Hermite series coefficients *c* differentiated *m* times along *axis*. At each iteration the result is multiplied by *scl* (the scaling factor is for use in a linear change of variable). The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `1*H_0 + 2*H_1 + 3*H_2` while [[1,2],[1,2]] represents `1*H_0(x)*H_0(y) + 1*H_1(x)*H_0(y) + 2*H_0(x)*H_1(y) + 2*H_1(x)*H_1(y)` if axis=0 is `x` and axis=1 is `y`.

Parameters

c : array_like

Array of Hermite series coefficients. If *c* is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

m : int, optional

Number of derivatives taken, must be non-negative. (Default: 1)

scl : scalar, optional

Each differentiation is multiplied by *scl*. The end result is multiplication by `scl**m`. This is for use in a linear change of variable. (Default: 1)

axis : int, optional

Axis over which the derivative is taken. (Default: 0).

New in version 1.7.0.

Returns

der : ndarray

Hermite series of the derivative.

See also:

hermint

Notes

In general, the result of differentiating a Hermite series does not resemble the same operation on a power series. Thus the result of this function may be "unintuitive," albeit correct; see Examples section below.

Examples

```
>>> from numpy.polynomial.hermite import hermder
>>> hermder([ 1. ,  0.5,  0.5,  0.5])
array([ 1.,  2.,  3.])
>>> hermder([-0.5,  1./2.,  1./8.,  1./12.,  1./16.], m=2)
array([ 1.,  2.,  3.])
```

numpy.polynomial.hermite.**hermint**(*c*, *m=1*, *k=[]*, *lbnd=0*, *scl=1*, *axis=0*)

Integrate a Hermite series.

Returns the Hermite series coefficients *c* integrated *m* times from *lbnd* along *axis*. At each iteration the resulting series is **multiplied** by *scl* and an integration constant, *k*, is added. The scaling factor is for use in a linear change of variable. ("Buyer beware": note that, depending on what one is doing, one may want *scl* to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `H_0 + 2*H_1 + 3*H_2` while [[1,2],[1,2]] represents `1*H_0(x)*H_0(y) + 1*H_1(x)*H_0(y) + 2*H_0(x)*H_1(y) + 2*H_1(x)*H_1(y)` if axis=0 is `x` and axis=1 is `y`.

Parameters

c : array_like

Array of Hermite series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

m : int, optional

Order of integration, must be positive. (Default: 1)

k : {[], list, scalar}, optional

Integration constant(s). The value of the first integral at `lbnd` is the first value in the list, the value of the second integral at `lbnd` is the second value, etc. If `k == []` (the default), all constants are set to zero. If `m == 1`, a single scalar can be given instead of a list.

lbnd : scalar, optional

The lower bound of the integral. (Default: 0)

scl : scalar, optional

Following each integration the result is *multiplied* by *scl* before the integration constant is added. (Default: 1)

axis : int, optional

Axis over which the integral is taken. (Default: 0).

New in version 1.7.0.

Returns

S : ndarray

Hermite series coefficients of the integral.

Raises

ValueError

If `m < 0`, `len(k) > m`, `np.isscalar(lbnd) == False`, or `np.isscalar(scl) == False`.

See also:

hermder

Notes

Note that the result of each integration is *multiplied* by *scl*. Why is this important to note? Say one is making a linear change of variable $u = ax + b$ in an integral relative to *x*. Then .. math::*dx* = *du*/*a*, so one will need to set *scl* equal to $1/a$ - perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be "reprojected" onto the C-series basis set. Thus, typically, the result of this function is "unintuitive," albeit correct; see Examples section below.

Examples

```
>>> from numpy.polynomial.hermite import hermint
>>> hermint([1,2,3]) # integrate once, value 0 at 0.
array([ 1. ,  0.5,  0.5,  0.5])
>>> hermint([1,2,3], m=2) # integrate twice, value & deriv 0 at 0
array([-0.5       ,  0.5       ,  0.125     ,  0.08333333,  0.0625    ])
>>> hermint([1,2,3], k=1) # integrate once, value 1 at 0.
array([ 2. ,  0.5,  0.5,  0.5])
>>> hermint([1,2,3], lbnd=-1) # integrate once, value 0 at -1
array([-2. ,  0.5,  0.5,  0.5])
>>> hermint([1,2,3], m=2, k=[1,2], lbnd=-1)
array([ 1.66666667, -0.5       ,  0.125     ,  0.08333333,  0.0625    ])
```

hermadd(c1, c2)	Add one Hermite series to another.
hermsub(c1, c2)	Subtract one Hermite series from another.
hermmul(c1, c2)	Multiply one Hermite series by another.
hermmulx(c)	Multiply a Hermite series by x.
hermdiv(c1, c2)	Divide one Hermite series by another.
hermpow(c, pow[, maxpower])	Raise a Hermite series to a power.

Algebra

numpy.polynomial.hermite.**hermadd**(*c1*, *c2*)

Add one Hermite series to another.

Returns the sum of two Hermite series *c1* + *c2*. The arguments are sequences of coefficients ordered from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Hermite series coefficients ordered from low to high.

Returns

out : ndarray

Array representing the Hermite series of their sum.

See also:

hermsub, *hermmul*, *hermdiv*, *hermpow*

Notes

Unlike multiplication, division, etc., the sum of two Hermite series is a Hermite series (without having to "re-project" the result onto the basis set) so addition, just like that of "standard" polynomials, is simply "component-wise."

Examples

```
>>> from numpy.polynomial.hermite import hermadd
>>> hermadd([1, 2, 3], [1, 2, 3, 4])
array([ 2.,  4.,  6.,  4.])
```

numpy.polynomial.hermite.**hermsub**(*c1*, *c2*)

Subtract one Hermite series from another.

Returns the difference of two Hermite series *c1* - *c2*. The sequences of coefficients are from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Hermite series coefficients ordered from low to high.

Returns

out : ndarray

Of Hermite series coefficients representing their difference.

See also:

hermadd, *hermmul*, *hermdiv*, *hermpow*

Notes

Unlike multiplication, division, etc., the difference of two Hermite series is a Hermite series (without having to "reproject" the result onto the basis set) so subtraction, just like that of "standard" polynomials, is simply "component-wise."

Examples

```
>>> from numpy.polynomial.hermite import hermsub
>>> hermsub([1, 2, 3, 4], [1, 2, 3])
array([ 0.,  0.,  0.,  4.])
```

numpy.polynomial.hermite.**hermmul**(*c1*, *c2*)
Multiply one Hermite series by another.

Returns the product of two Hermite series *c1* * *c2*. The arguments are sequences of coefficients, from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters
c1, c2 : array_like

1-D arrays of Hermite series coefficients ordered from low to high.

Returns
out : ndarray

Of Hermite series coefficients representing their product.

See also:

hermadd, *hermsub*, *hermdiv*, *hermpow*

Notes

In general, the (polynomial) product of two C-series results in terms that are not in the Hermite polynomial basis set. Thus, to express the product as a Hermite series, it is necessary to "reproject" the product onto said basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

Examples

```
>>> from numpy.polynomial.hermite import hermmul
>>> hermmul([1, 2, 3], [0, 1, 2])
array([ 52.,  29.,  52.,   7.,   6.])
```

numpy.polynomial.hermite.**hermmulx**(*c*)
Multiply a Hermite series by x.

Multiply the Hermite series *c* by x, where x is the independent variable.

Parameters
c : array_like

1-D array of Hermite series coefficients ordered from low to high.

Returns
out : ndarray

Array representing the result of the multiplication.

Notes

The multiplication uses the recursion relationship for Hermite polynomials in the form

xP_i(x) = (P_{i + 1}(x)/2 + i*P_{i - 1}(x))

Examples

```
>>> from numpy.polynomial.hermite import hermmulx
>>> hermmulx([1, 2, 3])
array([ 2. ,  6.5,  1. ,  1.5])
```

numpy.polynomial.hermite.**hermdiv**(*c1*, *c2*)
Divide one Hermite series by another.

Returns the quotient-with-remainder of two Hermite series *c1* / *c2*. The arguments are sequences of coefficients from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Hermite series coefficients ordered from low to high.

Returns

[quo, rem] : ndarrays

Of Hermite series coefficients representing the quotient and remainder.

See also:

hermadd, *hermsub*, *hermmul*, *hermpow*

Notes

In general, the (polynomial) division of one Hermite series by another results in quotient and remainder terms that are not in the Hermite polynomial basis set. Thus, to express these results as a Hermite series, it is necessary to "reproject" the results onto the Hermite basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

Examples

```
>>> from numpy.polynomial.hermite import hermdiv
>>> hermdiv([ 52.,  29.,  52.,   7.,   6.], [0, 1, 2])
(array([ 1.,  2.,  3.]), array([ 0.]))
>>> hermdiv([ 54.,  31.,  52.,   7.,   6.], [0, 1, 2])
(array([ 1.,  2.,  3.]), array([ 2.,  2.]))
>>> hermdiv([ 53.,  30.,  52.,   7.,   6.], [0, 1, 2])
(array([ 1.,  2.,  3.]), array([ 1.,  1.]))
```

numpy.polynomial.hermite.**hermpow**(*c*, *pow*, *maxpower=16*)

Raise a Hermite series to a power.

Returns the Hermite series *c* raised to the power *pow*. The argument *c* is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c : array_like

1-D array of Hermite series coefficients ordered from low to high.

pow : integer

Power to which the series will be raised

maxpower : integer, optional

Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

Returns

coef : ndarray

Hermite series of power.

See also:

hermadd, *hermsub*, *hermmul*, *hermdiv*

Examples

```
>>> from numpy.polynomial.hermite import hermpow
>>> hermpow([1, 2, 3], 2)
array([ 81.,  52.,  82.,  12.,   9.])
```

hermgauss(deg)	Gauss-Hermite quadrature.
hermweight(x)	Weight function of the Hermite polynomials.

Quadrature

numpy.polynomial.hermite.**hermgauss**(*deg*)

Gauss-Hermite quadrature.

Computes the sample points and weights for Gauss-Hermite quadrature. These sample points and weights will correctly integrate polynomials of degree $2*deg-1$ or less over the interval $[-\inf, \inf]$ with the weight function $f(x) = \exp(-x^2)$.

Parameters

deg : int

Number of sample points and weights. It must be >= 1.

Returns

x : ndarray

1-D ndarray containing the sample points.

y : ndarray

1-D ndarray containing the weights.

Notes

The results have only been tested up to degree 100, higher degrees may be problematic. The weights are determined by using the fact that

$$w_k = c/(H'_n(x_k) * H_{n-1}(x_k))$$

where c is a constant independent of k and x_k is the k'th root of H_n, and then scaling the results to get the right value when integrating 1.

numpy.polynomial.hermite.**hermweight**(*x*)

Weight function of the Hermite polynomials.

The weight function is $\exp(-x^2)$ and the interval of integration is $[-\inf, \inf]$. the Hermite polynomials are orthogonal, but not normalized, with respect to this weight function.

Parameters

x : array_like

Values at which the weight function will be computed.

Returns

w : ndarray

The weight function at *x*.

Notes

hermcompanion(c)	Return the scaled companion matrix of c.
hermdomain	

Continued on next page

Table 3.157 – continued from previous page

hermzero	
hermone	
hermx	
hermline(off, scl)	Hermite series whose graph is a straight line.
herm2poly(c)	Convert a Hermite series to a polynomial.
poly2herm(pol)	Convert a polynomial to a Hermite series.

Miscellaneous

numpy.polynomial.hermite.**hermcompanion**(*c*)

Return the scaled companion matrix of c.

The basis polynomials are scaled so that the companion matrix is symmetric when *c* is an Hermite basis polynomial. This provides better eigenvalue estimates than the unscaled case and for basis polynomials the eigenvalues are guaranteed to be real if `numpy.linalg.eigvalsh` is used to obtain them.

Parameters

c : array_like

1-D array of Hermite series coefficients ordered from low to high degree.

Returns

mat : ndarray

Scaled companion matrix of dimensions (deg, deg).

Notes

numpy.polynomial.hermite.**hermdomain = array([-1, 1])**

numpy.polynomial.hermite.**hermzero = array([0])**

numpy.polynomial.hermite.**hermone = array([1])**

numpy.polynomial.hermite.**hermx = array([0. , 0.5])**

numpy.polynomial.hermite.**hermline**(*off*, *scl*)

Hermite series whose graph is a straight line.

Parameters

off, scl : scalars

The specified line is given by `off + scl*x`.

Returns

y : ndarray

This module's representation of the Hermite series for `off + scl*x`.

See also:

`polyline`, `chebline`

Examples

```
>>> from numpy.polynomial.hermite import hermline, hermval
>>> hermval(0,hermline(3, 2))
3.0
```

```
    >>> hermval(1,hermline(3, 2))
    5.0
```

numpy.polynomial.hermite.**herm2poly**(*c*)

Convert a Hermite series to a polynomial.

Convert an array representing the coefficients of a Hermite series, ordered from lowest degree to highest, to an array of the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest to highest degree.

Parameters

c : array_like

1-D array containing the Hermite series coefficients, ordered from lowest order term to highest.

Returns

pol : ndarray

1-D array containing the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest order term to highest.

See also:

poly2herm

Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```
    >>> from numpy.polynomial.hermite import herm2poly
    >>> herm2poly([ 1.    ,  2.75 ,  0.5  ,  0.375])
    array([ 0.,  1.,  2.,  3.])
```

numpy.polynomial.hermite.**poly2herm**(*pol*)

Convert a polynomial to a Hermite series.

Convert an array representing the coefficients of a polynomial (relative to the "standard" basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Hermite series, ordered from lowest to highest degree.

Parameters

pol : array_like

1-D array containing the polynomial coefficients

Returns

c : ndarray

1-D array containing the coefficients of the equivalent Hermite series.

See also:

herm2poly

Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```
>>> from numpy.polynomial.hermite import poly2herm
>>> poly2herm(np.arange(4))
array([ 1.   ,  2.75 ,  0.5  ,  0.375])
```

HermiteE Module, "Probabilists"' (`numpy.polynomial.hermite_e`)

New in version 1.6.0.

This module provides a number of objects (mostly functions) useful for dealing with HermiteE series, including a *HermiteE* class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its "parent" sub-package, `numpy.polynomial`).

HermiteE(coef[, domain, window])	An HermiteE series class.

HermiteE Class

class `numpy.polynomial.hermite_e.`**HermiteE**(*coef*, *domain=None*, *window=None*)

An HermiteE series class.

The HermiteE class provides the standard Python numerical methods '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the attributes and methods listed in the `ABCPolyBase` documentation.

Parameters

coef : array_like

HermiteE coefficients in order of increasing degree, i.e, `(1, 2, 3)` gives `1*He_0(x) + 2*He_1(X) + 3*He_2(x)`.

domain : (2,) array_like, optional

Domain to use. The interval `[domain[0], domain[1]]` is mapped to the interval `[window[0], window[1]]` by shifting and scaling. The default value is [-1, 1].

window : (2,) array_like, optional

Window, see `domain` for its use. The default value is [-1, 1].

New in version 1.6.0.

Methods

hermeval(x, c[, tensor])	Evaluate an HermiteE series at points x.
hermeval2d(x, y, c)	Evaluate a 2-D HermiteE series at points (x, y).
hermeval3d(x, y, z, c)	Evaluate a 3-D Hermite_e series at points (x, y, z).
hermegrid2d(x, y, c)	Evaluate a 2-D HermiteE series on the Cartesian product of x and y.
hermegrid3d(x, y, z, c)	Evaluate a 3-D HermiteE series on the Cartesian product of x, y, and z.
hermeroots(c)	Compute the roots of a HermiteE series.
hermefromroots(roots)	Generate a HermiteE series with given roots.

Basics

`numpy.polynomial.hermite_e.`**hermeval**(*x*, *c*, *tensor=True*)

Evaluate an HermiteE series at points x.

If *c* is of length *n + 1*, this function returns the value:

$$p(x) = c_0 * He_0(x) + c_1 * He_1(x) + ... + c_n * He_n(x)$$

The parameter *x* is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either *x* or its elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array, then *p(x)* will have the same shape as *x*. If *c* is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor* is true the shape will be c.shape[1:] + x.shape. If *tensor* is false the shape will be c.shape[1:]. Note that scalars have shape (,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

Parameters

x : array_like, compatible object

If *x* is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, *x* or its elements must support addition and multiplication with with themselves and with the elements of *c*.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree n are contained in c[n]. If *c* is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of *c*.

tensor : boolean, optional

If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of *x*. Scalars have dimension 0 for this action. The result is that every column of coefficients in *c* is evaluated for every element of *x*. If False, *x* is broadcast over the columns of *c* for the evaluation. This keyword is useful when *c* is multidimensional. The default value is True.

New in version 1.7.0.

Returns

values : ndarray, algebra_like

The shape of the return value is described above.

See also:

hermeval2d, *hermegrid2d*, *hermeval3d*, *hermegrid3d*

Notes

The evaluation uses Clenshaw recursion, aka synthetic division.

Examples

```
>>> from numpy.polynomial.hermite_e import hermeval
>>> coef = [1,2,3]
>>> hermeval(1, coef)
3.0
>>> hermeval([[1,2],[3,4]], coef)
array([[  3.,  14.],
       [ 31.,  54.]])
```

numpy.polynomial.hermite_e.**hermeval2d**(*x*, *y*, *c*)

Evaluate a 2-D HermiteE series at points (x, y).

This function returns the values:

$$p(x,y) = \sum_{i,j} c_{i,j} * He_i(x) * He_j(y)$$

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array a one is implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

Parameters

x, y : array_like, compatible objects

The two dimensional series is evaluated at the points *(x, y)*, where *x* and *y* must have the same shape. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional polynomial at points formed with pairs of corresponding values from *x* and *y*.

See also:

hermeval, *hermegrid2d*, *hermeval3d*, *hermegrid3d*

Notes

numpy.polynomial.hermite_e.**hermeval3d**(*x*, *y*, *z*, *c*)

Evaluate a 3-D Hermite_e series at points (x, y, z).

This function returns the values:

$$p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * He_i(x) * He_j(y) * He_k(z)$$

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape.

Parameters

x, y, z : array_like, compatible object

The three dimensional series is evaluated at the points *(x, y, z)*, where *x*, *y*, and *z* must have the same shape. If any of *x*, *y*, or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j,k is contained in `c[i,j,k]`. If *c* has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the multidimensional polynomial on points formed with triples of corresponding values from *x*, *y*, and *z*.

See also:

hermeval, *hermeval2d*, *hermegrid2d*, *hermegrid3d*

Notes

numpy.polynomial.hermite_e.**hermegrid2d**(*x*, *y*, *c*)

Evaluate a 2-D HermiteE series on the Cartesian product of x and y.

This function returns the values:

$$p(a,b) = \sum_{i,j} c_{i,j} * H_i(a) * H_j(b)$$

where the points *(a, b)* consist of all pairs formed by taking *a* from *x* and *b* from *y*. The resulting points form a grid with *x* in the first dimension and *y* in the second.

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

Parameters

x, y : array_like, compatible objects

The two dimensional series is evaluated at the points in the Cartesian product of *x* and *y*. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

See also:

hermeval, *hermeval2d*, *hermeval3d*, *hermegrid3d*

Notes

numpy.polynomial.hermite_e.**hermegrid3d**(*x*, *y*, *z*, *c*)

Evaluate a 3-D HermiteE series on the Cartesian product of x, y, and z.

This function returns the values:

$$p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * He_i(a) * He_j(b) * He_k(c)$$

where the points *(a, b, c)* consist of all triples formed by taking *a* from *x*, *b* from *y*, and *c* from *z*. The resulting points form a grid with *x* in the first dimension, *y* in the second, and *z* in the third.

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape + y.shape + z.shape.

Parameters

x, y, z : array_like, compatible objects

The three dimensional series is evaluated at the points in the Cartesian product of *x*, *y*, and *z*. If *x*,'y', or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

See also:

hermeval, *hermeval2d*, *hermegrid2d*, *hermeval3d*

Notes

`numpy.polynomial.hermite_e.`**`hermeroots`**`(c)`

Compute the roots of a HermiteE series.

Return the roots (a.k.a. "zeros") of the polynomial

$$p(x) = \sum_i c[i] * He_i(x).$$

Parameters

c : 1-D array_like

1-D array of coefficients.

Returns

out : ndarray

Array of the roots of the series. If all the roots are real, then *out* is also real, otherwise it is complex.

See also:

`polyroots`, `legroots`, `lagroots`, `hermroots`, `chebroots`

Notes

The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the complex plane may have large errors due to the numerical instability of the series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton's method.

The HermiteE series basis polynomials aren't powers of *x* so the results of this function may seem unintuitive.

Examples

```
>>> from numpy.polynomial.hermite_e import hermeroots, hermefromroots
>>> coef = hermefromroots([-1, 0, 1])
>>> coef
array([ 0.,  2.,  0.,  1.])
>>> hermeroots(coef)
array([-1.,  0.,  1.])
```

numpy.polynomial.hermite_e.**hermefromroots**(*roots*)

Generate a HermiteE series with given roots.

The function returns the coefficients of the polynomial

$$p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),$$

in HermiteE form, where the *r_n* are the roots specified in *roots*. If a zero has multiplicity n, then it must appear in *roots* n times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then *roots* looks something like [2, 2, 2, 3, 3]. The roots can appear in any order.

If the returned coefficients are *c*, then

$$p(x) = c_0 + c_1 * He_1(x) + ... + c_n * He_n(x)$$

The coefficient of the last term is not generally 1 for monic polynomials in HermiteE form.

Parameters

roots : array_like

Sequence containing the roots.

Returns

out : ndarray

1-D array of coefficients. If all roots are real then *out* is a real array, if some of the roots are complex, then *out* is complex even if all the coefficients in the result are real (see Examples below).

See also:

`polyfromroots`, `legfromroots`, `lagfromroots`, `hermfromroots`, `chebfromroots`.

Examples

```
>>> from numpy.polynomial.hermite_e import hermefromroots, hermeval
>>> coef = hermefromroots((-1, 0, 1))
>>> hermeval((-1, 0, 1), coef)
array([ 0.,  0.,  0.])
>>> coef = hermefromroots((-1j, 1j))
>>> hermeval((-1j, 1j), coef)
array([ 0.+0.j,  0.+0.j])
```

hermefit(x, y, deg[, rcond, full, w])	Least squares fit of Hermite series to data.
hermevander(x, deg)	Pseudo-Vandermonde matrix of given degree.
hermevander2d(x, y, deg)	Pseudo-Vandermonde matrix of given degrees.
hermevander3d(x, y, z, deg)	Pseudo-Vandermonde matrix of given degrees.

Fitting

numpy.polynomial.hermite_e.**hermefit**(*x*, *y*, *deg*, *rcond=None*, *full=False*, *w=None*)

Least squares fit of Hermite series to data.

Return the coefficients of a HermiteE series of degree *deg* that is the least squares fit to the data values *y* given at points *x*. If *y* is 1-D the returned coefficients will also be 1-D. If *y* is 2-D multiple fits are done, one for each column of *y*, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

$$p(x) = c_0 + c_1 * He_1(x) + ... + c_n * He_n(x),$$

where *n* is *deg*.

Parameters

x : array_like, shape (M,)

x-coordinates of the M sample points `(x[i], y[i])`.

y : array_like, shape (M,) or (M, K)

y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

deg : int or 1-D array_like

Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For Numpy versions >= 1.11 a list of integers specifying the degrees of the terms to include may be used instead.

rcond : float, optional

Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

full : bool, optional

Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

w : array_like, shape (*M*,), optional

Weights. If not None, the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

Returns

coef : ndarray, shape (M,) or (M, K)

Hermite coefficients ordered from low to high. If *y* was 2-D, the coefficients for the data in column k of *y* are in column *k*.

[residuals, rank, singular_values, rcond] : list

These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

Warns

RankWarning

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if *full* = False. The warnings can be turned off by

```
>>> import warnings
>>> warnings.simplefilter('ignore', RankWarning)
```

See also:

`chebfit`, `legfit`, `polyfit`, `hermfit`, `polyfit`

hermeval
Evaluates a Hermite series.

hermevander
pseudo Vandermonde matrix of Hermite series.

hermeweight
HermiteE weight function.

linalg.lstsq
Computes a least-squares fit from the matrix.

scipy.interpolate.UnivariateSpline
Computes spline fits.

Notes

The solution is the coefficients of the HermiteE series *p* that minimizes the sum of the weighted squared errors

$$E = \sum_j w_j^2 * |y_j - p(x_j)|^2,$$

where the w_j are the weights. This problem is solved by setting up the (typically) overdetermined matrix equation

$$V(x) * c = w * y,$$

where *V* is the pseudo Vandermonde matrix of *x*, the elements of *c* are the coefficients to be solved for, and the elements of *y* are the observed values. This equation is then solved using the singular value decomposition of *V*.

If some of the singular values of *V* are so small that they are neglected, then a *RankWarning* will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The *rcond* parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using HermiteE series are probably most useful when the data can be approximated by `sqrt(w(x)) * p(x)`, where *w(x)* is the HermiteE weight. In that case the weight `sqrt(w(x[i])` should be used together with data values `y[i]/sqrt(w(x[i])`. The weight function is available as *hermeweight*.

References

[R62]

Examples

```
>>> from numpy.polynomial.hermite_e import hermefik, hermeval
>>> x = np.linspace(-10, 10)
>>> err = np.random.randn(len(x))/10
>>> y = hermeval(x, [1, 2, 3]) + err
>>> hermefit(x, y, 2)
array([ 1.01690445,  1.99951418,  2.99948696])
```

numpy.polynomial.hermite_e.**hermevander**(*x*, *deg*)

Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree *deg* and sample points *x*. The pseudo-Vandermonde matrix is defined by

$$V[..., i] = He_i(x),$$

where *0 <= i <= deg*. The leading indices of *V* index the elements of *x* and the last index is the degree of the HermiteE polynomial.

If *c* is a 1-D array of coefficients of length *n + 1* and *V* is the array `V = hermevander(x, n)`, then `np.dot(V, c)` and `hermeval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of HermiteE series of the same degree and sample points.

Parameters

x : array_like

Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If *x* is scalar it is converted to a 1-D array.

deg : int

Degree of the resulting matrix.

Returns

vander : ndarray

The pseudo-Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where The last index is the degree of the corresponding HermiteE polynomial. The dtype will be the same as the converted *x*.

Examples

```
>>> from numpy.polynomial.hermite_e import hermevander
>>> x = np.array([-1, 0, 1])
>>> hermevander(x, 3)
array([[ 1., -1.,  0.,  2.],
       [ 1.,  0., -1., -0.],
       [ 1.,  1.,  0., -2.]])
```

numpy.polynomial.hermite_e.**hermevander2d**(*x*, *y*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y)*. The pseudo-Vandermonde matrix is defined by

$$V[..., deg[1] * i + j] = He_i(x) * He_j(y),$$

where *0 <= i <= deg[0]* and *0 <= j <= deg[1]*. The leading indices of *V* index the points *(x, y)* and the last index encodes the degrees of the HermiteE polynomials.

If `V = hermevander2d(x, y, [xdeg, ydeg])`, then the columns of *V* correspond to the elements of a 2-D coefficient array *c* of shape (xdeg + 1, ydeg + 1) in the order

$$c_{00}, c_{01}, c_{02}..., c_{10}, c_{11}, c_{12}...$$

and `np.dot(V, c.flat)` and `hermeval2d(x, y, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D HermiteE series of the same degrees and sample points.

Parameters

x, y : array_like

Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

deg : list of ints

List of maximum degrees of the form [x_deg, y_deg].

Returns

vander2d : ndarray

The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0]+1)*(deg([1]+1)$. The dtype will be the same as the converted *x* and *y*.

See also:

hermevander, hermevander3d., *hermeval3d*

Notes

numpy.polynomial.hermite_e.**hermevander3d**(*x*, *y*, *z*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y, z)*. If *l, m, n* are the given degrees in *x, y, z*, then Hehe pseudo-Vandermonde matrix is defined by

$$V[..., (m+1)(n+1)i + (n+1)j + k] = He_i(x) * He_j(y) * He_k(z),$$

where *0 <= i <= l*, *0 <= j <= m*, and *0 <= j <= n*. The leading indices of *V* index the points *(x, y, z)* and the last index encodes the degrees of the HermiteE polynomials.

If `V = hermevander3d(x, y, z, [xdeg, ydeg, zdeg])`, then the columns of *V* correspond to the elements of a 3-D coefficient array *c* of shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order

$$c_{000}, c_{001}, c_{002}, ..., c_{010}, c_{011}, c_{012}, ...$$

and `np.dot(V, c.flat)` and `hermeval3d(x, y, z, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D HermiteE series of the same degrees and sample points.

Parameters

x, y, z : array_like

Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

deg : list of ints

List of maximum degrees of the form [x_deg, y_deg, z_deg].

Returns

vander3d : ndarray

The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0]+1)*(deg([1]+1)*(deg[2]+1)$. The dtype will be the same as the converted *x*, *y*, and *z*.

See also:

hermevander, hermevander3d., *hermeval3d*

Notes

hermeder(c[, m, scl, axis])	Differentiate a Hermite_e series.
hermeint(c[, m, k, lbnd, scl, axis])	Integrate a Hermite_e series.

Calculus

numpy.polynomial.hermite_e.**hermeder**(*c*, *m=1*, *scl=1*, *axis=0*)

Differentiate a Hermite_e series.

Returns the series coefficients *c* differentiated *m* times along *axis*. At each iteration the result is multiplied by *scl* (the scaling factor is for use in a linear change of variable). The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `1*He_0 + 2*He_1 + 3*He_2` while [[1,2],[1,2]] represents `1*He_0(x)*He_0(y) + 1*He_1(x)*He_0(y) + 2*He_0(x)*He_1(y) + 2*He_1(x)*He_1(y)` if axis=0 is `x` and axis=1 is `y`.

Parameters

c : array_like

Array of Hermite_e series coefficients. If *c* is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

m : int, optional

Number of derivatives taken, must be non-negative. (Default: 1)

scl : scalar, optional

Each differentiation is multiplied by *scl*. The end result is multiplication by `scl**m`. This is for use in a linear change of variable. (Default: 1)

axis : int, optional

Axis over which the derivative is taken. (Default: 0).

New in version 1.7.0.

Returns

der : ndarray

Hermite series of the derivative.

See also:

hermeint

Notes

In general, the result of differentiating a Hermite series does not resemble the same operation on a power series. Thus the result of this function may be "unintuitive," albeit correct; see Examples section below.

Examples

```
>>> from numpy.polynomial.hermite_e import hermeder
>>> hermeder([ 1.,  1.,  1.,  1.])
array([ 1.,  2.,  3.])
>>> hermeder([-0.25,  1.,  1./2.,  1./3.,  1./4 ], m=2)
array([ 1.,  2.,  3.])
```

numpy.polynomial.hermite_e.**hermeint**(*c*, *m=1*, *k=[]*, *lbnd=0*, *scl=1*, *axis=0*)

Integrate a Hermite_e series.

Returns the Hermite_e series coefficients *c* integrated *m* times from *lbnd* along *axis*. At each iteration the resulting series is **multiplied** by *scl* and an integration constant, *k*, is added. The scaling factor is for use in a linear change of variable. ("Buyer beware": note that, depending on what one is doing, one may want *scl* to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `H_0 + 2*H_1 + 3*H_2` while [[1,2],[1,2]] represents `1*H_0(x)*H_0(y) + 1*H_1(x)*H_0(y) + 2*H_0(x)*H_1(y) + 2*H_1(x)*H_1(y)` if axis=0 is `x` and axis=1 is `y`.

Parameters

c : array_like

Array of Hermite_e series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

m : int, optional

Order of integration, must be positive. (Default: 1)

k : {[], list, scalar}, optional

Integration constant(s). The value of the first integral at `lbnd` is the first value in the list, the value of the second integral at `lbnd` is the second value, etc. If `k == []` (the default), all constants are set to zero. If `m == 1`, a single scalar can be given instead of a list.

lbnd : scalar, optional

The lower bound of the integral. (Default: 0)

scl : scalar, optional

Following each integration the result is *multiplied* by *scl* before the integration constant is added. (Default: 1)

axis : int, optional

Axis over which the integral is taken. (Default: 0).

New in version 1.7.0.

Returns

S : ndarray

Hermite_e series coefficients of the integral.

Raises

ValueError

If `m < 0`, `len(k) > m`, `np.isscalar(lbnd) == False`, or `np.isscalar(scl) == False`.

See also:

hermeder

Notes

Note that the result of each integration is *multiplied* by *scl*. Why is this important to note? Say one is making a linear change of variable $u = ax + b$ in an integral relative to *x*. Then .. math::*dx* = *du/a*, so one will need to set *scl* equal to $1/a$ - perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be "reprojected" onto the C-series basis set. Thus, typically, the result of this function is "unintuitive," albeit correct; see Examples section below.

Examples

```
>>> from numpy.polynomial.hermite_e import hermeint
>>> hermeint([1, 2, 3]) # integrate once, value 0 at 0.
array([ 1.,  1.,  1.,  1.])
>>> hermeint([1, 2, 3], m=2) # integrate twice, value & deriv 0 at 0
array([-0.25      ,  1.        ,  0.5       ,  0.33333333,  0.25      ])
>>> hermeint([1, 2, 3], k=1) # integrate once, value 1 at 0.
array([ 2.,  1.,  1.,  1.])
>>> hermeint([1, 2, 3], lbnd=-1) # integrate once, value 0 at -1
array([-1.,  1.,  1.,  1.])
>>> hermeint([1, 2, 3], m=2, k=[1, 2], lbnd=-1)
array([ 1.83333333,  0.        ,  0.5       ,  0.33333333,  0.25      ])
```

hermeadd(c1, c2)	Add one Hermite series to another.
hermesub(c1, c2)	Subtract one Hermite series from another.
hermemul(c1, c2)	Multiply one Hermite series by another.
hermemulx(c)	Multiply a Hermite series by x.
hermediv(c1, c2)	Divide one Hermite series by another.
hermepow(c, pow[, maxpower])	Raise a Hermite series to a power.

Algebra

numpy.polynomial.hermite_e.**hermeadd**(*c1*, *c2*)

Add one Hermite series to another.

Returns the sum of two Hermite series *c1* + *c2*. The arguments are sequences of coefficients ordered from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Hermite series coefficients ordered from low to high.

Returns

out : ndarray

Array representing the Hermite series of their sum.

See also:

hermesub, *hermemul*, *hermediv*, *hermepow*

Notes

Unlike multiplication, division, etc., the sum of two Hermite series is a Hermite series (without having to "re-project" the result onto the basis set) so addition, just like that of "standard" polynomials, is simply "component-wise."

Examples

```
>>> from numpy.polynomial.hermite_e import hermeadd
>>> hermeadd([1, 2, 3], [1, 2, 3, 4])
array([ 2.,  4.,  6.,  4.])
```

numpy.polynomial.hermite_e.**hermesub**(*c1*, *c2*)

Subtract one Hermite series from another.

Returns the difference of two Hermite series *c1* - *c2*. The sequences of coefficients are from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Hermite series coefficients ordered from low to high.

Returns

out : ndarray

Of Hermite series coefficients representing their difference.

See also:

hermeadd, *hermemul*, *hermediv*, *hermepow*

Notes

Unlike multiplication, division, etc., the difference of two Hermite series is a Hermite series (without having to "reproject" the result onto the basis set) so subtraction, just like that of "standard" polynomials, is simply "component-wise."

Examples

```
>>> from numpy.polynomial.hermite_e import hermesub
>>> hermesub([1, 2, 3, 4], [1, 2, 3])
array([ 0.,  0.,  0.,  4.])
```

numpy.polynomial.hermite_e.**hermemul**(*c1*, *c2*)

Multiply one Hermite series by another.

Returns the product of two Hermite series *c1* * *c2*. The arguments are sequences of coefficients, from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Hermite series coefficients ordered from low to high.

Returns

out : ndarray

Of Hermite series coefficients representing their product.

See also:

hermeadd, *hermesub*, *hermediv*, *hermepow*

Notes

In general, the (polynomial) product of two C-series results in terms that are not in the Hermite polynomial basis set. Thus, to express the product as a Hermite series, it is necessary to "reproject" the product onto said basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

Examples

```
>>> from numpy.polynomial.hermite_e import hermemul
>>> hermemul([1, 2, 3], [0, 1, 2])
array([ 14.,  15.,  28.,   7.,   6.])
```

numpy.polynomial.hermite_e.**hermemulx**(*c*)

Multiply a Hermite series by x.

Multiply the Hermite series *c* by x, where x is the independent variable.

Parameters

c : array_like

1-D array of Hermite series coefficients ordered from low to high.

Returns

out : ndarray

Array representing the result of the multiplication.

Notes

The multiplication uses the recursion relationship for Hermite polynomials in the form

xP_i(x) = (P_{i + 1}(x) + iP_{i - 1}(x)))

Examples

```
>>> from numpy.polynomial.hermite_e import hermemulx
>>> hermemulx([1, 2, 3])
array([ 2.,  7.,  2.,  3.])
```

numpy.polynomial.hermite_e.**hermediv**(*c1*, *c2*)

Divide one Hermite series by another.

Returns the quotient-with-remainder of two Hermite series *c1* / *c2*. The arguments are sequences of coefficients from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c1, c2 : array_like

1-D arrays of Hermite series coefficients ordered from low to high.

Returns

[quo, rem] : ndarrays

Of Hermite series coefficients representing the quotient and remainder.

See also:

hermeadd, *hermesub*, *hermemul*, *hermepow*

Notes

In general, the (polynomial) division of one Hermite series by another results in quotient and remainder terms that are not in the Hermite polynomial basis set. Thus, to express these results as a Hermite series, it is necessary to "reproject" the results onto the Hermite basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

Examples

```
>>> from numpy.polynomial.hermite_e import hermediv
>>> hermediv([ 14.,  15.,  28.,   7.,   6.], [0, 1, 2])
(array([ 1.,  2.,  3.]), array([ 0.]))
>>> hermediv([ 15.,  17.,  28.,   7.,   6.], [0, 1, 2])
(array([ 1.,  2.,  3.]), array([ 1.,  2.]))
```

numpy.polynomial.hermite_e.**hermepow**(*c*, *pow*, *maxpower=16*)

Raise a Hermite series to a power.

Returns the Hermite series *c* raised to the power *pow*. The argument *c* is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series `P_0 + 2*P_1 + 3*P_2`.

Parameters

c : array_like

1-D array of Hermite series coefficients ordered from low to high.

pow : integer

Power to which the series will be raised

maxpower : integer, optional

Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

Returns

coef : ndarray

Hermite series of power.

See also:

hermeadd, *hermesub*, *hermemul*, *hermediv*

Examples

```
>>> from numpy.polynomial.hermite_e import hermepow
>>> hermepow([1, 2, 3], 2)
array([ 23.,  28.,  46.,  12.,   9.])
```

hermegauss(deg)	Gauss-HermiteE quadrature.
hermeweight(x)	Weight function of the Hermite_e polynomials.

Quadrature

numpy.polynomial.hermite_e.**hermegauss**(*deg*)

Gauss-HermiteE quadrature.

Computes the sample points and weights for Gauss-HermiteE quadrature. These sample points and weights will correctly integrate polynomials of degree $2 * deg - 1$ or less over the interval $[-\inf, \inf]$ with the weight function $f(x) = \exp(-x^2/2)$.

Parameters

deg : int

Number of sample points and weights. It must be >= 1.

Returns

x : ndarray

1-D ndarray containing the sample points.

y : ndarray

1-D ndarray containing the weights.

Notes

The results have only been tested up to degree 100, higher degrees may be problematic. The weights are determined by using the fact that

$$w_k = c/(He'_n(x_k) * He_{n-1}(x_k))$$

where c is a constant independent of k and x_k is the k'th root of He_n, and then scaling the results to get the right value when integrating 1.

numpy.polynomial.hermite_e.**hermeweight**(*x*)

Weight function of the Hermite_e polynomials.

The weight function is $\exp(-x^2/2)$ and the interval of integration is $[-\inf, \inf]$. the HermiteE polynomials are orthogonal, but not normalized, with respect to this weight function.

Parameters

x : array_like

Values at which the weight function will be computed.

Returns

w : ndarray

The weight function at *x*.

Notes

hermecompanion(c)	Return the scaled companion matrix of c.
hermedomain	
hermezero	
hermeone	
hermex	
hermeline(off, scl)	Hermite series whose graph is a straight line.
herme2poly(c)	Convert a Hermite series to a polynomial.
poly2herme(pol)	Convert a polynomial to a Hermite series.

Miscellaneous

numpy.polynomial.hermite_e.**hermecompanion**(*c*)

Return the scaled companion matrix of c.

The basis polynomials are scaled so that the companion matrix is symmetric when *c* is an HermiteE basis polynomial. This provides better eigenvalue estimates than the unscaled case and for basis polynomials the eigenvalues are guaranteed to be real if `numpy.linalg.eigvalsh` is used to obtain them.

Parameters

c : array_like

1-D array of HermiteE series coefficients ordered from low to high degree.

Returns

mat : ndarray

Scaled companion matrix of dimensions (deg, deg).

Notes

numpy.polynomial.hermite_e.**hermedomain** = array([-1, 1])

numpy.polynomial.hermite_e.**hermezero** = array([0])

numpy.polynomial.hermite_e.**hermeone** = array([1])

numpy.polynomial.hermite_e.**hermex** = array([0, 1])

numpy.polynomial.hermite_e.**hermeline**(*off*, *scl*)

Hermite series whose graph is a straight line.

Parameters

off, scl : scalars

The specified line is given by `off + scl*x`.

Returns

y : ndarray

This module's representation of the Hermite series for `off + scl*x`.

See also:

`polyline`, `chebline`

Examples

```
>>> from numpy.polynomial.hermite_e import hermeline
>>> from numpy.polynomial.hermite_e import hermeline, hermeval
>>> hermeval(0,hermeline(3, 2))
3.0
>>> hermeval(1,hermeline(3, 2))
5.0
```

numpy.polynomial.hermite_e.**herme2poly**(*c*)

Convert a Hermite series to a polynomial.

Convert an array representing the coefficients of a Hermite series, ordered from lowest degree to highest, to an array of the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest to highest degree.

Parameters

c : array_like

1-D array containing the Hermite series coefficients, ordered from lowest order term to highest.

Returns

pol : ndarray

1-D array containing the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest order term to highest.

See also:

poly2herme

Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```
>>> from numpy.polynomial.hermite_e import herme2poly
>>> herme2poly([  2.,  10.,   2.,   3.])
array([ 0.,  1.,  2.,  3.])
```

numpy.polynomial.hermite_e.**poly2herme**(*pol*)

Convert a polynomial to a Hermite series.

Convert an array representing the coefficients of a polynomial (relative to the "standard" basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Hermite series, ordered from lowest to highest degree.

Parameters

pol : array_like

1-D array containing the polynomial coefficients

Returns

c : ndarray

1-D array containing the coefficients of the equivalent Hermite series.

See also:

herme2poly

Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```
>>> from numpy.polynomial.hermite_e import poly2herme
>>> poly2herme(np.arange(4))
array([  2.,  10.,   2.,   3.])
```

Poly1d

Basics

Fitting

Calculus

Arithmetic

Warnings

3.25 Random sampling (numpy.random)

3.25.1 Simple random data

rand(d0, d1, ..., dn)	Random values in a given shape.
randn(d0, d1, ..., dn)	Return a sample (or samples) from the "standard normal" distribution.
randint(low[, high, size, dtype])	Return random integers from *low* (inclusive) to *high* (exclusive).
random_integers(low[, high, size])	Random integers of type np.int between *low* and *high*, inclusive.
random_sample([size])	Return random floats in the half-open interval [0.0, 1.0).
random([size])	Return random floats in the half-open interval [0.0, 1.0).
ranf([size])	Return random floats in the half-open interval [0.0, 1.0).
sample([size])	Return random floats in the half-open interval [0.0, 1.0).
choice(a[, size, replace, p])	Generates a random sample from a given 1-D array
bytes(length)	Return random bytes.

numpy.random.**rand**(*d0*, *d1*, ..., *dn*)

Random values in a given shape.

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)`.

Parameters

d0, d1, ..., dn : int, optional

The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

Returns

out : ndarray, shape `(d0, d1, ..., dn)`

Random values.

See also:

random

Notes

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random_sample .

Examples

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

numpy.random.**randn**(*d0*, *d1*, ..., *dn*)

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, *randn* generates an array of shape `(d0, d1, ..., dn)`, filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and

variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use *numpy.random.standard_normal* instead.

Parameters

d0, d1, ..., dn : int, optional

The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.

Returns

Z : ndarray or float

A `(d0, d1, ..., dn)`-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

See also:

`random.standard_normal`
Similar, but takes a tuple as its argument.

Notes

For random samples from $N(\mu, \sigma^2)$, use:

`sigma * np.random.randn(...) + mu`

Examples

```
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

```
>>> 2.5 * np.random.randn(2, 4) + 3
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],  #random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]]) #random
```

`numpy.random.`**`randint`**(*low*, *high=None*, *size=None*, *dtype='l'*)
Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution of the specified dtype in the "half-open" interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

Parameters

low : int

Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).

high : int, optional

If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

dtype : dtype, optional

Desired dtype of the result. All dtypes are determined by their name, i.e., 'int64', 'int', etc, so byteorder is not available and a specific precision may have different C types depending on the platform. The default value is 'np.int'.

New in version 1.11.0.

Returns

out : int or ndarray of ints

size-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

See also:

`random.random_integers`
similar to `randint`, only for the closed interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

Examples

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1],
       [3, 2, 2, 0]])
```

numpy.random.**random_integers**(*low*, *high=None*, *size=None*)

Random integers of type np.int between *low* and *high*, inclusive.

Return random integers of type np.int from the "discrete uniform" distribution in the closed interval [*low*, *high*]. If *high* is None (the default), then results are from [1, *low*]. The np.int type translates to the C long type used by Python 2 for "short" integers and its precision is platform dependent.

This function has been deprecated. Use randint instead.

Deprecated since version 1.11.0.

Parameters

low : int

Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).

high : int, optional

If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out : int or ndarray of ints

size-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

See also:

random.randint
Similar to *random_integers*, only for the half-open interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

Notes

To sample from N evenly spaced floating-point numbers between a and b, use:

```
a + (b - a) * (np.random.random_integers(N) - 1) / (N - 1.)
```

Examples

```
>>> np.random.random_integers(5)
4
>>> type(np.random.random_integers(5))
<type 'int'>
>>> np.random.random_integers(5, size=(3.,2.))
array([[5, 4],
       [3, 3],
       [4, 5]])
```

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (*i.e.*, from the set $0, 5/8, 10/8, 15/8, 20/8$):

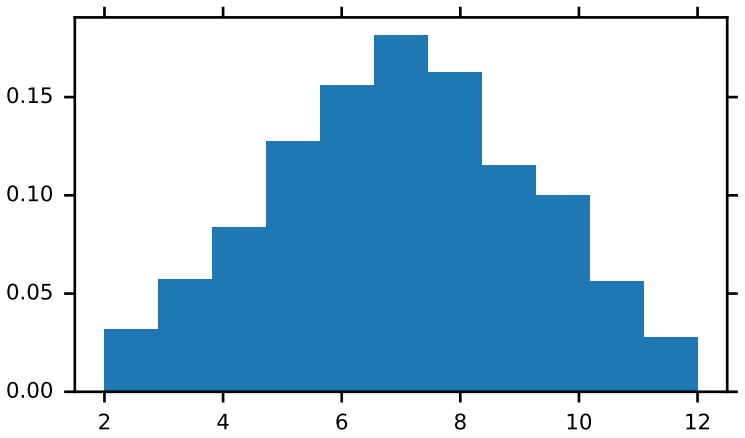
```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4.
array([ 0.625,  1.25 ,  0.625,  0.625,  2.5  ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```



numpy.random.**random_sample**(*size=None*)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of `random_sample` by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

Parameters

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out : float or ndarray of floats

Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

Examples

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

numpy.random.**random**(*size=None*)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

Parameters

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out : float or ndarray of floats

Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

Examples

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

numpy.random.**ranf**(*size=None*)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

Parameters

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out : float or ndarray of floats

Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

Examples

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
```

```
    >>> np.random.random_sample((5,))
    array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
    >>> 5 * np.random.random_sample((3, 2)) - 5
    array([[-3.99149989, -0.52338984],
           [-2.99091858, -0.79479508],
           [-1.23204345, -1.75224494]])
```

numpy.random.**sample**(*size=None*)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
    (b - a) * random_sample() + a
```

Parameters

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out : float or ndarray of floats

Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

Examples

```
    >>> np.random.random_sample()
    0.47108547995356098
    >>> type(np.random.random_sample())
    <type 'float'>
    >>> np.random.random_sample((5,))
    array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
    >>> 5 * np.random.random_sample((3, 2)) - 5
    array([[-3.99149989, -0.52338984],
           [-2.99091858, -0.79479508],
           [-1.23204345, -1.75224494]])
```

numpy.random.**choice**(*a*, *size=None*, *replace=True*, *p=None*)

Generates a random sample from a given 1-D array

New in version 1.7.0.

Parameters

a : 1-D array-like or int

If an ndarray, a random sample is generated from its elements. If an int, the random sample is generated as if a was np.arange(n)

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

replace : boolean, optional

Whether the sample is with or without replacement

p : 1-D array-like, optional

The probabilities associated with each entry in a. If not given the sample assumes a uniform distribution over all entries in a.

Returns

samples : 1-D ndarray, shape (size,)

The generated random samples

Raises

ValueError

If a is an int and less than zero, if a or p are not 1-dimensional, if a is an array-like of size 0, if p is not a vector of probabilities, if a and p have different lengths, or if replace=False and the sample size is greater than the population size

See also:

randint, *shuffle*, *permutation*

Examples

Generate a uniform random sample from np.arange(5) of size 3:

```
>>> np.random.choice(5, 3)
array([0, 3, 4])
>>> #This is equivalent to np.random.randint(0,5,3)
```

Generate a non-uniform random sample from np.arange(5) of size 3:

```
>>> np.random.choice(5, 3, p=[0.1, 0, 0.3, 0.6, 0])
array([3, 3, 0])
```

Generate a uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> np.random.choice(5, 3, replace=False)
array([3,1,0])
>>> #This is equivalent to np.random.permutation(np.arange(5))[:3]
```

Generate a non-uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> np.random.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0])
array([2, 3, 0])
```

Any of the above can be repeated with an arbitrary array-like instead of just integers. For instance:

```
>>> aa_milne_arr = ['pooh', 'rabbit', 'piglet', 'Christopher']
>>> np.random.choice(aa_milne_arr, 5, p=[0.5, 0.1, 0.1, 0.3])
array(['pooh', 'pooh', 'pooh', 'Christopher', 'piglet'],
      dtype='|S11')
```

numpy.random.**bytes**(*length*)

Return random bytes.

Parameters

length : int

Number of random bytes.

Returns
out : str

String of length *length*.

Examples

```
>>> np.random.bytes(10)
' eh\x85\x022SZ\xbf\xa4' #random
```

3.25.2 Permutations

shuffle(x)	Modify a sequence in-place by shuffling its contents.
permutation(x)	Randomly permute a sequence, or return a permuted range.

numpy.random.**shuffle**(*x*)
Modify a sequence in-place by shuffling its contents.

Parameters
x : array_like

The array or list to be shuffled.

Returns
None

Examples

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]
```

This function only shuffles the array along the first index of a multi-dimensional array:

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.shuffle(arr)
>>> arr
array([[3, 4, 5],
       [6, 7, 8],
       [0, 1, 2]])
```

numpy.random.**permutation**(*x*)
Randomly permute a sequence, or return a permuted range.

If *x* is a multi-dimensional array, it is only shuffled along its first index.

Parameters
x : int or array_like

If *x* is an integer, randomly permute `np.arange(x)`. If *x* is an array, make a copy and shuffle the elements randomly.

Returns
out : ndarray

Permuted sequence or array range.

Examples

```
>>> np.random.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6])
```

```
>>> np.random.permutation([1, 4, 9, 12, 15])
array([15,  1,  9,  4, 12])
```

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.permutation(arr)
array([[6, 7, 8],
       [0, 1, 2],
       [3, 4, 5]])
```

3.25.3 Distributions

beta(a, b[, size])	Draw samples from a Beta distribution.
binomial(n, p[, size])	Draw samples from a binomial distribution.
chisquare(df[, size])	Draw samples from a chi-square distribution.
dirichlet(alpha[, size])	Draw samples from the Dirichlet distribution.
exponential([scale, size])	Draw samples from an exponential distribution.
f(dfnum, dfden[, size])	Draw samples from an F distribution.
gamma(shape[, scale, size])	Draw samples from a Gamma distribution.
geometric(p[, size])	Draw samples from the geometric distribution.
gumbel([loc, scale, size])	Draw samples from a Gumbel distribution.
hypergeometric(ngood, nbad, nsample[, size])	Draw samples from a Hypergeometric distribution.
laplace([loc, scale, size])	Draw samples from the Laplace or double exponential distribution with specifie
logistic([loc, scale, size])	Draw samples from a logistic distribution.
lognormal([mean, sigma, size])	Draw samples from a log-normal distribution.
logseries(p[, size])	Draw samples from a logarithmic series distribution.
multinomial(n, pvals[, size])	Draw samples from a multinomial distribution.
multivariate_normal(mean, cov[, size])	Draw random samples from a multivariate normal distribution.
negative_binomial(n, p[, size])	Draw samples from a negative binomial distribution.
noncentral_chisquare(df, nonc[, size])	Draw samples from a noncentral chi-square distribution.
noncentral_f(dfnum, dfden, nonc[, size])	Draw samples from the noncentral F distribution.
normal([loc, scale, size])	Draw random samples from a normal (Gaussian) distribution.
pareto(a[, size])	Draw samples from a Pareto II or Lomax distribution with specified shape.
poisson([lam, size])	Draw samples from a Poisson distribution.
power(a[, size])	Draws samples in [0, 1] from a power distribution with positive exponent a - 1.
rayleigh([scale, size])	Draw samples from a Rayleigh distribution.
standard_cauchy([size])	Draw samples from a standard Cauchy distribution with mode = 0.
standard_exponential([size])	Draw samples from the standard exponential distribution.
standard_gamma(shape[, size])	Draw samples from a standard Gamma distribution.
standard_normal([size])	Draw samples from a standard Normal distribution (mean=0, stdev=1).
standard_t(df[, size])	Draw samples from a standard Student's t distribution with *df* degrees of freedo
triangular(left, mode, right[, size])	Draw samples from the triangular distribution.
uniform([low, high, size])	Draw samples from a uniform distribution.
vonmises(mu, kappa[, size])	Draw samples from a von Mises distribution.
wald(mean, scale[, size])	Draw samples from a Wald, or inverse Gaussian, distribution.
weibull(a[, size])	Draw samples from a Weibull distribution.
zipf(a[, size])	Draw samples from a Zipf distribution.

numpy.random.**beta**(*a*, *b*, *size=None*)

Draw samples from a Beta distribution.

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1}(1-x)^{\beta-1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1} dt.$$

It is often seen in Bayesian inference and order statistics.

Parameters

a : float

Alpha, non-negative.

b : float

Beta, non-negative.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out : ndarray

Array of the given shape, containing values drawn from a Beta distribution.

numpy.random.**binomial**(*n*, *p*, *size=None*)

Draw samples from a binomial distribution.

Samples are drawn from a binomial distribution with specified parameters, n trials and p probability of success where n an integer >= 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

Parameters

n : float (but truncated to an integer)

parameter, >= 0.

p : float

parameter, >= 0 and <=1.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar

where the values are all integers in [0, n].

See also:

scipy.stats.distributions.binom

probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product p*n <=5, where p = population proportion estimate, and n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

References

[R208], *[R209]*, *[R210]*, *[R211]*, *[R212]*

Examples

Draw samples from the distribution:

```
>>> n, p = 10, .5  # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9, 0.1, 20000) == 0)/20000.
# answer = 0.38885, or 38%.
```

numpy.random.**chisquare**(*df*, *size=None*)

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

Parameters

df : int

Number of degrees of freedom.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

output : ndarray

Samples drawn from the distribution, packed in a *size*-shaped array.

Raises

ValueError

When *df* <= 0 or when an inappropriate *size* (e.g. `size=-1`) is given.

Notes

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\texttt{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2.$$

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

References

[R213]

Examples

```
>>> np.random.chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272])
```

numpy.random.**dirichlet**(*alpha*, *size=None*)

Draw samples from the Dirichlet distribution.

Draw *size* samples of dimension k from a Dirichlet distribution. A Dirichlet-distributed random variable can be seen as a multivariate generalization of a Beta distribution. Dirichlet pdf is the conjugate prior of a multinomial in Bayesian inference.

Parameters

alpha : array

Parameter of the distribution (k dimension for sample of dimension k).

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray,

The drawn samples, of shape (size, alpha.ndim).

Notes

$$X \approx \prod_{i=1}^{k} x_i^{\alpha_i - 1}$$

Uses the following property for computation: for each dimension, draw a random sample y_i from a standard gamma generator of shape *alpha_i*, then $X = \frac{1}{\sum_{i=1}^{k} y_i}(y_1, \ldots, y_n)$ is Dirichlet distributed.

References

[R214], *[R215]*

Examples

Taking an example cited in Wikipedia, this distribution can be used if one wanted to cut strings (each of initial length 1.0) into K pieces with different lengths, where each piece had, on average, a designated average length, but allowing some variation in the relative sizes of the pieces.

```
>>> s = np.random.dirichlet((10, 5, 3), 20).transpose()
```

```
>>> plt.barh(range(20), s[0])
>>> plt.barh(range(20), s[1], left=s[0], color='g')
>>> plt.barh(range(20), s[2], left=s[0]+s[1], color='r')
>>> plt.title("Lengths of Strings")
```

numpy.random.**exponential**(*scale=1.0*, *size=None*)

Draw samples from an exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for `x > 0` and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution *[R218]*.

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms *[R216]*, or the time between page requests to Wikipedia *[R217]*.

Parameters

scale : float

The scale parameter, $\beta = 1/\lambda$.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

References

[R216], *[R217]*, *[R218]*

numpy.random.**f**(*dfnum*, *dfden*, *size=None*)

Draw samples from an F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

Parameters

dfnum : float

Degrees of freedom in numerator. Should be greater than zero.

dfden : float

Degrees of freedom in denominator. Should be greater than zero.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar

Samples from the Fisher distribution.

See also:

`scipy.stats.distributions.f`
probability density function, distribution or cumulative density function, etc.

Notes

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

References

[R219], *[R220]*

Examples

An example from Glantz[1], pp 47-40:

Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is :

```
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

`numpy.random.`**`gamma`**(*shape*, *scale=1.0*, *size=None*)
Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

Parameters

shape : scalar > 0

The shape of the gamma distribution.

scale : scalar > 0, optional

The scale of the gamma distribution. Default is equal to 1.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out : ndarray, float

Returns one sample unless *size* parameter is specified.

See also:

`scipy.stats.distributions.gamma`
probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

References

[R221], *[R222]*

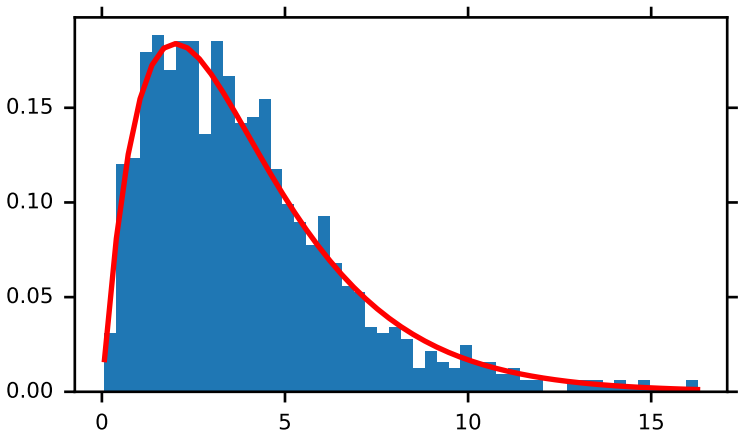
Examples

Draw samples from the distribution:

```
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> y = bins**(shape-1)*(np.exp(-bins/scale) /
...                      (sps.gamma(shape)*scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```



numpy.random.**geometric**(*p*, *size=None*)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, `k = 1, 2, ...`.

The probability mass function of the geometric distribution is

$$f(k) = (1-p)^{k-1}p$$

where p is the probability of success of an individual trial.

Parameters

p : float

The probability of success of an individual trial.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out : ndarray

Samples from the geometric distribution, shaped according to *size*.

Examples

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000.
0.34889999999999999 #random
```

numpy.random.**gumbel**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from a Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

Parameters

loc : float

The location of the mode of the distribution.

scale : float

The scale parameter of the distribution.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar

See also:

`scipy.stats.gumbel_l`, `scipy.stats.gumbel_r`, `scipy.stats.genextreme`, *weibull*

Notes

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

References

[R223], *[R224]*

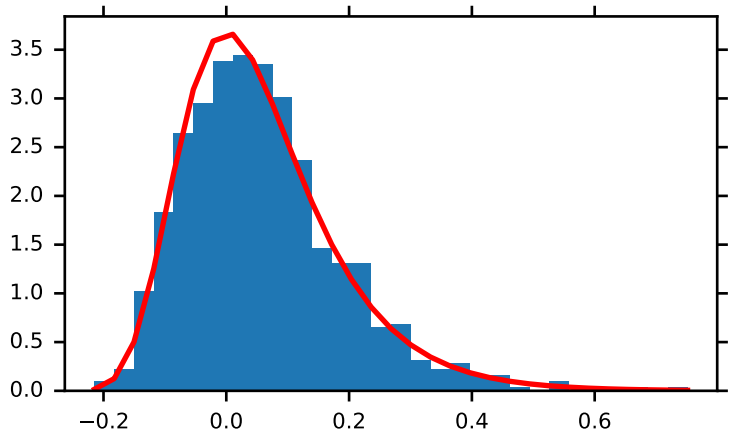
Examples

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

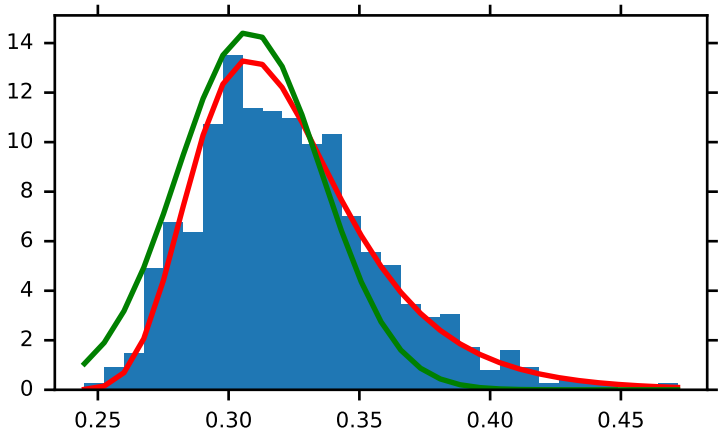
Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp( -np.exp( -(bins - mu) /beta) ),
...          linewidth=2, color='r')
>>> plt.show()
```



Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
...    a = np.random.normal(mu, beta, 1000)
...    means.append(a.mean())
...    maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima) * np.sqrt(6) / np.pi
>>> mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp(-np.exp(-(bins - mu)/beta)),
...          linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
...          * np.exp(-(bins - mu)**2 / (2 * beta**2)),
...          linewidth=2, color='g')
>>> plt.show()
```



numpy.random.**hypergeometric**(*ngood*, *nbad*, *nsample*, *size=None*)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

Parameters

ngood : int or array_like

Number of ways to make a good selection. Must be nonnegative.

nbad : int or array_like

Number of ways to make a bad selection. Must be nonnegative.

nsample : int or array_like

Number of items sampled. Must be at least 1 and at most `ngood + nbad`.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar

The values are all integers in [0, n].

See also:

scipy.stats.distributions.hypergeom

probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n}\binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \leq x \leq m$ and $n + m - N \leq x \leq n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the binomial.

References

[R225], *[R226]*, *[R227]*

Examples

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
#   note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
#   answer = 0.003 ... pretty unlikely!
```

numpy.random.**laplace**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

Parameters

loc : float, optional

The position, μ, of the distribution peak.

scale : float, optional

λ, the exponential decay.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or float

Notes

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in economics and health sciences, this distribution seems to model the data better than the standard Gaussian distribution.

References

[R228], *[R229]*, *[R230]*, *[R231]*

Examples

Draw samples from the distribution

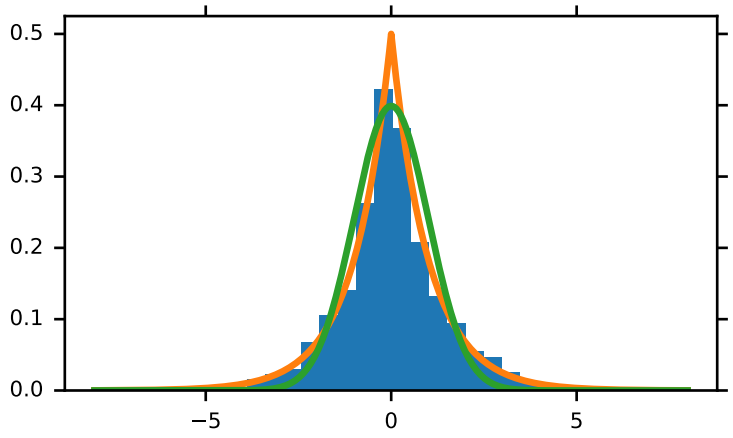
```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc)/scale)/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

```
>>> g = (1/(scale * np.sqrt(2 * np.pi)) *
...      np.exp(-(x - loc)**2 / (2 * scale**2)))
>>> plt.plot(x,g)
```



`numpy.random.`**`logistic`**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from a logistic distribution.

Samples are drawn from a logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

Parameters

loc : float

scale : float > 0.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar

where the values are all integers in [0, n].

See also:

`scipy.stats.distributions.logistic`
probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1+e^{-(x-\mu)/s})^2},$$

where μ = location and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

References

[R232], *[R233]*, *[R234]*

Examples

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...     return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

`numpy.random.`**`lognormal`**(*mean=0.0*, *sigma=1.0*, *size=None*)
Draw samples from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

Parameters

mean : float

Mean value of the underlying normal distribution

sigma : float, > 0.

Standard deviation of the underlying normal distribution

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or float

The desired samples. An array of the same shape as *size* if given, if *size* is None a float is returned.

See also:

`scipy.stats.lognorm`
probability density function, distribution, cumulative density function, etc.

Notes

A variable *x* has a log-normal distribution if *log(x)* is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{(-\frac{(ln(x)-\mu)^2}{2\sigma^2})}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

References

[R235], *[R236]*

Examples

Draw samples from the distribution:

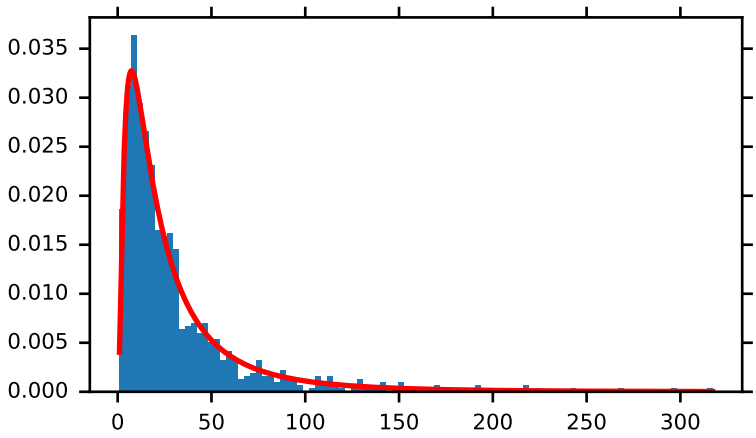
```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
```

```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```



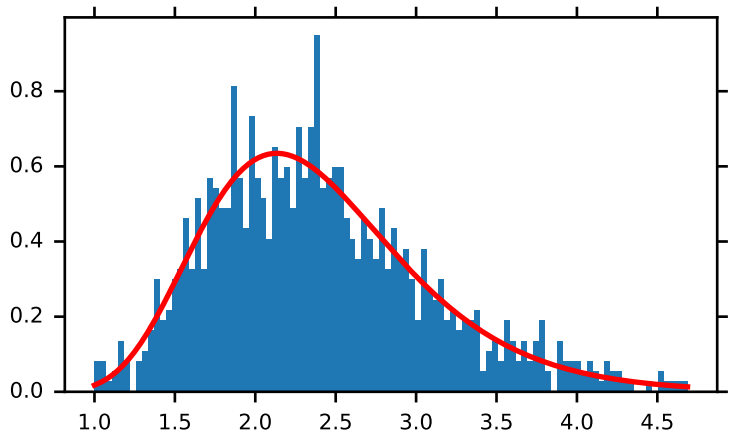
Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

```
>>> # Generate a thousand samples: each is the product of 100 random
>>> # values, drawn from a normal distribution.
>>> b = []
>>> for i in range(1000):
...     a = 10. + np.random.random(100)
...     b.append(np.product(a))
```

```
>>> b = np.array(b) / np.min(b) # scale values to be positive
>>> count, bins, ignored = plt.hist(b, 100, normed=True, align='mid')
>>> sigma = np.std(np.log(b))
>>> mu = np.mean(np.log(b))
```

```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, color='r', linewidth=2)
>>> plt.show()
```



numpy.random.**logseries**(*p*, *size=None*)

Draw samples from a logarithmic series distribution.

Samples are drawn from a log series distribution with specified shape parameter, 0 < p < 1.

Parameters

loc : float

scale : float > 0.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar

where the values are all integers in [0, n].

See also:

scipy.stats.distributions.logser

probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The log series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

References

[R237], *[R238]*, *[R239]*, *[R240]*

Examples

Draw samples from the distribution:

```
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> count, bins, ignored = plt.hist(s)
```

plot against distribution

```
>>> def logseries(k, p):
...     return -p**k/(k*log(1-p))
>>> plt.plot(bins, logseries(bins, a)*count.max()/
             logseries(bins, a).max(), 'r')
>>> plt.show()
```

numpy.random.**multinomial**(*n*, *pvals*, *size=None*)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of `p` possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents *n* such experiments. Its values, `X_i = [X_0, X_1, ..., X_p]`, represent the number of times the outcome was `i`.

Parameters

n : int

Number of experiments.

pvals : sequence of floats, length p

Probabilities of each of the `p` different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as `sum(pvals[:-1]) <= 1)`.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out : ndarray

The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

Examples

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1)
array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

```
>>> np.random.multinomial(20, [1/6.]*6, size=2)
array([[3, 4, 3, 3, 4, 3],
       [2, 4, 3, 4, 0, 7]])
```

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded die is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5 + [2/7.])
array([11, 16, 14, 17, 16, 26])
```

The probability inputs should be normalized. As an implementation detail, the value of the last entry is ignored and assumed to take up any leftover probability mass, but this should not be relied on. A biased coin which has twice as much weight on one side as on the other should be sampled like so:

```
>>> np.random.multinomial(100, [1.0 / 3, 2.0 / 3])  # RIGHT
array([38, 62])
```

not like:

```
>>> np.random.multinomial(100, [1.0, 2.0])  # WRONG
array([100,   0])
```

numpy.random.**multivariate_normal**(*mean*, *cov*[, *size*])

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

Parameters

mean : 1-D array_like, of length N

Mean of the N-dimensional distribution.

cov : 2-D array_like, of shape (N, N)

Covariance matrix of the distribution. It must be symmetric and positive-semidefinite for proper sampling.

size : int or tuple of ints, optional

Given a shape of, for example, `(m,n,k)`, `m*n*k` samples are generated, and packed in an *m*-by-*n*-by-*k* arrangement. Because each sample is *N*-dimensional, the output shape is `(m,n,k,N)`. If no shape is specified, a single (*N*-D) sample is returned.

Returns

out : ndarray

The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

Notes

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

•Spherical covariance (*cov* is a multiple of the identity matrix)

•Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0, 0]
>>> cov = [[1, 0], [0, 100]]  # diagonal covariance
```

Diagonal covariance means that points are oriented along x or y-axis:

```
>>> import matplotlib.pyplot as plt
>>> x, y = np.random.multivariate_normal(mean, cov, 5000).T
>>> plt.plot(x, y, 'x')
>>> plt.axis('equal')
>>> plt.show()
```

Note that the covariance matrix must be positive semidefinite (a.k.a. nonnegative-definite). Otherwise, the behavior of this method is undefined and backwards compatibility is not guaranteed.

References

[R241], *[R242]*

Examples

```
>>> mean = (1, 2)
>>> cov = [[1, 0], [0, 1]]
>>> x = np.random.multivariate_normal(mean, cov, (3, 3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> list((x[0,0,:] - mean) < 0.6)
[True, True]
```

numpy.random.**negative_binomial**(*n*, *p*, *size=None*)

Draw samples from a negative binomial distribution.

Samples are drawn from a negative binomial distribution with specified parameters, *n* trials and *p* probability of success where *n* is an integer > 0 and *p* is in the interval [0, 1].

Parameters

n : int

Parameter, > 0.

p : float

Parameter, >= 0 and <=1.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : int or ndarray of ints

Drawn samples.

Notes

The probability density for the negative binomial distribution is

$$P(N;n,p) = \binom{N+n-1}{n-1} p^n (1-p)^N,$$

where $n-1$ is the number of successes, p is the probability of success, and $N+n-1$ is the number of trials. The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

References

[R243], *[R244]*

Examples

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
...     probability = sum(s<i) / 100000.
...     print i, "wells drilled, probability of one success =", probability
```

numpy.random.**noncentral_chisquare**(*df*, *nonc*, *size=None*)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

Parameters

df : int

Degrees of freedom, should be > 0 as of Numpy 1.10, should be > 1 for earlier versions.

nonc : float

Non-centrality, should be non-negative.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Notes

The probability density function for the noncentral Chi-square distribution is

$$P(x;df,nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^i}{i!} \P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

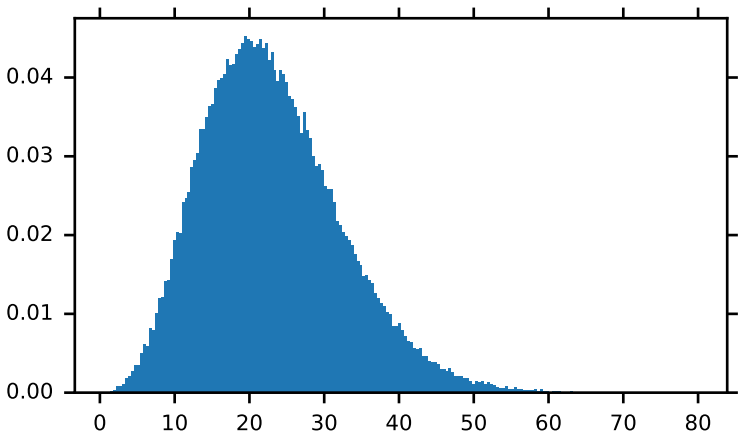
References

[R245], *[R246]*

Examples

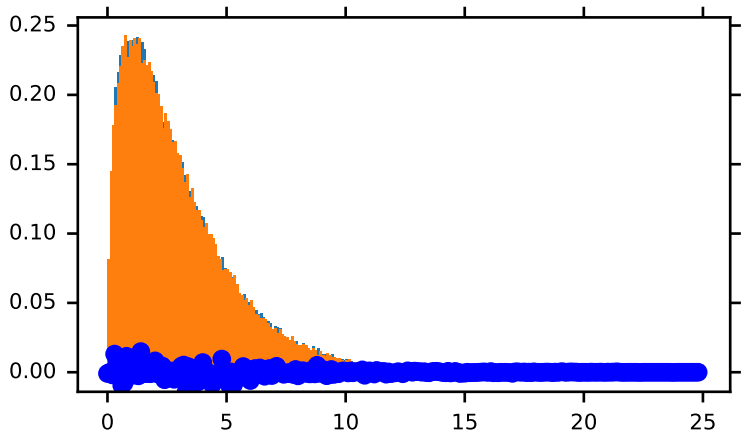
Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, normed=True)
>>> plt.show()
```



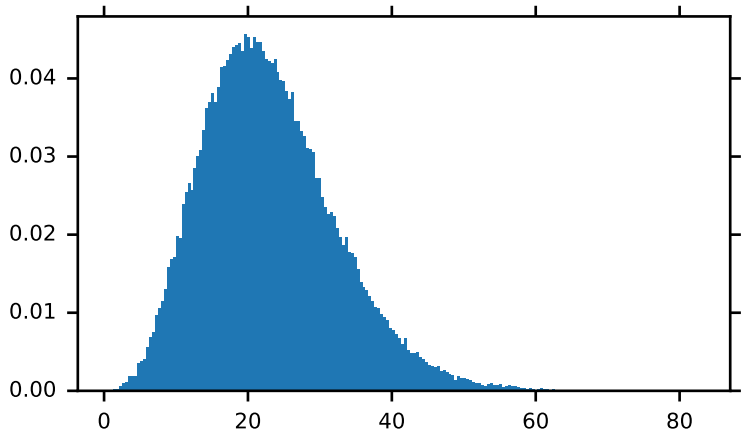
Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
...                   bins=np.arange(0., 25, .1), normed=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
...                    bins=np.arange(0., 25, .1), normed=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```



Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, normed=True)
>>> plt.show()
```



numpy.random.**noncentral_f**(*dfnum*, *dfden*, *nonc*, *size=None*)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

Parameters

dfnum : int

Parameter, should be > 1.

dfden : int

Parameter, should be > 1.

nonc : float

Parameter, should be >= 0.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : scalar or ndarray

Drawn samples.

Notes

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

References

[R247], *[R248]*

Examples

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

numpy.random.**normal**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently *[R250]*, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution *[R250]*.

Parameters

loc : float

Mean ("centre") of the distribution.

scale : float

Standard deviation (spread or "width") of the distribution.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

See also:

`scipy.stats.distributions.norm`
probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2, is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ *[R250]*). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

References

[R249], *[R250]*

Examples

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

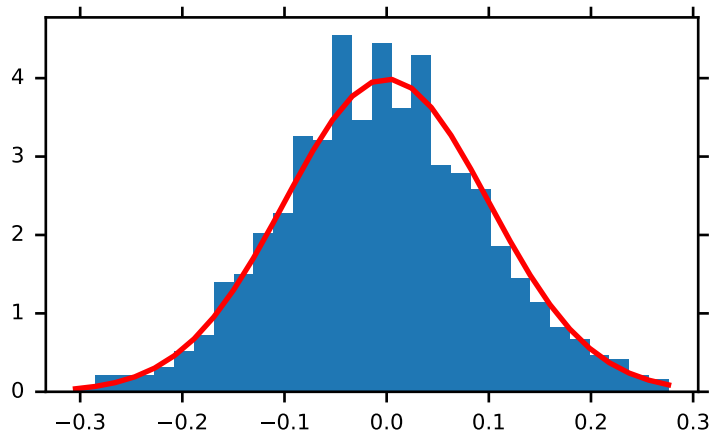
Verify the mean and the variance:

```
>>> abs(mu - np.mean(s)) < 0.01
True
```

```
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
...                np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
...          linewidth=2, color='r')
>>> plt.show()
```



numpy.random.**pareto**(*a*, *size=None*)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding 1 and multiplying by the scale parameter `m` (see Notes). The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is `mu`, where the standard Pareto distribution has location `mu = 1`. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

Parameters

shape : float, > 0.

Shape of the distribution.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

See also:

scipy.stats.distributions.lomax.pdf
probability density function, distribution or cumulative density function, etc.

scipy.stats.distributions.genpareto.pdf
probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the scale.

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge *[R251]*. It is one of the so-called "fat-tailed" distributions.

References

[R251], *[R252]*, *[R253]*, *[R254]*

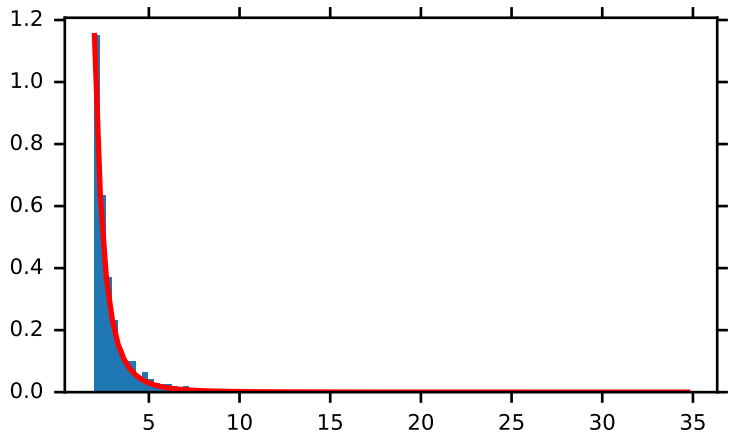
Examples

Draw samples from the distribution:

```
>>> a, m = 3., 2.  # shape and mode
>>> s = (np.random.pareto(a, 1000) + 1) * m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, _ = plt.hist(s, 100, normed=True)
>>> fit = a*m**a / bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit), linewidth=2, color='r')
>>> plt.show()
```



numpy.random.**poisson**(*lam=1.0*, *size=None*)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the binomial distribution for large N.

Parameters

lam : float or sequence of float

Expectation of interval, should be >= 0. A sequence of expectation intervals must be broadcastable over the requested size.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar

The drawn samples, of shape *size*, if it was provided.

Notes

The Poisson distribution

$$f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ.

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

References

[R255], *[R256]*

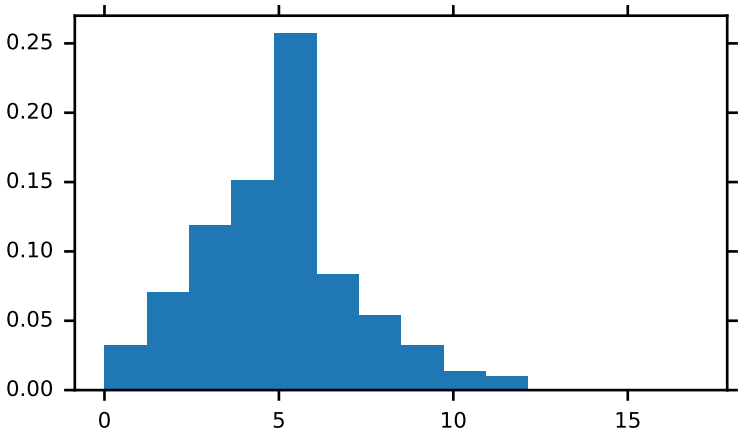
Examples

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)
```

Display histogram of the sample:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()
```



Draw each 100 values for lambda 100 and 500:

```
>>> s = np.random.poisson(lam=(100., 500.), size=(100, 2))
```

numpy.random.**power**(*a*, *size=None*)

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

Parameters

a : float

parameter, > 0

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar

The returned samples lie in [0, 1].

Raises

ValueError

If a < 1.

Notes

The probability density function is

$$P(x; a) = ax^{a-1}, 0 \leq x \leq 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

References

[R257], *[R258]*

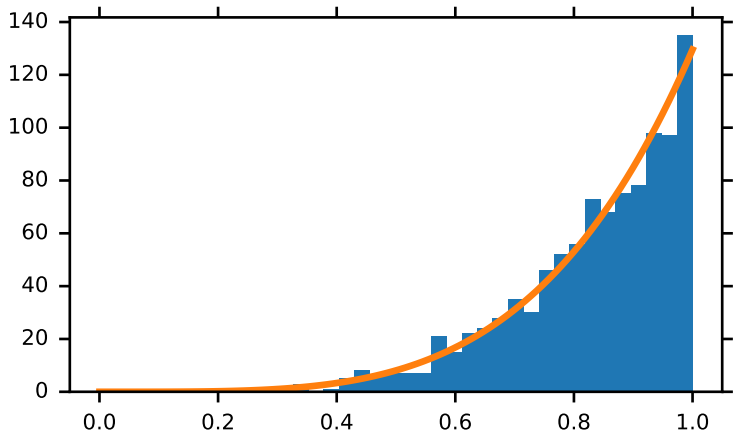
Examples

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```



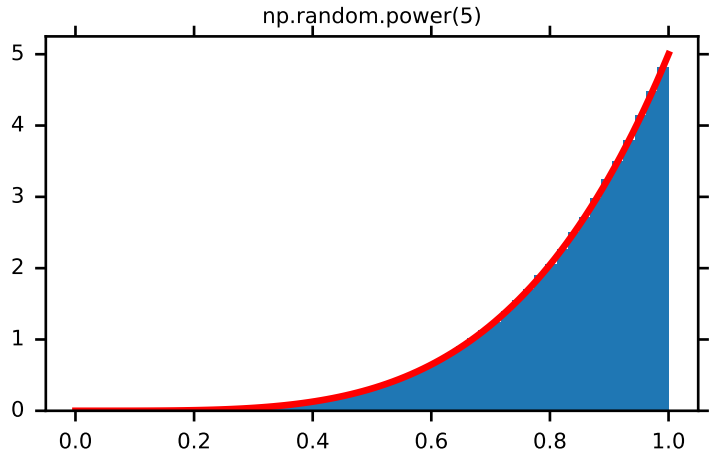
Compare the power function distribution to the inverse of the Pareto.

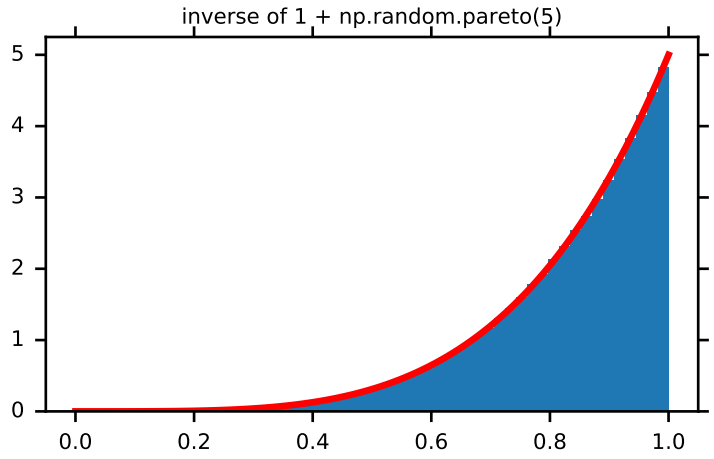
```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
```

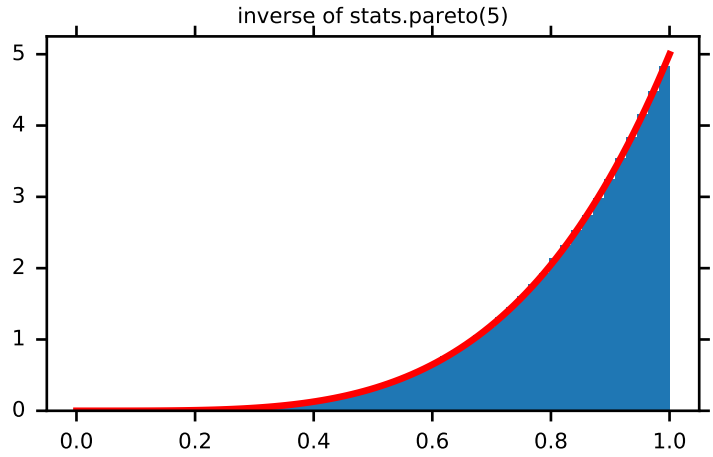
```
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
```


np.random.power(5)
0
1
2
3
4
5
0.0
0.2
0.4
0.6
0.8
1.0


inverse of 1 + np.random.pareto(5)
0
1
2
3
4
5
0.0
0.2
0.4
0.6
0.8
1.0



numpy.random.**rayleigh**(*scale=1.0*, *size=None*)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

Parameters

scale : scalar

Scale, also equals the mode. Should be >= 0.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Notes

The probability density function for the Rayleigh distribution is

$$P(x;scale) = \frac{x}{scale^2}e^{\frac{-x^2}{2\cdot scale^2}}$$

The Rayleigh distribution would arise, for example, if the East and North components of the wind velocity had identical zero-mean Gaussian distributions. Then the wind speed would have a Rayleigh distribution.

References

[R259], *[R260]*

Examples

Draw values from the distribution and plot the histogram

```
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.087300000000000003
```

numpy.random.**standard_cauchy**(*size=None*)

Draw samples from a standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

Parameters

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar

The drawn samples.

Notes

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi\gamma\left[1 + (\frac{x-x_0}{\gamma})^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

References

[R262], *[R263]*, *[R264]*

Examples

Draw samples and plot the distribution:

```
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)]  # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

numpy.random.**standard_exponential**(*size=None*)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

Parameters

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out : float or ndarray

Drawn samples.

Examples

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

numpy.random.**standard_gamma**(*shape*, *size=None*)

Draw samples from a standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

Parameters

shape : float

Parameter, should be > 0.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar

The drawn samples.

See also:

scipy.stats.distributions.gamma

probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

References

[R265], *[R266]*

Examples

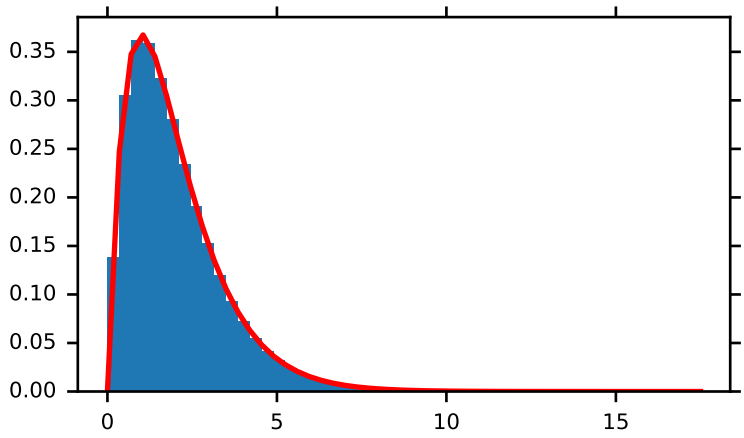
Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> y = bins**(shape-1) * ((np.exp(-bins/scale))/ \
```

```
...                       (sps.gamma(shape) * scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```



numpy.random.**standard_normal**(*size=None*)

Draw samples from a standard Normal distribution (mean=0, stdev=1).

Parameters

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out : float or ndarray

Drawn samples.

Examples

```
>>> s = np.random.standard_normal(8000)
>>> s
array([ 0.6888893 ,  0.78096262, -0.89086505, ...,  0.49876311, #random
       -0.38672696, -0.4685006 ])                               #random
>>> s.shape
(8000,)
>>> s = np.random.standard_normal(size=(3, 4, 2))
>>> s.shape
(3, 4, 2)
```

numpy.random.**standard_t**(*df*, *size=None*)

Draw samples from a standard Student's t distribution with *df* degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard_normal*).

Parameters

df : int

Degrees of freedom, should be > 0.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar

Drawn samples.

Notes

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df}\,\Gamma(\frac{df}{2})}\left(1+\frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was first published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

References

[R267], *[R268]*

Examples

From Dalgaard page 83 *[R267]*, suppose the daily energy intake for 11 women in Kj is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
...                    7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

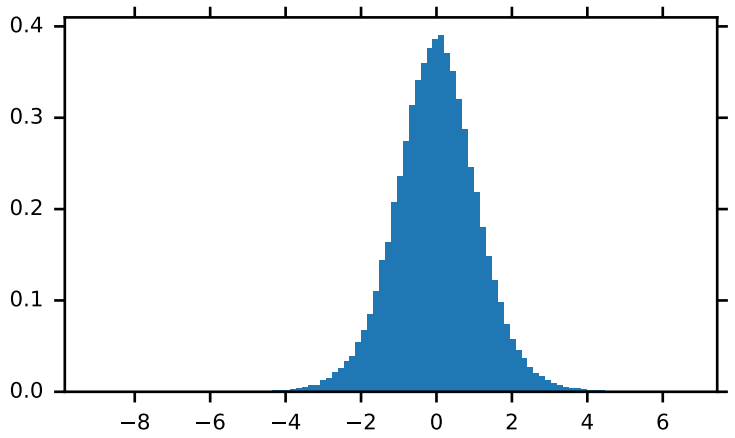
Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> np.sum(s<t) / float(len(s))
0.0090699999999999999  #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.



numpy.random.**triangular**(*left*, *mode*, *right*, *size=None*)

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

Parameters

left : scalar

Lower limit.

mode : scalar

The value where the peak of the distribution occurs. The value should fulfill the condition `left <= mode <= right`.

right : scalar

Upper limit, should be larger than *left*.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar

The returned samples all lie in the interval [left, right].

Notes

The probability density function for the triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(r-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

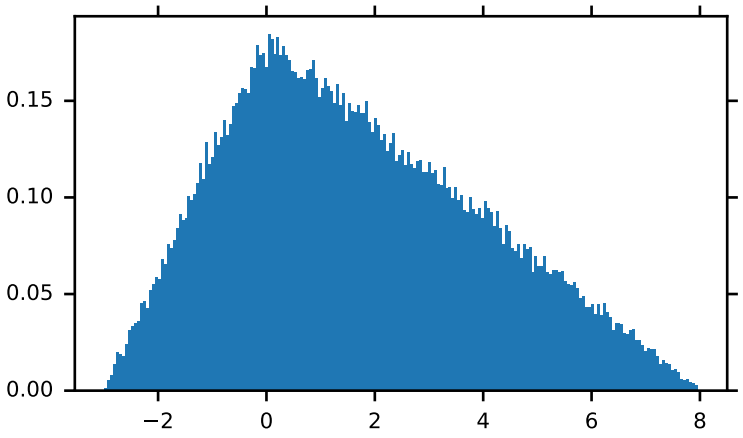
References

[R269]

Examples

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.triangular(-3, 0, 8, 100000), bins=200,
...              normed=True)
>>> plt.show()
```



numpy.random.**uniform**(*low=0.0*, *high=1.0*, *size=None*)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

Parameters

low : float, optional

Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high : float

Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out : ndarray

Drawn samples, with shape *size*.

See also:

randint
Discrete uniform distribution, yielding integers.

random_integers
Discrete uniform distribution over the closed interval `[low, high]`.

random_sample
Floats uniformly distributed over `[0, 1)`.

random
Alias for *random_sample*.

rand
Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

Notes

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

Examples

Draw samples from the distribution:

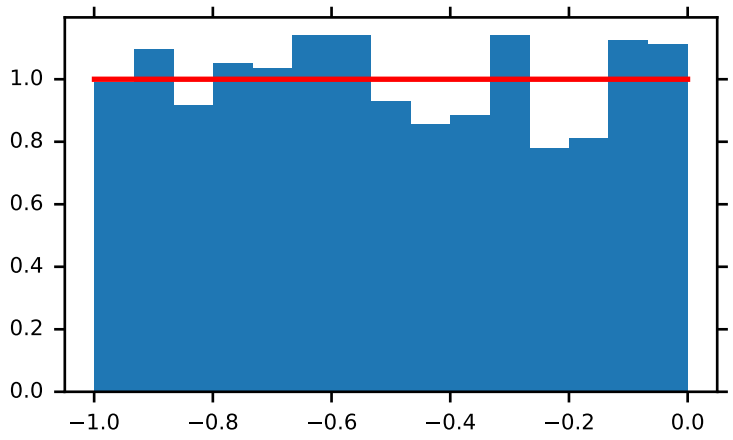
```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```



numpy.random.**vonmises**(*mu*, *kappa*, *size=None*)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

Parameters

mu : float

Mode ("center") of the distribution.

kappa : float

Dispersion of the distribution, has to be >=0.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : scalar or ndarray

The returned samples, which are in the interval [-pi, pi].

See also:

scipy.stats.distributions.vonmises

probability density function, distribution, or cumulative density function, etc.

Notes

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

References

[R270], *[R271]*

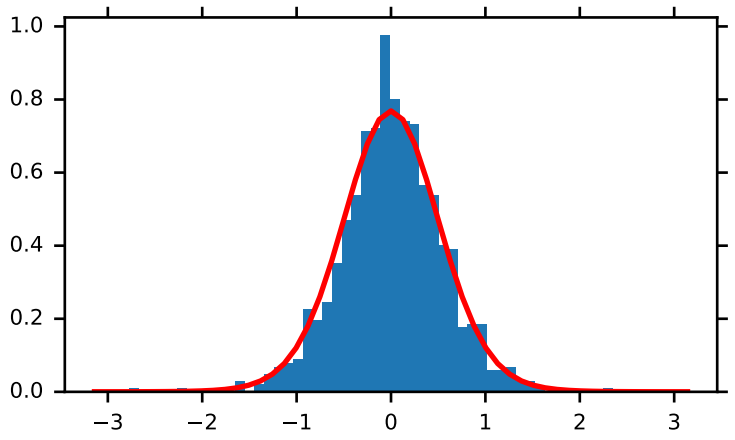
Examples

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy.special import i0
>>> plt.hist(s, 50, normed=True)
>>> x = np.linspace(-np.pi, np.pi, num=51)
>>> y = np.exp(kappa*np.cos(x-mu))/(2*np.pi*i0(kappa))
>>> plt.plot(x, y, linewidth=2, color='r')
>>> plt.show()
```



numpy.random.**wald**(*mean*, *scale*, *size=None*)

Draw samples from a Wald, or inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian. Some references claim that the Wald is an inverse Gaussian with mean equal to 1, but this is by no means universal.

The inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

Parameters

mean : scalar

Distribution mean, should be > 0.

scale : scalar

Scale parameter, should be >= 0.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar

Drawn sample, all greater than zero.

Notes

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the inverse Gaussian distribution first arise from attempts to model Brownian motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

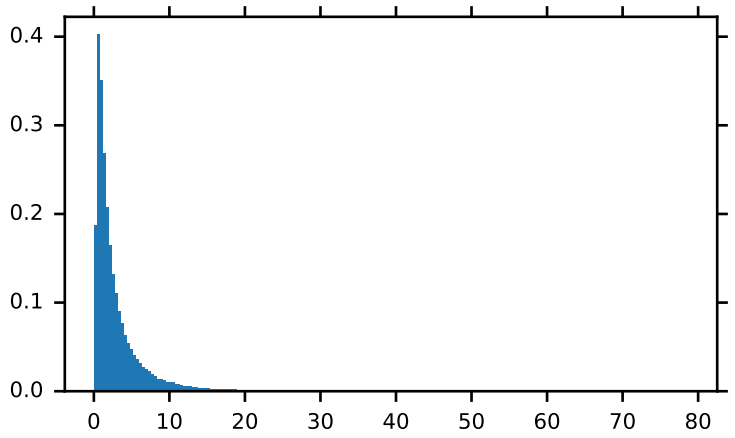
References

[R272], *[R273]*, *[R274]*

Examples

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
```



`numpy.random.`**`weibull`**(*a*, *size=None*)

Draw samples from a Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda(-ln(U))^{1/a}$.

Parameters

a : float

Shape of the distribution.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : ndarray

See also:

`scipy.stats.distributions.weibull_max`, `scipy.stats.distributions.weibull_min`, `scipy.stats.distributions.genextreme`, *gumbel*

Notes

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda}(\frac{x}{\lambda})^{a-1}e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When `a = 1`, the Weibull distribution reduces to the exponential distribution.

References

[R275], *[R276]*, *[R277]*

Examples

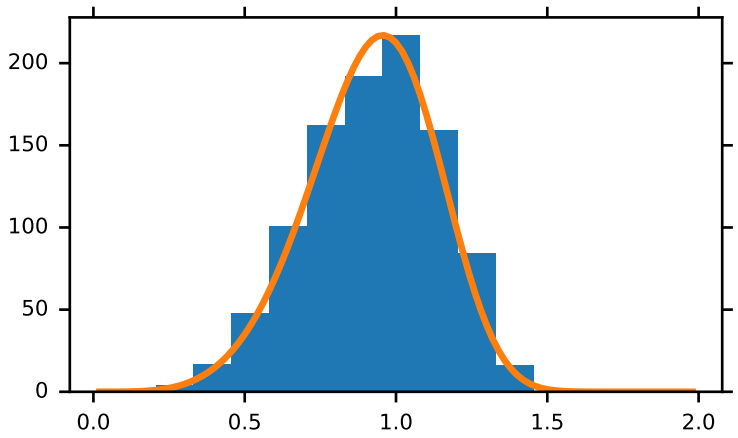
Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
```

```
>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```



numpy.random.**zipf**(*a*, *size=None*)

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter $a > 1$.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

Parameters

a : float > 1

Distribution parameter.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples : scalar or ndarray

The returned samples are greater than or equal to one.

See also:

scipy.stats.distributions.zipf

probability density function, distribution, or cumulative density function, etc.

Notes

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

References

[R278]

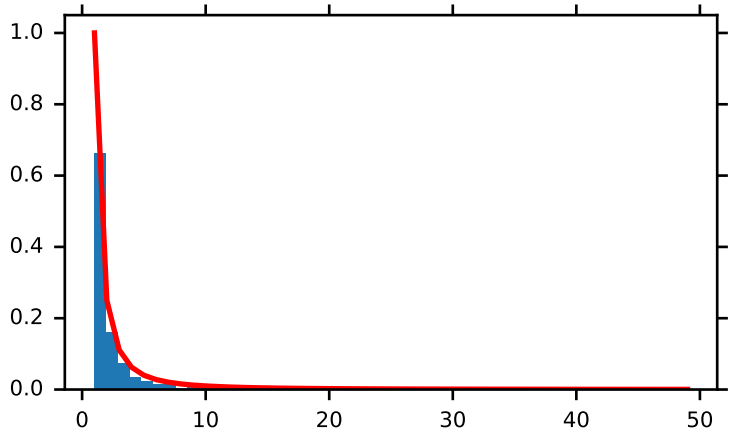
Examples

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```



3.25.4 Random generator

seed([seed])	Seed the generator.
get_state()	Return a tuple representing the internal state of the generator.
set_state(state)	Set the internal state of the generator from a tuple.

numpy.random.**seed**(*seed=None*)

Seed the generator.

This method is called when RandomState is initialized. It can be called again to re-seed the generator. For details, see RandomState.

Parameters

seed : int or array_like, optional

Seed for `RandomState`. Must be convertible to 32 bit unsigned integers.

See also:

`RandomState`

`numpy.random.get_state()`
Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

Returns
out : tuple(str, ndarray of 624 uints, int, int, float)

The returned tuple has the following items:

1. the string 'MT19937'.
2. a 1-D array of 624 unsigned integer keys.
3. an integer `pos`.
4. an integer `has_gauss`.
5. a float `cached_gaussian`.

See also:

set_state

Notes

set_state and *get_state* are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

`numpy.random.set_state(state)`
Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the "Mersenne Twister"*[R261]* pseudo-random number generating algorithm.

Parameters
state : tuple(str, ndarray of 624 uints, int, int, float)

The *state* tuple has the following items:

1. the string 'MT19937', specifying the Mersenne Twister algorithm.
2. a 1-D array of 624 unsigned integers `keys`.
3. an integer `pos`.
4. an integer `has_gauss`.
5. a float `cached_gaussian`.

Returns
out : None

Returns 'None' on success.

See also:

get_state

Notes

set_state and *get_state* are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: `state = ('MT19937', keys, pos)`.

References

[R261]

3.26 Set routines

3.26.1 Making proper sets

3.26.2 Boolean operations

3.27 Sorting, searching, and counting

3.27.1 Sorting

ndarray.sort([axis, kind, order])	Sort an array, in-place.

ndarray.**sort**(*axis=-1*, *kind='quicksort'*, *order=None*)

Sort an array, in-place.

Parameters

axis : int, optional

Axis along which to sort. Default is -1, which means sort along the last axis.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional

Sorting algorithm. Default is 'quicksort'.

order : str or list of str, optional

When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See also:

numpy.sort

Return a sorted copy of an array.

argsort

Indirect sort.

lexsort
Indirect stable sort on multiple keys.

searchsorted
Find elements in sorted array.

partition
Partial sort.

Notes

See `sort` for notes on the different sorting algorithms.

Examples

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

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

3.27.2 Searching

3.27.3 Counting

3.28 Statistics

3.28.1 Order statistics

3.28.2 Averages and variances

3.28.3 Correlating

3.28.4 Histograms

3.29 Test Support (`numpy.testing`)

Common test support for all numpy test scripts.

This single module should provide all the common functionality for numpy tests in a single location, so that test scripts can just import it and work right away.

3.29.1 Asserts

3.29.2 Decorators

decorators.deprecated([conditional])	Filter deprecation warnings while running the test suite.
decorators.knownfailureif(fail_condition[, msg])	Make function raise KnownFailureException exception if given condit
decorators.setastest([tf])	Signals to nose that this function is or is not a test.
decorators.skipif(skip_condition[, msg])	Make function raise SkipTest exception if a given condition is true.
decorators.slow(t)	Label a test as 'slow'.

numpy.testing.decorators.**deprecated**(*conditional=True*)

Filter deprecation warnings while running the test suite.

This decorator can be used to filter DeprecationWarning's, to avoid printing them during the test suite run, while checking that the test actually raises a DeprecationWarning.

Parameters

conditional : bool or callable, optional

Flag to determine whether to mark test as deprecated or not. If the condition is a callable, it is used at runtime to dynamically make the decision. Default is True.

Returns

decorator : function

The *deprecated* decorator itself.

Notes

New in version 1.4.0.

numpy.testing.decorators.**knownfailureif**(*fail_condition*, *msg=None*)

Make function raise KnownFailureException exception if given condition is true.

If the condition is a callable, it is used at runtime to dynamically make the decision. This is useful for tests that may require costly imports, to delay the cost until the test suite is actually executed.

Parameters

fail_condition : bool or callable

Flag to determine whether to mark the decorated test as a known failure (if True) or not (if False).

msg : str, optional

Message to give on raising a KnownFailureException exception. Default is None.

Returns

decorator : function

Decorator, which, when applied to a function, causes KnownFailureException to be raised when *fail_condition* is True, and the function to be called normally otherwise.

Notes

The decorator itself is decorated with the `nose.tools.make_decorator` function in order to transmit function name, and various other metadata.

`numpy.testing.decorators.`**`setastest`**(*tf=True*)

Signals to nose that this function is or is not a test.

Parameters

tf : bool

If True, specifies that the decorated callable is a test. If False, specifies that the decorated callable is not a test. Default is True.

Notes

This decorator can't use the nose namespace, because it can be called from a non-test module. See also `istest` and `nottest` in `nose.tools`.

Examples

`setastest` can be used in the following way:

```
from numpy.testing.decorators import setastest

@setastest(False)
def func_with_test_in_name(arg1, arg2):
    pass
```

`numpy.testing.decorators.`**`skipif`**(*skip_condition*, *msg=None*)

Make function raise SkipTest exception if a given condition is true.

If the condition is a callable, it is used at runtime to dynamically make the decision. This is useful for tests that may require costly imports, to delay the cost until the test suite is actually executed.

Parameters

skip_condition : bool or callable

Flag to determine whether to skip the decorated test.

msg : str, optional

Message to give on raising a SkipTest exception. Default is None.

Returns

decorator : function

Decorator which, when applied to a function, causes SkipTest to be raised when *skip_condition* is True, and the function to be called normally otherwise.

Notes

The decorator itself is decorated with the `nose.tools.make_decorator` function in order to transmit function name, and various other metadata.

`numpy.testing.decorators.`**`slow`**(*t*)

Label a test as 'slow'.

The exact definition of a slow test is obviously both subjective and hardware-dependent, but in general any individual test that requires more than a second or two should be labeled as slow (the whole suite consits of thousands of tests, so even a second is significant).

Parameters

t : callable

The test to label as slow.

Returns

t : callable

The decorated test *t*.

Examples

The `numpy.testing` module includes `import decorators as dec`. A test can be decorated as slow like this:

```
from numpy.testing import *

@dec.slow
def test_big(self):
    print('Big, slow test')
```

3.29.3 Test Running

3.30 Window functions

3.30.1 Various windows

CHAPTER

FOUR

PACKAGING (NUMPY.DISTUTILS)

NumPy provides enhanced distutils functionality to make it easier to build and install sub-packages, auto-generate code, and extension modules that use Fortran-compiled libraries. To use features of NumPy distutils, use the `setup` command from `numpy.distutils.core`. A useful *Configuration* class is also provided in *numpy.distutils.misc_util* that can make it easier to construct keyword arguments to pass to the setup function (by passing the dictionary obtained from the todict() method of the class). More information is available in the NumPy Distutils Users Guide in `<site-packages>/numpy/doc/DISTUTILS.txt`.

4.1 Modules in numpy.distutils

4.1.1 misc_util

get_numpy_include_dirs()	
dict_append(d, **kws)	
appendpath(prefix, path)	
allpath(name)	Convert a /-separated pathname to one using the OS's path separator.
dot_join(*args)	
generate_config_py(target)	Generate config.py file containing system_info information used during building the package.
get_cmd(cmdname[, _cache])	
terminal_has_colors()	
red_text(s)	
green_text(s)	
yellow_text(s)	
blue_text(s)	
cyan_text(s)	
cyg2win32(path)	
all_strings(lst)	Return True if all items in lst are string objects.
has_f_sources(sources)	Return True if sources contains Fortran files
has_cxx_sources(sources)	Return True if sources contains C++ files
filter_sources(sources)	Return four lists of filenames containing C, C++, Fortran, and Fortran 90 module sources, respec
get_dependencies(sources)	
is_local_src_dir(directory)	Return true if directory is local directory.
get_ext_source_files(ext)	
get_script_files(scripts)	

numpy.distutils.misc_util.**get_numpy_include_dirs**()

numpy.distutils.misc_util.**dict_append**(*d*, ***kws*)

numpy.distutils.misc_util.**appendpath**(*prefix*, *path*)

numpy.distutils.misc_util.**allpath**(*name*)
Convert a /-separated pathname to one using the OS's path separator.

numpy.distutils.misc_util.**dot_join**(**args*)

numpy.distutils.misc_util.**generate_config_py**(*target*)
Generate config.py file containing system_info information used during building the package.

Usage:
config['py_modules'].append((packagename, '__config__',generate_config_py))

numpy.distutils.misc_util.**get_cmd**(*cmdname*, *_cache={}*)

numpy.distutils.misc_util.**terminal_has_colors**()

numpy.distutils.misc_util.**red_text**(*s*)

numpy.distutils.misc_util.**green_text**(*s*)

numpy.distutils.misc_util.**yellow_text**(*s*)

numpy.distutils.misc_util.**blue_text**(*s*)

numpy.distutils.misc_util.**cyan_text**(*s*)

numpy.distutils.misc_util.**cyg2win32**(*path*)

numpy.distutils.misc_util.**all_strings**(*lst*)
Return True if all items in lst are string objects.

numpy.distutils.misc_util.**has_f_sources**(*sources*)
Return True if sources contains Fortran files

numpy.distutils.misc_util.**has_cxx_sources**(*sources*)
Return True if sources contains C++ files

numpy.distutils.misc_util.**filter_sources**(*sources*)
Return four lists of filenames containing C, C++, Fortran, and Fortran 90 module sources, respectively.

numpy.distutils.misc_util.**get_dependencies**(*sources*)

numpy.distutils.misc_util.**is_local_src_dir**(*directory*)
Return true if directory is local directory.

numpy.distutils.misc_util.**get_ext_source_files**(*ext*)

numpy.distutils.misc_util.**get_script_files**(*scripts*)

class numpy.distutils.misc_util.**Configuration**(*package_name=None*, *parent_name=None*, *top_path=None*, *package_path=None*, ***attrs*)

Construct a configuration instance for the given package name. If *parent_name* is not None, then construct the package as a sub-package of the *parent_name* package. If *top_path* and *package_path* are None then they are assumed equal to the path of the file this instance was created in. The setup.py files in the numpy distribution are good examples of how to use the *Configuration* instance.

todict()

Return a dictionary compatible with the keyword arguments of distutils setup function.

Examples

```
>>> setup(**config.todict())
```

get_distribution()

Return the distutils distribution object for self.

get_subpackage(*subpackage_name*, *subpackage_path=None*, *parent_name=None*, *caller_level=1*)

Return list of subpackage configurations.

Parameters

subpackage_name : str or None

Name of the subpackage to get the configuration. '*' in subpackage_name is handled as a wildcard.

subpackage_path : str

If None, then the path is assumed to be the local path plus the subpackage_name. If a setup.py file is not found in the subpackage_path, then a default configuration is used.

parent_name : str

Parent name.

add_subpackage(*subpackage_name*, *subpackage_path=None*, *standalone=False*)

Add a sub-package to the current Configuration instance.

This is useful in a setup.py script for adding sub-packages to a package.

Parameters

subpackage_name : str

name of the subpackage

subpackage_path : str

if given, the subpackage path such as the subpackage is in subpackage_path / subpackage_name. If None,the subpackage is assumed to be located in the local path / subpackage_name.

standalone : bool

add_data_files(**files*)

Add data files to configuration data_files.

Parameters

files : sequence

Argument(s) can be either

- 2-sequence (<datadir prefix>,<path to data file(s)>)
- paths to data files where python datadir prefix defaults to package dir.

Notes

The form of each element of the files sequence is very flexible allowing many combinations of where to get the files from the package and where they should ultimately be installed on the system. The most basic usage is for an element of the files argument sequence to be a simple filename. This will cause that file from the local path to be installed to the installation path of the self.name package (package path). The file argument can also be a relative path in which case the entire relative path will be installed into the package directory. Finally, the file can be an absolute path name in which case the file will be found at the absolute path name but installed to the package path.

This basic behavior can be augmented by passing a 2-tuple in as the file argument. The first element of the tuple should specify the relative path (under the package install directory) where the remaining sequence of files should be installed to (it has nothing to do with the file-names in the source distribution). The second element of the tuple is the sequence of files that should be installed. The files in this sequence can be filenames, relative paths, or absolute paths. For absolute paths the file will be installed in the top-level package installation directory (regardless of the first argument). Filenames and relative path names will be installed in the package install directory under the path name given as the first element of the tuple.

Rules for installation paths:

1. file.txt -> (., file.txt)-> parent/file.txt
2. foo/file.txt -> (foo, foo/file.txt) -> parent/foo/file.txt
3. /foo/bar/file.txt -> (., /foo/bar/file.txt) -> parent/file.txt
4. *.txt -> parent/a.txt, parent/b.txt
5. foo/*.txt -> parent/foo/a.txt, parent/foo/b.txt
6. */*.txt -> (*, */*.txt) -> parent/c/a.txt, parent/d/b.txt
7. (sun, file.txt) -> parent/sun/file.txt
8. (sun, bar/file.txt) -> parent/sun/file.txt
9. (sun, /foo/bar/file.txt) -> parent/sun/file.txt
10. (sun, *.txt) -> parent/sun/a.txt, parent/sun/b.txt
11. (sun, bar/*.txt) -> parent/sun/a.txt, parent/sun/b.txt
12. (sun/*, */*.txt) -> parent/sun/c/a.txt, parent/d/b.txt

An additional feature is that the path to a data-file can actually be a function that takes no arguments and returns the actual path(s) to the data-files. This is useful when the data files are generated while building the package.

Examples

Add files to the list of data_files to be included with the package.

```
>>> self.add_data_files('foo.dat',
...     ('fun', ['gun.dat', 'nun/pun.dat', '/tmp/sun.dat']),
...     'bar/cat.dat',
...     '/full/path/to/can.dat')
```

will install these data files to:

```
<package install directory>/
 foo.dat
 fun/
   gun.dat
   nun/
```

```
        pun.dat
    sun.dat
    bar/
      car.dat
    can.dat
```

where <package install directory> is the package (or sub-package) directory such as '/usr/lib/python2.4/site-packages/mypackage' ('C: Python2.4 Lib site-packages mypackage') or '/usr/lib/python2.4/site- packages/mypackage/mysubpackage' ('C: Python2.4 Lib site-packages mypackage mysubpackage').

add_data_dir(*data_path*)

Recursively add files under data_path to data_files list.

Recursively add files under data_path to the list of data_files to be installed (and distributed). The data_path can be either a relative path-name, or an absolute path-name, or a 2-tuple where the first argument shows where in the install directory the data directory should be installed to.

Parameters

data_path : seq or str

Argument can be either

- 2-sequence (<datadir suffix>, <path to data directory>)
- path to data directory where python datadir suffix defaults to package dir.

Notes

Rules for installation paths:

foo/bar -> (foo/bar, foo/bar) -> parent/foo/bar (gun, foo/bar) -> parent/gun foo/* -> (foo/a, foo/a), (foo/b, foo/b) -> parent/foo/a, parent/foo/b (gun, foo/*) -> *(gun, foo/a), (gun, foo/b)* -> *gun (gun/*, foo/*)* -> *parent/gun/a, parent/gun/b /foo/bar -> (bar, /foo/bar) -> parent/bar (gun, /foo/bar) -> parent/gun (fun/*/gun/*, sun/foo/bar) -> parent/fun/foo/gun/bar

Examples

For example suppose the source directory contains fun/foo.dat and fun/bar/car.dat:

```
>>> self.add_data_dir('fun')
>>> self.add_data_dir(('sun', 'fun'))
>>> self.add_data_dir(('gun', '/full/path/to/fun'))
```

Will install data-files to the locations:

```
<package install directory>/
  fun/
    foo.dat
    bar/
      car.dat
  sun/
    foo.dat
    bar/
      car.dat
  gun/
    foo.dat
    car.dat
```

add_include_dirs(**paths*)

Add paths to configuration include directories.

Add the given sequence of paths to the beginning of the include_dirs list. This list will be visible to all extension modules of the current package.

add_headers (**files*)
Add installable headers to configuration.

Add the given sequence of files to the beginning of the headers list. By default, headers will be installed under <python- include>/<self.name.replace('.','/')>/ directory. If an item of files is a tuple, then its first argument specifies the actual installation location relative to the <python-include> path.

Parameters
files : str or seq

Argument(s) can be either:

- 2-sequence (<includedir suffix>,<path to header file(s)>)
- path(s) to header file(s) where python includedir suffix will default to package name.

add_extension (*name*, *sources*, ***kw*)
Add extension to configuration.

Create and add an Extension instance to the ext_modules list. This method also takes the following optional keyword arguments that are passed on to the Extension constructor.

Parameters
name : str

name of the extension

sources : seq

list of the sources. The list of sources may contain functions (called source generators) which must take an extension instance and a build directory as inputs and return a source file or list of source files or None. If None is returned then no sources are generated. If the Extension instance has no sources after processing all source generators, then no extension module is built.

include_dirs :

define_macros :

undef_macros :

library_dirs :

libraries :

runtime_library_dirs :

extra_objects :

extra_compile_args :

extra_link_args :

extra_f77_compile_args :

extra_f90_compile_args :

export_symbols :

swig_opts :

depends :

The depends list contains paths to files or directories that the sources of the extension module depend on. If any path in the depends list is newer than the extension module, then the module will be rebuilt.

language :

f2py_options :

module_dirs :

extra_info : dict or list

dict or list of dict of keywords to be appended to keywords.

Notes

The self.paths(...) method is applied to all lists that may contain paths.

add_library(*name*, *sources*, ***build_info*)
Add library to configuration.

Parameters
name : str

Name of the extension.

sources : sequence

List of the sources. The list of sources may contain functions (called source generators) which must take an extension instance and a build directory as inputs and return a source file or list of source files or None. If None is returned then no sources are generated. If the Extension instance has no sources after processing all source generators, then no extension module is built.

build_info : dict, optional

The following keys are allowed:

- depends
- macros
- include_dirs
- extra_compiler_args
- extra_f77_compiler_args
- extra_f90_compiler_args
- f2py_options
- language

add_scripts(**files*)
Add scripts to configuration.

Add the sequence of files to the beginning of the scripts list. Scripts will be installed under the <prefix>/bin/ directory.

add_installed_library(*name*, *sources*, *install_dir*, *build_info=None*)
Similar to add_library, but the specified library is installed.

Most C libraries used with `distutils` are only used to build python extensions, but libraries built through this method will be installed so that they can be reused by third-party packages.

Parameters

name : str

Name of the installed library.

sources : sequence

List of the library's source files. See *add_library* for details.

install_dir : str

Path to install the library, relative to the current sub-package.

build_info : dict, optional

The following keys are allowed:

- depends
- macros
- include_dirs
- extra_compiler_args
- extra_f77_compiler_args
- extra_f90_compiler_args
- f2py_options
- language

Returns

None

See also:

add_library, *add_npy_pkg_config*, *get_info*

Notes

The best way to encode the options required to link against the specified C libraries is to use a "libname.ini" file, and use *get_info* to retrieve the required options (see *add_npy_pkg_config* for more information).

add_npy_pkg_config(*template*, *install_dir*, *subst_dict=None*)

Generate and install a npy-pkg config file from a template.

The config file generated from *template* is installed in the given install directory, using *subst_dict* for variable substitution.

Parameters

template : str

The path of the template, relatively to the current package path.

install_dir : str

Where to install the npy-pkg config file, relatively to the current package path.

subst_dict : dict, optional

If given, any string of the form `@key@` will be replaced by `subst_dict[key]` in the template file when installed. The install prefix is always available through the variable `@prefix@`, since the install prefix is not easy to get reliably from setup.py.

See also:

add_installed_library, *get_info*

Notes

This works for both standard installs and in-place builds, i.e. the `@prefix@` refer to the source directory for in-place builds.

Examples

```
config.add_npy_pkg_config('foo.ini.in', 'lib', {'foo': bar})
```

Assuming the foo.ini.in file has the following content:

```
[meta]
Name=@foo@
Version=1.0
Description=dummy description

[default]
Cflags=-I@prefix@/include
Libs=
```

The generated file will have the following content:

```
[meta]
Name=bar
Version=1.0
Description=dummy description

[default]
Cflags=-Iprefix_dir/include
Libs=
```

and will be installed as foo.ini in the 'lib' subpath.

paths(**paths*, ***kws*)
Apply glob to paths and prepend local_path if needed.

Applies glob.glob(...) to each path in the sequence (if needed) and pre-pends the local_path if needed. Because this is called on all source lists, this allows wildcard characters to be specified in lists of sources for extension modules and libraries and scripts and allows path-names be relative to the source directory.

get_config_cmd()
Returns the numpy.distutils config command instance.

get_build_temp_dir()
Return a path to a temporary directory where temporary files should be placed.

have_f77c()
Check for availability of Fortran 77 compiler.

Use it inside source generating function to ensure that setup distribution instance has been initialized.

Notes

True if a Fortran 77 compiler is available (because a simple Fortran 77 code was able to be compiled successfully).

have_f90c()
Check for availability of Fortran 90 compiler.

Use it inside source generating function to ensure that setup distribution instance has been initialized.

Notes

True if a Fortran 90 compiler is available (because a simple Fortran 90 code was able to be compiled successfully)

get_version(*version_file=None*, *version_variable=None*)
Try to get version string of a package.

Return a version string of the current package or None if the version information could not be detected.

Notes

This method scans files named __version__.py, <packagename>_version.py, version.py, and __svn_version__.py for string variables version, __version__, and <packagename>_version, until a version number is found.

make_svn_version_py(*delete=True*)
Appends a data function to the data_files list that will generate __svn_version__.py file to the current package directory.

Generate package __svn_version__.py file from SVN revision number, it will be removed after python exits but will be available when sdist, etc commands are executed.

Notes

If __svn_version__.py existed before, nothing is done.

This is intended for working with source directories that are in an SVN repository.

make_config_py(*name='__config__'*)
Generate package __config__.py file containing system_info information used during building the package.

This file is installed to the package installation directory.

get_info(**names*)
Get resources information.

Return information (from system_info.get_info) for all of the names in the argument list in a single dictionary.

4.1.2 Other modules

system_info.get_info(name[, notfound_action])	notfound_action:
system_info.get_standard_file(fname)	Returns a list of files named 'fname' from
cpuinfo.cpu	
log.set_verbosity(v[, force])	
exec_command	exec_command

numpy.distutils.system_info.**get_info**(*name*, *notfound_action=0*)

notfound_action:
0 - do nothing 1 - display warning message 2 - raise error

numpy.distutils.system_info.**get_standard_file**(*fname*)
Returns a list of files named 'fname' from 1) System-wide directory (directory-location of this module) 2)

Users HOME directory (os.environ['HOME']) 3) Local directory

numpy.distutils.cpuinfo.**cpu = <numpy.distutils.cpuinfo.LinuxCPUInfo object>**

numpy.distutils.log.**set_verbosity**(*v*, *force=False*)

exec_command

Implements exec_command function that is (almost) equivalent to commands.getstatusoutput function but on NT, DOS systems the returned status is actually correct (though, the returned status values may be different by a factor). In addition, exec_command takes keyword arguments for (re-)defining environment variables.

Provides functions:

exec_command — execute command in a specified directory and
in the modified environment.

find_executable — locate a command using info from environment
variable PATH. Equivalent to posix *which* command.

Author: Pearu Peterson <pearu@cens.ioc.ee> Created: 11 January 2003

Requires: Python 2.x

Succesfully tested on:

os.name	sys.platform	comments
posix	linux2	Debian (sid) Linux, Python 2.1.3+, 2.2.3+, 2.3.3 PyCrust 0.9.3, Idle 1.0.2
posix	linux2	Red Hat 9 Linux, Python 2.1.3, 2.2.2, 2.3.2
posix	sunos5	SunOS 5.9, Python 2.2, 2.3.2
posix	darwin	Darwin 7.2.0, Python 2.3
nt	win32	Windows Me Python 2.3(EE), Idle 1.0, PyCrust 0.7.2 Python 2.1.1 Idle 0.8
nt	win32	Windows 98, Python 2.1.1. Idle 0.8
nt	win32	Cygwin 98-4.10, Python 2.1.1(MSC) - echo tests fail i.e. redefining environment variables may not work. FIXED: don't use cygwin echo! Comment: also *cmd /c echo* will not work but redefining environment variables do work.
posix	cygwin	Cygwin 98-4.10, Python 2.3.3(cygming special)
nt	win32	Windows XP, Python 2.3.3

Known bugs:

- Tests, that send messages to stderr, fail when executed from MSYS prompt because the messages are lost at some point.

Functions

exec_command(command[, execute_in, ...])	Return (status,output) of executed command.
find_executable(exe[, path, _cache])	Return full path of a executable or None.
get_pythonexe()	
quote_arg(arg)	
temp_file_name()	
test(**kws)	
test_cl(**kws)	
test_execute_in(**kws)	
test_nt(**kws)	
test_posix(**kws)	
test_svn(**kws)	

4.2 Building Installable C libraries

Conventional C libraries (installed through *add_library*) are not installed, and are just used during the build (they are statically linked). An installable C library is a pure C library, which does not depend on the python C runtime, and is installed such that it may be used by third-party packages. To build and install the C library, you just use the method *add_installed_library* instead of *add_library*, which takes the same arguments except for an additional `install_dir` argument:

```
>>> config.add_installed_library('foo', sources=['foo.c'], install_dir='lib')
```

4.2.1 npy-pkg-config files

To make the necessary build options available to third parties, you could use the *npy-pkg-config* mechanism implemented in *numpy.distutils*. This mechanism is based on a .ini file which contains all the options. A .ini file is very similar to .pc files as used by the pkg-config unix utility:

```
[meta]
Name: foo
Version: 1.0
Description: foo library

[variables]
prefix = /home/user/local
libdir = ${prefix}/lib
includedir = ${prefix}/include

[default]
cflags = -I${includedir}
libs = -L${libdir} -lfoo
```

Generally, the file needs to be generated during the build, since it needs some information known at build time only (e.g. prefix). This is mostly automatic if one uses the *Configuration* method *add_npy_pkg_config*. Assuming we have a template file foo.ini.in as follows:

```
[meta]
Name: foo
Version: @version@
Description: foo library

[variables]
prefix = @prefix@
libdir = ${prefix}/lib
includedir = ${prefix}/include

[default]
cflags = -I${includedir}
libs = -L${libdir} -lfoo
```

and the following code in setup.py:

```
>>> config.add_installed_library('foo', sources=['foo.c'], install_dir='lib')
>>> subst = {'version': '1.0'}
>>> config.add_npy_pkg_config('foo.ini.in', 'lib', subst_dict=subst)
```

This will install the file foo.ini into the directory package_dir/lib, and the foo.ini file will be generated from foo.ini.in, where each `@version@` will be replaced by `subst_dict['version']`. The dictionary has an additional prefix substitution rule automatically added, which contains the install prefix (since this is not easy to get from setup.py).

npy-pkg-config files can also be installed at the same location as used for numpy, using the path returned from *get_npy_pkg_dir* function.

4.2.2 Reusing a C library from another package

Info are easily retrieved from the *get_info* function in *numpy.distutils.misc_util*:

```
>>> info = get_info('npymath')
>>> config.add_extension('foo', sources=['foo.c'], extra_info=**info)
```

An additional list of paths to look for .ini files can be given to *get_info*.

4.3 Conversion of `.src` files

NumPy distutils supports automatic conversion of source files named <somefile>.src. This facility can be used to maintain very similar code blocks requiring only simple changes between blocks. During the build phase of setup, if a template file named <somefile>.src is encountered, a new file named <somefile> is constructed from the template and placed in the build directory to be used instead. Two forms of template conversion are supported. The first form occurs for files named named <file>.ext.src where ext is a recognized Fortran extension (f, f90, f95, f77, for, ftn, pyf). The second form is used for all other cases.

4.3.1 Fortran files

This template converter will replicate all **function** and **subroutine** blocks in the file with names that contain '<...>' according to the rules in '<...>'. The number of comma-separated words in '<...>' determines the number of times the block is repeated. What these words are indicates what that repeat rule, '<...>', should be replaced with in each block. All of the repeat rules in a block must contain the same number of comma-separated words indicating the number of times that block should be repeated. If the word in the repeat rule needs a comma, leftarrow, or rightarrow, then prepend it with a backslash ' '. If a word in the repeat rule matches ' \<index>' then it will be replaced with the <index>-th word in the same repeat specification. There are two forms for the repeat rule: named and short.

Named repeat rule

A named repeat rule is useful when the same set of repeats must be used several times in a block. It is specified using <rule1=item1, item2, item3,..., itemN>, where N is the number of times the block should be repeated. On each repeat of the block, the entire expression, '<...>' will be replaced first with item1, and then with item2, and so forth until N repeats are accomplished. Once a named repeat specification has been introduced, the same repeat rule may be used **in the current block** by referring only to the name (i.e. <rule1>.

Short repeat rule

A short repeat rule looks like <item1, item2, item3, ..., itemN>. The rule specifies that the entire expression, '<...>' should be replaced first with item1, and then with item2, and so forth until N repeats are accomplished.

Pre-defined names

The following predefined named repeat rules are available:

- <prefix=s,d,c,z>

- <_c=s,d,c,z>
- <_t=real, double precision, complex, double complex>
- <ftype=real, double precision, complex, double complex>
- <ctype=float, double, complex_float, complex_double>
- <ftypereal=float, double precision, \0, \1>
- <ctypereal=float, double, \0, \1>

4.3.2 Other files

Non-Fortran files use a separate syntax for defining template blocks that should be repeated using a variable expansion similar to the named repeat rules of the Fortran-specific repeats. The template rules for these files are:

1. "/**begin repeat "on a line by itself marks the beginning of a segment that should be repeated.
2. Named variable expansions are defined using #name=item1, item2, item3, ..., itemN# and placed on successive lines. These variables are replaced in each repeat block with corresponding word. All named variables in the same repeat block must define the same number of words.
3. In specifying the repeat rule for a named variable, item*N is short- hand for item, item, ..., item repeated N times. In addition, parenthesis in combination with *N can be used for grouping several items that should be repeated. Thus, #name=(item1, item2)*4# is equivalent to #name=item1, item2, item1, item2, item1, item2, item1, item2#
4. "*/ "on a line by itself marks the end of the the variable expansion naming. The next line is the first line that will be repeated using the named rules.
5. Inside the block to be repeated, the variables that should be expanded are specified as @name@.
6. "/**end repeat**/ "on a line by itself marks the previous line as the last line of the block to be repeated.

CHAPTER

FIVE

NUMPY C-API

Beware of the man who won't be bothered with details.
— *William Feather, Sr.*

The truth is out there.
— *Chris Carter, The X Files*

NumPy provides a C-API to enable users to extend the system and get access to the array object for use in other routines. The best way to truly understand the C-API is to read the source code. If you are unfamiliar with (C) source code, however, this can be a daunting experience at first. Be assured that the task becomes easier with practice, and you may be surprised at how simple the C-code can be to understand. Even if you don't think you can write C-code from scratch, it is much easier to understand and modify already-written source code then create it *de novo*.

Python extensions are especially straightforward to understand because they all have a very similar structure. Admittedly, NumPy is not a trivial extension to Python, and may take a little more snooping to grasp. This is especially true because of the code-generation techniques, which simplify maintenance of very similar code, but can make the code a little less readable to beginners. Still, with a little persistence, the code can be opened to your understanding. It is my hope, that this guide to the C-API can assist in the process of becoming familiar with the compiled-level work that can be done with NumPy in order to squeeze that last bit of necessary speed out of your code.

5.1 Python Types and C-Structures

Several new types are defined in the C-code. Most of these are accessible from Python, but a few are not exposed due to their limited use. Every new Python type has an associated `PyObject *` with an internal structure that includes a pointer to a "method table" that defines how the new object behaves in Python. When you receive a Python object into C code, you always get a pointer to a `PyObject` structure. Because a `PyObject` structure is very generic and defines only `PyObject_HEAD`, by itself it is not very interesting. However, different objects contain more details after the `PyObject_HEAD` (but you have to cast to the correct type to access them — or use accessor functions or macros).

5.1.1 New Python Types Defined

Python types are the functional equivalent in C of classes in Python. By constructing a new Python type you make available a new object for Python. The ndarray object is an example of a new type defined in C. New types are defined in C by two basic steps:

1. creating a C-structure (usually named `Py{Name}Object`) that is binary- compatible with the `PyObject` structure itself but holds the additional information needed for that particular object;

2. populating the PyTypeObject table (pointed to by the ob_type member of the PyObject structure) with pointers to functions that implement the desired behavior for the type.

Instead of special method names which define behavior for Python classes, there are "function tables" which point to functions that implement the desired results. Since Python 2.2, the PyTypeObject itself has become dynamic which allows C types that can be "sub-typed "from other C-types in C, and sub-classed in Python. The children types inherit the attributes and methods from their parent(s).

There are two major new types: the ndarray (PyArray_Type) and the ufunc (PyUFunc_Type). Additional types play a supportive role: the PyArrayIter_Type, the PyArrayMultiIter_Type, and the PyArrayDescr_Type . The PyArrayIter_Type is the type for a flat iterator for an ndarray (the object that is returned when getting the flat attribute). The PyArrayMultiIter_Type is the type of the object returned when calling broadcast (). It handles iteration and broadcasting over a collection of nested sequences. Also, the PyArrayDescr_Type is the data-type-descriptor type whose instances describe the data. Finally, there are 21 new scalar-array types which are new Python scalars corresponding to each of the fundamental data types available for arrays. An additional 10 other types are place holders that allow the array scalars to fit into a hierarchy of actual Python types.

PyArray_Type

PyArrayObject

The *PyArrayObject* C-structure contains all of the required information for an array. All instances of an ndarray (and its subclasses) will have this structure. For future compatibility, these structure members should normally be accessed using the provided macros. If you need a shorter name, then you can make use of NPY_AO which is defined to be equivalent to *PyArrayObject*.

```
typedef struct PyArrayObject {
    PyObject_HEAD
    char *data;
    int nd;
    npy_intp *dimensions;
    npy_intp *strides;
    PyObject *base;
    PyArray_Descr *descr;
    int flags;
    PyObject *weakreflist;
} PyArrayObject;
```

char ***PyArrayObject.data**

A pointer to the first element of the array. This pointer can (and normally should) be recast to the data type of the array.

int **PyArrayObject.nd**

An integer providing the number of dimensions for this array. When nd is 0, the array is sometimes called a rank-0 array. Such arrays have undefined dimensions and strides and cannot be accessed. *NPY_MAXDIMS* is the largest number of dimensions for any array.

npy_intp **PyArrayObject.dimensions**

An array of integers providing the shape in each dimension as long as nd ≥ 1. The integer is always large enough to hold a pointer on the platform, so the dimension size is only limited by memory.

npy_intp ***PyArrayObject.strides**

An array of integers providing for each dimension the number of bytes that must be skipped to get to the next element in that dimension.

PyObject ***PyArrayObject.base**

This member is used to hold a pointer to another Python object that is related to this array. There are two use cases: 1) If this array does not own its own memory, then base points to the Python object that owns it (perhaps

another array object), 2) If this array has the *NPY_ARRAY_UPDATEIFCOPY* flag set, then this array is a working copy of a "misbehaved" array. As soon as this array is deleted, the array pointed to by base will be updated with the contents of this array.

PyArray_Descr ***PyArrayObject.descr**
A pointer to a data-type descriptor object (see below). The data-type descriptor object is an instance of a new built-in type which allows a generic description of memory. There is a descriptor structure for each data type supported. This descriptor structure contains useful information about the type as well as a pointer to a table of function pointers to implement specific functionality.

int **PyArrayObject.flags**
Flags indicating how the memory pointed to by data is to be interpreted. Possible flags are *NPY_ARRAY_C_CONTIGUOUS*, *NPY_ARRAY_F_CONTIGUOUS*, *NPY_ARRAY_OWNDATA*, *NPY_ARRAY_ALIGNED*, *NPY_ARRAY_WRITEABLE*, and *NPY_ARRAY_UPDATEIFCOPY*.

PyObject ***PyArrayObject.weakreflist**
This member allows array objects to have weak references (using the weakref module).

PyArrayDescr_Type

PyArray_Descr
The format of the *PyArray_Descr* structure that lies at the heart of the `PyArrayDescr_Type` is

```
typedef struct {
    PyObject_HEAD
    PyTypeObject *typeobj;
    char kind;
    char type;
    char byteorder;
    char unused;
    int flags;
    int type_num;
    int elsize;
    int alignment;
    PyArray_ArrayDescr *subarray;
    PyObject *fields;
    PyArray_ArrFuncs *f;
} PyArray_Descr;
```

PyTypeObject ***PyArray_Descr.typeobj**
Pointer to a typeobject that is the corresponding Python type for the elements of this array. For the builtin types, this points to the corresponding array scalar. For user-defined types, this should point to a user-defined typeobject. This typeobject can either inherit from array scalars or not. If it does not inherit from array scalars, then the `NPY_USE_GETITEM` and `NPY_USE_SETITEM` flags should be set in the `flags` member.

char **PyArray_Descr.kind**
A character code indicating the kind of array (using the array interface typestring notation). A 'b' represents Boolean, a 'i' represents signed integer, a 'u' represents unsigned integer, 'f' represents floating point, 'c' represents complex floating point, 'S' represents 8-bit character string, 'U' represents 32-bit/character unicode string, and 'V' repesents arbitrary.

char **PyArray_Descr.type**
A traditional character code indicating the data type.

char **PyArray_Descr.byteorder**
A character indicating the byte-order: '>' (big-endian), '<' (little- endian), '=' (native), 'I' (irrelevant, ignore). All builtin data- types have byteorder '='.

int **PyArray_Descr.flags**
A data-type bit-flag that determines if the data-type exhibits object- array like behavior. Each bit in this member is a flag which are named as:

PyDataType_FLAGCHK (*PyArray_Descr* *dtype*, int *flags*)
Return true if all the given flags are set for the data-type object.

PyDataType_REFCHK (*PyArray_Descr* *dtype*)
Equivalent to *PyDataType_FLAGCHK* (*dtype*, NPY_ITEM_REFCOUNT).

int **PyArray_Descr.type_num**
A number that uniquely identifies the data type. For new data-types, this number is assigned when the data-type is registered.

int **PyArray_Descr.elsize**
For data types that are always the same size (such as long), this holds the size of the data type. For flexible data types where different arrays can have a different elementsize, this should be 0.

int **PyArray_Descr.alignment**
A number providing alignment information for this data type. Specifically, it shows how far from the start of a 2-element structure (whose first element is a `char`), the compiler places an item of this type: `offsetof(struct {char c; type v;}, v)`

PyArray_ArrayDescr ***PyArray_Descr.subarray**
If this is non- NULL, then this data-type descriptor is a C-style contiguous array of another data-type descriptor. In other-words, each element that this descriptor describes is actually an array of some other base descriptor. This is most useful as the data-type descriptor for a field in another data-type descriptor. The fields member should be NULL if this is non- NULL (the fields member of the base descriptor can be non- NULL however). The PyArray_ArrayDescr structure is defined using

```
typedef struct {
    PyArray_Descr *base;
    PyObject *shape;
} PyArray_ArrayDescr;
```

The elements of this structure are:

PyArray_Descr ***PyArray_ArrayDescr.base**
The data-type-descriptor object of the base-type.

PyObject ***PyArray_ArrayDescr.shape**
The shape (always C-style contiguous) of the sub-array as a Python tuple.

PyObject ***PyArray_Descr.fields**
If this is non-NULL, then this data-type-descriptor has fields described by a Python dictionary whose keys are names (and also titles if given) and whose values are tuples that describe the fields. Recall that a data-type-descriptor always describes a fixed-length set of bytes. A field is a named sub-region of that total, fixed-length collection. A field is described by a tuple composed of another data- type-descriptor and a byte offset. Optionally, the tuple may contain a title which is normally a Python string. These tuples are placed in this dictionary keyed by name (and also title if given).

PyArray_ArrFuncs ***PyArray_Descr.f**
A pointer to a structure containing functions that the type needs to implement internal features. These functions are not the same thing as the universal functions (ufuncs) described later. Their signatures can vary arbitrarily.

PyArray_ArrFuncs
Functions implementing internal features. Not all of these function pointers must be defined for a given type. The required members are `nonzero`, `copyswap`, `copyswapn`, `setitem`, `getitem`, and `cast`. These are assumed to be non- NULL and NULL entries will cause a program crash. The other functions may be NULL

which will just mean reduced functionality for that data-type. (Also, the nonzero function will be filled in with a default function if it is NULL when you register a user-defined data-type).

```
typedef struct {
    PyArray_VectorUnaryFunc *cast[NPY_NTYPES];
    PyArray_GetItemFunc *getitem;
    PyArray_SetItemFunc *setitem;
    PyArray_CopySwapNFunc *copyswapn;
    PyArray_CopySwapFunc *copyswap;
    PyArray_CompareFunc *compare;
    PyArray_ArgFunc *argmax;
    PyArray_DotFunc *dotfunc;
    PyArray_ScanFunc *scanfunc;
    PyArray_FromStrFunc *fromstr;
    PyArray_NonzeroFunc *nonzero;
    PyArray_FillFunc *fill;
    PyArray_FillWithScalarFunc *fillwithscalar;
    PyArray_SortFunc *sort[NPY_NSORTS];
    PyArray_ArgSortFunc *argsort[NPY_NSORTS];
    PyObject *castdict;
    PyArray_ScalarKindFunc *scalarkind;
    int **cancastscalarkindto;
    int *cancastto;
    PyArray_FastClipFunc *fastclip;
    PyArray_FastPutmaskFunc *fastputmask;
    PyArray_FastTakeFunc *fasttake;
    PyArray_ArgFunc *argmin;
} PyArray_ArrFuncs;
```

The concept of a behaved segment is used in the description of the function pointers. A behaved segment is one that is aligned and in native machine byte-order for the data-type. The `nonzero`, `copyswap`, `copyswapn`, `getitem`, and `setitem` functions can (and must) deal with mis-behaved arrays. The other functions require behaved memory segments.

void **cast**(void **from*, void **to*, npy_intp *n*, void **fromarr*, void **toarr*)
: An array of function pointers to cast from the current type to all of the other builtin types. Each function casts a contiguous, aligned, and notswapped buffer pointed at by *from* to a contiguous, aligned, and notswapped buffer pointed at by *to* The number of items to cast is given by *n*, and the arguments *fromarr* and *toarr* are interpreted as PyArrayObjects for flexible arrays to get itemsize information.

PyObject ***getitem**(void **data*, void **arr*)
: A pointer to a function that returns a standard Python object from a single element of the array object *arr* pointed to by *data*. This function must be able to deal with "misbehaved "(misaligned and/or swapped) arrays correctly.

int **setitem**(PyObject **item*, void **data*, void **arr*)
: A pointer to a function that sets the Python object *item* into the array, *arr*, at the position pointed to by *data* . This function deals with "misbehaved" arrays. If successful, a zero is returned, otherwise, a negative one is returned (and a Python error set).

void **copyswapn**(void **dest*, npy_intp *dstride*, void **src*, npy_intp *sstride*, npy_intp *n*, int *swap*, void **arr*)

void **copyswap**(void **dest*, void **src*, int *swap*, void **arr*)
: These members are both pointers to functions to copy data from *src* to *dest* and *swap* if indicated. The value of arr is only used for flexible (*NPY_STRING*, *NPY_UNICODE*, and *NPY_VOID*) arrays (and is obtained from `arr->descr->elsize`). The second function copies a single value, while the first loops over n values with the provided strides. These functions can deal with misbehaved *src* data. If *src* is NULL then no copy is performed. If *swap* is 0, then no byteswapping occurs. It is assumed that *dest* and

src do not overlap. If they overlap, then use `memmove` (...) first followed by `copyswap(n)` with NULL valued `src`.

int **compare**(const void* *d1*, const void* *d2*, void* *arr*)
A pointer to a function that compares two elements of the array, `arr`, pointed to by `d1` and `d2`. This function requires behaved (aligned and not swapped) arrays. The return value is 1 if * `d1` > * `d2`, 0 if * `d1` == * `d2`, and -1 if * `d1` < * `d2`. The array object `arr` is used to retrieve itemsize and field information for flexible arrays.

int **argmax**(void* *data*, npy_intp *n*, npy_intp* *max_ind*, void* *arr*)
A pointer to a function that retrieves the index of the largest of `n` elements in `arr` beginning at the element pointed to by `data`. This function requires that the memory segment be contiguous and behaved. The return value is always 0. The index of the largest element is returned in `max_ind`.

void **dotfunc**(void* *ip1*, npy_intp *is1*, void* *ip2*, npy_intp *is2*, void* *op*, npy_intp *n*, void* *arr*)
A pointer to a function that multiplies two `n` -length sequences together, adds them, and places the result in element pointed to by `op` of `arr`. The start of the two sequences are pointed to by `ip1` and `ip2`. To get to the next element in each sequence requires a jump of `is1` and `is2` *bytes*, respectively. This function requires behaved (though not necessarily contiguous) memory.

int **scanfunc**(FILE* *fd*, void* *ip*, void* *sep*, void* *arr*)
A pointer to a function that scans (scanf style) one element of the corresponding type from the file descriptor `fd` into the array memory pointed to by `ip`. The array is assumed to be behaved. If `sep` is not NULL, then a separator string is also scanned from the file before returning. The last argument `arr` is the array to be scanned into. A 0 is returned if the scan is successful. A negative number indicates something went wrong: -1 means the end of file was reached before the separator string could be scanned, -4 means that the end of file was reached before the element could be scanned, and -3 means that the element could not be interpreted from the format string. Requires a behaved array.

int **fromstr**(char* *str*, void* *ip*, char** *endptr*, void* *arr*)
A pointer to a function that converts the string pointed to by `str` to one element of the corresponding type and places it in the memory location pointed to by `ip`. After the conversion is completed, `*endptr` points to the rest of the string. The last argument `arr` is the array into which ip points (needed for variable-size data- types). Returns 0 on success or -1 on failure. Requires a behaved array.

Bool **nonzero**(void* *data*, void* *arr*)
A pointer to a function that returns TRUE if the item of `arr` pointed to by `data` is nonzero. This function can deal with misbehaved arrays.

void **fill**(void* *data*, npy_intp *length*, void* *arr*)
A pointer to a function that fills a contiguous array of given length with data. The first two elements of the array must already be filled- in. From these two values, a delta will be computed and the values from item 3 to the end will be computed by repeatedly adding this computed delta. The data buffer must be well-behaved.

void **fillwithscalar**(void* *buffer*, npy_intp *length*, void* *value*, void* *arr*)
A pointer to a function that fills a contiguous `buffer` of the given `length` with a single scalar `value` whose address is given. The final argument is the array which is needed to get the itemsize for variable-length arrays.

int **sort**(void* *start*, npy_intp *length*, void* *arr*)
An array of function pointers to a particular sorting algorithms. A particular sorting algorithm is obtained using a key (so far `NPY_QUICKSORT`, :data‘NPY_HEAPSORT‘, and `NPY_MERGESORT` are defined). These sorts are done in-place assuming contiguous and aligned data.

int **argsort**(void* *start*, npy_intp* *result*, npy_intp *length*, void **arr*)
An array of function pointers to sorting algorithms for this data type. The same sorting algorithms as for sort are available. The indices producing the sort are returned in `result` (which must be initialized with indices 0 to `length-1` inclusive).

PyObject ***castdict**
: Either `NULL` or a dictionary containing low-level casting functions for user- defined data-types. Each function is wrapped in a `PyCObject *` and keyed by the data-type number.

NPY_SCALARKIND **scalarkind**(*PyArrayObject** *arr*)
: A function to determine how scalars of this type should be interpreted. The argument is `NULL` or a 0-dimensional array containing the data (if that is needed to determine the kind of scalar). The return value must be of type *NPY_SCALARKIND*.

int ****cancastscalarkindto**
: Either `NULL` or an array of *NPY_NSCALARKINDS* pointers. These pointers should each be either `NULL` or a pointer to an array of integers (terminated by *NPY_NOTYPE*) indicating data-types that a scalar of this data-type of the specified kind can be cast to safely (this usually means without losing precision).

int ***cancastto**
: Either `NULL` or an array of integers (terminated by *NPY_NOTYPE*) indicated data-types that this data-type can be cast to safely (this usually means without losing precision).

void **fastclip**(void **in*, npy_intp *n_in*, void **min*, void **max*, void **out*)
: A function that reads `n_in` items from `in`, and writes to `out` the read value if it is within the limits pointed to by `min` and `max`, or the corresponding limit if outside. The memory segments must be contiguous and behaved, and either `min` or `max` may be `NULL`, but not both.

void **fastputmask**(void **in*, void **mask*, npy_intp *n_in*, void **values*, npy_intp *nv*)
: A function that takes a pointer `in` to an array of `n_in` items, a pointer `mask` to an array of `n_in` boolean values, and a pointer `vals` to an array of `nv` items. Items from `vals` are copied into `in` wherever the value in `mask` is non-zero, tiling `vals` as needed if `nv < n_in`. All arrays must be contiguous and behaved.

void **fasttake**(void **dest*, void **src*, npy_intp **indarray*, npy_intp *nindarray*, npy_intp *n_outer*, npy_intp *m_middle*, npy_intp *nelem*, *NPY_CLIPMODE* *clipmode*)
: A function that takes a pointer `src` to a C contiguous, behaved segment, interpreted as a 3-dimensional array of shape `(n_outer, nindarray, nelem)`, a pointer `indarray` to a contiguous, behaved segment of `m_middle` integer indices, and a pointer `dest` to a C contiguous, behaved segment, interpreted as a 3-dimensional array of shape `(n_outer, m_middle, nelem)`. The indices in `indarray` are used to index `src` along the second dimension, and copy the corresponding chunks of `nelem` items into `dest`. `clipmode` (which can take on the values *NPY_RAISE*, *NPY_WRAP* or *NPY_CLIP*) determines how will indices smaller than 0 or larger than `nindarray` will be handled.

int **argmin**(void* *data*, npy_intp *n*, npy_intp* *min_ind*, void* *arr*)
: A pointer to a function that retrieves the index of the smallest of `n` elements in `arr` beginning at the element pointed to by `data`. This function requires that the memory segment be contiguous and behaved. The return value is always 0. The index of the smallest element is returned in `min_ind`.

The `PyArray_Type` typeobject implements many of the features of Python objects including the tp_as_number, tp_as_sequence, tp_as_mapping, and tp_as_buffer interfaces. The rich comparison (tp_richcompare) is also used along with new-style attribute lookup for methods (tp_methods) and properties (tp_getset). The `PyArray_Type` can also be sub-typed.

Tip: The tp_as_number methods use a generic approach to call whatever function has been registered for handling the operation. The function PyNumeric_SetOps(..) can be used to register functions to handle particular mathematical operations (for all arrays). When the umath module is imported, it sets the numeric operations for all arrays to the corresponding ufuncs. The tp_str and tp_repr methods can also be altered using PyString_SetStringFunction(...).

PyUFunc_Type

PyUFuncObject

The core of the ufunc is the *PyUFuncObject* which contains all the information needed to call the underlying C-code loops that perform the actual work. It has the following structure:

```
typedef struct {
    PyObject_HEAD
    int nin;
    int nout;
    int nargs;
    int identity;
    PyUFuncGenericFunction *functions;
    void **data;
    int ntypes;
    int reserved1;
    const char *name;
    char *types;
    const char *doc;
    void *ptr;
    PyObject *obj;
    PyObject *userloops;
    npy_uint32 *op_flags;
    npy_uint32 *iter_flags;
} PyUFuncObject;
```

int **PyUFuncObject.nin**

The number of input arguments.

int **PyUFuncObject.nout**

The number of output arguments.

int **PyUFuncObject.nargs**

The total number of arguments (*nin* + *nout*). This must be less than NPY_MAXARGS.

int **PyUFuncObject.identity**

Either PyUFunc_One, PyUFunc_Zero, or PyUFunc_None to indicate the identity for this operation. It is only used for a reduce-like call on an empty array.

void PyUFuncObject.functions(char args, npy_intp* dims,**

npy_intp* steps, void* extradata)

An array of function pointers — one for each data type supported by the ufunc. This is the vector loop that is called to implement the underlying function *dims* [0] times. The first argument, *args*, is an array of *nargs* pointers to behaved memory. Pointers to the data for the input arguments are first, followed by the pointers to the data for the output arguments. How many bytes must be skipped to get to the next element in the sequence is specified by the corresponding entry in the *steps* array. The last argument allows the loop to receive extra information. This is commonly used so that a single, generic vector loop can be used for multiple functions. In this case, the actual scalar function to call is passed in as *extradata*. The size of this function pointer array is ntypes.

void ***PyUFuncObject.data**

Extra data to be passed to the 1-d vector loops or NULL if no extra-data is needed. This C-array must be the same size (*i.e.* ntypes) as the functions array. NULL is used if extra_data is not needed. Several C-API calls for UFuncs are just 1-d vector loops that make use of this extra data to receive a pointer to the actual function to call.

int **PyUFuncObject.ntypes**

The number of supported data types for the ufunc. This number specifies how many different 1-d loops

(of the builtin data types) are available.

char ***PyUFuncObject.name**
A string name for the ufunc. This is used dynamically to build the __doc__ attribute of ufuncs.

char ***PyUFuncObject.types**
An array of *nargs* × *ntypes* 8-bit type_numbers which contains the type signature for the function for each of the supported (builtin) data types. For each of the *ntypes* functions, the corresponding set of type numbers in this array shows how the *args* argument should be interpreted in the 1-d vector loop. These type numbers do not have to be the same type and mixed-type ufuncs are supported.

char ***PyUFuncObject.doc**
Documentation for the ufunc. Should not contain the function signature as this is generated dynamically when __doc__ is retrieved.

void ***PyUFuncObject.ptr**
Any dynamically allocated memory. Currently, this is used for dynamic ufuncs created from a python function to store room for the types, data, and name members.

PyObject ***PyUFuncObject.obj**
For ufuncs dynamically created from python functions, this member holds a reference to the underlying Python function.

PyObject ***PyUFuncObject.userloops**
A dictionary of user-defined 1-d vector loops (stored as CObject ptrs) for user-defined types. A loop may be registered by the user for any user-defined type. It is retrieved by type number. User defined type numbers are always larger than *NPY_USERDEF*.

npy_uint32 **PyUFuncObject.op_flags**
Override the default operand flags for each ufunc operand.

npy_uint32 **PyUFuncObject.iter_flags**
Override the default nditer flags for the ufunc.

PyArrayIter_Type

PyArrayIterObject
The C-structure corresponding to an object of `PyArrayIter_Type` is the *PyArrayIterObject*. The *PyArrayIterObject* is used to keep track of a pointer into an N-dimensional array. It contains associated information used to quickly march through the array. The pointer can be adjusted in three basic ways: 1) advance to the "next" position in the array in a C-style contiguous fashion, 2) advance to an arbitrary N-dimensional coordinate in the array, and 3) advance to an arbitrary one-dimensional index into the array. The members of the *PyArrayIterObject* structure are used in these calculations. Iterator objects keep their own dimension and strides information about an array. This can be adjusted as needed for "broadcasting," or to loop over only specific dimensions.

```
typedef struct {
    PyObject_HEAD
    int   nd_m1;
    npy_intp  index;
    npy_intp  size;
    npy_intp  coordinates[NPY_MAXDIMS];
    npy_intp  dims_m1[NPY_MAXDIMS];
    npy_intp  strides[NPY_MAXDIMS];
    npy_intp  backstrides[NPY_MAXDIMS];
    npy_intp  factors[NPY_MAXDIMS];
    PyArrayObject *ao;
    char  *dataptr;
```

```
    Bool  contiguous;
} PyArrayIterObject;
```

int **PyArrayIterObject.nd_m1**
: $N-1$ where N is the number of dimensions in the underlying array.

npy_intp **PyArrayIterObject.index**
: The current 1-d index into the array.

npy_intp **PyArrayIterObject.size**
: The total size of the underlying array.

npy_intp ***PyArrayIterObject.coordinates**
: An N -dimensional index into the array.

npy_intp ***PyArrayIterObject.dims_m1**
: The size of the array minus 1 in each dimension.

npy_intp ***PyArrayIterObject.strides**
: The strides of the array. How many bytes needed to jump to the next element in each dimension.

npy_intp ***PyArrayIterObject.backstrides**
: How many bytes needed to jump from the end of a dimension back to its beginning. Note that *backstrides* [k]= *strides* [k]*d *ims_m1* [k], but it is stored here as an optimization.

npy_intp ***PyArrayIterObject.factors**
: This array is used in computing an N-d index from a 1-d index. It contains needed products of the dimensions.

PyArrayObject ***PyArrayIterObject.ao**
: A pointer to the underlying ndarray this iterator was created to represent.

char ***PyArrayIterObject.dataptr**
: This member points to an element in the ndarray indicated by the index.

Bool **PyArrayIterObject.contiguous**
: This flag is true if the underlying array is *NPY_ARRAY_C_CONTIGUOUS*. It is used to simplify calculations when possible.

How to use an array iterator on a C-level is explained more fully in later sections. Typically, you do not need to concern yourself with the internal structure of the iterator object, and merely interact with it through the use of the macros *PyArray_ITER_NEXT* (it), *PyArray_ITER_GOTO* (it, dest), or *PyArray_ITER_GOTO1D* (it, index). All of these macros require the argument *it* to be a *PyArrayIterObject **.

PyArrayMultiIter_Type

PyArrayMultiIterObject

```
typedef struct {
    PyObject_HEAD
    int numiter;
    npy_intp size;
    npy_intp index;
    int nd;
    npy_intp dimensions[NPY_MAXDIMS];
    PyArrayIterObject *iters[NPY_MAXDIMS];
} PyArrayMultiIterObject;
```

int **PyArrayMultiIterObject.numiter**
The number of arrays that need to be broadcast to the same shape.

npy_intp **PyArrayMultiIterObject.size**
The total broadcasted size.

npy_intp **PyArrayMultiIterObject.index**
The current (1-d) index into the broadcasted result.

int **PyArrayMultiIterObject.nd**
The number of dimensions in the broadcasted result.

npy_intp ***PyArrayMultiIterObject.dimensions**
The shape of the broadcasted result (only `nd` slots are used).

PyArrayIterObject ****PyArrayMultiIterObject.iters**
An array of iterator objects that holds the iterators for the arrays to be broadcast together. On return, the iterators are adjusted for broadcasting.

PyArrayNeighborhoodIter_Type

PyArrayNeighborhoodIterObject
The C-structure corresponding to an object of `PyArrayNeighborhoodIter_Type` is the *PyArrayNeighborhoodIterObject*.

PyArrayFlags_Type

ScalarArrayTypes

There is a Python type for each of the different built-in data types that can be present in the array Most of these are simple wrappers around the corresponding data type in C. The C-names for these types are `Py{TYPE}ArrType_Type` where `{TYPE}` can be

Bool, **Byte**, **Short**, **Int**, **Long**, **LongLong**, **UByte**, **UShort**, **UInt**, **ULong**, **ULongLong**, **Half**, **Float**, **Double**, **LongDouble**, **CFloat**, **CDouble**, **CLongDouble**, **String**, **Unicode**, **Void**, and **Object**.

These type names are part of the C-API and can therefore be created in extension C-code. There is also a `PyIntpArrType_Type` and a `PyUIntpArrType_Type` that are simple substitutes for one of the integer types that can hold a pointer on the platform. The structure of these scalar objects is not exposed to C-code. The function *PyArray_ScalarAsCtype* (..) can be used to extract the C-type value from the array scalar and the function *PyArray_Scalar* (...) can be used to construct an array scalar from a C-value.

5.1.2 Other C-Structures

A few new C-structures were found to be useful in the development of NumPy. These C-structures are used in at least one C-API call and are therefore documented here. The main reason these structures were defined is to make it easy to use the Python ParseTuple C-API to convert from Python objects to a useful C-Object.

PyArray_Dims

PyArray_Dims
This structure is very useful when shape and/or strides information is supposed to be interpreted. The structure is:

```
typedef struct {
    npy_intp *ptr;
    int len;
} PyArray_Dims;
```

The members of this structure are

npy_intp ***PyArray_Dims.ptr***
: A pointer to a list of (`npy_intp`) integers which usually represent array shape or array strides.

int ***PyArray_Dims.len***
: The length of the list of integers. It is assumed safe to access *ptr* [0] to *ptr* [len-1].

PyArray_Chunk

PyArray_Chunk
: This is equivalent to the buffer object structure in Python up to the ptr member. On 32-bit platforms (*i.e.* if *NPY_SIZEOF_INT* == `NPY_SIZEOF_INTP`), the len member also matches an equivalent member of the buffer object. It is useful to represent a generic single-segment chunk of memory.

```
typedef struct {
    PyObject_HEAD
    PyObject *base;
    void *ptr;
    npy_intp len;
    int flags;
} PyArray_Chunk;
```

The members are

PyObject ***PyArray_Chunk.base***
: The Python object this chunk of memory comes from. Needed so that memory can be accounted for properly.

void ***PyArray_Chunk.ptr***
: A pointer to the start of the single-segment chunk of memory.

npy_intp ***PyArray_Chunk.len***
: The length of the segment in bytes.

int ***PyArray_Chunk.flags***
: Any data flags (*e.g.* *NPY_ARRAY_WRITEABLE*) that should be used to interpret the memory.

PyArrayInterface

See also:

The Array Interface

PyArrayInterface
: The *PyArrayInterface* structure is defined so that NumPy and other extension modules can use the rapid array interface protocol. The `__array_struct__` method of an object that supports the rapid array interface protocol should return a `PyCObject` that contains a pointer to a *PyArrayInterface* structure with the relevant details of the array. After the new array is created, the attribute should be `DECREF`'d which will free the *PyArrayInterface* structure. Remember to `INCREF` the object (whose `__array_struct__` attribute was retrieved) and point the base member of the new *PyArrayObject* to this same object. In this way the memory for the array will be managed correctly.

```
typedef struct {
    int two;
    int nd;
    char typekind;
    int itemsize;
    int flags;
    npy_intp *shape;
    npy_intp *strides;
    void *data;
    PyObject *descr;
} PyArrayInterface;
```

int **PyArrayInterface.two**
: the integer 2 as a sanity check.

int **PyArrayInterface.nd**
: the number of dimensions in the array.

char **PyArrayInterface.typekind**
: A character indicating what kind of array is present according to the typestring convention with 't' -> bitfield, 'b' -> Boolean, 'i' -> signed integer, 'u' -> unsigned integer, 'f' -> floating point, 'c' -> complex floating point, 'O' -> object, 'S' -> (byte-)string, 'U' -> unicode, 'V' -> void.

int **PyArrayInterface.itemsize**
: The number of bytes each item in the array requires.

int **PyArrayInterface.flags**
: Any of the bits *NPY_ARRAY_C_CONTIGUOUS* (1), *NPY_ARRAY_F_CONTIGUOUS* (2), *NPY_ARRAY_ALIGNED* (0x100), *NPY_ARRAY_NOTSWAPPED* (0x200), or *NPY_ARRAY_WRITEABLE* (0x400) to indicate something about the data. The *NPY_ARRAY_ALIGNED*, *NPY_ARRAY_C_CONTIGUOUS*, and *NPY_ARRAY_F_CONTIGUOUS* flags can actually be determined from the other parameters. The flag NPY_ARR_HAS_DESCR (0x800) can also be set to indicate to objects consuming the version 3 array interface that the descr member of the structure is present (it will be ignored by objects consuming version 2 of the array interface).

npy_intp ***PyArrayInterface.shape**
: An array containing the size of the array in each dimension.

npy_intp ***PyArrayInterface.strides**
: An array containing the number of bytes to jump to get to the next element in each dimension.

void ***PyArrayInterface.data**
: A pointer *to* the first element of the array.

PyObject ***PyArrayInterface.descr**
: A Python object describing the data-type in more detail (same as the *descr* key in *__array_interface__*). This can be NULL if *typekind* and *itemsize* provide enough information. This field is also ignored unless ARR_HAS_DESCR flag is on in *flags*.

Internally used structures

Internally, the code uses some additional Python objects primarily for memory management. These types are not accessible directly from Python, and are not exposed to the C-API. They are included here only for completeness and assistance in understanding the code.

PyUFuncLoopObject
: A loose wrapper for a C-structure that contains the information needed for looping. This is useful if you are

trying to understand the ufunc looping code. The *PyUFuncLoopObject* is the associated C-structure. It is defined in the `ufuncobject.h` header.

PyUFuncReduceObject
A loose wrapper for the C-structure that contains the information needed for reduce-like methods of ufuncs. This is useful if you are trying to understand the reduce, accumulate, and reduce-at code. The *PyUFuncReduceObject* is the associated C-structure. It is defined in the `ufuncobject.h` header.

PyUFunc_Loop1d
A simple linked-list of C-structures containing the information needed to define a 1-d loop for a ufunc for every defined signature of a user-defined data-type.

5.2 System configuration

When NumPy is built, information about system configuration is recorded, and is made available for extension modules using Numpy's C API. These are mostly defined in `numpyconfig.h` (included in `ndarrayobject.h`). The public symbols are prefixed by `NPY_*`. Numpy also offers some functions for querying information about the platform in use.

For private use, Numpy also constructs a `config.h` in the NumPy include directory, which is not exported by Numpy (that is a python extension which use the numpy C API will not see those symbols), to avoid namespace pollution.

5.2.1 Data type sizes

The `NPY_SIZEOF_{CTYPE}` constants are defined so that sizeof information is available to the pre-processor.

NPY_SIZEOF_SHORT

NPY_SIZEOF_INT

NPY_SIZEOF_LONG

NPY_SIZEOF_LONGLONG
sizeof(longlong) where longlong is defined appropriately on the platform.

NPY_SIZEOF_PY_LONG_LONG

NPY_SIZEOF_FLOAT

NPY_SIZEOF_DOUBLE

NPY_SIZEOF_LONG_DOUBLE

NPY_SIZEOF_PY_INTPTR_T
Size of a pointer on this platform (sizeof(void *)) (A macro defines NPY_SIZEOF_INTP as well.)

5.2.2 Platform information

NPY_CPU_X86

NPY_CPU_AMD64

NPY_CPU_IA64

NPY_CPU_PPC

NPY_CPU_PPC64

NPY_CPU_SPARC

NPY_CPU_SPARC64

NPY_CPU_S390

NPY_CPU_PARISC
New in version 1.3.0.

CPU architecture of the platform; only one of the above is defined.

Defined in `numpy/npy_cpu.h`

NPY_LITTLE_ENDIAN

NPY_BIG_ENDIAN

NPY_BYTE_ORDER
New in version 1.3.0.

Portable alternatives to the `endian.h` macros of GNU Libc. If big endian, *NPY_BYTE_ORDER* == *NPY_BIG_ENDIAN*, and similarly for little endian architectures.

Defined in `numpy/npy_endian.h`.

PyArray_GetEndianness()
New in version 1.3.0.

Returns the endianness of the current platform. One of `NPY_CPU_BIG`, `NPY_CPU_LITTLE`, or `NPY_CPU_UNKNOWN_ENDIAN`.

5.3 Data Type API

The standard array can have 24 different data types (and has some support for adding your own types). These data types all have an enumerated type, an enumerated type-character, and a corresponding array scalar Python type object (placed in a hierarchy). There are also standard C typedefs to make it easier to manipulate elements of the given data type. For the numeric types, there are also bit-width equivalent C typedefs and named typenumbers that make it easier to select the precision desired.

Warning: The names for the types in c code follows c naming conventions more closely. The Python names for these types follow Python conventions. Thus, *NPY_FLOAT* picks up a 32-bit float in C, but `numpy.float_` in Python corresponds to a 64-bit double. The bit-width names can be used in both Python and C for clarity.

5.3.1 Enumerated Types

There is a list of enumerated types defined providing the basic 24 data types plus some useful generic names. Whenever the code requires a type number, one of these enumerated types is requested. The types are all called `NPY_{NAME}`:

NPY_BOOL
The enumeration value for the boolean type, stored as one byte. It may only be set to the values 0 and 1.

NPY_BYTE

NPY_INT8
The enumeration value for an 8-bit/1-byte signed integer.

NPY_SHORT

NPY_INT16
The enumeration value for a 16-bit/2-byte signed integer.

NPY_INT

NPY_INT32
The enumeration value for a 32-bit/4-byte signed integer.

NPY_LONG
Equivalent to either NPY_INT or NPY_LONGLONG, depending on the platform.

NPY_LONGLONG

NPY_INT64
The enumeration value for a 64-bit/8-byte signed integer.

NPY_UBYTE

NPY_UINT8
The enumeration value for an 8-bit/1-byte unsigned integer.

NPY_USHORT

NPY_UINT16
The enumeration value for a 16-bit/2-byte unsigned integer.

NPY_UINT

NPY_UINT32
The enumeration value for a 32-bit/4-byte unsigned integer.

NPY_ULONG
Equivalent to either NPY_UINT or NPY_ULONGLONG, depending on the platform.

NPY_ULONGLONG

NPY_UINT64
The enumeration value for a 64-bit/8-byte unsigned integer.

NPY_HALF

NPY_FLOAT16
The enumeration value for a 16-bit/2-byte IEEE 754-2008 compatible floating point type.

NPY_FLOAT

NPY_FLOAT32
The enumeration value for a 32-bit/4-byte IEEE 754 compatible floating point type.

NPY_DOUBLE

NPY_FLOAT64
The enumeration value for a 64-bit/8-byte IEEE 754 compatible floating point type.

NPY_LONGDOUBLE
The enumeration value for a platform-specific floating point type which is at least as large as NPY_DOUBLE, but larger on many platforms.

NPY_CFLOAT

NPY_COMPLEX64
The enumeration value for a 64-bit/8-byte complex type made up of two NPY_FLOAT values.

NPY_CDOUBLE

NPY_COMPLEX128
The enumeration value for a 128-bit/16-byte complex type made up of two NPY_DOUBLE values.

NPY_CLONGDOUBLE
The enumeration value for a platform-specific complex floating point type which is made up of two NPY_LONGDOUBLE values.

NPY_DATETIME
The enumeration value for a data type which holds dates or datetimes with a precision based on selectable date or time units.

NPY_TIMEDELTA
The enumeration value for a data type which holds lengths of times in integers of selectable date or time units.

NPY_STRING
The enumeration value for ASCII strings of a selectable size. The strings have a fixed maximum size within a given array.

NPY_UNICODE
The enumeration value for UCS4 strings of a selectable size. The strings have a fixed maximum size within a given array.

NPY_OBJECT
The enumeration value for references to arbitrary Python objects.

NPY_VOID
Primarily used to hold struct dtypes, but can contain arbitrary binary data.

Some useful aliases of the above types are

NPY_INTP
The enumeration value for a signed integer type which is the same size as a (void *) pointer. This is the type used by all arrays of indices.

NPY_UINTP
The enumeration value for an unsigned integer type which is the same size as a (void *) pointer.

NPY_MASK
The enumeration value of the type used for masks, such as with the *NPY_ITER_ARRAYMASK* iterator flag. This is equivalent to *NPY_UINT8*.

NPY_DEFAULT_TYPE
The default type to use when no dtype is explicitly specified, for example when calling np.zero(shape). This is equivalent to *NPY_DOUBLE*.

Other useful related constants are

NPY_NTYPES
The total number of built-in NumPy types. The enumeration covers the range from 0 to NPY_NTYPES-1.

NPY_NOTYPE
A signal value guaranteed not to be a valid type enumeration number.

NPY_USERDEF
The start of type numbers used for Custom Data types.

The various character codes indicating certain types are also part of an enumerated list. References to type characters (should they be needed at all) should always use these enumerations. The form of them is `NPY_{NAME}LTR` where `{NAME}` can be

BOOL, **BYTE**, **UBYTE**, **SHORT**, **USHORT**, **INT**, **UINT**, **LONG**, **ULONG**, **LONGLONG**, **ULONGLONG**, **HALF**, **FLOAT**, **DOUBLE**, **LONGDOUBLE**, **CFLOAT**, **CDOUBLE**, **CLONGDOUBLE**, **DATETIME**, **TIMEDELTA**, **OBJECT**, **STRING**, **VOID**

INTP, **UINTP**

GENBOOL, **SIGNED**, **UNSIGNED**, **FLOATING**, **COMPLEX**

The latter group of `{NAME}s` corresponds to letters used in the array interface typestring specification.

5.3.2 Defines

Max and min values for integers

NPY_MAX_INT{bits}

NPY_MAX_UINT{bits}

NPY_MIN_INT{bits}
These are defined for `{bits}` = 8, 16, 32, 64, 128, and 256 and provide the maximum (minimum) value of the corresponding (unsigned) integer type. Note: the actual integer type may not be available on all platforms (i.e. 128-bit and 256-bit integers are rare).

NPY_MIN_{type}
This is defined for `{type}` = **BYTE**, **SHORT**, **INT**, **LONG**, **LONGLONG**, **INTP**

NPY_MAX_{type}
This is defined for all defined for `{type}` = **BYTE**, **UBYTE**, **SHORT**, **USHORT**, **INT**, **UINT**, **LONG**, **ULONG**, **LONGLONG**, **ULONGLONG**, **INTP**, **UINTP**

Number of bits in data types

All NPY_SIZEOF_{CTYPE} constants have corresponding NPY_BITSOF_{CTYPE} constants defined. The NPY_BITSOF_{CTYPE} constants provide the number of bits in the data type. Specifically, the available {CTYPE}s are

BOOL, **CHAR**, **SHORT**, **INT**, **LONG**, **LONGLONG**, **FLOAT**, **DOUBLE**, **LONGDOUBLE**

Bit-width references to enumerated typenums

All of the numeric data types (integer, floating point, and complex) have constants that are defined to be a specific enumerated type number. Exactly which enumerated type a bit-width type refers to is platform dependent. In particular, the constants available are PyArray_{NAME}{BITS} where {NAME} is **INT**, **UINT**, **FLOAT**, **COMPLEX** and {BITS} can be 8, 16, 32, 64, 80, 96, 128, 160, 192, 256, and 512. Obviously not all bit-widths are available on all platforms for all the kinds of numeric types. Commonly 8-, 16-, 32-, 64-bit integers; 32-, 64-bit floats; and 64-, 128-bit complex types are available.

Integer that can hold a pointer

The constants **NPY_INTP** and **NPY_UINTP** refer to an enumerated integer type that is large enough to hold a pointer on the platform. Index arrays should always be converted to **NPY_INTP** , because the dimension of the array is of type npy_intp.

5.3.3 C-type names

There are standard variable types for each of the numeric data types and the bool data type. Some of these are already available in the C-specification. You can create variables in extension code with these types.

Boolean

npy_bool
: unsigned char; The constants *NPY_FALSE* and *NPY_TRUE* are also defined.

(Un)Signed Integer

Unsigned versions of the integers can be defined by pre-pending a 'u' to the front of the integer name.

npy_(u)byte
: (unsigned) char

npy_(u)short
: (unsigned) short

npy_(u)int
: (unsigned) int

npy_(u)long
: (unsigned) long int

npy_(u)longlong
: (unsigned long long int)

npy_(u)intp
: (unsigned) Py_intptr_t (an integer that is the size of a pointer on the platform).

(Complex) Floating point

npy_(c)float
float

npy_(c)double
double

npy_(c)longdouble
long double

complex types are structures with **.real** and **.imag** members (in that order).

Bit-width names

There are also typedefs for signed integers, unsigned integers, floating point, and complex floating point types of specific bit- widths. The available type names are

`npy_int{bits}`, `npy_uint{bits}`, `npy_float{bits}`, and `npy_complex{bits}`

where `{bits}` is the number of bits in the type and can be **8**, **16**, **32**, **64**, 128, and 256 for integer types; 16, **32** , **64**, 80, 96, 128, and 256 for floating-point types; and 32, **64**, **128**, 160, 192, and 512 for complex-valued types. Which bit-widths are available is platform dependent. The bolded bit-widths are usually available on all platforms.

5.3.4 Printf Formatting

For help in printing, the following strings are defined as the correct format specifier in printf and related commands.

`NPY_LONGLONG_FMT`, `NPY_ULONGLONG_FMT`, `NPY_INTP_FMT`, `NPY_UINTP_FMT`, `NPY_LONGDOUBLE_FMT`

5.4 Array API

The test of a first-rate intelligence is the ability to hold two
opposed ideas in the mind at the same time, and still retain the
ability to function.
— *F. Scott Fitzgerald*

For a successful technology, reality must take precedence over public
relations, for Nature cannot be fooled.
— *Richard P. Feynman*

5.4.1 Array structure and data access

These macros all access the *PyArrayObject* structure members. The input argument, arr, can be any `PyObject *` that is directly interpretable as a *PyArrayObject* * (any instance of the `PyArray_Type` and its sub-types).

int **PyArray_NDIM**(*PyArrayObject* *arr)
The number of dimensions in the array.

npy_intp ***PyArray_DIMS**(*PyArrayObject *arr*)
Returns a pointer to the dimensions/shape of the array. The number of elements matches the number of dimensions of the array.

npy_intp ***PyArray_SHAPE**(*PyArrayObject *arr*)
New in version 1.7.

A synonym for PyArray_DIMS, named to be consistent with the 'shape' usage within Python.

void ***PyArray_DATA**(*PyArrayObject *arr*)

char ***PyArray_BYTES**(*PyArrayObject *arr*)
These two macros are similar and obtain the pointer to the data-buffer for the array. The first macro can (and should be) assigned to a particular pointer where the second is for generic processing. If you have not guaranteed a contiguous and/or aligned array then be sure you understand how to access the data in the array to avoid memory and/or alignment problems.

npy_intp ***PyArray_STRIDES**(*PyArrayObject* *arr*)
Returns a pointer to the strides of the array. The number of elements matches the number of dimensions of the array.

npy_intp **PyArray_DIM**(*PyArrayObject* *arr*, int *n*)
Return the shape in the *n* th dimension.

npy_intp **PyArray_STRIDE**(*PyArrayObject* *arr*, int *n*)
Return the stride in the *n* th dimension.

PyObject ***PyArray_BASE**(*PyArrayObject* *arr*)
This returns the base object of the array. In most cases, this means the object which owns the memory the array is pointing at.

If you are constructing an array using the C API, and specifying your own memory, you should use the function `PyArray_SetBaseObject` to set the base to an object which owns the memory.

If the `NPY_ARRAY_UPDATEIFCOPY` flag is set, it has a different meaning, namely base is the array into which the current array will be copied upon destruction. This overloading of the base property for two functions is likely to change in a future version of NumPy.

PyArray_Descr ***PyArray_DESCR**(*PyArrayObject* *arr*)
Returns a borrowed reference to the dtype property of the array.

PyArray_Descr ***PyArray_DTYPE**(*PyArrayObject* *arr*)
New in version 1.7.

A synonym for PyArray_DESCR, named to be consistent with the 'dtype' usage within Python.

void **PyArray_ENABLEFLAGS**(*PyArrayObject* *arr*, int *flags*)
New in version 1.7.

Enables the specified array flags. This function does no validation, and assumes that you know what you're doing.

void **PyArray_CLEARFLAGS**(*PyArrayObject* *arr*, int *flags*)
New in version 1.7.

Clears the specified array flags. This function does no validation, and assumes that you know what you're doing.

int **PyArray_FLAGS**(*PyArrayObject* *arr*)

npy_intp **PyArray_ITEMSIZE**(*PyArrayObject* *arr*)
Return the itemsize for the elements of this array.

Note that, in the old API that was deprecated in version 1.7, this function had the return type `int`.

int **PyArray_TYPE**(*PyArrayObject* arr*)
Return the (builtin) typenumber for the elements of this array.

PyObject ***PyArray_GETITEM**(*PyArrayObject* arr*, void* *itemptr*)
Get a Python object from the ndarray, *arr*, at the location pointed to by itemptr. Return `NULL` on failure.

int **PyArray_SETITEM**(*PyArrayObject* arr*, void* *itemptr*, PyObject* *obj*)
Convert obj and place it in the ndarray, *arr*, at the place pointed to by itemptr. Return -1 if an error occurs or 0 on success.

npy_intp **PyArray_SIZE**(*PyArrayObject* arr*)
Returns the total size (in number of elements) of the array.

npy_intp **PyArray_Size**(*PyArrayObject* obj*)
Returns 0 if *obj* is not a sub-class of bigndarray. Otherwise, returns the total number of elements in the array. Safer version of *PyArray_SIZE* (*obj*).

npy_intp **PyArray_NBYTES**(*PyArrayObject* arr*)
Returns the total number of bytes consumed by the array.

Data access

These functions and macros provide easy access to elements of the ndarray from C. These work for all arrays. You may need to take care when accessing the data in the array, however, if it is not in machine byte-order, misaligned, or not writeable. In other words, be sure to respect the state of the flags unless you know what you are doing, or have previously guaranteed an array that is writeable, aligned, and in machine byte-order using *PyArray_FromAny*. If you wish to handle all types of arrays, the copyswap function for each type is useful for handling misbehaved arrays. Some platforms (e.g. Solaris) do not like misaligned data and will crash if you de-reference a misaligned pointer. Other platforms (e.g. x86 Linux) will just work more slowly with misaligned data.

void* **PyArray_GetPtr**(*PyArrayObject* aobj*, npy_intp* *ind*)
Return a pointer to the data of the ndarray, *aobj*, at the N-dimensional index given by the c-array, *ind*, (which must be at least *aobj* ->nd in size). You may want to typecast the returned pointer to the data type of the ndarray.

void* **PyArray_GETPTR1**(*PyArrayObject* obj*, npy_intp *i*)

void* **PyArray_GETPTR2**(*PyArrayObject* obj*, npy_intp *i*, npy_intp *j*)

void* **PyArray_GETPTR3**(*PyArrayObject* obj*, npy_intp *i*, npy_intp *j*, npy_intp *k*)

void* **PyArray_GETPTR4**(*PyArrayObject* obj*, npy_intp *i*, npy_intp *j*, npy_intp *k*, npy_intp *l*)
Quick, inline access to the element at the given coordinates in the ndarray, *obj*, which must have respectively 1, 2, 3, or 4 dimensions (this is not checked). The corresponding *i*, *j*, *k*, and *l* coordinates can be any integer but will be interpreted as `npy_intp`. You may want to typecast the returned pointer to the data type of the ndarray.

5.4.2 Creating arrays

From scratch

PyObject* **PyArray_NewFromDescr**(PyTypeObject* *subtype*, *PyArray_Descr** *descr*, int *nd*, npy_intp* *dims*, npy_intp* *strides*, void* *data*, int *flags*, PyObject* *obj*)
This function steals a reference to *descr*.

This is the main array creation function. Most new arrays are created with this flexible function.

The returned object is an object of Python-type *subtype*, which must be a subtype of `PyArray_Type`. The array has *nd* dimensions, described by *dims*. The data-type descriptor of the new array is *descr*.

If *subtype* is of an array subclass instead of the base `&PyArray_Type`, then *obj* is the object to pass to the `__array_finalize__` method of the subclass.

If *data* is `NULL`, then new memory will be allocated and *flags* can be non-zero to indicate a Fortran-style contiguous array. If *data* is not `NULL`, then it is assumed to point to the memory to be used for the array and the *flags* argument is used as the new flags for the array (except the state of `NPY_OWNDATA` and *NPY_ARRAY_UPDATEIFCOPY* flags of the new array will be reset).

In addition, if *data* is non-NULL, then *strides* can also be provided. If *strides* is `NULL`, then the array strides are computed as C-style contiguous (default) or Fortran-style contiguous (*flags* is nonzero for *data* = `NULL` or *flags* & *NPY_ARRAY_F_CONTIGUOUS* is nonzero non-NULL *data*). Any provided *dims* and *strides* are copied into newly allocated dimension and strides arrays for the new array object.

PyObject* **PyArray_NewLikeArray**(*PyArrayObject** *prototype*, *NPY_ORDER* *order*, *PyArray_Descr** *descr*, int *subok*)

New in version 1.6.

This function steals a reference to *descr* if it is not NULL.

This array creation routine allows for the convenient creation of a new array matching an existing array's shapes and memory layout, possibly changing the layout and/or data type.

When *order* is *NPY_ANYORDER*, the result order is *NPY_FORTRANORDER* if *prototype* is a fortran array, *NPY_CORDER* otherwise. When *order* is *NPY_KEEPORDER*, the result order matches that of *prototype*, even when the axes of *prototype* aren't in C or Fortran order.

If *descr* is NULL, the data type of *prototype* is used.

If *subok* is 1, the newly created array will use the sub-type of *prototype* to create the new array, otherwise it will create a base-class array.

PyObject* **PyArray_New**(PyTypeObject* *subtype*, int *nd*, npy_intp* *dims*, int *type_num*, npy_intp* *strides*, void* *data*, int *itemsize*, int *flags*, PyObject* *obj*)

This is similar to *PyArray_DescrNew* (...) except you specify the data-type descriptor with *type_num* and *itemsize*, where *type_num* corresponds to a builtin (or user-defined) type. If the type always has the same number of bytes, then itemsize is ignored. Otherwise, itemsize specifies the particular size of this array.

Warning: If data is passed to *PyArray_NewFromDescr* or *PyArray_New*, this memory must not be deallocated until the new array is deleted. If this data came from another Python object, this can be accomplished using `Py_INCREF` on that object and setting the base member of the new array to point to that object. If strides are passed in they must be consistent with the dimensions, the itemsize, and the data of the array.

PyObject* **PyArray_SimpleNew**(int *nd*, npy_intp* *dims*, int *typenum*)

Create a new unitialized array of type, *typenum*, whose size in each of *nd* dimensions is given by the integer array, *dims*. This function cannot be used to create a flexible-type array (no itemsize given).

PyObject* **PyArray_SimpleNewFromData**(int *nd*, npy_intp* *dims*, int *typenum*, void* *data*)

Create an array wrapper around *data* pointed to by the given pointer. The array flags will have a default that the data area is well-behaved and C-style contiguous. The shape of the array is given by the *dims* c-array of length *nd*. The data-type of the array is indicated by *typenum*.

PyObject* **PyArray_SimpleNewFromDescr**(int *nd*, npy_intp* *dims*, *PyArray_Descr** *descr*)

This function steals a reference to *descr* if it is not NULL.

Create a new array with the provided data-type descriptor, *descr* , of the shape determined by *nd* and *dims*.

PyArray_FILLWBYTE (PyObject* *obj*, int *val*)
Fill the array pointed to by *obj* —which must be a (subclass of) bigndarray—with the contents of *val* (evaluated as a byte). This macro calls memset, so obj must be contiguous.

PyObject* **PyArray_Zeros** (int *nd*, npy_intp* *dims*, *PyArray_Descr** *dtype*, int *fortran*)
Construct a new *nd* -dimensional array with shape given by *dims* and data type given by *dtype*. If *fortran* is non-zero, then a Fortran-order array is created, otherwise a C-order array is created. Fill the memory with zeros (or the 0 object if *dtype* corresponds to *NPY_OBJECT*).

PyObject* **PyArray_ZEROS** (int *nd*, npy_intp* *dims*, int *type_num*, int *fortran*)
Macro form of *PyArray_Zeros* which takes a type-number instead of a data-type object.

PyObject* **PyArray_Empty** (int *nd*, npy_intp* *dims*, *PyArray_Descr** *dtype*, int *fortran*)
Construct a new *nd* -dimensional array with shape given by *dims* and data type given by *dtype*. If *fortran* is non-zero, then a Fortran-order array is created, otherwise a C-order array is created. The array is uninitialized unless the data type corresponds to *NPY_OBJECT* in which case the array is filled with `Py_None`.

PyObject* **PyArray_EMPTY** (int *nd*, npy_intp* *dims*, int *typenum*, int *fortran*)
Macro form of *PyArray_Empty* which takes a type-number, *typenum*, instead of a data-type object.

PyObject* **PyArray_Arange** (double *start*, double *stop*, double *step*, int *typenum*)
Construct a new 1-dimensional array of data-type, *typenum*, that ranges from *start* to *stop* (exclusive) in increments of *step* . Equivalent to **arange** (*start*, *stop*, *step*, dtype).

PyObject* **PyArray_ArangeObj** (PyObject* *start*, PyObject* *stop*, PyObject* *step*, *PyArray_Descr** *descr*)
Construct a new 1-dimensional array of data-type determined by `descr`, that ranges from `start` to `stop` (exclusive) in increments of `step`. Equivalent to arange(`start`, `stop`, `step`, `typenum`).

int **PyArray_SetBaseObject** (*PyArrayObject** *arr*, PyObject* *obj*)
New in version 1.7.

This function **steals a reference** to `obj` and sets it as the base property of `arr`.

If you construct an array by passing in your own memory buffer as a parameter, you need to set the array's *base* property to ensure the lifetime of the memory buffer is appropriate.

The return value is 0 on success, -1 on failure.

If the object provided is an array, this function traverses the chain of *base* pointers so that each array points to the owner of the memory directly. Once the base is set, it may not be changed to another value.

From other objects

PyObject* **PyArray_FromAny** (PyObject* *op*, *PyArray_Descr** *dtype*, int *min_depth*, int *max_depth*, int *requirements*, PyObject* *context*)
This is the main function used to obtain an array from any nested sequence, or object that exposes the array interface, *op*. The parameters allow specification of the required *dtype*, the minimum (*min_depth*) and maximum (*max_depth*) number of dimensions acceptable, and other *requirements* for the array. The *dtype* argument needs to be a *PyArray_Descr* structure indicating the desired data-type (including required byteorder). The *dtype* argument may be NULL, indicating that any data-type (and byteorder) is acceptable. Unless `FORCECAST` is present in `flags`, this call will generate an error if the data type cannot be safely obtained from the object. If you want to use `NULL` for the *dtype* and ensure the array is notswapped then use *PyArray_CheckFromAny*. A value of 0 for either of the depth parameters causes the parameter to be ignored. Any of the following array flags can be added (*e.g.* using |) to get the *requirements* argument. If your code can handle general (*e.g.* strided, byte-swapped, or unaligned arrays) then *requirements* may be 0. Also, if *op* is not already an array (or does not expose the array interface), then a new array will be created (and filled from *op* using the sequence protocol). The new array will have `NPY_DEFAULT` as its flags member. The *context* argument is passed to the

__array__ method of *op* and is only used if the array is constructed that way. Almost always this parameter is NULL.

In versions 1.6 and earlier of NumPy, the following flags did not have the _ARRAY_ macro namespace in them. That form of the constant names is deprecated in 1.7.

NPY_ARRAY_C_CONTIGUOUS
Make sure the returned array is C-style contiguous

NPY_ARRAY_F_CONTIGUOUS
Make sure the returned array is Fortran-style contiguous.

NPY_ARRAY_ALIGNED
Make sure the returned array is aligned on proper boundaries for its data type. An aligned array has the data pointer and every strides factor as a multiple of the alignment factor for the data-type- descriptor.

NPY_ARRAY_WRITEABLE
Make sure the returned array can be written to.

NPY_ARRAY_ENSURECOPY
Make sure a copy is made of *op*. If this flag is not present, data is not copied if it can be avoided.

NPY_ARRAY_ENSUREARRAY
Make sure the result is a base-class ndarray or bigndarray. By default, if *op* is an instance of a subclass of the bigndarray, an instance of that same subclass is returned. If this flag is set, an ndarray object will be returned instead.

NPY_ARRAY_FORCECAST
Force a cast to the output type even if it cannot be done safely. Without this flag, a data cast will occur only if it can be done safely, otherwise an error is reaised.

NPY_ARRAY_UPDATEIFCOPY
If *op* is already an array, but does not satisfy the requirements, then a copy is made (which will satisfy the requirements). If this flag is present and a copy (of an object that is already an array) must be made, then the corresponding *NPY_ARRAY_UPDATEIFCOPY* flag is set in the returned copy and *op* is made to be read-only. When the returned copy is deleted (presumably after your calculations are complete), its contents will be copied back into *op* and the *op* array will be made writeable again. If *op* is not writeable to begin with, then an error is raised. If *op* is not already an array, then this flag has no effect.

NPY_ARRAY_BEHAVED
NPY_ARRAY_ALIGNED | *NPY_ARRAY_WRITEABLE*

NPY_ARRAY_CARRAY
NPY_ARRAY_C_CONTIGUOUS | *NPY_ARRAY_BEHAVED*

NPY_ARRAY_CARRAY_RO
NPY_ARRAY_C_CONTIGUOUS | *NPY_ARRAY_ALIGNED*

NPY_ARRAY_FARRAY
NPY_ARRAY_F_CONTIGUOUS | *NPY_ARRAY_BEHAVED*

NPY_ARRAY_FARRAY_RO
NPY_ARRAY_F_CONTIGUOUS | *NPY_ARRAY_ALIGNED*

NPY_ARRAY_DEFAULT
NPY_ARRAY_CARRAY

NPY_ARRAY_IN_ARRAY
NPY_ARRAY_C_CONTIGUOUS | *NPY_ARRAY_ALIGNED*

NPY_ARRAY_IN_FARRAY
NPY_ARRAY_F_CONTIGUOUS | *NPY_ARRAY_ALIGNED*

NPY_OUT_ARRAY
NPY_ARRAY_C_CONTIGUOUS | *NPY_ARRAY_WRITEABLE* | *NPY_ARRAY_ALIGNED*

NPY_ARRAY_OUT_FARRAY
NPY_ARRAY_F_CONTIGUOUS | *NPY_ARRAY_WRITEABLE* | *NPY_ARRAY_ALIGNED*

NPY_ARRAY_INOUT_ARRAY
NPY_ARRAY_C_CONTIGUOUS | *NPY_ARRAY_WRITEABLE* | *NPY_ARRAY_ALIGNED* | *NPY_ARRAY_UPDATEIFCOPY*

NPY_ARRAY_INOUT_FARRAY
NPY_ARRAY_F_CONTIGUOUS | *NPY_ARRAY_WRITEABLE* | *NPY_ARRAY_ALIGNED* | *NPY_ARRAY_UPDATEIFCOPY*

int **PyArray_GetArrayParamsFromObject**(PyObject* *op*, *PyArray_Descr** *requested_dtype*, *npy_bool* *writeable*, *PyArray_Descr*** *out_dtype*, int* *out_ndim*, npy_intp* *out_dims*, *PyArrayObject*** *out_arr*, PyObject* *context*)

New in version 1.6.

Retrieves the array parameters for viewing/converting an arbitrary PyObject* to a NumPy array. This allows the "innate type and shape" of Python list-of-lists to be discovered without actually converting to an array. PyArray_FromAny calls this function to analyze its input.

In some cases, such as structured arrays and the __array__ interface, a data type needs to be used to make sense of the object. When this is needed, provide a Descr for 'requested_dtype', otherwise provide NULL. This reference is not stolen. Also, if the requested dtype doesn't modify the interpretation of the input, out_dtype will still get the "innate" dtype of the object, not the dtype passed in 'requested_dtype'.

If writing to the value in 'op' is desired, set the boolean 'writeable' to 1. This raises an error when 'op' is a scalar, list of lists, or other non-writeable 'op'. This differs from passing NPY_ARRAY_WRITEABLE to PyArray_FromAny, where the writeable array may be a copy of the input.

When success (0 return value) is returned, either out_arr is filled with a non-NULL PyArrayObject and the rest of the parameters are untouched, or out_arr is filled with NULL, and the rest of the parameters are filled.

Typical usage:

```
PyArrayObject *arr = NULL;
PyArray_Descr *dtype = NULL;
int ndim = 0;
npy_intp dims[NPY_MAXDIMS];

if (PyArray_GetArrayParamsFromObject(op, NULL, 1, &dtype,
                                     &ndim, &dims, &arr, NULL) < 0) {
    return NULL;
}
if (arr == NULL) {
    ... validate/change dtype, validate flags, ndim, etc ...
    // Could make custom strides here too
    arr = PyArray_NewFromDescr(&PyArray_Type, dtype, ndim,
                               dims, NULL,
                               fortran ? NPY_ARRAY_F_CONTIGUOUS : 0,
                               NULL);
    if (arr == NULL) {
        return NULL;
    }
    if (PyArray_CopyObject(arr, op) < 0) {
        Py_DECREF(arr);
        return NULL;
    }
```

```
    }
    else {
        ... in this case the other parameters weren't filled, just
            validate and possibly copy arr itself ...
    }
    ... use arr ...
```

PyObject* **PyArray_CheckFromAny**(PyObject* *op*, *PyArray_Descr** *dtype*, int *min_depth*, int *max_depth*, int *requirements*, PyObject* *context*)

Nearly identical to *PyArray_FromAny* (...) except *requirements* can contain *NPY_ARRAY_NOTSWAPPED* (over-riding the specification in *dtype*) and *NPY_ARRAY_ELEMENTSTRIDES* which indicates that the array should be aligned in the sense that the strides are multiples of the element size.

In versions 1.6 and earlier of NumPy, the following flags did not have the _ARRAY_ macro namespace in them. That form of the constant names is deprecated in 1.7.

NPY_ARRAY_NOTSWAPPED

Make sure the returned array has a data-type descriptor that is in machine byte-order, over-riding any specification in the *dtype* argument. Normally, the byte-order requirement is determined by the *dtype* argument. If this flag is set and the dtype argument does not indicate a machine byte-order descriptor (or is NULL and the object is already an array with a data-type descriptor that is not in machine byte- order), then a new data-type descriptor is created and used with its byte-order field set to native.

NPY_ARRAY_BEHAVED_NS

NPY_ARRAY_ALIGNED | *NPY_ARRAY_WRITEABLE* | *NPY_ARRAY_NOTSWAPPED*

NPY_ARRAY_ELEMENTSTRIDES

Make sure the returned array has strides that are multiples of the element size.

PyObject* **PyArray_FromArray**(*PyArrayObject** *op*, *PyArray_Descr** *newtype*, int *requirements*)

Special case of *PyArray_FromAny* for when *op* is already an array but it needs to be of a specific *newtype* (including byte-order) or has certain *requirements*.

PyObject* **PyArray_FromStructInterface**(PyObject* *op*)

Returns an ndarray object from a Python object that exposes the `__array_struct__` attribute and follows the array interface protocol. If the object does not contain this attribute then a borrowed reference to `Py_NotImplemented` is returned.

PyObject* **PyArray_FromInterface**(PyObject* *op*)

Returns an ndarray object from a Python object that exposes the *__array_interface__* attribute following the array interface protocol. If the object does not contain this attribute then a borrowed reference to `Py_NotImplemented` is returned.

PyObject* **PyArray_FromArrayAttr**(PyObject* *op*, *PyArray_Descr** *dtype*, PyObject* *context*)

Return an ndarray object from a Python object that exposes the `__array__` method. The `__array__` method can take 0, 1, or 2 arguments ([dtype, context]) where *context* is used to pass information about where the `__array__` method is being called from (currently only used in ufuncs).

PyObject* **PyArray_ContiguousFromAny**(PyObject* *op*, int *typenum*, int *min_depth*, int *max_depth*)

This function returns a (C-style) contiguous and behaved function array from any nested sequence or array interface exporting object, *op*, of (non-flexible) type given by the enumerated *typenum*, of minimum depth *min_depth*, and of maximum depth *max_depth*. Equivalent to a call to *PyArray_FromAny* with requirements set to `NPY_DEFAULT` and the type_num member of the type argument set to *typenum*.

PyObject *__PyArray_FromObject__(PyObject *__op__, int *typenum*, int *min_depth*, int *max_depth*)

Return an aligned and in native-byteorder array from any nested sequence or array-interface exporting object, op, of a type given by the enumerated typenum. The minimum number of dimensions the array can have is given by min_depth while the maximum is max_depth. This is equivalent to a call to *PyArray_FromAny* with requirements set to BEHAVED.

PyObject* **PyArray_EnsureArray**(PyObject* *op*)

This function **steals a reference** to `op` and makes sure that `op` is a base-class ndarray. It special cases array scalars, but otherwise calls *PyArray_FromAny* (`op`, NULL, 0, 0, `NPY_ARRAY_ENSUREARRAY`).

PyObject* **PyArray_FromString**(char* *string*, npy_intp *slen*, *PyArray_Descr** *dtype*, npy_intp *num*, char* *sep*)

Construct a one-dimensional ndarray of a single type from a binary or (ASCII) text `string` of length `slen`. The data-type of the array to-be-created is given by `dtype`. If num is -1, then **copy** the entire string and return an appropriately sized array, otherwise, `num` is the number of items to **copy** from the string. If `sep` is NULL (or ""), then interpret the string as bytes of binary data, otherwise convert the sub-strings separated by `sep` to items of data-type `dtype`. Some data-types may not be readable in text mode and an error will be raised if that occurs. All errors return NULL.

PyObject* **PyArray_FromFile**(FILE* *fp*, *PyArray_Descr** *dtype*, npy_intp *num*, char* *sep*)

Construct a one-dimensional ndarray of a single type from a binary or text file. The open file pointer is `fp`, the data-type of the array to be created is given by `dtype`. This must match the data in the file. If `num` is -1, then read until the end of the file and return an appropriately sized array, otherwise, `num` is the number of items to read. If `sep` is NULL (or ""), then read from the file in binary mode, otherwise read from the file in text mode with `sep` providing the item separator. Some array types cannot be read in text mode in which case an error is raised.

PyObject* **PyArray_FromBuffer**(PyObject* *buf*, *PyArray_Descr** *dtype*, npy_intp *count*, npy_intp *offset*)

Construct a one-dimensional ndarray of a single type from an object, `buf`, that exports the (single-segment) buffer protocol (or has an attribute __buffer__ that returns an object that exports the buffer protocol). A writeable buffer will be tried first followed by a read- only buffer. The *NPY_ARRAY_WRITEABLE* flag of the returned array will reflect which one was successful. The data is assumed to start at `offset` bytes from the start of the memory location for the object. The type of the data in the buffer will be interpreted depending on the data- type descriptor, `dtype.` If `count` is negative then it will be determined from the size of the buffer and the requested itemsize, otherwise, `count` represents how many elements should be converted from the buffer.

int **PyArray_CopyInto**(*PyArrayObject** *dest*, *PyArrayObject** *src*)

Copy from the source array, `src`, into the destination array, `dest`, performing a data-type conversion if necessary. If an error occurs return -1 (otherwise 0). The shape of `src` must be broadcastable to the shape of `dest`. The data areas of dest and src must not overlap.

int **PyArray_MoveInto**(*PyArrayObject** *dest*, *PyArrayObject** *src*)

Move data from the source array, `src`, into the destination array, `dest`, performing a data-type conversion if necessary. If an error occurs return -1 (otherwise 0). The shape of `src` must be broadcastable to the shape of `dest`. The data areas of dest and src may overlap.

*PyArrayObject** **PyArray_GETCONTIGUOUS**(PyObject* *op*)

If `op` is already (C-style) contiguous and well-behaved then just return a reference, otherwise return a (contiguous and well-behaved) copy of the array. The parameter op must be a (sub-class of an) ndarray and no checking for that is done.

PyObject* **PyArray_FROM_O**(PyObject* *obj*)

Convert `obj` to an ndarray. The argument can be any nested sequence or object that exports the array interface. This is a macro form of *PyArray_FromAny* using `NULL`, 0, 0, 0 for the other arguments. Your code must be able to handle any data-type descriptor and any combination of data-flags to use this macro.

PyObject* **PyArray_FROM_OF**(PyObject* *obj*, int *requirements*)

Similar to *PyArray_FROM_O* except it can take an argument of *requirements* indicating properties the resulting array must have. Available requirements that can be enforced are *NPY_ARRAY_C_CONTIGUOUS*, *NPY_ARRAY_F_CONTIGUOUS*, *NPY_ARRAY_ALIGNED*, *NPY_ARRAY_WRITEABLE*, *NPY_ARRAY_NOTSWAPPED*, NPY_ARRAY_ENSURECOPY, *NPY_ARRAY_UPDATEIFCOPY*, NPY_ARRAY_FORCECAST, and NPY_ARRAY_ENSUREARRAY. Standard combinations of flags can

also be used:

PyObject* **PyArray_FROM_OT**(PyObject* *obj*, int *typenum*)
Similar to *PyArray_FROM_O* except it can take an argument of *typenum* specifying the type-number the returned array.

PyObject* **PyArray_FROM_OTF**(PyObject* *obj*, int *typenum*, int *requirements*)
Combination of *PyArray_FROM_OF* and *PyArray_FROM_OT* allowing both a *typenum* and a *flags* argument to be provided..

PyObject* **PyArray_FROMANY**(PyObject* *obj*, int *typenum*, int *min*, int *max*, int *requirements*)
Similar to *PyArray_FromAny* except the data-type is specified using a typenumber. *PyArray_DescrFromType* (*typenum*) is passed directly to *PyArray_FromAny*. This macro also adds `NPY_DEFAULT` to requirements if `NPY_ARRAY_ENSURECOPY` is passed in as requirements.

PyObject ***PyArray_CheckAxis**(PyObject* *obj*, int* *axis*, int *requirements*)
Encapsulate the functionality of functions and methods that take the axis= keyword and work properly with None as the axis argument. The input array is `obj`, while `*axis` is a converted integer (so that >=MAXDIMS is the None value), and `requirements` gives the needed properties of `obj`. The output is a converted version of the input so that requirements are met and if needed a flattening has occurred. On output negative values of `*axis` are converted and the new value is checked to ensure consistency with the shape of `obj`.

5.4.3 Dealing with types

General check of Python Type

PyArray_Check(op)
Evaluates true if *op* is a Python object whose type is a sub-type of `PyArray_Type`.

PyArray_CheckExact(op)
Evaluates true if *op* is a Python object with type `PyArray_Type`.

PyArray_HasArrayInterface(op, out)
If `op` implements any part of the array interface, then `out` will contain a new reference to the newly created ndarray using the interface or `out` will contain `NULL` if an error during conversion occurs. Otherwise, out will contain a borrowed reference to `Py_NotImplemented` and no error condition is set.

PyArray_HasArrayInterfaceType(op, type, context, out)
If `op` implements any part of the array interface, then `out` will contain a new reference to the newly created ndarray using the interface or `out` will contain `NULL` if an error during conversion occurs. Otherwise, out will contain a borrowed reference to Py_NotImplemented and no error condition is set. This version allows setting of the type and context in the part of the array interface that looks for the `__array__` attribute.

PyArray_IsZeroDim(op)
Evaluates true if *op* is an instance of (a subclass of) `PyArray_Type` and has 0 dimensions.

PyArray_IsScalar(op, cls)
Evaluates true if *op* is an instance of `Py{cls}ArrType_Type`.

PyArray_CheckScalar(op)
Evaluates true if *op* is either an array scalar (an instance of a sub-type of `PyGenericArr_Type`), or an instance of (a sub-class of) `PyArray_Type` whose dimensionality is 0.

PyArray_IsPythonNumber(op)
Evaluates true if *op* is an instance of a builtin numeric type (int, float, complex, long, bool)

PyArray_IsPythonScalar(op)
Evaluates true if *op* is a builtin Python scalar object (int, float, complex, str, unicode, long, bool).

PyArray_IsAnyScalar(op)
Evaluates true if *op* is either a Python scalar object (see *PyArray_IsPythonScalar*) or an array scalar (an instance of a sub- type of `PyGenericArr_Type`).

PyArray_CheckAnyScalar(op)
Evaluates true if *op* is a Python scalar object (see *PyArray_IsPythonScalar*), an array scalar (an instance of a sub-type of `PyGenericArr_Type`) or an instance of a sub-type of `PyArray_Type` whose dimensionality is 0.

Data-type checking

For the typenum macros, the argument is an integer representing an enumerated array data type. For the array type checking macros the argument must be a `PyObject *` that can be directly interpreted as a *PyArrayObject **.

PyTypeNum_ISUNSIGNED(num)

PyDataType_ISUNSIGNED(descr)

PyArray_ISUNSIGNED(obj)
Type represents an unsigned integer.

PyTypeNum_ISSIGNED(num)

PyDataType_ISSIGNED(descr)

PyArray_ISSIGNED(obj)
Type represents a signed integer.

PyTypeNum_ISINTEGER(num)

PyDataType_ISINTEGER(descr)

PyArray_ISINTEGER(obj)
Type represents any integer.

PyTypeNum_ISFLOAT(num)

PyDataType_ISFLOAT(descr)

PyArray_ISFLOAT(obj)
Type represents any floating point number.

PyTypeNum_ISCOMPLEX(num)

PyDataType_ISCOMPLEX(descr)

PyArray_ISCOMPLEX(obj)
Type represents any complex floating point number.

PyTypeNum_ISNUMBER(num)

PyDataType_ISNUMBER(descr)

PyArray_ISNUMBER(obj)
Type represents any integer, floating point, or complex floating point number.

PyTypeNum_ISSTRING(num)

PyDataType_ISSTRING(descr)

PyArray_ISSTRING(obj)
Type represents a string data type.

PyTypeNum_ISPYTHON(num)

PyDataType_ISPYTHON(descr)

PyArray_ISPYTHON(obj)
Type represents an enumerated type corresponding to one of the standard Python scalar (bool, int, float, or complex).

PyTypeNum_ISFLEXIBLE(num)

PyDataType_ISFLEXIBLE(descr)

PyArray_ISFLEXIBLE(obj)
Type represents one of the flexible array types (*NPY_STRING*, *NPY_UNICODE*, or *NPY_VOID*).

PyTypeNum_ISUSERDEF(num)

PyDataType_ISUSERDEF(descr)

PyArray_ISUSERDEF(obj)
Type represents a user-defined type.

PyTypeNum_ISEXTENDED(num)

PyDataType_ISEXTENDED(descr)

PyArray_ISEXTENDED(obj)
Type is either flexible or user-defined.

PyTypeNum_ISOBJECT(num)

PyDataType_ISOBJECT(descr)

PyArray_ISOBJECT(obj)
Type represents object data type.

PyTypeNum_ISBOOL(num)

PyDataType_ISBOOL(descr)

PyArray_ISBOOL(obj)
Type represents Boolean data type.

PyDataType_HASFIELDS(descr)

PyArray_HASFIELDS(obj)
Type has fields associated with it.

PyArray_ISNOTSWAPPED(m)
Evaluates true if the data area of the ndarray *m* is in machine byte-order according to the array's data-type descriptor.

PyArray_ISBYTESWAPPED(m)
Evaluates true if the data area of the ndarray *m* is **not** in machine byte-order according to the array's data-type descriptor.

Bool **PyArray_EquivTypes**(*PyArray_Descr** *type1*, *PyArray_Descr** *type2*)
Return *NPY_TRUE* if *type1* and *type2* actually represent equivalent types for this platform (the fortran member of each type is ignored). For example, on 32-bit platforms, *NPY_LONG* and *NPY_INT* are equivalent. Otherwise return *NPY_FALSE*.

Bool **PyArray_EquivArrTypes**(*PyArrayObject** *a1*, *PyArrayObject* * *a2*)
Return *NPY_TRUE* if *a1* and *a2* are arrays with equivalent types for this platform.

Bool **PyArray_EquivTypenums**(int *typenum1*, int *typenum2*)
Special case of *PyArray_EquivTypes*(...) that does not accept flexible data types but may be easier to call.

int **PyArray_EquivByteorders**({byteorder} *b1*, {byteorder} *b2*)
True if byteorder characters (NPY_LITTLE, NPY_BIG, NPY_NATIVE, NPY_IGNORE) are either equal or equivalent as to their specification of a native byte order. Thus, on a little-endian machine NPY_LITTLE and NPY_NATIVE are equivalent where they are not equivalent on a big-endian machine.

Converting data types

PyObject* **PyArray_Cast**(*PyArrayObject** *arr*, int *typenum*)
Mainly for backwards compatibility to the Numeric C-API and for simple casts to non-flexible types. Return a new array object with the elements of *arr* cast to the data-type *typenum* which must be one of the enumerated types and not a flexible type.

PyObject* **PyArray_CastToType**(*PyArrayObject** *arr*, *PyArray_Descr** *type*, int *fortran*)
Return a new array of the *type* specified, casting the elements of *arr* as appropriate. The fortran argument specifies the ordering of the output array.

int **PyArray_CastTo**(*PyArrayObject** *out*, *PyArrayObject** *in*)
As of 1.6, this function simply calls *PyArray_CopyInto*, which handles the casting.

Cast the elements of the array *in* into the array *out*. The output array should be writeable, have an integer-multiple of the number of elements in the input array (more than one copy can be placed in out), and have a data type that is one of the builtin types. Returns 0 on success and -1 if an error occurs.

PyArray_VectorUnaryFunc* **PyArray_GetCastFunc**(*PyArray_Descr** *from*, int *totype*)
Return the low-level casting function to cast from the given descriptor to the builtin type number. If no casting function exists return NULL and set an error. Using this function instead of direct access to *from* ->f->cast will allow support of any user-defined casting functions added to a descriptors casting dictionary.

int **PyArray_CanCastSafely**(int *fromtype*, int *totype*)
Returns non-zero if an array of data type *fromtype* can be cast to an array of data type *totype* without losing information. An exception is that 64-bit integers are allowed to be cast to 64-bit floating point values even though this can lose precision on large integers so as not to proliferate the use of long doubles without explict requests. Flexible array types are not checked according to their lengths with this function.

int **PyArray_CanCastTo**(*PyArray_Descr** *fromtype*, *PyArray_Descr** *totype*)
PyArray_CanCastTypeTo supercedes this function in NumPy 1.6 and later.

Equivalent to PyArray_CanCastTypeTo(fromtype, totype, NPY_SAFE_CASTING).

int **PyArray_CanCastTypeTo**(*PyArray_Descr** *fromtype*, *PyArray_Descr** *totype*, *NPY_CASTING casting*)
New in version 1.6.

Returns non-zero if an array of data type *fromtype* (which can include flexible types) can be cast safely to an array of data type *totype* (which can include flexible types) according to the casting rule *casting*. For simple types with *NPY_SAFE_CASTING*, this is basically a wrapper around *PyArray_CanCastSafely*, but for flexible types such as strings or unicode, it produces results taking into account their sizes. Integer and float types can only be cast to a string or unicode type using *NPY_SAFE_CASTING* if the string or unicode type is big enough to hold the max value of the integer/float type being cast from.

int **PyArray_CanCastArrayTo**(*PyArrayObject** *arr*, *PyArray_Descr** *totype*, *NPY_CASTING casting*)
New in version 1.6.

Returns non-zero if *arr* can be cast to *totype* according to the casting rule given in *casting*. If *arr* is an array scalar, its value is taken into account, and non-zero is also returned when the value will not overflow or be truncated to an integer when converting to a smaller type.

This is almost the same as the result of PyArray_CanCastTypeTo(PyArray_MinScalarType(arr), totype, casting), but it also handles a special case arising because the set of uint values is not a subset of the int values for types with the same number of bits.

*PyArray_Descr** **PyArray_MinScalarType**(*PyArrayObject** *arr*)
New in version 1.6.

If *arr* is an array, returns its data type descriptor, but if *arr* is an array scalar (has 0 dimensions), it finds the data type of smallest size to which the value may be converted without overflow or truncation to an integer.

This function will not demote complex to float or anything to boolean, but will demote a signed integer to an unsigned integer when the scalar value is positive.

*PyArray_Descr** **PyArray_PromoteTypes**(*PyArray_Descr** *type1*, *PyArray_Descr** *type2*)
New in version 1.6.

Finds the data type of smallest size and kind to which *type1* and *type2* may be safely converted. This function is symmetric and associative. A string or unicode result will be the proper size for storing the max value of the input types converted to a string or unicode.

*PyArray_Descr** **PyArray_ResultType**(npy_intp *narrs*, *PyArrayObject***arrs, npy_intp *ndtypes*, *PyArray_Descr***dtypes)
New in version 1.6.

This applies type promotion to all the inputs, using the NumPy rules for combining scalars and arrays, to determine the output type of a set of operands. This is the same result type that ufuncs produce. The specific algorithm used is as follows.

Categories are determined by first checking which of boolean, integer (int/uint), or floating point (float/complex) the maximum kind of all the arrays and the scalars are.

If there are only scalars or the maximum category of the scalars is higher than the maximum category of the arrays, the data types are combined with *PyArray_PromoteTypes* to produce the return value.

Otherwise, PyArray_MinScalarType is called on each array, and the resulting data types are all combined with *PyArray_PromoteTypes* to produce the return value.

The set of int values is not a subset of the uint values for types with the same number of bits, something not reflected in *PyArray_MinScalarType*, but handled as a special case in PyArray_ResultType.

int **PyArray_ObjectType**(PyObject* *op*, int *mintype*)
This function is superceded by *PyArray_MinScalarType* and/or *PyArray_ResultType*.

This function is useful for determining a common type that two or more arrays can be converted to. It only works for non-flexible array types as no itemsize information is passed. The *mintype* argument represents the minimum type acceptable, and *op* represents the object that will be converted to an array. The return value is the enumerated typenumber that represents the data-type that *op* should have.

void **PyArray_ArrayType**(PyObject* *op*, *PyArray_Descr** *mintype*, *PyArray_Descr** *outtype*)
This function is superceded by *PyArray_ResultType*.

This function works similarly to *PyArray_ObjectType* (...) except it handles flexible arrays. The *mintype* argument can have an itemsize member and the *outtype* argument will have an itemsize member at least as big but perhaps bigger depending on the object *op*.

*PyArrayObject*** **PyArray_ConvertToCommonType**(PyObject* *op*, int* *n*)
The functionality this provides is largely superceded by iterator *NpyIter* introduced in 1.6, with flag *NPY_ITER_COMMON_DTYPE* or with the same dtype parameter for all operands.

Convert a sequence of Python objects contained in *op* to an array of ndarrays each having the same data type. The type is selected based on the typenumber (larger type number is chosen over a smaller one) ignoring objects that are only scalars. The length of the sequence is returned in *n*, and an *n* -length array of *PyArrayObject* pointers is the return value (or NULL if an error occurs). The returned array must be freed by the caller of this routine (using *PyDataMem_FREE*) and all the array objects in it DECREF 'd or a memory-leak will occur. The example template-code below shows a typically usage:

```
mps = PyArray_ConvertToCommonType(obj, &n);
if (mps==NULL) return NULL;
{code}
<before return>
for (i=0; i<n; i++) Py_DECREF(mps[i]);
PyDataMem_FREE(mps);
{return}
```

char* **PyArray_Zero**(*PyArrayObject** *arr*)
A pointer to newly created memory of size *arr* ->itemsize that holds the representation of 0 for that type. The returned pointer, *ret*, **must be freed** using *PyDataMem_FREE* (ret) when it is not needed anymore.

char* **PyArray_One**(*PyArrayObject** *arr*)
A pointer to newly created memory of size *arr* ->itemsize that holds the representation of 1 for that type. The returned pointer, *ret*, **must be freed** using *PyDataMem_FREE* (ret) when it is not needed anymore.

int **PyArray_ValidType**(int *typenum*)
Returns *NPY_TRUE* if *typenum* represents a valid type-number (builtin or user-defined or character code). Otherwise, this function returns *NPY_FALSE*.

New data types

void **PyArray_InitArrFuncs**(*PyArray_ArrFuncs** *f*)
Initialize all function pointers and members to NULL.

int **PyArray_RegisterDataType**(*PyArray_Descr** *dtype*)
Register a data-type as a new user-defined data type for arrays. The type must have most of its entries filled

in. This is not always checked and errors can produce segfaults. In particular, the typeobj member of the `dtype` structure must be filled with a Python type that has a fixed-size element-size that corresponds to the elsize member of *dtype*. Also the `f` member must have the required functions: nonzero, copyswap, copyswapn, getitem, setitem, and cast (some of the cast functions may be `NULL` if no support is desired). To avoid confusion, you should choose a unique character typecode but this is not enforced and not relied on internally.

A user-defined type number is returned that uniquely identifies the type. A pointer to the new structure can then be obtained from *PyArray_DescrFromType* using the returned type number. A -1 is returned if an error occurs. If this *dtype* has already been registered (checked only by the address of the pointer), then return the previously-assigned type-number.

int **PyArray_RegisterCastFunc** (*PyArray_Descr* descr*, int *totype*, PyArray_VectorUnaryFunc* *castfunc*)

Register a low-level casting function, *castfunc*, to convert from the data-type, *descr*, to the given data-type number, *totype*. Any old casting function is over-written. A `0` is returned on success or a `-1` on failure.

int **PyArray_RegisterCanCast** (*PyArray_Descr* descr*, int *totype*, *NPY_SCALARKIND scalar*)

Register the data-type number, *totype*, as castable from data-type object, *descr*, of the given *scalar* kind. Use *scalar* = `NPY_NOSCALAR` to register that an array of data-type *descr* can be cast safely to a data-type whose type_number is *totype*.

Special functions for NPY_OBJECT

int **PyArray_INCREF** (*PyArrayObject* op*)

Used for an array, *op*, that contains any Python objects. It increments the reference count of every object in the array according to the data-type of *op*. A -1 is returned if an error occurs, otherwise 0 is returned.

void **PyArray_Item_INCREF** (char* *ptr*, *PyArray_Descr* dtype*)

A function to INCREF all the objects at the location *ptr* according to the data-type *dtype*. If *ptr* is the start of a structured type with an object at any offset, then this will (recursively) increment the reference count of all object-like items in the structured type.

int **PyArray_XDECREF** (*PyArrayObject* op*)

Used for an array, *op*, that contains any Python objects. It decrements the reference count of every object in the array according to the data-type of *op*. Normal return value is 0. A -1 is returned if an error occurs.

void **PyArray_Item_XDECREF** (char* *ptr*, *PyArray_Descr* dtype*)

A function to XDECREF all the object-like items at the location *ptr* as recorded in the data-type, *dtype*. This works recursively so that if `dtype` itself has fields with data-types that contain object-like items, all the object-like fields will be XDECREF `'d`.

void **PyArray_FillObjectArray** (*PyArrayObject* arr*, PyObject* *obj*)

Fill a newly created array with a single value obj at all locations in the structure with object data-types. No checking is performed but *arr* must be of data-type *NPY_OBJECT* and be single-segment and uninitialized (no previous objects in position). Use `PyArray_DECREF` (*arr*) if you need to decrement all the items in the object array prior to calling this function.

5.4.4 Array flags

The `flags` attribute of the `PyArrayObject` structure contains important information about the memory used by the array (pointed to by the data member) This flag information must be kept accurate or strange results and even segfaults may result.

There are 6 (binary) flags that describe the memory area used by the data buffer. These constants are defined in `arrayobject.h` and determine the bit-position of the flag. Python exposes a nice attribute- based interface as well as a dictionary-like interface for getting (and, if appropriate, setting) these flags.

Memory areas of all kinds can be pointed to by an ndarray, necessitating these flags. If you get an arbitrary `PyArrayObject` in C-code, you need to be aware of the flags that are set. If you need to guarantee a certain kind of array (like *NPY_ARRAY_C_CONTIGUOUS* and *NPY_ARRAY_BEHAVED*), then pass these requirements into the PyArray_FromAny function.

Basic Array Flags

An ndarray can have a data segment that is not a simple contiguous chunk of well-behaved memory you can manipulate. It may not be aligned with word boundaries (very important on some platforms). It might have its data in a different byte-order than the machine recognizes. It might not be writeable. It might be in Fortan-contiguous order. The array flags are used to indicate what can be said about data associated with an array.

In versions 1.6 and earlier of NumPy, the following flags did not have the _ARRAY_ macro namespace in them. That form of the constant names is deprecated in 1.7.

NPY_ARRAY_C_CONTIGUOUS
: The data area is in C-style contiguous order (last index varies the fastest).

NPY_ARRAY_F_CONTIGUOUS
: The data area is in Fortran-style contiguous order (first index varies the fastest).

Note: Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true. The correct way to access the `itemsize` of an array from the C API is `PyArray_ITEMSIZE(arr)`.

See also:

Internal memory layout of an ndarray

NPY_ARRAY_OWNDATA
: The data area is owned by this array.

NPY_ARRAY_ALIGNED
: The data area and all array elements are aligned appropriately.

NPY_ARRAY_WRITEABLE
: The data area can be written to.

  Notice that the above 3 flags are are defined so that a new, well- behaved array has these flags defined as true.

NPY_ARRAY_UPDATEIFCOPY
: The data area represents a (well-behaved) copy whose information should be transferred back to the original when this array is deleted.

  This is a special flag that is set if this array represents a copy made because a user required certain flags in *PyArray_FromAny* and a copy had to be made of some other array (and the user asked for this flag to be set in such a situation). The base attribute then points to the "misbehaved" array (which is set read_only). When the array with this flag set is deallocated, it will copy its contents back to the "misbehaved" array (casting if necessary) and will reset the "misbehaved" array to *NPY_ARRAY_WRITEABLE*. If the "misbehaved" array was not *NPY_ARRAY_WRITEABLE* to begin with then *PyArray_FromAny* would have returned an error because *NPY_ARRAY_UPDATEIFCOPY* would not have been possible.

PyArray_UpdateFlags (obj, flags) will update the `obj->flags` for `flags` which can be any of *NPY_ARRAY_C_CONTIGUOUS*, *NPY_ARRAY_F_CONTIGUOUS*, *NPY_ARRAY_ALIGNED*, or *NPY_ARRAY_WRITEABLE*.

Combinations of array flags

NPY_ARRAY_BEHAVED
NPY_ARRAY_ALIGNED | *NPY_ARRAY_WRITEABLE*

NPY_ARRAY_CARRAY
NPY_ARRAY_C_CONTIGUOUS | *NPY_ARRAY_BEHAVED*

NPY_ARRAY_CARRAY_RO
NPY_ARRAY_C_CONTIGUOUS | *NPY_ARRAY_ALIGNED*

NPY_ARRAY_FARRAY
NPY_ARRAY_F_CONTIGUOUS | *NPY_ARRAY_BEHAVED*

NPY_ARRAY_FARRAY_RO
NPY_ARRAY_F_CONTIGUOUS | *NPY_ARRAY_ALIGNED*

NPY_ARRAY_DEFAULT
NPY_ARRAY_CARRAY

NPY_ARRAY_UPDATE_ALL
NPY_ARRAY_C_CONTIGUOUS | *NPY_ARRAY_F_CONTIGUOUS* | *NPY_ARRAY_ALIGNED*

Flag-like constants

These constants are used in *PyArray_FromAny* (and its macro forms) to specify desired properties of the new array.

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 (or bigndarray), and not a sub-class.

NPY_ARRAY_NOTSWAPPED
Only used in *PyArray_CheckFromAny* to over-ride the byteorder of the data-type object passed in.

NPY_ARRAY_BEHAVED_NS
NPY_ARRAY_ALIGNED | *NPY_ARRAY_WRITEABLE* | *NPY_ARRAY_NOTSWAPPED*

Flag checking

For all of these macros *arr* must be an instance of a (subclass of) `PyArray_Type`, but no checking is done.

PyArray_CHKFLAGS (arr, flags)
The first parameter, arr, must be an ndarray or subclass. The parameter, *flags*, should be an integer consisting of bitwise combinations of the possible flags an array can have: *NPY_ARRAY_C_CONTIGUOUS*, *NPY_ARRAY_F_CONTIGUOUS*, *NPY_ARRAY_OWNDATA*, *NPY_ARRAY_ALIGNED*, *NPY_ARRAY_WRITEABLE*, *NPY_ARRAY_UPDATEIFCOPY*.

PyArray_IS_C_CONTIGUOUS (arr)
Evaluates true if *arr* is C-style contiguous.

PyArray_IS_F_CONTIGUOUS (arr)
Evaluates true if *arr* is Fortran-style contiguous.

PyArray_ISFORTRAN (arr)
Evaluates true if *arr* is Fortran-style contiguous and *not* C-style contiguous. *PyArray_IS_F_CONTIGUOUS* is the correct way to test for Fortran-style contiguity.

PyArray_ISWRITEABLE (arr)
Evaluates true if the data area of *arr* can be written to

PyArray_ISALIGNED (arr)
Evaluates true if the data area of *arr* is properly aligned on the machine.

PyArray_ISBEHAVED (arr)
Evalutes true if the data area of *arr* is aligned and writeable and in machine byte-order according to its descriptor.

PyArray_ISBEHAVED_RO (arr)
Evaluates true if the data area of *arr* is aligned and in machine byte-order.

PyArray_ISCARRAY (arr)
Evaluates true if the data area of *arr* is C-style contiguous, and *PyArray_ISBEHAVED* (*arr*) is true.

PyArray_ISFARRAY (arr)
Evaluates true if the data area of *arr* is Fortran-style contiguous and *PyArray_ISBEHAVED* (*arr*) is true.

PyArray_ISCARRAY_RO (arr)
Evaluates true if the data area of *arr* is C-style contiguous, aligned, and in machine byte-order.

PyArray_ISFARRAY_RO (arr)
Evaluates true if the data area of *arr* is Fortran-style contiguous, aligned, and in machine byte-order **.**

PyArray_ISONESEGMENT (arr)
Evaluates true if the data area of *arr* consists of a single (C-style or Fortran-style) contiguous segment.

void **PyArray_UpdateFlags** (*PyArrayObject** *arr*, int *flagmask*)
The *NPY_ARRAY_C_CONTIGUOUS*, *NPY_ARRAY_ALIGNED*, and *NPY_ARRAY_F_CONTIGUOUS* array flags can be "calculated" from the array object itself. This routine updates one or more of these flags of *arr* as specified in *flagmask* by performing the required calculation.

Warning: It is important to keep the flags updated (using *PyArray_UpdateFlags* can help) whenever a manipulation with an array is performed that might cause them to change. Later calculations in NumPy that rely on the state of these flags do not repeat the calculation to update them.

5.4.5 Array method alternative API

Conversion

PyObject* **PyArray_GetField** (*PyArrayObject** *self*, *PyArray_Descr** *dtype*, int *offset*)
Equivalent to `ndarray.getfield` (*self*, *dtype*, *offset*). Return a new array of the given *dtype* using the data in the current array at a specified *offset* in bytes. The *offset* plus the itemsize of the new array type must be less than *self* ->descr->elsize or an error is raised. The same shape and strides as the original array are used. Therefore, this function has the effect of returning a field from a structured array. But, it can also be used to select specific bytes or groups of bytes from any array type.

int **PyArray_SetField**(*PyArrayObject* self*, *PyArray_Descr* dtype*, int *offset*, PyObject* *val*)
Equivalent to `ndarray.setfield` (*self*, *val*, *dtype*, *offset*). Set the field starting at *offset* in bytes and of the given *dtype* to *val*. The *offset* plus *dtype* ->elsize must be less than *self* ->descr->elsize or an error is raised. Otherwise, the *val* argument is converted to an array and copied into the field pointed to. If necessary, the elements of *val* are repeated to fill the destination array, But, the number of elements in the destination must be an integer multiple of the number of elements in *val*.

PyObject* **PyArray_Byteswap**(*PyArrayObject* self*, Bool *inplace*)
Equivalent to `ndarray.byteswap` (*self*, *inplace*). Return an array whose data area is byteswapped. If *inplace* is non-zero, then do the byteswap inplace and return a reference to self. Otherwise, create a byteswapped copy and leave self unchanged.

PyObject* **PyArray_NewCopy**(*PyArrayObject* old*, *NPY_ORDER order*)
Equivalent to `ndarray.copy` (*self*, *fortran*). Make a copy of the *old* array. The returned array is always aligned and writeable with data interpreted the same as the old array. If *order* is *NPY_CORDER*, then a C-style contiguous array is returned. If *order* is *NPY_FORTRANORDER*, then a Fortran-style contiguous array is returned. If *order is* *NPY_ANYORDER*, then the array returned is Fortran-style contiguous only if the old one is; otherwise, it is C-style contiguous.

PyObject* **PyArray_ToList**(*PyArrayObject* self*)
Equivalent to `ndarray.tolist` (*self*). Return a nested Python list from *self*.

PyObject* **PyArray_ToString**(*PyArrayObject* self*, *NPY_ORDER order*)
Equivalent to `ndarray.tobytes` (*self*, *order*). Return the bytes of this array in a Python string.

PyObject* **PyArray_ToFile**(*PyArrayObject* self*, FILE* *fp*, char* *sep*, char* *format*)
Write the contents of *self* to the file pointer *fp* in C-style contiguous fashion. Write the data as binary bytes if *sep* is the string “”or `NULL`. Otherwise, write the contents of *self* as text using the *sep* string as the item separator. Each item will be printed to the file. If the *format* string is not `NULL` or “”, then it is a Python print statement format string showing how the items are to be written.

int **PyArray_Dump**(PyObject* *self*, PyObject* *file*, int *protocol*)
Pickle the object in *self* to the given *file* (either a string or a Python file object). If *file* is a Python string it is considered to be the name of a file which is then opened in binary mode. The given *protocol* is used (if *protocol* is negative, or the highest available is used). This is a simple wrapper around cPickle.dump(*self*, *file*, *protocol*).

PyObject* **PyArray_Dumps**(PyObject* *self*, int *protocol*)
Pickle the object in *self* to a Python string and return it. Use the Pickle *protocol* provided (or the highest available if *protocol* is negative).

int **PyArray_FillWithScalar**(*PyArrayObject* arr*, PyObject* *obj*)
Fill the array, *arr*, with the given scalar object, *obj*. The object is first converted to the data type of *arr*, and then copied into every location. A -1 is returned if an error occurs, otherwise 0 is returned.

PyObject* **PyArray_View**(*PyArrayObject* self*, *PyArray_Descr* dtype*, PyTypeObject **ptype*)
Equivalent to `ndarray.view` (*self*, *dtype*). Return a new view of the array *self* as possibly a different data-type, *dtype*, and different array subclass *ptype*.

If *dtype* is `NULL`, then the returned array will have the same data type as *self*. The new data-type must be consistent with the size of *self*. Either the itemsizes must be identical, or *self* must be single-segment and the total number of bytes must be the same. In the latter case the dimensions of the returned array will be altered in the last (or first for Fortran-style contiguous arrays) dimension. The data area of the returned array and self is exactly the same.

Shape Manipulation

PyObject* **PyArray_Newshape**(*PyArrayObject* self*, *PyArray_Dims* newshape*, *NPY_ORDER order*)
Result will be a new array (pointing to the same memory location as *self* if possible), but having a shape given

by *newshape*. If the new shape is not compatible with the strides of *self*, then a copy of the array with the new specified shape will be returned.

PyObject* **PyArray_Reshape**(*PyArrayObject* self*, PyObject* *shape*)
Equivalent to `ndarray.reshape` (*self*, *shape*) where *shape* is a sequence. Converts *shape* to a *PyArray_Dims* structure and calls *PyArray_Newshape* internally. For back-ward compatability – Not recommended

PyObject* **PyArray_Squeeze**(*PyArrayObject* self*)
Equivalent to `ndarray.squeeze` (*self*). Return a new view of *self* with all of the dimensions of length 1 removed from the shape.

Warning: matrix objects are always 2-dimensional. Therefore, *PyArray_Squeeze* has no effect on arrays of matrix sub-class.

PyObject* **PyArray_SwapAxes**(*PyArrayObject* self*, int *a1*, int *a2*)
Equivalent to `ndarray.swapaxes` (*self*, *a1*, *a2*). The returned array is a new view of the data in *self* with the given axes, *a1* and *a2*, swapped.

PyObject* **PyArray_Resize**(*PyArrayObject* self*, *PyArray_Dims* *newshape*, int *refcheck*, *NPY_ORDER fortran*)
Equivalent to `ndarray.resize` (*self*, *newshape*, refcheck = *refcheck*, order= fortran). This function only works on single-segment arrays. It changes the shape of *self* inplace and will reallocate the memory for *self* if *newshape* has a different total number of elements then the old shape. If reallocation is necessary, then *self* must own its data, have *self* - `>base==NULL`, have *self* - `>weakrefs==NULL`, and (unless refcheck is 0) not be referenced by any other array. A reference to the new array is returned. The fortran argument can be *NPY_ANYORDER*, *NPY_CORDER*, or *NPY_FORTRANORDER*. It currently has no effect. Eventually it could be used to determine how the resize operation should view the data when constructing a differently-dimensioned array.

PyObject* **PyArray_Transpose**(*PyArrayObject* self*, *PyArray_Dims* *permute*)
Equivalent to `ndarray.transpose` (*self*, *permute*). Permute the axes of the ndarray object *self* according to the data structure *permute* and return the result. If *permute* is `NULL`, then the resulting array has its axes reversed. For example if *self* has shape $10 \times 20 \times 30$, and *permute* `.ptr` is (0,2,1) the shape of the result is $10 \times 30 \times 20$. If *permute* is `NULL`, the shape of the result is $30 \times 20 \times 10$.

PyObject* **PyArray_Flatten**(*PyArrayObject* self*, *NPY_ORDER order*)
Equivalent to `ndarray.flatten` (*self*, *order*). Return a 1-d copy of the array. If *order* is *NPY_FORTRANORDER* the elements are scanned out in Fortran order (first-dimension varies the fastest). If *order* is *NPY_CORDER*, the elements of `self` are scanned in C-order (last dimension varies the fastest). If *order* *NPY_ANYORDER*, then the result of *PyArray_ISFORTRAN* (*self*) is used to determine which order to flatten.

PyObject* **PyArray_Ravel**(*PyArrayObject* self*, *NPY_ORDER order*)
Equivalent to *self*.ravel(*order*). Same basic functionality as *PyArray_Flatten* (*self*, *order*) except if *order* is 0 and *self* is C-style contiguous, the shape is altered but no copy is performed.

Item selection and manipulation

PyObject* **PyArray_TakeFrom**(*PyArrayObject* self*, PyObject* *indices*, int *axis*, *PyArrayObject* *ret*, *NPY_CLIPMODE clipmode*)
Equivalent to `ndarray.take` (*self*, *indices*, *axis*, *ret*, *clipmode*) except *axis* =None in Python is obtained by setting *axis* = *NPY_MAXDIMS* in C. Extract the items from self indicated by the integer-valued *indices* along the given *axis*. The clipmode argument can be *NPY_RAISE*, *NPY_WRAP*, or *NPY_CLIP* to indicate what to do with out-of-bound indices. The *ret* argument can specify an output array rather than having one created internally.

PyObject* **PyArray_PutTo**(*PyArrayObject** *self*, PyObject* *values*, PyObject* *indices*, *NPY_CLIPMODE* *clipmode*)

Equivalent to *self*.put(*values*, *indices*, *clipmode*). Put *values* into *self* at the corresponding (flattened) *indices*. If *values* is too small it will be repeated as necessary.

PyObject* **PyArray_PutMask**(*PyArrayObject** *self*, PyObject* *values*, PyObject* *mask*)

Place the *values* in *self* wherever corresponding positions (using a flattened context) in *mask* are true. The *mask* and *self* arrays must have the same total number of elements. If *values* is too small, it will be repeated as necessary.

PyObject* **PyArray_Repeat**(*PyArrayObject** *self*, PyObject* *op*, int *axis*)

Equivalent to `ndarray.repeat` (*self*, *op*, *axis*). Copy the elements of *self*, *op* times along the given *axis*. Either *op* is a scalar integer or a sequence of length *self* ->dimensions[*axis*] indicating how many times to repeat each item along the axis.

PyObject* **PyArray_Choose**(*PyArrayObject** *self*, PyObject* *op*, *PyArrayObject** *ret*, *NPY_CLIPMODE* *clipmode*)

Equivalent to `ndarray.choose` (*self*, *op*, *ret*, *clipmode*). Create a new array by selecting elements from the sequence of arrays in *op* based on the integer values in *self*. The arrays must all be broadcastable to the same shape and the entries in *self* should be between 0 and len(*op*). The output is placed in *ret* unless it is `NULL` in which case a new output is created. The *clipmode* argument determines behavior for when entries in *self* are not between 0 and len(*op*).

NPY_RAISE
: raise a ValueError;

NPY_WRAP
: wrap values < 0 by adding len(*op*) and values >=len(*op*) by subtracting len(*op*) until they are in range;

NPY_CLIP
: all values are clipped to the region [0, len(*op*)).

PyObject* **PyArray_Sort**(*PyArrayObject** *self*, int *axis*)

Equivalent to `ndarray.sort` (*self*, *axis*). Return an array with the items of *self* sorted along *axis*.

PyObject* **PyArray_ArgSort**(*PyArrayObject** *self*, int *axis*)

Equivalent to `ndarray.argsort` (*self*, *axis*). Return an array of indices such that selection of these indices along the given `axis` would return a sorted version of *self*. If *self* ->descr is a data-type with fields defined, then self->descr->names is used to determine the sort order. A comparison where the first field is equal will use the second field and so on. To alter the sort order of a structured array, create a new data-type with a different order of names and construct a view of the array with that new data-type.

PyObject* **PyArray_LexSort**(PyObject* *sort_keys*, int *axis*)

Given a sequence of arrays (*sort_keys*) of the same shape, return an array of indices (similar to *PyArray_ArgSort* (...)) that would sort the arrays lexicographically. A lexicographic sort specifies that when two keys are found to be equal, the order is based on comparison of subsequent keys. A merge sort (which leaves equal entries unmoved) is required to be defined for the types. The sort is accomplished by sorting the indices first using the first *sort_key* and then using the second *sort_key* and so forth. This is equivalent to the lexsort(*sort_keys*, *axis*) Python command. Because of the way the merge-sort works, be sure to understand the order the *sort_keys* must be in (reversed from the order you would use when comparing two elements).

If these arrays are all collected in a structured array, then *PyArray_Sort* (...) can also be used to sort the array directly.

PyObject* **PyArray_SearchSorted**(*PyArrayObject** *self*, PyObject* *values*, NPY_SEARCHSIDE *side*, PyObject* *perm*)

Equivalent to `ndarray.searchsorted` (*self*, *values*, *side*, *perm*). Assuming *self* is a 1-d array in ascending order, then the output is an array of indices the same shape as *values* such that, if the elements in *values* were

inserted before the indices, the order of *self* would be preserved. No checking is done on whether or not self is in ascending order.

The *side* argument indicates whether the index returned should be that of the first suitable location (if `NPY_SEARCHLEFT`) or of the last (if `NPY_SEARCHRIGHT`).

The *sorter* argument, if not `NULL`, must be a 1D array of integer indices the same length as *self*, that sorts it into ascending order. This is typically the result of a call to *PyArray_ArgSort* (...) Binary search is used to find the required insertion points.

int **PyArray_Partition**(*PyArrayObject* *self, *PyArrayObject* * ktharray, int axis, NPY_SELECTKIND *which*)

Equivalent to `ndarray.partition` (*self*, *ktharray*, *axis*, *kind*). Partitions the array so that the values of the element indexed by *ktharray* are in the positions they would be if the array is fully sorted and places all elements smaller than the kth before and all elements equal or greater after the kth element. The ordering of all elements within the partitions is undefined. If *self*->descr is a data-type with fields defined, then self->descr->names is used to determine the sort order. A comparison where the first field is equal will use the second field and so on. To alter the sort order of a structured array, create a new data-type with a different order of names and construct a view of the array with that new data-type. Returns zero on success and -1 on failure.

PyObject* **PyArray_ArgPartition**(*PyArrayObject* *op, *PyArrayObject* * ktharray, int axis, NPY_SELECTKIND *which*)

Equivalent to `ndarray.argpartition` (*self*, *ktharray*, *axis*, *kind*). Return an array of indices such that selection of these indices along the given `axis` would return a partitioned version of *self*.

PyObject* **PyArray_Diagonal**(*PyArrayObject** *self*, int *offset*, int *axis1*, int *axis2*)

Equivalent to `ndarray.diagonal` (*self*, *offset*, *axis1*, *axis2*). Return the *offset* diagonals of the 2-d arrays defined by *axis1* and *axis2*.

npy_intp **PyArray_CountNonzero**(*PyArrayObject** *self*)

New in version 1.6.

Counts the number of non-zero elements in the array object *self*.

PyObject* **PyArray_Nonzero**(*PyArrayObject** *self*)

Equivalent to `ndarray.nonzero` (*self*). Returns a tuple of index arrays that select elements of *self* that are nonzero. If (nd= *PyArray_NDIM* (`self`))==1, then a single index array is returned. The index arrays have data type *NPY_INTP*. If a tuple is returned (nd $\neq$ 1), then its length is nd.

PyObject* **PyArray_Compress**(*PyArrayObject** *self*, PyObject* *condition*, int *axis*, *PyArrayObject** *out*)

Equivalent to `ndarray.compress` (*self*, *condition*, *axis*). Return the elements along *axis* corresponding to elements of *condition* that are true.

Calculation

Tip: Pass in *NPY_MAXDIMS* for axis in order to achieve the same effect that is obtained by passing in *axis* = `None` in Python (treating the array as a 1-d array).

PyObject* **PyArray_ArgMax**(*PyArrayObject** *self*, int *axis*, *PyArrayObject** *out*)

Equivalent to `ndarray.argmax` (*self*, *axis*). Return the index of the largest element of *self* along *axis*.

PyObject* **PyArray_ArgMin**(*PyArrayObject** *self*, int *axis*, *PyArrayObject** *out*)

Equivalent to `ndarray.argmin` (*self*, *axis*). Return the index of the smallest element of *self* along *axis*.

Note: The out argument specifies where to place the result. If out is NULL, then the output array is created, otherwise the output is placed in out which must be the correct size and type. A new reference to the output array is always returned even when out is not NULL. The caller of the routine has the responsibility to `DECREF` out if not NULL or a memory-leak will occur.

PyObject* **PyArray_Max**(*PyArrayObject* self*, int *axis*, *PyArrayObject* out*)
Equivalent to `ndarray.max` (*self*, *axis*). Return the largest element of *self* along the given *axis*.

PyObject* **PyArray_Min**(*PyArrayObject* self*, int *axis*, *PyArrayObject* out*)
Equivalent to `ndarray.min` (*self*, *axis*). Return the smallest element of *self* along the given *axis*.

PyObject* **PyArray_Ptp**(*PyArrayObject* self*, int *axis*, *PyArrayObject* out*)
Equivalent to `ndarray.ptp` (*self*, *axis*). Return the difference between the largest element of *self* along *axis* and the smallest element of *self* along *axis*.

Note: The rtype argument specifies the data-type the reduction should take place over. This is important if the data-type of the array is not "large" enough to handle the output. By default, all integer data-types are made at least as large as *NPY_LONG* for the "add" and "multiply" ufuncs (which form the basis for mean, sum, cumsum, prod, and cumprod functions).

PyObject* **PyArray_Mean**(*PyArrayObject* self*, int *axis*, int *rtype*, *PyArrayObject* out*)
Equivalent to `ndarray.mean` (*self*, *axis*, *rtype*). Returns the mean of the elements along the given *axis*, using the enumerated type *rtype* as the data type to sum in. Default sum behavior is obtained using *NPY_NOTYPE* for *rtype*.

PyObject* **PyArray_Trace**(*PyArrayObject* self*, int *offset*, int *axis1*, int *axis2*, int *rtype*, *PyArrayObject* out*)
Equivalent to `ndarray.trace` (*self*, *offset*, *axis1*, *axis2*, *rtype*). Return the sum (using *rtype* as the data type of summation) over the *offset* diagonal elements of the 2-d arrays defined by *axis1* and *axis2* variables. A positive offset chooses diagonals above the main diagonal. A negative offset selects diagonals below the main diagonal.

PyObject* **PyArray_Clip**(*PyArrayObject* self*, PyObject* *min*, PyObject* *max*)
Equivalent to `ndarray.clip` (*self*, *min*, *max*). Clip an array, *self*, so that values larger than *max* are fixed to *max* and values less than *min* are fixed to *min*.

PyObject* **PyArray_Conjugate**(*PyArrayObject* self*)
Equivalent to `ndarray.conjugate` (*self*). Return the complex conjugate of *self*. If *self* is not of complex data type, then return *self* with an reference.

PyObject* **PyArray_Round**(*PyArrayObject* self*, int *decimals*, *PyArrayObject* out*)
Equivalent to `ndarray.round` (*self*, *decimals*, *out*). Returns the array with elements rounded to the nearest decimal place. The decimal place is defined as the $10^{-\text{decimals}}$ digit so that negative *decimals* cause rounding to the nearest 10's, 100's, etc. If out is `NULL`, then the output array is created, otherwise the output is placed in *out* which must be the correct size and type.

PyObject* **PyArray_Std**(*PyArrayObject* self*, int *axis*, int *rtype*, *PyArrayObject* out*)
Equivalent to `ndarray.std` (*self*, *axis*, *rtype*). Return the standard deviation using data along *axis* converted to data type *rtype*.

PyObject* **PyArray_Sum**(*PyArrayObject* self*, int *axis*, int *rtype*, *PyArrayObject* out*)
Equivalent to `ndarray.sum` (*self*, *axis*, *rtype*). Return 1-d vector sums of elements in *self* along *axis*. Perform the sum after converting data to data type *rtype*.

PyObject* **PyArray_CumSum**(*PyArrayObject* self*, int *axis*, int *rtype*, *PyArrayObject* out*)
Equivalent to `ndarray.cumsum` (*self*, *axis*, *rtype*). Return cumulative 1-d sums of elements in *self* along *axis*. Perform the sum after converting data to data type *rtype*.

PyObject* **PyArray_Prod**(*PyArrayObject* self*, int *axis*, int *rtype*, *PyArrayObject* out*)
Equivalent to `ndarray.prod` (*self*, *axis*, *rtype*). Return 1-d products of elements in *self* along *axis*. Perform the product after converting data to data type *rtype*.

PyObject* **PyArray_CumProd** (*PyArrayObject* self*, int *axis*, int *rtype*, *PyArrayObject* out*)
Equivalent to `ndarray.cumprod` (*self*, *axis*, *rtype*). Return 1-d cumulative products of elements in `self` along `axis`. Perform the product after converting data to data type `rtype`.

PyObject* **PyArray_All** (*PyArrayObject* self*, int *axis*, *PyArrayObject* out*)
Equivalent to `ndarray.all` (*self*, *axis*). Return an array with True elements for every 1-d sub-array of `self` defined by `axis` in which all the elements are True.

PyObject* **PyArray_Any** (*PyArrayObject* self*, int *axis*, *PyArrayObject* out*)
Equivalent to `ndarray.any` (*self*, *axis*). Return an array with True elements for every 1-d sub-array of *self* defined by *axis* in which any of the elements are True.

5.4.6 Functions

Array Functions

int **PyArray_AsCArray** (PyObject** *op*, void* *ptr*, npy_intp* *dims*, int *nd*, int *typenum*, int *itemsize*)
Sometimes it is useful to access a multidimensional array as a C-style multi-dimensional array so that algorithms can be implemented using C's a[i][j][k] syntax. This routine returns a pointer, *ptr*, that simulates this kind of C-style array, for 1-, 2-, and 3-d ndarrays.

Parameters

- **op** – The address to any Python object. This Python object will be replaced with an equivalent well-behaved, C-style contiguous, ndarray of the given data type specified by the last two arguments. Be sure that stealing a reference in this way to the input object is justified.
- **ptr** – The address to a (ctype* for 1-d, ctype** for 2-d or ctype*** for 3-d) variable where ctype is the equivalent C-type for the data type. On return, *ptr* will be addressable as a 1-d, 2-d, or 3-d array.
- **dims** – An output array that contains the shape of the array object. This array gives boundaries on any looping that will take place.
- **nd** – The dimensionality of the array (1, 2, or 3).
- **typenum** – The expected data type of the array.
- **itemsize** – This argument is only needed when *typenum* represents a flexible array. Otherwise it should be 0.

Note: The simulation of a C-style array is not complete for 2-d and 3-d arrays. For example, the simulated arrays of pointers cannot be passed to subroutines expecting specific, statically-defined 2-d and 3-d arrays. To pass to functions requiring those kind of inputs, you must statically define the required array and copy data.

---

int **PyArray_Free** (PyObject* *op*, void* *ptr*)
Must be called with the same objects and memory locations returned from *PyArray_AsCArray* (...). This function cleans up memory that otherwise would get leaked.

PyObject* **PyArray_Concatenate** (PyObject* *obj*, int *axis*)
Join the sequence of objects in *obj* together along *axis* into a single array. If the dimensions or types are not compatible an error is raised.

PyObject* **PyArray_InnerProduct** (PyObject* *obj1*, PyObject* *obj2*)
Compute a product-sum over the last dimensions of *obj1* and *obj2*. Neither array is conjugated.

PyObject* **PyArray_MatrixProduct**(PyObject* *obj1*, PyObject* *obj*)
Compute a product-sum over the last dimension of *obj1* and the second-to-last dimension of *obj2*. For 2-d arrays this is a matrix-product. Neither array is conjugated.

PyObject* **PyArray_MatrixProduct2**(PyObject* *obj1*, PyObject* *obj*, PyObject* *out*)
New in version 1.6.

Same as PyArray_MatrixProduct, but store the result in *out*. The output array must have the correct shape, type, and be C-contiguous, or an exception is raised.

PyObject* **PyArray_EinsteinSum**(char* *subscripts*, npy_intp *nop*, *PyArrayObject*** *op_in*, *PyArray_Descr** *dtype*, *NPY_ORDER* *order*, *NPY_CASTING* *casting*, *PyArrayObject** *out*)
New in version 1.6.

Applies the Einstein summation convention to the array operands provided, returning a new array or placing the result in *out*. The string in *subscripts* is a comma separated list of index letters. The number of operands is in *nop*, and *op_in* is an array containing those operands. The data type of the output can be forced with *dtype*, the output order can be forced with *order* (*NPY_KEEPORDER* is recommended), and when *dtype* is specified, *casting* indicates how permissive the data conversion should be.

See the `einsum` function for more details.

PyObject* **PyArray_CopyAndTranspose**(PyObject * *op*)
A specialized copy and transpose function that works only for 2-d arrays. The returned array is a transposed copy of *op*.

PyObject* **PyArray_Correlate**(PyObject* *op1*, PyObject* *op2*, int *mode*)
Compute the 1-d correlation of the 1-d arrays *op1* and *op2* . The correlation is computed at each output point by multiplying *op1* by a shifted version of *op2* and summing the result. As a result of the shift, needed values outside of the defined range of *op1* and *op2* are interpreted as zero. The mode determines how many shifts to return: 0 - return only shifts that did not need to assume zero- values; 1 - return an object that is the same size as *op1*, 2 - return all possible shifts (any overlap at all is accepted).

Notes

This does not compute the usual correlation: if op2 is larger than op1, the arguments are swapped, and the conjugate is never taken for complex arrays. See PyArray_Correlate2 for the usual signal processing correlation.

PyObject* **PyArray_Correlate2**(PyObject* *op1*, PyObject* *op2*, int *mode*)
Updated version of PyArray_Correlate, which uses the usual definition of correlation for 1d arrays. The correlation is computed at each output point by multiplying *op1* by a shifted version of *op2* and summing the result. As a result of the shift, needed values outside of the defined range of *op1* and *op2* are interpreted as zero. The mode determines how many shifts to return: 0 - return only shifts that did not need to assume zero- values; 1 - return an object that is the same size as *op1*, 2 - return all possible shifts (any overlap at all is accepted).

Notes

Compute z as follows:

```
z[k] = sum_n op1[n] * conj(op2[n+k])
```

PyObject* **PyArray_Where**(PyObject* *condition*, PyObject* *x*, PyObject* *y*)
If both `x` and `y` are `NULL`, then return *PyArray_Nonzero*(*condition*). Otherwise, both *x* and *y* must be given and the object returned is shaped like *condition* and has elements of *x* and *y* where *condition* is respectively True or False.

Other functions

Bool **PyArray_CheckStrides**(int *elsize*, int *nd*, npy_intp *numbytes*, npy_intp* *dims*, npy_intp* *newstrides*)

Determine if *newstrides* is a strides array consistent with the memory of an *nd* -dimensional array with shape `dims` and element-size, *elsize*. The *newstrides* array is checked to see if jumping by the provided number of bytes in each direction will ever mean jumping more than *numbytes* which is the assumed size of the available memory segment. If *numbytes* is 0, then an equivalent *numbytes* is computed assuming *nd*, *dims*, and *elsize* refer to a single-segment array. Return *NPY_TRUE* if *newstrides* is acceptable, otherwise return *NPY_FALSE*.

npy_intp **PyArray_MultiplyList**(npy_intp* *seq*, int *n*)

int **PyArray_MultiplyIntList**(int* *seq*, int *n*)

Both of these routines multiply an *n* -length array, *seq*, of integers and return the result. No overflow checking is performed.

int **PyArray_CompareLists**(npy_intp* *l1*, npy_intp* *l2*, int *n*)

Given two *n* -length arrays of integers, *l1*, and *l2*, return 1 if the lists are identical; otherwise, return 0.

5.4.7 Auxiliary Data With Object Semantics

New in version 1.7.0.

NpyAuxData

When working with more complex dtypes which are composed of other dtypes, such as the struct dtype, creating inner loops that manipulate the dtypes requires carrying along additional data. NumPy supports this idea through a struct *NpyAuxData*, mandating a few conventions so that it is possible to do this.

Defining an *NpyAuxData* is similar to defining a class in C++, but the object semantics have to be tracked manually since the API is in C. Here's an example for a function which doubles up an element using an element copier function as a primitive.:

```
typedef struct {
    NpyAuxData base;
    ElementCopier_Func *func;
    NpyAuxData *funcdata;
} eldoubler_aux_data;

void free_element_doubler_aux_data(NpyAuxData *data)
{
    eldoubler_aux_data *d = (eldoubler_aux_data *)data;
    /* Free the memory owned by this auxadata */
    NPY_AUXDATA_FREE(d->funcdata);
    PyArray_free(d);
}

NpyAuxData *clone_element_doubler_aux_data(NpyAuxData *data)
{
    eldoubler_aux_data *ret = PyArray_malloc(sizeof(eldoubler_aux_data));
    if (ret == NULL) {
        return NULL;
    }

    /* Raw copy of all data */
    memcpy(ret, data, sizeof(eldoubler_aux_data));
```

```
    /* Fix up the owned auxdata so we have our own copy */
    ret->funcdata = NPY_AUXDATA_CLONE(ret->funcdata);
    if (ret->funcdata == NULL) {
        PyArray_free(ret);
        return NULL;
    }

    return (NpyAuxData *)ret;
}

NpyAuxData *create_element_doubler_aux_data(
                            ElementCopier_Func *func,
                            NpyAuxData *funcdata)
{
    eldoubler_aux_data *ret = PyArray_malloc(sizeof(eldoubler_aux_data));
    if (ret == NULL) {
        PyErr_NoMemory();
        return NULL;
    }
    memset(&ret, 0, sizeof(eldoubler_aux_data));
    ret->base->free = &free_element_doubler_aux_data;
    ret->base->clone = &clone_element_doubler_aux_data;
    ret->func = func;
    ret->funcdata = funcdata;

    return (NpyAuxData *)ret;
}
```

NpyAuxData_FreeFunc
The function pointer type for NpyAuxData free functions.

NpyAuxData_CloneFunc
The function pointer type for NpyAuxData clone functions. These functions should never set the Python exception on error, because they may be called from a multi-threaded context.

NPY_AUXDATA_FREE(auxdata)
A macro which calls the auxdata's free function appropriately, does nothing if auxdata is NULL.

NPY_AUXDATA_CLONE(auxdata)
A macro which calls the auxdata's clone function appropriately, returning a deep copy of the auxiliary data.

5.4.8 Array Iterators

As of Numpy 1.6, these array iterators are superceded by the new array iterator, *NpyIter*.

An array iterator is a simple way to access the elements of an N-dimensional array quickly and efficiently. Section *2* provides more description and examples of this useful approach to looping over an array.

PyObject* **PyArray_IterNew**(PyObject* *arr*)
Return an array iterator object from the array, *arr*. This is equivalent to *arr*. **flat**. The array iterator object makes it easy to loop over an N-dimensional non-contiguous array in C-style contiguous fashion.

PyObject* **PyArray_IterAllButAxis**(PyObject* *arr*, int **axis*)
Return an array iterator that will iterate over all axes but the one provided in **axis*. The returned iterator cannot be used with *PyArray_ITER_GOTO1D*. This iterator could be used to write something similar to what ufuncs do wherein the loop over the largest axis is done by a separate sub-routine. If **axis* is negative then **axis* will be set to the axis having the smallest stride and that axis will be used.

PyObject ***PyArray_BroadcastToShape** (PyObject* *arr*, npy_intp **dimensions*, int *nd*)
Return an array iterator that is broadcast to iterate as an array of the shape provided by *dimensions* and *nd*.

int **PyArrayIter_Check** (PyObject* *op*)
Evaluates true if *op* is an array iterator (or instance of a subclass of the array iterator type).

void **PyArray_ITER_RESET** (PyObject* *iterator*)
Reset an *iterator* to the beginning of the array.

void **PyArray_ITER_NEXT** (PyObject* *iterator*)
Incremement the index and the dataptr members of the *iterator* to point to the next element of the array. If the array is not (C-style) contiguous, also increment the N-dimensional coordinates array.

void ***PyArray_ITER_DATA** (PyObject* *iterator*)
A pointer to the current element of the array.

void **PyArray_ITER_GOTO** (PyObject* *iterator*, npy_intp* *destination*)
Set the *iterator* index, dataptr, and coordinates members to the location in the array indicated by the N-dimensional c-array, *destination*, which must have size at least *iterator* ->nd_m1+1.

PyArray_ITER_GOTO1D (PyObject* *iterator*, npy_intp *index*)
Set the *iterator* index and dataptr to the location in the array indicated by the integer *index* which points to an element in the C-styled flattened array.

int **PyArray_ITER_NOTDONE** (PyObject* *iterator*)
Evaluates TRUE as long as the iterator has not looped through all of the elements, otherwise it evaluates FALSE.

5.4.9 Broadcasting (multi-iterators)

PyObject* **PyArray_MultiIterNew** (int *num*, ...)
A simplified interface to broadcasting. This function takes the number of arrays to broadcast and then *num* extra (`PyObject *`) arguments. These arguments are converted to arrays and iterators are created. *PyArray_Broadcast* is then called on the resulting multi-iterator object. The resulting, broadcasted mult-iterator object is then returned. A broadcasted operation can then be performed using a single loop and using *PyArray_MultiIter_NEXT* (..)

void **PyArray_MultiIter_RESET** (PyObject* *multi*)
Reset all the iterators to the beginning in a multi-iterator object, *multi*.

void **PyArray_MultiIter_NEXT** (PyObject* *multi*)
Advance each iterator in a multi-iterator object, *multi*, to its next (broadcasted) element.

void ***PyArray_MultiIter_DATA** (PyObject* *multi*, int *i*)
Return the data-pointer of the i^{th} iterator in a multi-iterator object.

void **PyArray_MultiIter_NEXTi** (PyObject* *multi*, int *i*)
Advance the pointer of only the i^{th} iterator.

void **PyArray_MultiIter_GOTO** (PyObject* *multi*, npy_intp* *destination*)
Advance each iterator in a multi-iterator object, *multi*, to the given N -dimensional *destination* where N is the number of dimensions in the broadcasted array.

void **PyArray_MultiIter_GOTO1D** (PyObject* *multi*, npy_intp *index*)
Advance each iterator in a multi-iterator object, *multi*, to the corresponding location of the *index* into the flattened broadcasted array.

int **PyArray_MultiIter_NOTDONE** (PyObject* *multi*)
Evaluates TRUE as long as the multi-iterator has not looped through all of the elements (of the broadcasted result), otherwise it evaluates FALSE.

int **PyArray_Broadcast**(*PyArrayMultiIterObject* mit*)

This function encapsulates the broadcasting rules. The *mit* container should already contain iterators for all the arrays that need to be broadcast. On return, these iterators will be adjusted so that iteration over each simultaneously will accomplish the broadcasting. A negative number is returned if an error occurs.

int **PyArray_RemoveSmallest**(*PyArrayMultiIterObject* mit*)

This function takes a multi-iterator object that has been previously "broadcasted," finds the dimension with the smallest "sum of strides" in the broadcasted result and adapts all the iterators so as not to iterate over that dimension (by effectively making them of length-1 in that dimension). The corresponding dimension is returned unless *mit* ->nd is 0, then -1 is returned. This function is useful for constructing ufunc-like routines that broadcast their inputs correctly and then call a strided 1-d version of the routine as the inner-loop. This 1-d version is usually optimized for speed and for this reason the loop should be performed over the axis that won't require large stride jumps.

5.4.10 Neighborhood iterator

New in version 1.4.0.

Neighborhood iterators are subclasses of the iterator object, and can be used to iter over a neighborhood of a point. For example, you may want to iterate over every voxel of a 3d image, and for every such voxel, iterate over an hypercube. Neighborhood iterator automatically handle boundaries, thus making this kind of code much easier to write than manual boundaries handling, at the cost of a slight overhead.

PyObject* **PyArray_NeighborhoodIterNew**(*PyArrayIterObject* iter*, npy_intp *bounds*, int *mode*, *PyArrayObject* fill_value*)

This function creates a new neighborhood iterator from an existing iterator. The neighborhood will be computed relatively to the position currently pointed by *iter*, the bounds define the shape of the neighborhood iterator, and the mode argument the boundaries handling mode.

The *bounds* argument is expected to be a (2 * iter->ao->nd) arrays, such as the range bound[2*i]->bounds[2*i+1] defines the range where to walk for dimension i (both bounds are included in the walked coordinates). The bounds should be ordered for each dimension (bounds[2*i] <= bounds[2*i+1]).

The mode should be one of:

- NPY_NEIGHBORHOOD_ITER_ZERO_PADDING: zero padding. Outside bounds values will be 0.
- NPY_NEIGHBORHOOD_ITER_ONE_PADDING: one padding, Outside bounds values will be 1.
- NPY_NEIGHBORHOOD_ITER_CONSTANT_PADDING: constant padding. Outside bounds values will be the same as the first item in fill_value.
- NPY_NEIGHBORHOOD_ITER_MIRROR_PADDING: mirror padding. Outside bounds values will be as if the array items were mirrored. For example, for the array [1, 2, 3, 4], x[-2] will be 2, x[-2] will be 1, x[4] will be 4, x[5] will be 1, etc...
- NPY_NEIGHBORHOOD_ITER_CIRCULAR_PADDING: circular padding. Outside bounds values will be as if the array was repeated. For example, for the array [1, 2, 3, 4], x[-2] will be 3, x[-2] will be 4, x[4] will be 1, x[5] will be 2, etc...

If the mode is constant filling (NPY_NEIGHBORHOOD_ITER_CONSTANT_PADDING), fill_value should point to an array object which holds the filling value (the first item will be the filling value if the array contains more than one item). For other cases, fill_value may be NULL.

- The iterator holds a reference to iter
- Return NULL on failure (in which case the reference count of iter is not changed)
- iter itself can be a Neighborhood iterator: this can be useful for .e.g automatic boundaries handling
- the object returned by this function should be safe to use as a normal iterator

- If the position of iter is changed, any subsequent call to PyArrayNeighborhoodIter_Next is undefined behavior, and PyArrayNeighborhoodIter_Reset must be called.

```
PyArrayIterObject \*iter;
PyArrayNeighborhoodIterObject \*neigh_iter;
iter = PyArray_IterNew(x);

//For a 3x3 kernel
bounds = {-1, 1, -1, 1};
neigh_iter = (PyArrayNeighborhoodIterObject*)PyArrayNeighborhoodIter_New(
     iter, bounds, NPY_NEIGHBORHOOD_ITER_ZERO_PADDING, NULL);

for(i = 0; i < iter->size; ++i) {
     for (j = 0; j < neigh_iter->size; ++j) {
             // Walk around the item currently pointed by iter->dataptr
             PyArrayNeighborhoodIter_Next(neigh_iter);
     }

     // Move to the next point of iter
     PyArrayIter_Next(iter);
     PyArrayNeighborhoodIter_Reset(neigh_iter);
}
```

int **PyArrayNeighborhoodIter_Reset**(*PyArrayNeighborhoodIterObject* iter*)
Reset the iterator position to the first point of the neighborhood. This should be called whenever the iter argument given at PyArray_NeighborhoodIterObject is changed (see example)

int **PyArrayNeighborhoodIter_Next**(*PyArrayNeighborhoodIterObject* iter*)
After this call, iter->dataptr points to the next point of the neighborhood. Calling this function after every point of the neighborhood has been visited is undefined.

5.4.11 Array Scalars

PyObject* **PyArray_Return**(*PyArrayObject* arr*)
This function steals a reference to *arr*.

This function checks to see if *arr* is a 0-dimensional array and, if so, returns the appropriate array scalar. It should be used whenever 0-dimensional arrays could be returned to Python.

PyObject* **PyArray_Scalar**(void* *data*, *PyArray_Descr* dtype*, PyObject* *itemsize*)
Return an array scalar object of the given enumerated *typenum* and *itemsize* by **copying** from memory pointed to by *data* . If *swap* is nonzero then this function will byteswap the data if appropriate to the data-type because array scalars are always in correct machine-byte order.

PyObject* **PyArray_ToScalar**(void* *data*, *PyArrayObject* arr*)
Return an array scalar object of the type and itemsize indicated by the array object *arr* copied from the memory pointed to by *data* and swapping if the data in *arr* is not in machine byte-order.

PyObject* **PyArray_FromScalar**(PyObject* *scalar*, *PyArray_Descr* outcode*)
Return a 0-dimensional array of type determined by *outcode* from *scalar* which should be an array-scalar object. If *outcode* is NULL, then the type is determined from *scalar*.

void **PyArray_ScalarAsCtype**(PyObject* *scalar*, void* *ctypeptr*)
Return in *ctypeptr* a pointer to the actual value in an array scalar. There is no error checking so *scalar* must be an array-scalar object, and ctypeptr must have enough space to hold the correct type. For flexible-sized types, a pointer to the data is copied into the memory of *ctypeptr*, for all other types, the actual data is copied into the address pointed to by *ctypeptr*.

void **PyArray_CastScalarToCtype** (PyObject* *scalar*, void* *ctypeptr*, *PyArray_Descr** *outcode*)
Return the data (cast to the data type indicated by *outcode*) from the array-scalar, *scalar*, into the memory pointed to by *ctypeptr* (which must be large enough to handle the incoming memory).

PyObject* **PyArray_TypeObjectFromType** (int *type*)
Returns a scalar type-object from a type-number, *type* . Equivalent to *PyArray_DescrFromType* (*type*)->typeobj except for reference counting and error-checking. Returns a new reference to the typeobject on success or NULL on failure.

NPY_SCALARKIND **PyArray_ScalarKind** (int *typenum*, *PyArrayObject*** *arr*)
See the function *PyArray_MinScalarType* for an alternative mechanism introduced in NumPy 1.6.0.

Return the kind of scalar represented by *typenum* and the array in **arr* (if *arr* is not NULL). The array is assumed to be rank-0 and only used if *typenum* represents a signed integer. If *arr* is not NULL and the first element is negative then NPY_INTNEG_SCALAR is returned, otherwise NPY_INTPOS_SCALAR is returned. The possible return values are NPY_{kind}_SCALAR where {kind} can be **INTPOS**, **INTNEG**, **FLOAT**, **COMPLEX**, **BOOL**, or **OBJECT**. NPY_NOSCALAR is also an enumerated value *NPY_SCALARKIND* variables can take on.

int **PyArray_CanCoerceScalar** (char *thistype*, char *neededtype*, *NPY_SCALARKIND* *scalar*)
See the function *PyArray_ResultType* for details of NumPy type promotion, updated in NumPy 1.6.0.

Implements the rules for scalar coercion. Scalars are only silently coerced from thistype to neededtype if this function returns nonzero. If scalar is NPY_NOSCALAR, then this function is equivalent to *PyArray_CanCastSafely*. The rule is that scalars of the same KIND can be coerced into arrays of the same KIND. This rule means that high-precision scalars will never cause low-precision arrays of the same KIND to be upcast.

5.4.12 Data-type descriptors

Warning: Data-type objects must be reference counted so be aware of the action on the data-type reference of different C-API calls. The standard rule is that when a data-type object is returned it is a new reference. Functions that take *PyArray_Descr * * objects and return arrays steal references to the data-type their inputs unless otherwise noted. Therefore, you must own a reference to any data-type object used as input to such a function.

int **PyArray_DescrCheck** (PyObject* *obj*)
Evaluates as true if *obj* is a data-type object (*PyArray_Descr * *).

*PyArray_Descr** **PyArray_DescrNew** (*PyArray_Descr** *obj*)
Return a new data-type object copied from *obj* (the fields reference is just updated so that the new object points to the same fields dictionary if any).

*PyArray_Descr** **PyArray_DescrNewFromType** (int *typenum*)
Create a new data-type object from the built-in (or user-registered) data-type indicated by *typenum*. All builtin types should not have any of their fields changed. This creates a new copy of the *PyArray_Descr* structure so that you can fill it in as appropriate. This function is especially needed for flexible data-types which need to have a new elsize member in order to be meaningful in array construction.

*PyArray_Descr** **PyArray_DescrNewByteorder** (*PyArray_Descr** *obj*, char *newendian*)
Create a new data-type object with the byteorder set according to *newendian*. All referenced data-type objects (in subdescr and fields members of the data-type object) are also changed (recursively). If a byteorder of NPY_IGNORE is encountered it is left alone. If newendian is NPY_SWAP, then all byte-orders are swapped. Other valid newendian values are NPY_NATIVE, NPY_LITTLE, and NPY_BIG which all cause the returned data-typed descriptor (and all it's referenced data-type descriptors) to have the corresponding byte- order.

*PyArray_Descr** **PyArray_DescrFromObject** (PyObject* *op*, *PyArray_Descr** *mintype*)
Determine an appropriate data-type object from the object *op* (which should be a "nested" sequence object) and the minimum data-type descriptor mintype (which can be `NULL`). Similar in behavior to array(*op*).dtype. Don't confuse this function with *PyArray_DescrConverter*. This function essentially looks at all the objects in the (nested) sequence and determines the data-type from the elements it finds.

*PyArray_Descr** **PyArray_DescrFromScalar** (PyObject* *scalar*)
Return a data-type object from an array-scalar object. No checking is done to be sure that *scalar* is an array scalar. If no suitable data-type can be determined, then a data-type of *NPY_OBJECT* is returned by default.

*PyArray_Descr** **PyArray_DescrFromType** (int *typenum*)
Returns a data-type object corresponding to *typenum*. The *typenum* can be one of the enumerated types, a character code for one of the enumerated types, or a user-defined type.

int **PyArray_DescrConverter** (PyObject* *obj*, *PyArray_Descr*** *dtype*)
Convert any compatible Python object, *obj*, to a data-type object in *dtype*. A large number of Python objects can be converted to data-type objects. See *Data type objects (dtype)* for a complete description. This version of the converter converts None objects to a *NPY_DEFAULT_TYPE* data-type object. This function can be used with the "O&" character code in `PyArg_ParseTuple` processing.

int **PyArray_DescrConverter2** (PyObject* *obj*, *PyArray_Descr*** *dtype*)
Convert any compatible Python object, *obj*, to a data-type object in *dtype*. This version of the converter converts None objects so that the returned data-type is `NULL`. This function can also be used with the "O&" character in PyArg_ParseTuple processing.

int **Pyarray_DescrAlignConverter** (PyObject* *obj*, *PyArray_Descr*** *dtype*)
Like *PyArray_DescrConverter* except it aligns C-struct-like objects on word-boundaries as the compiler would.

int **Pyarray_DescrAlignConverter2** (PyObject* *obj*, *PyArray_Descr*** *dtype*)
Like *PyArray_DescrConverter2* except it aligns C-struct-like objects on word-boundaries as the compiler would.

PyObject ***PyArray_FieldNames** (PyObject* *dict*)
Take the fields dictionary, *dict*, such as the one attached to a data-type object and construct an ordered-list of field names such as is stored in the names field of the *PyArray_Descr* object.

5.4.13 Conversion Utilities

For use with `PyArg_ParseTuple`

All of these functions can be used in `PyArg_ParseTuple` (...) with the "O&" format specifier to automatically convert any Python object to the required C-object. All of these functions return *NPY_SUCCEED* if successful and *NPY_FAIL* if not. The first argument to all of these function is a Python object. The second argument is the **address** of the C-type to convert the Python object to.

> **Warning:** Be sure to understand what steps you should take to manage the memory when using these conversion functions. These functions can require freeing memory, and/or altering the reference counts of specific objects based on your use.

int **PyArray_Converter** (PyObject* *obj*, PyObject** *address*)
Convert any Python object to a *PyArrayObject*. If *PyArray_Check* (*obj*) is TRUE then its reference count is incremented and a reference placed in *address*. If *obj* is not an array, then convert it to an array using *PyArray_FromAny* . No matter what is returned, you must DECREF the object returned by this routine in *address* when you are done with it.

int **PyArray_OutputConverter**(PyObject* *obj*, *PyArrayObject*** *address*)
This is a default converter for output arrays given to functions. If *obj* is `Py_None` or `NULL`, then **address* will be `NULL` but the call will succeed. If *PyArray_Check* (*obj*) is TRUE then it is returned in **address* without incrementing its reference count.

int **PyArray_IntpConverter**(PyObject* *obj*, *PyArray_Dims** *seq*)
Convert any Python sequence, *obj*, smaller than *NPY_MAXDIMS* to a C-array of `npy_intp`. The Python object could also be a single number. The *seq* variable is a pointer to a structure with members ptr and len. On successful return, *seq* ->ptr contains a pointer to memory that must be freed to avoid a memory leak. The restriction on memory size allows this converter to be conveniently used for sequences intended to be interpreted as array shapes.

int **PyArray_BufferConverter**(PyObject* *obj*, *PyArray_Chunk** *buf*)
Convert any Python object, *obj*, with a (single-segment) buffer interface to a variable with members that detail the object's use of its chunk of memory. The *buf* variable is a pointer to a structure with base, ptr, len, and flags members. The *PyArray_Chunk* structure is binary compatible with the Python's buffer object (through its len member on 32-bit platforms and its ptr member on 64-bit platforms or in Python 2.5). On return, the base member is set to *obj* (or its base if *obj* is already a buffer object pointing to another object). If you need to hold on to the memory be sure to INCREF the base member. The chunk of memory is pointed to by *buf* ->ptr member and has length *buf* ->len. The flags member of *buf* is `NPY_BEHAVED_RO` with the *NPY_ARRAY_WRITEABLE* flag set if *obj* has a writeable buffer interface.

int **PyArray_AxisConverter**(PyObject * *obj*, int* *axis*)
Convert a Python object, *obj*, representing an axis argument to the proper value for passing to the functions that take an integer axis. Specifically, if *obj* is None, *axis* is set to *NPY_MAXDIMS* which is interpreted correctly by the C-API functions that take axis arguments.

int **PyArray_BoolConverter**(PyObject* *obj*, Bool* *value*)
Convert any Python object, *obj*, to *NPY_TRUE* or *NPY_FALSE*, and place the result in *value*.

int **PyArray_ByteorderConverter**(PyObject* *obj*, char* *endian*)
Convert Python strings into the corresponding byte-order character: '>', '<', 's', '=', or 'l'.

int **PyArray_SortkindConverter**(PyObject* *obj*, *NPY_SORTKIND** *sort*)
Convert Python strings into one of `NPY_QUICKSORT` (starts with 'q' or 'Q') , `NPY_HEAPSORT` (starts with 'h' or 'H'), or `NPY_MERGESORT` (starts with 'm' or 'M').

int **PyArray_SearchsideConverter**(PyObject* *obj*, NPY_SEARCHSIDE* *side*)
Convert Python strings into one of `NPY_SEARCHLEFT` (starts with 'l' or 'L'), or `NPY_SEARCHRIGHT` (starts with 'r' or 'R').

int **PyArray_OrderConverter**(PyObject* *obj*, *NPY_ORDER** *order*)
Convert the Python strings 'C', 'F', 'A', and 'K' into the *NPY_ORDER* enumeration *NPY_CORDER*, *NPY_FORTRANORDER*, *NPY_ANYORDER*, and *NPY_KEEPORDER*.

int **PyArray_CastingConverter**(PyObject* *obj*, *NPY_CASTING** *casting*)
Convert the Python strings 'no', 'equiv', 'safe', 'same_kind', and 'unsafe' into the *NPY_CASTING* enumeration *NPY_NO_CASTING*, *NPY_EQUIV_CASTING*, *NPY_SAFE_CASTING*, *NPY_SAME_KIND_CASTING*, and *NPY_UNSAFE_CASTING*.

int **PyArray_ClipmodeConverter**(PyObject* *object*, *NPY_CLIPMODE** *val*)
Convert the Python strings 'clip', 'wrap', and 'raise' into the *NPY_CLIPMODE* enumeration *NPY_CLIP*, *NPY_WRAP*, and *NPY_RAISE*.

int **PyArray_ConvertClipmodeSequence**(PyObject* *object*, *NPY_CLIPMODE** *modes*, int *n*)
Converts either a sequence of clipmodes or a single clipmode into a C array of *NPY_CLIPMODE* values. The number of clipmodes *n* must be known before calling this function. This function is provided to help functions allow a different clipmode for each dimension.

Other conversions

int **PyArray_PyIntAsInt** (PyObject* *op*)
Convert all kinds of Python objects (including arrays and array scalars) to a standard integer. On error, -1 is returned and an exception set. You may find useful the macro:

```
#define error_converting(x)  (((x) == -1) && PyErr_Occurred()
```

npy_intp **PyArray_PyIntAsIntp** (PyObject* *op*)
Convert all kinds of Python objects (including arrays and array scalars) to a (platform-pointer-sized) integer. On error, -1 is returned and an exception set.

int **PyArray_IntpFromSequence** (PyObject* *seq*, npy_intp* *vals*, int *maxvals*)
Convert any Python sequence (or single Python number) passed in as *seq* to (up to) *maxvals* pointer-sized integers and place them in the *vals* array. The sequence can be smaller then *maxvals* as the number of converted objects is returned.

int **PyArray_TypestrConvert** (int *itemsize*, int *gentype*)
Convert typestring characters (with *itemsize*) to basic enumerated data types. The typestring character corresponding to signed and unsigned integers, floating point numbers, and complex-floating point numbers are recognized and converted. Other values of gentype are returned. This function can be used to convert, for example, the string 'f4' to *NPY_FLOAT32*.

5.4.14 Miscellaneous

Importing the API

In order to make use of the C-API from another extension module, the `import_array` () command must be used. If the extension module is self-contained in a single .c file, then that is all that needs to be done. If, however, the extension module involves multiple files where the C-API is needed then some additional steps must be taken.

void **import_array** (void)
This function must be called in the initialization section of a module that will make use of the C-API. It imports the module where the function-pointer table is stored and points the correct variable to it.

PY_ARRAY_UNIQUE_SYMBOL

NO_IMPORT_ARRAY
Using these #defines you can use the C-API in multiple files for a single extension module. In each file you must define *PY_ARRAY_UNIQUE_SYMBOL* to some name that will hold the C-API (*e.g.* myextension_ARRAY_API). This must be done **before** including the numpy/arrayobject.h file. In the module initialization routine you call `import_array` (). In addition, in the files that do not have the module initialization sub_routine define *NO_IMPORT_ARRAY* prior to including numpy/arrayobject.h.

Suppose I have two files coolmodule.c and coolhelper.c which need to be compiled and linked into a single extension module. Suppose coolmodule.c contains the required initcool module initialization function (with the import_array() function called). Then, coolmodule.c would have at the top:

```
#define PY_ARRAY_UNIQUE_SYMBOL cool_ARRAY_API
#include numpy/arrayobject.h
```

On the other hand, coolhelper.c would contain at the top:

```
#define NO_IMPORT_ARRAY
#define PY_ARRAY_UNIQUE_SYMBOL cool_ARRAY_API
#include numpy/arrayobject.h
```

You can also put the common two last lines into an extension-local header file as long as you make sure that NO_IMPORT_ARRAY is #defined before #including that file.

Checking the API Version

Because python extensions are not used in the same way as usual libraries on most platforms, some errors cannot be automatically detected at build time or even runtime. For example, if you build an extension using a function available only for numpy >= 1.3.0, and you import the extension later with numpy 1.2, you will not get an import error (but almost certainly a segmentation fault when calling the function). That's why several functions are provided to check for numpy versions. The macros *NPY_VERSION* and NPY_FEATURE_VERSION corresponds to the numpy version used to build the extension, whereas the versions returned by the functions PyArray_GetNDArrayCVersion and PyArray_GetNDArrayCFeatureVersion corresponds to the runtime numpy's version.

The rules for ABI and API compatibilities can be summarized as follows:

- Whenever *NPY_VERSION* != PyArray_GetNDArrayCVersion, the extension has to be recompiled (ABI incompatibility).
- *NPY_VERSION* == PyArray_GetNDArrayCVersion and NPY_FEATURE_VERSION <= PyArray_GetNDArrayCFeatureVersion means backward compatible changes.

ABI incompatibility is automatically detected in every numpy's version. API incompatibility detection was added in numpy 1.4.0. If you want to supported many different numpy versions with one extension binary, you have to build your extension with the lowest NPY_FEATURE_VERSION as possible.

unsigned int **PyArray_GetNDArrayCVersion** (void)
This just returns the value *NPY_VERSION*. *NPY_VERSION* changes whenever a backward incompatible change at the ABI level. Because it is in the C-API, however, comparing the output of this function from the value defined in the current header gives a way to test if the C-API has changed thus requiring a re-compilation of extension modules that use the C-API. This is automatically checked in the function import_array.

unsigned int **PyArray_GetNDArrayCFeatureVersion** (void)
New in version 1.4.0.

This just returns the value NPY_FEATURE_VERSION. NPY_FEATURE_VERSION changes whenever the API changes (e.g. a function is added). A changed value does not always require a recompile.

Internal Flexibility

int **PyArray_SetNumericOps** (PyObject* *dict*)
NumPy stores an internal table of Python callable objects that are used to implement arithmetic operations for arrays as well as certain array calculation methods. This function allows the user to replace any or all of these Python objects with their own versions. The keys of the dictionary, *dict*, are the named functions to replace and the paired value is the Python callable object to use. Care should be taken that the function used to replace an internal array operation does not itself call back to that internal array operation (unless you have designed the function to handle that), or an unchecked infinite recursion can result (possibly causing program crash). The key names that represent operations that can be replaced are:

add, **subtract**, **multiply**, **divide**, **remainder**, **power**, **square**, **reciprocal**, **ones_like**, **sqrt**, **negative**, **absolute**, **invert**, **left_shift**, **right_shift**, **bitwise_and**, **bitwise_xor**, **bitwise_or**, **less**, **less_equal**, **equal**, **not_equal**, **greater**, **greater_equal**, **floor_divide**, **true_divide**, **logical_or**, **logical_and**, **floor**, **ceil**, **maximum**, **minimum**, **rint**.

These functions are included here because they are used at least once in the array object's methods. The function returns -1 (without setting a Python Error) if one of the objects being assigned is not callable.

PyObject* **PyArray_GetNumericOps** (void)
Return a Python dictionary containing the callable Python objects stored in the the internal arithmetic operation table. The keys of this dictionary are given in the explanation for *PyArray_SetNumericOps*.

void **PyArray_SetStringFunction** (PyObject* *op*, int *repr*)
This function allows you to alter the tp_str and tp_repr methods of the array object to any Python function. Thus you can alter what happens for all arrays when str(arr) or repr(arr) is called from Python. The function to be called is passed in as *op*. If *repr* is non-zero, then this function will be called in response to repr(arr), otherwise the function will be called in response to str(arr). No check on whether or not *op* is callable is performed. The callable passed in to *op* should expect an array argument and should return a string to be printed.

Memory management

char* **PyDataMem_NEW** (size_t *nbytes*)

PyDataMem_FREE (char* *ptr*)

char* **PyDataMem_RENEW** (void * *ptr*, size_t *newbytes*)
Macros to allocate, free, and reallocate memory. These macros are used internally to create arrays.

npy_intp* **PyDimMem_NEW** (nd)

PyDimMem_FREE (npy_intp* *ptr*)

npy_intp* **PyDimMem_RENEW** (npy_intp* *ptr*, npy_intp *newnd*)
Macros to allocate, free, and reallocate dimension and strides memory.

PyArray_malloc (nbytes)

PyArray_free (ptr)

PyArray_realloc (ptr, nbytes)
These macros use different memory allocators, depending on the constant `NPY_USE_PYMEM`. The system malloc is used when `NPY_USE_PYMEM` is 0, if `NPY_USE_PYMEM` is 1, then the Python memory allocator is used.

Threading support

These macros are only meaningful if `NPY_ALLOW_THREADS` evaluates True during compilation of the extension module. Otherwise, these macros are equivalent to whitespace. Python uses a single Global Interpreter Lock (GIL) for each Python process so that only a single thread may execute at a time (even on multi-cpu machines). When calling out to a compiled function that may take time to compute (and does not have side-effects for other threads like updated global variables), the GIL should be released so that other Python threads can run while the time-consuming calculations are performed. This can be accomplished using two groups of macros. Typically, if one macro in a group is used in a code block, all of them must be used in the same code block. Currently, `NPY_ALLOW_THREADS` is defined to the python-defined `WITH_THREADS` constant unless the environment variable `NPY_NOSMP` is set in which case `NPY_ALLOW_THREADS` is defined to be 0.

Group 1

This group is used to call code that may take some time but does not use any Python C-API calls. Thus, the GIL should be released during its calculation.

NPY_BEGIN_ALLOW_THREADS
Equivalent to `Py_BEGIN_ALLOW_THREADS` except it uses `NPY_ALLOW_THREADS` to determine if the macro if replaced with white-space or not.

NPY_END_ALLOW_THREADS
Equivalent to `Py_END_ALLOW_THREADS` except it uses `NPY_ALLOW_THREADS` to determine if the macro if replaced with white-space or not.

NPY_BEGIN_THREADS_DEF
Place in the variable declaration area. This macro sets up the variable needed for storing the Python state.

NPY_BEGIN_THREADS
Place right before code that does not need the Python interpreter (no Python C-API calls). This macro saves the Python state and releases the GIL.

NPY_END_THREADS
Place right after code that does not need the Python interpreter. This macro acquires the GIL and restores the Python state from the saved variable.

NPY_BEGIN_THREADS_DESCR(*PyArray_Descr *dtype*)
Useful to release the GIL only if *dtype* does not contain arbitrary Python objects which may need the Python interpreter during execution of the loop. Equivalent to

NPY_END_THREADS_DESCR(*PyArray_Descr *dtype*)
Useful to regain the GIL in situations where it was released using the BEGIN form of this macro.

NPY_BEGIN_THREADS_THRESHOLDED(int *loop_size*)
Useful to release the GIL only if *loop_size* exceeds a minimum threshold, currently set to 500. Should be matched with a .. c:macro::*NPY_END_THREADS* to regain the GIL.

Group 2

This group is used to re-acquire the Python GIL after it has been released. For example, suppose the GIL has been released (using the previous calls), and then some path in the code (perhaps in a different subroutine) requires use of the Python C-API, then these macros are useful to acquire the GIL. These macros accomplish essentially a reverse of the previous three (acquire the LOCK saving what state it had) and then re-release it with the saved state.

NPY_ALLOW_C_API_DEF
Place in the variable declaration area to set up the necessary variable.

NPY_ALLOW_C_API
Place before code that needs to call the Python C-API (when it is known that the GIL has already been released).

NPY_DISABLE_C_API
Place after code that needs to call the Python C-API (to re-release the GIL).

Tip: Never use semicolons after the threading support macros.

Priority

NPY_PRIORITY
Default priority for arrays.

NPY_SUBTYPE_PRIORITY
Default subtype priority.

NPY_SCALAR_PRIORITY
Default scalar priority (very small)

double **PyArray_GetPriority**(PyObject* *obj*, double *def*)
Return the `__array_priority__` attribute (converted to a double) of *obj* or *def* if no attribute of that name exists. Fast returns that avoid the attribute lookup are provided for objects of type `PyArray_Type`.

Default buffers

NPY_BUFSIZE
Default size of the user-settable internal buffers.

NPY_MIN_BUFSIZE
Smallest size of user-settable internal buffers.

NPY_MAX_BUFSIZE
Largest size allowed for the user-settable buffers.

Other constants

NPY_NUM_FLOATTYPE
The number of floating-point types

NPY_MAXDIMS
The maximum number of dimensions allowed in arrays.

NPY_VERSION
The current version of the ndarray object (check to see if this variable is defined to guarantee the numpy/arrayobject.h header is being used).

NPY_FALSE
Defined as 0 for use with Bool.

NPY_TRUE
Defined as 1 for use with Bool.

NPY_FAIL
The return value of failed converter functions which are called using the "O&" syntax in `PyArg_ParseTuple`-like functions.

NPY_SUCCEED
The return value of successful converter functions which are called using the "O&" syntax in `PyArg_ParseTuple`-like functions.

Miscellaneous Macros

PyArray_SAMESHAPE(a1, a2)
Evaluates as True if arrays *a1* and *a2* have the same shape.

PyArray_MAX(a, b)
Returns the maximum of *a* and *b*. If (*a*) or (*b*) are expressions they are evaluated twice.

PyArray_MIN(a, b)
Returns the minimum of *a* and *b*. If (*a*) or (*b*) are expressions they are evaluated twice.

PyArray_CLT(a, b)

PyArray_CGT(a, b)

PyArray_CLE(a, b)

PyArray_CGE(a, b)

PyArray_CEQ(a, b)

PyArray_CNE(a, b)
Implements the complex comparisons between two complex numbers (structures with a real and imag member) using NumPy's definition of the ordering which is lexicographic: comparing the real parts first and then the complex parts if the real parts are equal.

PyArray_REFCOUNT(PyObject* *op*)
Returns the reference count of any Python object.

PyArray_XDECREF_ERR(PyObject **obj*)
DECREF's an array object which may have the *NPY_ARRAY_UPDATEIFCOPY* flag set without causing the contents to be copied back into the original array. Resets the *NPY_ARRAY_WRITEABLE* flag on the base object. This is useful for recovering from an error condition when *NPY_ARRAY_UPDATEIFCOPY* is used.

Enumerated Types

NPY_SORTKIND
A special variable-type which can take on the values `NPY_{KIND}` where `{KIND}` is

QUICKSORT, **HEAPSORT**, **MERGESORT**

NPY_NSORTS
Defined to be the number of sorts.

NPY_SCALARKIND
A special variable type indicating the number of "kinds" of scalars distinguished in determining scalar-coercion rules. This variable can take on the values `NPY_{KIND}` where `{KIND}` can be

NOSCALAR, **BOOL_SCALAR**, **INTPOS_SCALAR**, **INTNEG_SCALAR**, **FLOAT_SCALAR**, **COMPLEX_SCALAR**, **OBJECT_SCALAR**

NPY_NSCALARKINDS
Defined to be the number of scalar kinds (not including `NPY_NOSCALAR`).

NPY_ORDER
An enumeration type indicating the element order that an array should be interpreted in. When a brand new array is created, generally only **NPY_CORDER** and **NPY_FORTRANORDER** are used, whereas when one or more inputs are provided, the order can be based on them.

NPY_ANYORDER
Fortran order if all the inputs are Fortran, C otherwise.

NPY_CORDER
C order.

NPY_FORTRANORDER
Fortran order.

NPY_KEEPORDER
An order as close to the order of the inputs as possible, even if the input is in neither C nor Fortran order.

NPY_CLIPMODE
A variable type indicating the kind of clipping that should be applied in certain functions.

NPY_RAISE
The default for most operations, raises an exception if an index is out of bounds.

NPY_CLIP
Clips an index to the valid range if it is out of bounds.

NPY_WRAP
Wraps an index to the valid range if it is out of bounds.

NPY_CASTING
New in version 1.6.

An enumeration type indicating how permissive data conversions should be. This is used by the iterator added in NumPy 1.6, and is intended to be used more broadly in a future version.

NPY_NO_CASTING
Only allow identical types.

NPY_EQUIV_CASTING
Allow identical and casts involving byte swapping.

NPY_SAFE_CASTING
Only allow casts which will not cause values to be rounded, truncated, or otherwise changed.

NPY_SAME_KIND_CASTING
Allow any safe casts, and casts between types of the same kind. For example, float64 -> float32 is permitted with this rule.

NPY_UNSAFE_CASTING
Allow any cast, no matter what kind of data loss may occur.

5.5 Array Iterator API

New in version 1.6.

5.5.1 Array Iterator

The array iterator encapsulates many of the key features in ufuncs, allowing user code to support features like output parameters, preservation of memory layouts, and buffering of data with the wrong alignment or type, without requiring difficult coding.

This page documents the API for the iterator. The iterator is named `NpyIter` and functions are named `NpyIter_*`.

There is an *introductory guide to array iteration* which may be of interest for those using this C API. In many instances, testing out ideas by creating the iterator in Python is a good idea before writing the C iteration code.

5.5.2 Simple Iteration Example

The best way to become familiar with the iterator is to look at its usage within the NumPy codebase itself. For example, here is a slightly tweaked version of the code for *`PyArray_CountNonzero`*, which counts the number of non-zero elements in an array.

```
npy_intp PyArray_CountNonzero(PyArrayObject* self)
{
    /* Nonzero boolean function */
    PyArray_NonzeroFunc* nonzero = PyArray_DESCR(self)->f->nonzero;

    NpyIter* iter;
    NpyIter_IterNextFunc *iternext;
    char** dataptr;
    npy_intp nonzero_count;
    npy_intp* strideptr,* innersizeptr;

    /* Handle zero-sized arrays specially */
    if (PyArray_SIZE(self) == 0) {
        return 0;
    }

    /*
     * Create and use an iterator to count the nonzeros.
     *   flag NPY_ITER_READONLY
     *     - The array is never written to.
     *   flag NPY_ITER_EXTERNAL_LOOP
     *     - Inner loop is done outside the iterator for efficiency.
     *   flag NPY_ITER_NPY_ITER_REFS_OK
     *     - Reference types are acceptable.
     *   order NPY_KEEPORDER
     *     - Visit elements in memory order, regardless of strides.
     *       This is good for performance when the specific order
     *       elements are visited is unimportant.
     *   casting NPY_NO_CASTING
     *     - No casting is required for this operation.
     */
    iter = NpyIter_New(self, NPY_ITER_READONLY|
                             NPY_ITER_EXTERNAL_LOOP|
                             NPY_ITER_REFS_OK,
                        NPY_KEEPORDER, NPY_NO_CASTING,
                        NULL);
    if (iter == NULL) {
        return -1;
    }

    /*
     * The iternext function gets stored in a local variable
     * so it can be called repeatedly in an efficient manner.
     */
    iternext = NpyIter_GetIterNext(iter, NULL);
    if (iternext == NULL) {
        NpyIter_Deallocate(iter);
        return -1;
    }
    /* The location of the data pointer which the iterator may update */
    dataptr = NpyIter_GetDataPtrArray(iter);
    /* The location of the stride which the iterator may update */
    strideptr = NpyIter_GetInnerStrideArray(iter);
    /* The location of the inner loop size which the iterator may update */
    innersizeptr = NpyIter_GetInnerLoopSizePtr(iter);

    nonzero_count = 0;
    do {
```

```
        /* Get the inner loop data/stride/count values */
        char* data = *dataptr;
        npy_intp stride = *strideptr;
        npy_intp count = *innersizeptr;

        /* This is a typical inner loop for NPY_ITER_EXTERNAL_LOOP */
        while (count--) {
            if (nonzero(data, self)) {
                ++nonzero_count;
            }
            data += stride;
        }

        /* Increment the iterator to the next inner loop */
    } while(iternext(iter));

    NpyIter_Deallocate(iter);

    return nonzero_count;
}
```

5.5.3 Simple Multi-Iteration Example

Here is a simple copy function using the iterator. The `order` parameter is used to control the memory layout of the allocated result, typically *NPY_KEEPORDER* is desired.

```
PyObject *CopyArray(PyObject *arr, NPY_ORDER order)
{
    NpyIter *iter;
    NpyIter_IterNextFunc *iternext;
    PyObject *op[2], *ret;
    npy_uint32 flags;
    npy_uint32 op_flags[2];
    npy_intp itemsize, *innersizeptr, innerstride;
    char **dataptrarray;

    /*
     * No inner iteration - inner loop is handled by CopyArray code
     */
    flags = NPY_ITER_EXTERNAL_LOOP;
    /*
     * Tell the constructor to automatically allocate the output.
     * The data type of the output will match that of the input.
     */
    op[0] = arr;
    op[1] = NULL;
    op_flags[0] = NPY_ITER_READONLY;
    op_flags[1] = NPY_ITER_WRITEONLY | NPY_ITER_ALLOCATE;

    /* Construct the iterator */
    iter = NpyIter_MultiNew(2, op, flags, order, NPY_NO_CASTING,
                            op_flags, NULL);
    if (iter == NULL) {
        return NULL;
    }

    /*
```

```
     * Make a copy of the iternext function pointer and
     * a few other variables the inner loop needs.
     */
    iternext = NpyIter_GetIterNext(iter, NULL);
    innerstride = NpyIter_GetInnerStrideArray(iter)[0];
    itemsize = NpyIter_GetDescrArray(iter)[0]->elsize;
    /*
     * The inner loop size and data pointers may change during the
     * loop, so just cache the addresses.
     */
    innersizeptr = NpyIter_GetInnerLoopSizePtr(iter);
    dataptrarray = NpyIter_GetDataPtrArray(iter);

    /*
     * Note that because the iterator allocated the output,
     * it matches the iteration order and is packed tightly,
     * so we don't need to check it like the input.
     */
    if (innerstride == itemsize) {
        do {
            memcpy(dataptrarray[1], dataptrarray[0],
                                    itemsize * (*innersizeptr));
        } while (iternext(iter));
    } else {
        /* For efficiency, should specialize this based on item size... */
        npy_intp i;
        do {
            npy_intp size = *innersizeptr;
            char *src = dataptrarray[0], *dst = dataptrarray[1];
            for(i = 0; i < size; i++, src += innerstride, dst += itemsize) {
                memcpy(dst, src, itemsize);
            }
        } while (iternext(iter));
    }

    /* Get the result from the iterator object array */
    ret = NpyIter_GetOperandArray(iter)[1];
    Py_INCREF(ret);

    if (NpyIter_Deallocate(iter) != NPY_SUCCEED) {
        Py_DECREF(ret);
        return NULL;
    }

    return ret;
}
```

5.5.4 Iterator Data Types

The iterator layout is an internal detail, and user code only sees an incomplete struct.

NpyIter
: This is an opaque pointer type for the iterator. Access to its contents can only be done through the iterator API.

NpyIter_Type
: This is the type which exposes the iterator to Python. Currently, no API is exposed which provides access to the values of a Python-created iterator. If an iterator is created in Python, it must be used in Python and vice

versa. Such an API will likely be created in a future version.

NpyIter_IterNextFunc
This is a function pointer for the iteration loop, returned by *NpyIter_GetIterNext*.

NpyIter_GetMultiIndexFunc
This is a function pointer for getting the current iterator multi-index, returned by *NpyIter_GetGetMultiIndex*.

5.5.5 Construction and Destruction

*NpyIter** **NpyIter_New**(*PyArrayObject** *op*, npy_uint32 *flags*, *NPY_ORDER* *order*, *NPY_CASTING* *casting*, *PyArray_Descr** *dtype*)
Creates an iterator for the given numpy array object `op`.

Flags that may be passed in `flags` are any combination of the global and per-operand flags documented in *NpyIter_MultiNew*, except for *NPY_ITER_ALLOCATE*.

Any of the *NPY_ORDER* enum values may be passed to `order`. For efficient iteration, *NPY_KEEPORDER* is the best option, and the other orders enforce the particular iteration pattern.

Any of the *NPY_CASTING* enum values may be passed to `casting`. The values include *NPY_NO_CASTING*, *NPY_EQUIV_CASTING*, *NPY_SAFE_CASTING*, *NPY_SAME_KIND_CASTING*, and *NPY_UNSAFE_CASTING*. To allow the casts to occur, copying or buffering must also be enabled.

If `dtype` isn't `NULL`, then it requires that data type. If copying is allowed, it will make a temporary copy if the data is castable. If *NPY_ITER_UPDATEIFCOPY* is enabled, it will also copy the data back with another cast upon iterator destruction.

Returns NULL if there is an error, otherwise returns the allocated iterator.

To make an iterator similar to the old iterator, this should work.

```
iter = NpyIter_New(op, NPY_ITER_READWRITE,
                       NPY_CORDER, NPY_NO_CASTING, NULL);
```

If you want to edit an array with aligned `double` code, but the order doesn't matter, you would use this.

```
dtype = PyArray_DescrFromType(NPY_DOUBLE);
iter = NpyIter_New(op, NPY_ITER_READWRITE|
                       NPY_ITER_BUFFERED|
                       NPY_ITER_NBO|
                       NPY_ITER_ALIGNED,
                       NPY_KEEPORDER,
                       NPY_SAME_KIND_CASTING,
                       dtype);
Py_DECREF(dtype);
```

*NpyIter** **NpyIter_MultiNew**(npy_intp *nop*, *PyArrayObject*** *op*, npy_uint32 *flags*, *NPY_ORDER* *order*, *NPY_CASTING* *casting*, npy_uint32* *op_flags*, *PyArray_Descr*** *op_dtypes*)
Creates an iterator for broadcasting the `nop` array objects provided in `op`, using regular NumPy broadcasting rules.

Any of the *NPY_ORDER* enum values may be passed to `order`. For efficient iteration, *NPY_KEEPORDER* is the best option, and the other orders enforce the particular iteration pattern. When using *NPY_KEEPORDER*, if you also want to ensure that the iteration is not reversed along an axis, you should pass the flag *NPY_ITER_DONT_NEGATE_STRIDES*.

Any of the *NPY_CASTING* enum values may be passed to `casting`. The values include *NPY_NO_CASTING*, *NPY_EQUIV_CASTING*, *NPY_SAFE_CASTING*, *NPY_SAME_KIND_CASTING*, and *NPY_UNSAFE_CASTING*. To allow the casts to occur, copying or buffering must also be enabled.

If `op_dtypes` isn't `NULL`, it specifies a data type or `NULL` for each `op[i]`.

Returns NULL if there is an error, otherwise returns the allocated iterator.

Flags that may be passed in `flags`, applying to the whole iterator, are:

NPY_ITER_C_INDEX
Causes the iterator to track a raveled flat index matching C order. This option cannot be used with *NPY_ITER_F_INDEX*.

NPY_ITER_F_INDEX
Causes the iterator to track a raveled flat index matching Fortran order. This option cannot be used with *NPY_ITER_C_INDEX*.

NPY_ITER_MULTI_INDEX
Causes the iterator to track a multi-index. This prevents the iterator from coalescing axes to produce bigger inner loops. If the loop is also not buffered and no index is being tracked (*NpyIter_RemoveAxis* can be called), then the iterator size can be `-1` to indicate that the iterator is too large. This can happen due to complex broadcasting and will result in errors being created when the setting the iterator range, removing the multi index, or getting the next function. However, it is possible to remove axes again and use the iterator normally if the size is small enough after removal.

NPY_ITER_EXTERNAL_LOOP
Causes the iterator to skip iteration of the innermost loop, requiring the user of the iterator to handle it.

This flag is incompatible with *NPY_ITER_C_INDEX*, *NPY_ITER_F_INDEX*, and *NPY_ITER_MULTI_INDEX*.

NPY_ITER_DONT_NEGATE_STRIDES
This only affects the iterator when *NPY_KEEPORDER* is specified for the order parameter. By default with *NPY_KEEPORDER*, the iterator reverses axes which have negative strides, so that memory is traversed in a forward direction. This disables this step. Use this flag if you want to use the underlying memory-ordering of the axes, but don't want an axis reversed. This is the behavior of `numpy.ravel(a, order='K')`, for instance.

NPY_ITER_COMMON_DTYPE
Causes the iterator to convert all the operands to a common data type, calculated based on the ufunc type promotion rules. Copying or buffering must be enabled.

If the common data type is known ahead of time, don't use this flag. Instead, set the requested dtype for all the operands.

NPY_ITER_REFS_OK
Indicates that arrays with reference types (object arrays or structured arrays containing an object type) may be accepted and used in the iterator. If this flag is enabled, the caller must be sure to check whether `NpyIter_IterationNeedsAPI(iter)` is true, in which case it may not release the GIL during iteration.

NPY_ITER_ZEROSIZE_OK
Indicates that arrays with a size of zero should be permitted. Since the typical iteration loop does not naturally work with zero-sized arrays, you must check that the IterSize is larger than zero before entering the iteration loop. Currently only the operands are checked, not a forced shape.

NPY_ITER_REDUCE_OK
Permits writeable operands with a dimension with zero stride and size greater than one. Note that such operands must be read/write.

When buffering is enabled, this also switches to a special buffering mode which reduces the loop length as necessary to not trample on values being reduced.

Note that if you want to do a reduction on an automatically allocated output, you must use *NpyIter_GetOperandArray* to get its reference, then set every value to the reduction unit before doing the iteration loop. In the case of a buffered reduction, this means you must also specify the flag *NPY_ITER_DELAY_BUFALLOC*, then reset the iterator after initializing the allocated operand to prepare the buffers.

NPY_ITER_RANGED
Enables support for iteration of sub-ranges of the full `iterindex` range `[0, NpyIter_IterSize(iter))`. Use the function *NpyIter_ResetToIterIndexRange* to specify a range for iteration.

This flag can only be used with *NPY_ITER_EXTERNAL_LOOP* when *NPY_ITER_BUFFERED* is enabled. This is because without buffering, the inner loop is always the size of the innermost iteration dimension, and allowing it to get cut up would require special handling, effectively making it more like the buffered version.

NPY_ITER_BUFFERED
Causes the iterator to store buffering data, and use buffering to satisfy data type, alignment, and byte-order requirements. To buffer an operand, do not specify the *NPY_ITER_COPY* or *NPY_ITER_UPDATEIFCOPY* flags, because they will override buffering. Buffering is especially useful for Python code using the iterator, allowing for larger chunks of data at once to amortize the Python interpreter overhead.

If used with *NPY_ITER_EXTERNAL_LOOP*, the inner loop for the caller may get larger chunks than would be possible without buffering, because of how the strides are laid out.

Note that if an operand is given the flag *NPY_ITER_COPY* or *NPY_ITER_UPDATEIFCOPY*, a copy will be made in preference to buffering. Buffering will still occur when the array was broadcast so elements need to be duplicated to get a constant stride.

In normal buffering, the size of each inner loop is equal to the buffer size, or possibly larger if *NPY_ITER_GROWINNER* is specified. If *NPY_ITER_REDUCE_OK* is enabled and a reduction occurs, the inner loops may become smaller depending on the structure of the reduction.

NPY_ITER_GROWINNER
When buffering is enabled, this allows the size of the inner loop to grow when buffering isn't necessary. This option is best used if you're doing a straight pass through all the data, rather than anything with small cache-friendly arrays of temporary values for each inner loop.

NPY_ITER_DELAY_BUFALLOC
When buffering is enabled, this delays allocation of the buffers until *NpyIter_Reset* or another reset function is called. This flag exists to avoid wasteful copying of buffer data when making multiple copies of a buffered iterator for multi-threaded iteration.

Another use of this flag is for setting up reduction operations. After the iterator is created, and a reduction output is allocated automatically by the iterator (be sure to use READWRITE access), its value may be initialized to the reduction unit. Use *NpyIter_GetOperandArray* to get the object. Then, call *NpyIter_Reset* to allocate and fill the buffers with their initial values.

Flags that may be passed in `op_flags[i]`, where `0 <= i < nop`:

NPY_ITER_READWRITE

NPY_ITER_READONLY

NPY_ITER_WRITEONLY
Indicate how the user of the iterator will read or write to `op[i]`. Exactly one of these flags must be specified per operand.

NPY_ITER_COPY
Allow a copy of `op[i]` to be made if it does not meet the data type or alignment requirements as specified by the constructor flags and parameters.

NPY_ITER_UPDATEIFCOPY
Triggers *NPY_ITER_COPY*, and when an array operand is flagged for writing and is copied, causes the data in a copy to be copied back to `op[i]` when the iterator is destroyed.

If the operand is flagged as write-only and a copy is needed, an uninitialized temporary array will be created and then copied to back to `op[i]` on destruction, instead of doing the unecessary copy operation.

NPY_ITER_NBO

NPY_ITER_ALIGNED

NPY_ITER_CONTIG
Causes the iterator to provide data for `op[i]` that is in native byte order, aligned according to the dtype requirements, contiguous, or any combination.

By default, the iterator produces pointers into the arrays provided, which may be aligned or unaligned, and with any byte order. If copying or buffering is not enabled and the operand data doesn't satisfy the constraints, an error will be raised.

The contiguous constraint applies only to the inner loop, successive inner loops may have arbitrary pointer changes.

If the requested data type is in non-native byte order, the NBO flag overrides it and the requested data type is converted to be in native byte order.

NPY_ITER_ALLOCATE
This is for output arrays, and requires that the flag *NPY_ITER_WRITEONLY* or *NPY_ITER_READWRITE* be set. If `op[i]` is NULL, creates a new array with the final broadcast dimensions, and a layout matching the iteration order of the iterator.

When `op[i]` is NULL, the requested data type `op_dtypes[i]` may be NULL as well, in which case it is automatically generated from the dtypes of the arrays which are flagged as readable. The rules for generating the dtype are the same is for UFuncs. Of special note is handling of byte order in the selected dtype. If there is exactly one input, the input's dtype is used as is. Otherwise, if more than one input dtypes are combined together, the output will be in native byte order.

After being allocated with this flag, the caller may retrieve the new array by calling *NpyIter_GetOperandArray* and getting the i-th object in the returned C array. The caller must call Py_INCREF on it to claim a reference to the array.

NPY_ITER_NO_SUBTYPE
For use with *NPY_ITER_ALLOCATE*, this flag disables allocating an array subtype for the output, forcing it to be a straight ndarray.

TODO: Maybe it would be better to introduce a function `NpyIter_GetWrappedOutput` and remove this flag?

NPY_ITER_NO_BROADCAST
Ensures that the input or output matches the iteration dimensions exactly.

NPY_ITER_ARRAYMASK
New in version 1.7.

Indicates that this operand is the mask to use for selecting elements when writing to operands which have the *NPY_ITER_WRITEMASKED* flag applied to them. Only one operand may have *NPY_ITER_ARRAYMASK* flag applied to it.

The data type of an operand with this flag should be either *NPY_BOOL*, *NPY_MASK*, or a struct dtype whose fields are all valid mask dtypes. In the latter case, it must match up with a struct operand being WRITEMASKED, as it is specifying a mask for each field of that array.

This flag only affects writing from the buffer back to the array. This means that if the operand is also *NPY_ITER_READWRITE* or *NPY_ITER_WRITEONLY*, code doing iteration can write to this operand to control which elements will be untouched and which ones will be modified. This is useful when the mask should be a combination of input masks, for example. Mask values can be created with the `NpyMask_Create` function.

NPY_ITER_WRITEMASKED
New in version 1.7.

Indicates that only elements which the operand with the ARRAYMASK flag indicates are intended to be modified by the iteration. In general, the iterator does not enforce this, it is up to the code doing the iteration to follow that promise. Code can use the `NpyMask_IsExposed` inline function to test whether the mask at a particular element allows writing.

When this flag is used, and this operand is buffered, this changes how data is copied from the buffer into the array. A masked copying routine is used, which only copies the elements in the buffer for which `NpyMask_IsExposed` returns true from the corresponding element in the ARRAYMASK operand.

*NpyIter** **NpyIter_AdvancedNew**(npy_intp *nop*, *PyArrayObject*** *op*, npy_uint32 *flags*, *NPY_ORDER* *order*, *NPY_CASTING* *casting*, npy_uint32* *op_flags*, *PyArray_Descr*** *op_dtypes*, int *oa_ndim*, int** *op_axes*, npy_intp* *itershape*, npy_intp *buffersize*)

Extends *NpyIter_MultiNew* with several advanced options providing more control over broadcasting and buffering.

If -1/NULL values are passed to `oa_ndim`, `op_axes`, `itershape`, and `buffersize`, it is equivalent to *NpyIter_MultiNew*.

The parameter `oa_ndim`, when not zero or -1, specifies the number of dimensions that will be iterated with customized broadcasting. If it is provided, `op_axes` must and `itershape` can also be provided. The `op_axes` parameter let you control in detail how the axes of the operand arrays get matched together and iterated. In `op_axes`, you must provide an array of `nop` pointers to `oa_ndim`-sized arrays of type `npy_intp`. If an entry in `op_axes` is NULL, normal broadcasting rules will apply. In `op_axes[j][i]` is stored either a valid axis of `op[j]`, or -1 which means `newaxis`. Within each `op_axes[j]` array, axes may not be repeated. The following example is how normal broadcasting applies to a 3-D array, a 2-D array, a 1-D array and a scalar.

Note: Before NumPy 1.8 `oa_ndim == 0` was used for signalling that that ``op_axes` and `itershape` are unused. This is deprecated and should be replaced with -1. Better backward compatibility may be achieved by using *NpyIter_MultiNew* for this case.

```
int oa_ndim = 3;                /* # iteration axes */
int op0_axes[] = {0, 1, 2};     /* 3-D operand */
int op1_axes[] = {-1, 0, 1};    /* 2-D operand */
int op2_axes[] = {-1, -1, 0};   /* 1-D operand */
```

```
    int op3_axes[] = {-1, -1, -1}  /* 0-D (scalar) operand */
    int* op_axes[] = {op0_axes, op1_axes, op2_axes, op3_axes};
```

The `itershape` parameter allows you to force the iterator to have a specific iteration shape. It is an array of length `oa_ndim`. When an entry is negative, its value is determined from the operands. This parameter allows automatically allocated outputs to get additional dimensions which don't match up with any dimension of an input.

If `buffersize` is zero, a default buffer size is used, otherwise it specifies how big of a buffer to use. Buffers which are powers of 2 such as 4096 or 8192 are recommended.

Returns NULL if there is an error, otherwise returns the allocated iterator.

*NpyIter** **NpyIter_Copy**(*NpyIter** *iter*)

Makes a copy of the given iterator. This function is provided primarily to enable multi-threaded iteration of the data.

TODO: Move this to a section about multithreaded iteration.

The recommended approach to multithreaded iteration is to first create an iterator with the flags *NPY_ITER_EXTERNAL_LOOP*, *NPY_ITER_RANGED*, *NPY_ITER_BUFFERED*, *NPY_ITER_DELAY_BUFALLOC*, and possibly *NPY_ITER_GROWINNER*. Create a copy of this iterator for each thread (minus one for the first iterator). Then, take the iteration index range `[0, NpyIter_GetIterSize(iter))` and split it up into tasks, for example using a TBB parallel_for loop. When a thread gets a task to execute, it then uses its copy of the iterator by calling *NpyIter_ResetToIterIndexRange* and iterating over the full range.

When using the iterator in multi-threaded code or in code not holding the Python GIL, care must be taken to only call functions which are safe in that context. *NpyIter_Copy* cannot be safely called without the Python GIL, because it increments Python references. The `Reset*` and some other functions may be safely called by passing in the `errmsg` parameter as non-NULL, so that the functions will pass back errors through it instead of setting a Python exception.

int NpyIter_RemoveAxis(NpyIter* iter, int axis)``

Removes an axis from iteration. This requires that *NPY_ITER_MULTI_INDEX* was set for iterator creation, and does not work if buffering is enabled or an index is being tracked. This function also resets the iterator to its initial state.

This is useful for setting up an accumulation loop, for example. The iterator can first be created with all the dimensions, including the accumulation axis, so that the output gets created correctly. Then, the accumulation axis can be removed, and the calculation done in a nested fashion.

WARNING: This function may change the internal memory layout of the iterator. Any cached functions or pointers from the iterator must be retrieved again! The iterator range will be reset as well.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

int **NpyIter_RemoveMultiIndex**(*NpyIter** *iter*)

If the iterator is tracking a multi-index, this strips support for them, and does further iterator optimizations that are possible if multi-indices are not needed. This function also resets the iterator to its initial state.

WARNING: This function may change the internal memory layout of the iterator. Any cached functions or pointers from the iterator must be retrieved again!

After calling this function, `NpyIter_HasMultiIndex(iter)` will return false.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

int **NpyIter_EnableExternalLoop**(*NpyIter** *iter*)

If *NpyIter_RemoveMultiIndex* was called, you may want to enable the flag *NPY_ITER_EXTERNAL_LOOP*. This flag is not permitted together with *NPY_ITER_MULTI_INDEX*,

so this function is provided to enable the feature after *NpyIter_RemoveMultiIndex* is called. This function also resets the iterator to its initial state.

WARNING: This function changes the internal logic of the iterator. Any cached functions or pointers from the iterator must be retrieved again!

Returns NPY_SUCCEED or NPY_FAIL.

int **NpyIter_Deallocate**(*NpyIter** *iter*)
Deallocates the iterator object. This additionally frees any copies made, triggering UPDATEIFCOPY behavior where necessary.

Returns NPY_SUCCEED or NPY_FAIL.

int **NpyIter_Reset**(*NpyIter** *iter*, char** *errmsg*)
Resets the iterator back to its initial state, at the beginning of the iteration range.

Returns NPY_SUCCEED or NPY_FAIL. If errmsg is non-NULL, no Python exception is set when NPY_FAIL is returned. Instead, *errmsg is set to an error message. When errmsg is non-NULL, the function may be safely called without holding the Python GIL.

int **NpyIter_ResetToIterIndexRange**(*NpyIter** *iter*, npy_intp *istart*, npy_intp *iend*, char** *errmsg*)
Resets the iterator and restricts it to the iterindex range [istart, iend). See *NpyIter_Copy* for an explanation of how to use this for multi-threaded iteration. This requires that the flag *NPY_ITER_RANGED* was passed to the iterator constructor.

If you want to reset both the iterindex range and the base pointers at the same time, you can do the following to avoid extra buffer copying (be sure to add the return code error checks when you copy this code).

```
/* Set to a trivial empty range */
NpyIter_ResetToIterIndexRange(iter, 0, 0);
/* Set the base pointers */
NpyIter_ResetBasePointers(iter, baseptrs);
/* Set to the desired range */
NpyIter_ResetToIterIndexRange(iter, istart, iend);
```

Returns NPY_SUCCEED or NPY_FAIL. If errmsg is non-NULL, no Python exception is set when NPY_FAIL is returned. Instead, *errmsg is set to an error message. When errmsg is non-NULL, the function may be safely called without holding the Python GIL.

int **NpyIter_ResetBasePointers**(*NpyIter* **iter*, char** *baseptrs*, char** *errmsg*)
Resets the iterator back to its initial state, but using the values in baseptrs for the data instead of the pointers from the arrays being iterated. This functions is intended to be used, together with the op_axes parameter, by nested iteration code with two or more iterators.

Returns NPY_SUCCEED or NPY_FAIL. If errmsg is non-NULL, no Python exception is set when NPY_FAIL is returned. Instead, *errmsg is set to an error message. When errmsg is non-NULL, the function may be safely called without holding the Python GIL.

TODO: Move the following into a special section on nested iterators.

Creating iterators for nested iteration requires some care. All the iterator operands must match exactly, or the calls to *NpyIter_ResetBasePointers* will be invalid. This means that automatic copies and output allocation should not be used haphazardly. It is possible to still use the automatic data conversion and casting features of the iterator by creating one of the iterators with all the conversion parameters enabled, then grabbing the allocated operands with the *NpyIter_GetOperandArray* function and passing them into the constructors for the rest of the iterators.

WARNING: When creating iterators for nested iteration, the code must not use a dimension more than once in the different iterators. If this is done, nested iteration will produce out-of-bounds pointers during iteration.

WARNING: When creating iterators for nested iteration, buffering can only be applied to the innermost iterator. If a buffered iterator is used as the source for `baseptrs`, it will point into a small buffer instead of the array and the inner iteration will be invalid.

The pattern for using nested iterators is as follows.

```
NpyIter *iter1, *iter1;
NpyIter_IterNextFunc *iternext1, *iternext2;
char **dataptrs1;

/*
 * With the exact same operands, no copies allowed, and
 * no axis in op_axes used both in iter1 and iter2.
 * Buffering may be enabled for iter2, but not for iter1.
 */
iter1 = ...; iter2 = ...;

iternext1 = NpyIter_GetIterNext(iter1);
iternext2 = NpyIter_GetIterNext(iter2);
dataptrs1 = NpyIter_GetDataPtrArray(iter1);

do {
    NpyIter_ResetBasePointers(iter2, dataptrs1);
    do {
        /* Use the iter2 values */
    } while (iternext2(iter2));
} while (iternext1(iter1));
```

int **NpyIter_GotoMultiIndex** (*NpyIter* iter*, npy_intp* *multi_index*)
Adjusts the iterator to point to the `ndim` indices pointed to by `multi_index`. Returns an error if a multi-index is not being tracked, the indices are out of bounds, or inner loop iteration is disabled.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

int **NpyIter_GotoIndex** (*NpyIter* iter*, npy_intp *index*)
Adjusts the iterator to point to the `index` specified. If the iterator was constructed with the flag *NPY_ITER_C_INDEX*, `index` is the C-order index, and if the iterator was constructed with the flag *NPY_ITER_F_INDEX*, `index` is the Fortran-order index. Returns an error if there is no index being tracked, the index is out of bounds, or inner loop iteration is disabled.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

npy_intp **NpyIter_GetIterSize** (*NpyIter* iter*)
Returns the number of elements being iterated. This is the product of all the dimensions in the shape. When a multi index is being tracked (and *NpyIter_RemoveAxis* may be called) the size may be `-1` to indicate an iterator is too large. Such an iterator is invalid, but may become valid after *NpyIter_RemoveAxis* is called. It is not necessary to check for this case.

npy_intp **NpyIter_GetIterIndex** (*NpyIter* iter*)
Gets the `iterindex` of the iterator, which is an index matching the iteration order of the iterator.

void **NpyIter_GetIterIndexRange** (*NpyIter* iter*, npy_intp* *istart*, npy_intp* *iend*)
Gets the `iterindex` sub-range that is being iterated. If *NPY_ITER_RANGED* was not specified, this always returns the range `[0, NpyIter_IterSize(iter))`.

int **NpyIter_GotoIterIndex** (*NpyIter* iter*, npy_intp *iterindex*)
Adjusts the iterator to point to the `iterindex` specified. The IterIndex is an index matching the iteration order of the iterator. Returns an error if the `iterindex` is out of bounds, buffering is enabled, or inner loop iteration is disabled.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

npy_bool **NpyIter_HasDelayedBufAlloc**(*NpyIter* iter*)
Returns 1 if the flag *NPY_ITER_DELAY_BUFALLOC* was passed to the iterator constructor, and no call to one of the Reset functions has been done yet, 0 otherwise.

npy_bool **NpyIter_HasExternalLoop**(*NpyIter* iter*)
Returns 1 if the caller needs to handle the inner-most 1-dimensional loop, or 0 if the iterator handles all looping. This is controlled by the constructor flag *NPY_ITER_EXTERNAL_LOOP* or *NpyIter_EnableExternalLoop*.

npy_bool **NpyIter_HasMultiIndex**(*NpyIter* iter*)
Returns 1 if the iterator was created with the *NPY_ITER_MULTI_INDEX* flag, 0 otherwise.

npy_bool **NpyIter_HasIndex**(*NpyIter* iter*)
Returns 1 if the iterator was created with the *NPY_ITER_C_INDEX* or *NPY_ITER_F_INDEX* flag, 0 otherwise.

npy_bool **NpyIter_RequiresBuffering**(*NpyIter* iter*)
Returns 1 if the iterator requires buffering, which occurs when an operand needs conversion or alignment and so cannot be used directly.

npy_bool **NpyIter_IsBuffered**(*NpyIter* iter*)
Returns 1 if the iterator was created with the *NPY_ITER_BUFFERED* flag, 0 otherwise.

npy_bool **NpyIter_IsGrowInner**(*NpyIter* iter*)
Returns 1 if the iterator was created with the *NPY_ITER_GROWINNER* flag, 0 otherwise.

npy_intp **NpyIter_GetBufferSize**(*NpyIter* iter*)
If the iterator is buffered, returns the size of the buffer being used, otherwise returns 0.

int **NpyIter_GetNDim**(*NpyIter* iter*)
Returns the number of dimensions being iterated. If a multi-index was not requested in the iterator constructor, this value may be smaller than the number of dimensions in the original objects.

int **NpyIter_GetNOp**(*NpyIter* iter*)
Returns the number of operands in the iterator.

When `NPY_ITER_USE_MASKNA` is used on an operand, a new operand is added to the end of the operand list in the iterator to track that operand's NA mask. Thus, this equals the number of construction operands plus the number of operands for which the flag `NPY_ITER_USE_MASKNA` was specified.

int **NpyIter_GetFirstMaskNAOp**(*NpyIter* iter*)
New in version 1.7.

Returns the index of the first NA mask operand in the array. This value is equal to the number of operands passed into the constructor.

npy_intp* **NpyIter_GetAxisStrideArray**(*NpyIter* iter*, int *axis*)
Gets the array of strides for the specified axis. Requires that the iterator be tracking a multi-index, and that buffering not be enabled.

This may be used when you want to match up operand axes in some fashion, then remove them with `NpyIter_RemoveAxis` to handle their processing manually. By calling this function before removing the axes, you can get the strides for the manual processing.

Returns `NULL` on error.

int **NpyIter_GetShape**(*NpyIter* iter*, npy_intp* *outshape*)
Returns the broadcast shape of the iterator in `outshape`. This can only be called on an iterator which is tracking a multi-index.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

*PyArray_Descr*** **NpyIter_GetDescrArray** (*NpyIter** *iter*)
This gives back a pointer to the `nop` data type Descrs for the objects being iterated. The result points into `iter`, so the caller does not gain any references to the Descrs.

This pointer may be cached before the iteration loop, calling `iternext` will not change it.

PyObject** **NpyIter_GetOperandArray** (*NpyIter** *iter*)
This gives back a pointer to the `nop` operand PyObjects that are being iterated. The result points into `iter`, so the caller does not gain any references to the PyObjects.

npy_int8* **NpyIter_GetMaskNAIndexArray** (*NpyIter** *iter*)
New in version 1.7.

This gives back a pointer to the `nop` indices which map construction operands with `NPY_ITER_USE_MASKNA` flagged to their corresponding NA mask operands and vice versa. For operands which were not flagged with `NPY_ITER_USE_MASKNA`, this array contains negative values.

PyObject* **NpyIter_GetIterView** (*NpyIter** *iter*, npy_intp *i*)
This gives back a reference to a new ndarray view, which is a view into the i-th object in the array *NpyIter_GetOperandArray*, whose dimensions and strides match the internal optimized iteration pattern. A C-order iteration of this view is equivalent to the iterator's iteration order.

For example, if an iterator was created with a single array as its input, and it was possible to rearrange all its axes and then collapse it into a single strided iteration, this would return a view that is a one-dimensional array.

void **NpyIter_GetReadFlags** (*NpyIter** *iter*, char* *outreadflags*)
Fills `nop` flags. Sets `outreadflags[i]` to 1 if `op[i]` can be read from, and to 0 if not.

void **NpyIter_GetWriteFlags** (*NpyIter** *iter*, char* *outwriteflags*)
Fills `nop` flags. Sets `outwriteflags[i]` to 1 if `op[i]` can be written to, and to 0 if not.

int **NpyIter_CreateCompatibleStrides** (*NpyIter** *iter*, npy_intp *itemsize*, npy_intp* *outstrides*)
Builds a set of strides which are the same as the strides of an output array created using the *NPY_ITER_ALLOCATE* flag, where NULL was passed for op_axes. This is for data packed contiguously, but not necessarily in C or Fortran order. This should be used together with *NpyIter_GetShape* and *NpyIter_GetNDim* with the flag *NPY_ITER_MULTI_INDEX* passed into the constructor.

A use case for this function is to match the shape and layout of the iterator and tack on one or more dimensions. For example, in order to generate a vector per input value for a numerical gradient, you pass in ndim*itemsize for itemsize, then add another dimension to the end with size ndim and stride itemsize. To do the Hessian matrix, you do the same thing but add two dimensions, or take advantage of the symmetry and pack it into 1 dimension with a particular encoding.

This function may only be called if the iterator is tracking a multi-index and if *NPY_ITER_DONT_NEGATE_STRIDES* was used to prevent an axis from being iterated in reverse order.

If an array is created with this method, simply adding 'itemsize' for each iteration will traverse the new array matching the iterator.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

npy_bool **NpyIter_IsFirstVisit** (*NpyIter** *iter*, int *iop*)
New in version 1.7.

Checks to see whether this is the first time the elements of the specified reduction operand which the iterator points at are being seen for the first time. The function returns a reasonable answer for reduction operands and when buffering is disabled. The answer may be incorrect for buffered non-reduction operands.

This function is intended to be used in EXTERNAL_LOOP mode only, and will produce some wrong answers when that mode is not enabled.

If this function returns true, the caller should also check the inner loop stride of the operand, because if that stride is 0, then only the first element of the innermost external loop is being visited for the first time.

WARNING: For performance reasons, 'iop' is not bounds-checked, it is not confirmed that 'iop' is actually a reduction operand, and it is not confirmed that EXTERNAL_LOOP mode is enabled. These checks are the responsibility of the caller, and should be done outside of any inner loops.

5.5.6 Functions For Iteration

*NpyIter_IterNextFunc** **NpyIter_GetIterNext** (*NpyIter** *iter*, char** *errmsg*)

Returns a function pointer for iteration. A specialized version of the function pointer may be calculated by this function instead of being stored in the iterator structure. Thus, to get good performance, it is required that the function pointer be saved in a variable rather than retrieved for each loop iteration.

Returns NULL if there is an error. If errmsg is non-NULL, no Python exception is set when `NPY_FAIL` is returned. Instead, *errmsg is set to an error message. When errmsg is non-NULL, the function may be safely called without holding the Python GIL.

The typical looping construct is as follows.

```
NpyIter_IterNextFunc *iternext = NpyIter_GetIterNext(iter, NULL);
char** dataptr = NpyIter_GetDataPtrArray(iter);

do {
    /* use the addresses dataptr[0], ... dataptr[nop-1] */
} while(iternext(iter));
```

When *NPY_ITER_EXTERNAL_LOOP* is specified, the typical inner loop construct is as follows.

```
NpyIter_IterNextFunc *iternext = NpyIter_GetIterNext(iter, NULL);
char** dataptr = NpyIter_GetDataPtrArray(iter);
npy_intp* stride = NpyIter_GetInnerStrideArray(iter);
npy_intp* size_ptr = NpyIter_GetInnerLoopSizePtr(iter), size;
npy_intp iop, nop = NpyIter_GetNOp(iter);

do {
    size = *size_ptr;
    while (size--) {
        /* use the addresses dataptr[0], ... dataptr[nop-1] */
        for (iop = 0; iop < nop; ++iop) {
            dataptr[iop] += stride[iop];
        }
    }
} while (iternext());
```

Observe that we are using the dataptr array inside the iterator, not copying the values to a local temporary. This is possible because when `iternext()` is called, these pointers will be overwritten with fresh values, not incrementally updated.

If a compile-time fixed buffer is being used (both flags *NPY_ITER_BUFFERED* and *NPY_ITER_EXTERNAL_LOOP*), the inner size may be used as a signal as well. The size is guaranteed to become zero when `iternext()` returns false, enabling the following loop construct. Note that if you use this construct, you should not pass *NPY_ITER_GROWINNER* as a flag, because it will cause larger sizes under some circumstances.

```
/* The constructor should have buffersize passed as this value */
#define FIXED_BUFFER_SIZE 1024
```

```
    NpyIter_IterNextFunc *iternext = NpyIter_GetIterNext(iter, NULL);
    char **dataptr = NpyIter_GetDataPtrArray(iter);
    npy_intp *stride = NpyIter_GetInnerStrideArray(iter);
    npy_intp *size_ptr = NpyIter_GetInnerLoopSizePtr(iter), size;
    npy_intp i, iop, nop = NpyIter_GetNOp(iter);

    /* One loop with a fixed inner size */
    size = *size_ptr;
    while (size == FIXED_BUFFER_SIZE) {
        /*
         * This loop could be manually unrolled by a factor
         * which divides into FIXED_BUFFER_SIZE
         */
        for (i = 0; i < FIXED_BUFFER_SIZE; ++i) {
            /* use the addresses dataptr[0], ... dataptr[nop-1] */
            for (iop = 0; iop < nop; ++iop) {
                dataptr[iop] += stride[iop];
            }
        }
        iternext();
        size = *size_ptr;
    }

    /* Finish-up loop with variable inner size */
    if (size > 0) do {
        size = *size_ptr;
        while (size--) {
            /* use the addresses dataptr[0], ... dataptr[nop-1] */
            for (iop = 0; iop < nop; ++iop) {
                dataptr[iop] += stride[iop];
            }
        }
    } while (iternext());
```

NpyIter_GetMultiIndexFunc ***NpyIter_GetGetMultiIndex**(*NpyIter* *iter*, char** *errmsg*)

Returns a function pointer for getting the current multi-index of the iterator. Returns NULL if the iterator is not tracking a multi-index. It is recommended that this function pointer be cached in a local variable before the iteration loop.

Returns NULL if there is an error. If errmsg is non-NULL, no Python exception is set when `NPY_FAIL` is returned. Instead, *errmsg is set to an error message. When errmsg is non-NULL, the function may be safely called without holding the Python GIL.

char** **NpyIter_GetDataPtrArray**(*NpyIter* *iter*)

This gives back a pointer to the `nop` data pointers. If *NPY_ITER_EXTERNAL_LOOP* was not specified, each data pointer points to the current data item of the iterator. If no inner iteration was specified, it points to the first data item of the inner loop.

This pointer may be cached before the iteration loop, calling `iternext` will not change it. This function may be safely called without holding the Python GIL.

char** **NpyIter_GetInitialDataPtrArray**(*NpyIter* *iter*)

Gets the array of data pointers directly into the arrays (never into the buffers), corresponding to iteration index 0.

These pointers are different from the pointers accepted by `NpyIter_ResetBasePointers`, because the direction along some axes may have been reversed.

This function may be safely called without holding the Python GIL.

npy_intp* **NpyIter_GetIndexPtr**(*NpyIter** *iter*)
This gives back a pointer to the index being tracked, or NULL if no index is being tracked. It is only useable if one of the flags *NPY_ITER_C_INDEX* or *NPY_ITER_F_INDEX* were specified during construction.

When the flag *NPY_ITER_EXTERNAL_LOOP* is used, the code needs to know the parameters for doing the inner loop. These functions provide that information.

npy_intp* **NpyIter_GetInnerStrideArray**(*NpyIter** *iter*)
Returns a pointer to an array of the `nop` strides, one for each iterated object, to be used by the inner loop.

This pointer may be cached before the iteration loop, calling `iternext` will not change it. This function may be safely called without holding the Python GIL.

WARNING: While the pointer may be cached, its values may change if the iterator is buffered.

npy_intp* **NpyIter_GetInnerLoopSizePtr**(*NpyIter** *iter*)
Returns a pointer to the number of iterations the inner loop should execute.

This address may be cached before the iteration loop, calling `iternext` will not change it. The value itself may change during iteration, in particular if buffering is enabled. This function may be safely called without holding the Python GIL.

void **NpyIter_GetInnerFixedStrideArray**(*NpyIter** *iter*, npy_intp* *out_strides*)
Gets an array of strides which are fixed, or will not change during the entire iteration. For strides that may change, the value NPY_MAX_INTP is placed in the stride.

Once the iterator is prepared for iteration (after a reset if `NPY_DELAY_BUFALLOC` was used), call this to get the strides which may be used to select a fast inner loop function. For example, if the stride is 0, that means the inner loop can always load its value into a variable once, then use the variable throughout the loop, or if the stride equals the itemsize, a contiguous version for that operand may be used.

This function may be safely called without holding the Python GIL.

5.5.7 Converting from Previous NumPy Iterators

The old iterator API includes functions like PyArrayIter_Check, PyArray_Iter* and PyArray_ITER_*. The multi-iterator array includes PyArray_MultiIter*, PyArray_Broadcast, and PyArray_RemoveSmallest. The new iterator design replaces all of this functionality with a single object and associated API. One goal of the new API is that all uses of the existing iterator should be replaceable with the new iterator without significant effort. In 1.6, the major exception to this is the neighborhood iterator, which does not have corresponding features in this iterator.

Here is a conversion table for which functions to use with the new iterator:

Iterator Functions	
PyArray_IterNew	*NpyIter_New*
PyArray_IterAllButAxis	*NpyIter_New* + `axes` parameter **or** Iterator flag *NPY_ITER_EXTERNAL_LOOP*
PyArray_BroadcastToShape	**NOT SUPPORTED** (Use the support for multiple operands instead.)
PyArrayIter_Check	Will need to add this in Python exposure
PyArray_ITER_RESET	*NpyIter_Reset*
PyArray_ITER_NEXT	Function pointer from *NpyIter_GetIterNext*
PyArray_ITER_DATA	c:func:*NpyIter_GetDataPtrArray*
PyArray_ITER_GOTO	*NpyIter_GotoMultiIndex*
PyArray_ITER_GOTO1D	*NpyIter_GotoIndex* or *NpyIter_GotoIterIndex*
PyArray_ITER_NOTDONE	Return value of `iternext` function pointer
Multi-iterator Functions	
PyArray_MultiIterNew	*NpyIter_MultiNew*
PyArray_MultiIter_RESET	*NpyIter_Reset*
PyArray_MultiIter_NEXT	Function pointer from *NpyIter_GetIterNext*
PyArray_MultiIter_DATA	*NpyIter_GetDataPtrArray*
PyArray_MultiIter_NEXTi	**NOT SUPPORTED** (always lock-step iteration)
PyArray_MultiIter_GOTO	*NpyIter_GotoMultiIndex*
PyArray_MultiIter_GOTO1D	*NpyIter_GotoIndex* or *NpyIter_GotoIterIndex*
PyArray_MultiIter_NOTDONE	Return value of `iternext` function pointer
PyArray_Broadcast	Handled by *NpyIter_MultiNew*
PyArray_RemoveSmallest	Iterator flag *NPY_ITER_EXTERNAL_LOOP*
Other Functions	
PyArray_ConvertToCommonType	Iterator flag *NPY_ITER_COMMON_DTYPE*

5.6 UFunc API

5.6.1 Constants

UFUNC_ERR_{HANDLER}
`{HANDLER}` can be **IGNORE**, **WARN**, **RAISE**, or **CALL**

UFUNC_{THING}_{ERR}
`{THING}` can be **MASK**, **SHIFT**, or **FPE**, and `{ERR}` can be **DIVIDEBYZERO**, **OVERFLOW**, **UNDERFLOW**, and **INVALID**.

PyUFunc_{VALUE}
`{VALUE}` can be **One** (1), **Zero** (0), or **None** (-1)

5.6.2 Macros

NPY_LOOP_BEGIN_THREADS
Used in universal function code to only release the Python GIL if loop->obj is not true (*i.e.* this is not an OBJECT array loop). Requires use of *NPY_BEGIN_THREADS_DEF* in variable declaration area.

NPY_LOOP_END_THREADS
Used in universal function code to re-acquire the Python GIL if it was released (because loop->obj was not true).

UFUNC_CHECK_ERROR(loop)
A macro used internally to check for errors and goto fail if found. This macro requires a fail label in the current

code block. The *loop* variable must have at least members (obj, errormask, and errorobj). If *loop* ->obj is nonzero, then `PyErr_Occurred` () is called (meaning the GIL must be held). If *loop* ->obj is zero, then if *loop* ->errormask is nonzero, *`PyUFunc_checkfperr`* is called with arguments *loop* ->errormask and *loop* ->errobj. If the result of this check of the IEEE floating point registers is true then the code redirects to the fail label which must be defined.

`UFUNC_CHECK_STATUS` (ret)

Deprecated: use npy_clear_floatstatus from npy_math.h instead.

A macro that expands to platform-dependent code. The *ret* variable can can be any integer. The `UFUNC_FPE_{ERR}` bits are set in *ret* according to the status of the corresponding error flags of the floating point processor.

5.6.3 Functions

`PyObject* PyUFunc_FromFuncAndData(PyUFuncGenericFunction* func,`

`void** data, char* types, int ntypes, int nin, int nout, int identity,`

`char* name, char* doc, int unused)`

Create a new broadcasting universal function from required variables. Each ufunc builds around the notion of an element-by-element operation. Each ufunc object contains pointers to 1-d loops implementing the basic functionality for each supported type.

Note: The *func*, *data*, *types*, *name*, and *doc* arguments are not copied by `PyUFunc_FromFuncAndData`. The caller must ensure that the memory used by these arrays is not freed as long as the ufunc object is alive.

Parameters

- **`func`** – Must to an array of length *ntypes* containing `PyUFuncGenericFunction` items. These items are pointers to functions that actually implement the underlying (element-by-element) function N times.
- **`data`** – Should be `NULL` or a pointer to an array of size *ntypes* . This array may contain arbitrary extra-data to be passed to the corresponding 1-d loop function in the func array.
- **`types`** – Must be of length (*nin* + *nout*) * *ntypes*, and it contains the data-types (built-in only) that the corresponding function in the *func* array can deal with.
- **`ntypes`** – How many different data-type "signatures" the ufunc has implemented.
- **`nin`** – The number of inputs to this operation.
- **`nout`** – The number of outputs
- **`name`** – The name for the ufunc. Specifying a name of 'add' or 'multiply' enables a special behavior for integer-typed reductions when no dtype is given. If the input type is an integer (or boolean) data type smaller than the size of the **int_** data type, it will be internally upcast to the **int_** (or uint) data type.
- **`doc`** – Allows passing in a documentation string to be stored with the ufunc. The documentation string should not contain the name of the function or the calling signature as that will be dynamically determined from the object and available when accessing the **__doc__** attribute of the ufunc.
- **`unused`** – Unused and present for backwards compatibility of the C-API.

PyObject* PyUFunc_FromFuncAndDataAndSignature(PyUFuncGenericFunction* func,

void data, char* types, int ntypes, int nin, int nout, int identity,**

char* name, char* doc, int unused, char *signature)

This function is very similar to PyUFunc_FromFuncAndData above, but has an extra *signature* argument, to define generalized universal functions. Similarly to how ufuncs are built around an element-by-element operation, gufuncs are around subarray-by-subarray operations, the signature defining the subarrays to operate on.

Parameters

- **signature** – The signature for the new gufunc. Setting it to NULL is equivalent to calling PyUFunc_FromFuncAndData. A copy of the string is made, so the passed in buffer can be freed.

int PyUFunc_RegisterLoopForType(PyUFuncObject* ufunc,

int usertype, PyUFuncGenericFunction function, int* arg_types, void* data)

This function allows the user to register a 1-d loop with an already- created ufunc to be used whenever the ufunc is called with any of its input arguments as the user-defined data-type. This is needed in order to make ufuncs work with built-in data-types. The data-type must have been previously registered with the numpy system. The loop is passed in as *function*. This loop can take arbitrary data which should be passed in as *data*. The data-types the loop requires are passed in as *arg_types* which must be a pointer to memory at least as large as ufunc->nargs.

int PyUFunc_RegisterLoopForDescr(PyUFuncObject* ufunc,

PyArray_Descr* userdtype, PyUFuncGenericFunction function,

PyArray_Descr arg_dtypes, void* data)**

This function behaves like PyUFunc_RegisterLoopForType above, except that it allows the user to register a 1-d loop using PyArray_Descr objects instead of dtype type num values. This allows a 1-d loop to be registered for structured array data-dtypes and custom data-types instead of scalar data-types.

int PyUFunc_ReplaceLoopBySignature(PyUFuncObject* ufunc,

PyUFuncGenericFunction newfunc, int* signature,

PyUFuncGenericFunction* oldfunc)

Replace a 1-d loop matching the given *signature* in the already-created *ufunc* with the new 1-d loop newfunc. Return the old 1-d loop function in *oldfunc*. Return 0 on success and -1 on failure. This function works only with built-in types (use `PyUFunc_RegisterLoopForType` for user-defined types). A signature is an array of data-type numbers indicating the inputs followed by the outputs assumed by the 1-d loop.

int PyUFunc_GenericFunction(PyUFuncObject* self,

PyObject* args, PyObject* kwds, PyArrayObject mps)**

A generic ufunc call. The ufunc is passed in as *self*, the arguments to the ufunc as *args* and *kwds*. The *mps* argument is an array of `PyArrayObject` pointers whose values are discarded and which receive the converted input arguments as well as the ufunc outputs when success is returned. The user is responsible for managing this array and receives a new reference for each array in *mps*. The total number of arrays in *mps* is given by *self* ->nin + *self* ->nout.

Returns 0 on success, -1 on error.

int **PyUFunc_checkfperr**(int *errmask*, PyObject* *errobj*)

A simple interface to the IEEE error-flag checking support. The *errmask* argument is a mask of `UFUNC_MASK_{ERR}` bitmasks indicating which errors to check for (and how to check for them). The *errobj* must be a Python tuple with two elements: a string containing the name which will be used in any communication of error and either a callable Python object (call-back function) or `Py_None`. The callable object will only be used if `UFUNC_ERR_CALL` is set as the desired error checking method. This routine manages the GIL and is safe to call even after releasing the GIL. If an error in the IEEE-compatibile hardware is determined a -1 is returned, otherwise a 0 is returned.

void **PyUFunc_clearfperr**()
Clear the IEEE error flags.

void PyUFunc_GetPyValues(char* name, int* bufsize,

int* errmask, PyObject errobj)**
Get the Python values used for ufunc processing from the thread-local storage area unless the defaults have been set in which case the name lookup is bypassed. The name is placed as a string in the first element of **errobj*. The second element is the looked-up function to call on error callback. The value of the looked-up buffer-size to use is passed into *bufsize*, and the value of the error mask is placed into *errmask*.

5.6.4 Generic functions

At the core of every ufunc is a collection of type-specific functions that defines the basic functionality for each of the supported types. These functions must evaluate the underlying function $N \geq 1$ times. Extra-data may be passed in that may be used during the calculation. This feature allows some general functions to be used as these basic looping functions. The general function has all the code needed to point variables to the right place and set up a function call. The general function assumes that the actual function to call is passed in as the extra data and calls it with the correct values. All of these functions are suitable for placing directly in the array of functions stored in the functions member of the PyUFuncObject structure.

void PyUFunc_f_f_As_d_d(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_d_d(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_f_f(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_g_g(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_F_F_As_D_D(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_F_F(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_D_D(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_G_G(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_e_e(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_e_e_As_f_f(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_e_e_As_d_d(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

Type specific, core 1-d functions for ufuncs where each calculation is obtained by calling a function taking one input argument and returning one output. This function is passed in `func`. The letters correspond to dtypechar's of the supported data types (`e` - half, `f` - float, `d` - double, `g` - long double, `F` - cfloat, `D` - cdouble, `G` - clongdouble). The argument *func* must support the same signature. The _As_X_X variants assume ndarray's of one data type but cast the values to use an underlying function that takes a different data type. Thus, `PyUFunc_f_f_As_d_d` uses ndarrays of data type *NPY_FLOAT* but calls out to a C-function that takes double and returns double.

void PyUFunc_ff_f_As_dd_d(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_ff_f(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_dd_d(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_gg_g(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_FF_F_As_DD_D(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_DD_D(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_FF_F(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_GG_G(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_ee_e(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_ee_e_As_ff_f(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_ee_e_As_dd_d(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

Type specific, core 1-d functions for ufuncs where each calculation is obtained by calling a function taking two input arguments and returning one output. The underlying function to call is passed in as *func*. The letters correspond to dtypechar's of the specific data type supported by the general-purpose function. The argument `func` must support the corresponding signature. The `_As_XX_X` variants assume ndarrays of one data type but cast the values at each iteration of the loop to use the underlying function that takes a different data type.

void PyUFunc_O_O(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

void PyUFunc_OO_O(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

One-input, one-output, and two-input, one-output core 1-d functions for the *NPY_OBJECT* data type. These functions handle reference count issues and return early on error. The actual function to call is *func* and it must accept calls with the signature `(PyObject*) (PyObject*)` for `PyUFunc_O_O` or `(PyObject*)(PyObject *, PyObject *)` for `PyUFunc_OO_O`.

void PyUFunc_O_O_method(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

This general purpose 1-d core function assumes that *func* is a string representing a method of the input object. For each iteration of the loop, the Python obejct is extracted from the array and its *func* method is called returning the result to the output array.

void PyUFunc_OO_O_method(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

This general purpose 1-d core function assumes that *func* is a string representing a method of the input object

that takes one argument. The first argument in *args* is the method whose function is called, the second argument in *args* is the argument passed to the function. The output of the function is stored in the third entry of *args*.

void PyUFunc_On_Om(char args, npy_intp* dimensions,**

npy_intp* steps, void* func)

This is the 1-d core function used by the dynamic ufuncs created by umath.frompyfunc(function, nin, nout). In this case *func* is a pointer to a *PyUFunc_PyFuncData* structure which has definition

PyUFunc_PyFuncData

```
typedef struct {
    int nin;
    int nout;
    PyObject *callable;
} PyUFunc_PyFuncData;
```

At each iteration of the loop, the *nin* input objects are exctracted from their object arrays and placed into an argument tuple, the Python *callable* is called with the input arguments, and the nout outputs are placed into their object arrays.

5.6.5 Importing the API

PY_UFUNC_UNIQUE_SYMBOL

NO_IMPORT_UFUNC

void **import_ufunc**(void)

These are the constants and functions for accessing the ufunc C-API from extension modules in precisely the same way as the array C-API can be accessed. The `import_ufunc` () function must always be called (in the initialization subroutine of the extension module). If your extension module is in one file then that is all that is required. The other two constants are useful if your extension module makes use of multiple files. In that case, define *PY_UFUNC_UNIQUE_SYMBOL* to something unique to your code and then in source files that do not contain the module initialization function but still need access to the UFUNC API, define *PY_UFUNC_UNIQUE_SYMBOL* to the same name used previously and also define *NO_IMPORT_UFUNC*.

The C-API is actually an array of function pointers. This array is created (and pointed to by a global variable) by import_ufunc. The global variable is either statically defined or allowed to be seen by other files depending on the state of `Py_UFUNC_UNIQUE_SYMBOL` and *NO_IMPORT_UFUNC*.

5.7 Generalized Universal Function API

There is a general need for looping over not only functions on scalars but also over functions on vectors (or arrays). This concept is realized in Numpy by generalizing the universal functions (ufuncs). In regular ufuncs, the elementary function is limited to element-by-element operations, whereas the generalized version (gufuncs) 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 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.

The dimensions beyond the core dimensions are called "loop" dimensions. In the above example, this corresponds to `(3, 5)`.

The signature determines how the dimensions of each input/output array are split into core and loop dimensions:

1. Each dimension in the signature is matched to a dimension of the corresponding passed-in array, starting from the end of the shape tuple. These are the core dimensions, and they must be present in the arrays, or an error will be raised.
2. Core dimensions assigned to the same label in the signature (e.g. the `i` in `inner1d`'s `(i),(i)->()`) must have exactly matching sizes, no broadcasting is performed.
3. The core dimensions are removed from all inputs and the remaining dimensions are broadcast together, defining the loop dimensions.
4. The shape of each output is determined from the loop dimensions plus the output's core dimensions

Typically, the size of all core dimensions in an output will be determined by the size of a core dimension with the same label in an input array. This is not a requirement, and it is possible to define a signature where a label comes up for the first time in an output, although some precautions must be taken when calling such a function. An example would be the function `euclidean_pdist(a)`, with signature `(n,d)->(p)`, that given an array of `n` `d`-dimensional vectors, computes all unique pairwise Euclidean distances among them. The output dimension `p` must therefore be equal to `n * (n - 1) / 2`, but it is the caller's responsibility to pass in an output array of the right size. If the size of a core dimension of an output cannot be determined from a passed in input or output array, an error will be raised.

Note: Prior to Numpy 1.10.0, less strict checks were in place: missing core dimensions were created by prepending 1's to the shape as necessary, core dimensions with the same label were broadcast together, and undetermined dimensions were created with size 1.

5.7.1 Definitions

Elementary Function
: Each ufunc consists of an elementary function that performs the most basic operation on the smallest portion of array arguments (e.g. adding two numbers is the most basic operation in adding two arrays). The ufunc applies the elementary function multiple times on different parts of the arrays. The input/output of elementary functions can be vectors; e.g., the elementary function of inner1d takes two vectors as input.

Signature
: A signature is a string describing the input/output dimensions of the elementary function of a ufunc. See section below for more details.

Core Dimension
: The dimensionality of each input/output of an elementary function is defined by its core dimensions (zero core dimensions correspond to a scalar input/output). The core dimensions are mapped to the last dimensions of the input/output arrays.

Dimension Name
: A dimension name represents a core dimension in the signature. Different dimensions may share a name, indicating that they are of the same size.

Dimension Index
A dimension index is an integer representing a dimension name. It enumerates the dimension names according to the order of the first occurrence of each name in the signature.

5.7.2 Details of Signature

The signature defines "core" dimensionality of input and output variables, and thereby also defines the contraction of the dimensions. The signature is represented by a string of the following format:

- Core dimensions of each input or output array are represented by a list of dimension names in parentheses, `(i_1,...,i_N)`; a scalar input/output is denoted by `()`. Instead of `i_1`, `i_2`, etc, one can use any valid Python variable name.
- Dimension lists for different arguments are separated by `","`. Input/output arguments are separated by `"->"`.
- If one uses the same dimension name in multiple locations, this enforces the same size of the corresponding dimensions.

The formal syntax of signatures is as follows:

```
<Signature>            ::= <Input arguments> "->" <Output arguments>
<Input arguments>      ::= <Argument list>
<Output arguments>     ::= <Argument list>
<Argument list>        ::= nil | <Argument> | <Argument> "," <Argument list>
<Argument>             ::= "(" <Core dimension list> ")"
<Core dimension list>  ::= nil | <Dimension name> |
                           <Dimension name> "," <Core dimension list>
<Dimension name>       ::= valid Python variable name
```

Notes:

1. All quotes are for clarity.
2. Core dimensions that share the same name must have the exact same size. Each dimension name typically corresponds to one level of looping in the elementary function's implementation.
3. White spaces are ignored.

Here are some examples of signatures:

add	`(),()->()`	
inner1d	`(i),(i)->()`	
sum1d	`(i)->()`	
dot2d	`(m,n),(n,p)->(m,p)`	matrix multiplication
outer_inner	`(i,t),(j,t)->(i,j)`	inner over the last dimension, outer over the second to last, and loop/broadcast over the rest.

5.7.3 C-API for implementing Elementary Functions

The current interface remains unchanged, and `PyUFunc_FromFuncAndData` can still be used to implement (specialized) ufuncs, consisting of scalar elementary functions.

One can use `PyUFunc_FromFuncAndDataAndSignature` to declare a more general ufunc. The argument list is the same as `PyUFunc_FromFuncAndData`, with an additional argument specifying the signature as C string.

Furthermore, the callback function is of the same type as before, `void (*foo)(char **args, intp *dimensions, intp *steps, void *func)`. When invoked, `args` is a list of length `nargs` containing the data of all input/output arguments. For a scalar elementary function, `steps` is also of length `nargs`, denoting the

strides used for the arguments. `dimensions` is a pointer to a single integer defining the size of the axis to be looped over.

For a non-trivial signature, `dimensions` will also contain the sizes of the core dimensions as well, starting at the second entry. Only one size is provided for each unique dimension name and the sizes are given according to the first occurrence of a dimension name in the signature.

The first `nargs` elements of `steps` remain the same as for scalar ufuncs. The following elements contain the strides of all core dimensions for all arguments in order.

For example, consider a ufunc with signature `(i,j),(i)->()`. In this case, `args` will contain three pointers to the data of the input/output arrays `a`, `b`, `c`. Furthermore, `dimensions` will be `[N, I, J]` to define the size of `N` of the loop and the sizes `I` and `J` for the core dimensions `i` and `j`. Finally, `steps` will be `[a_N, b_N, c_N, a_i, a_j, b_i]`, containing all necessary strides.

5.8 Numpy core libraries

New in version 1.3.0.

Starting from numpy 1.3.0, we are working on separating the pure C, "computational" code from the python dependent code. The goal is twofolds: making the code cleaner, and enabling code reuse by other extensions outside numpy (scipy, etc...).

5.8.1 Numpy core math library

The numpy core math library ('npymath') is a first step in this direction. This library contains most math-related C99 functionality, which can be used on platforms where C99 is not well supported. The core math functions have the same API as the C99 ones, except for the npy_* prefix.

The available functions are defined in <numpy/npy_math.h> - please refer to this header when in doubt.

Floating point classification

NPY_NAN
: This macro is defined to a NaN (Not a Number), and is guaranteed to have the signbit unset ('positive' NaN). The corresponding single and extension precision macro are available with the suffix F and L.

NPY_INFINITY
: This macro is defined to a positive inf. The corresponding single and extension precision macro are available with the suffix F and L.

NPY_PZERO
: This macro is defined to positive zero. The corresponding single and extension precision macro are available with the suffix F and L.

NPY_NZERO
: This macro is defined to negative zero (that is with the sign bit set). The corresponding single and extension precision macro are available with the suffix F and L.

int **npy_isnan**(x)
: This is a macro, and is equivalent to C99 isnan: works for single, double and extended precision, and return a non 0 value is x is a NaN.

int **npy_isfinite**(x)
: This is a macro, and is equivalent to C99 isfinite: works for single, double and extended precision, and return a non 0 value is x is neither a NaN nor an infinity.

int **npy_isinf**(x)
: This is a macro, and is equivalent to C99 isinf: works for single, double and extended precision, and return a non 0 value is x is infinite (positive and negative).

int **npy_signbit**(x)
: This is a macro, and is equivalent to C99 signbit: works for single, double and extended precision, and return a non 0 value is x has the signbit set (that is the number is negative).

double **npy_copysign**(double *x*, double *y*)
: This is a function equivalent to C99 copysign: return x with the same sign as y. Works for any value, including inf and nan. Single and extended precisions are available with suffix f and l.

  New in version 1.4.0.

Useful math constants

The following math constants are available in npy_math.h. Single and extended precision are also available by adding the F and L suffixes respectively.

NPY_E
: Base of natural logarithm (e)

NPY_LOG2E
: Logarithm to base 2 of the Euler constant ($\frac{\ln(e)}{\ln(2)}$)

NPY_LOG10E
: Logarithm to base 10 of the Euler constant ($\frac{\ln(e)}{\ln(10)}$)

NPY_LOGE2
: Natural logarithm of 2 ($\ln(2)$)

NPY_LOGE10
: Natural logarithm of 10 ($\ln(10)$)

NPY_PI
: Pi (π)

NPY_PI_2
: Pi divided by 2 ($\frac{\pi}{2}$)

NPY_PI_4
: Pi divided by 4 ($\frac{\pi}{4}$)

NPY_1_PI
: Reciprocal of pi ($\frac{1}{\pi}$)

NPY_2_PI
: Two times the reciprocal of pi ($\frac{2}{\pi}$)

NPY_EULER

  The Euler constant
  $\lim_{n\to\infty}(\sum_{k=1}^{n}\frac{1}{k} - \ln n)$

Low-level floating point manipulation

Those can be useful for precise floating point comparison.

double **npy_nextafter**(double *x*, double *y*)
This is a function equivalent to C99 nextafter: return next representable floating point value from x in the direction of y. Single and extended precisions are available with suffix f and l.

New in version 1.4.0.

double **npy_spacing**(double *x*)
This is a function equivalent to Fortran intrinsic. Return distance between x and next representable floating point value from x, e.g. spacing(1) == eps. spacing of nan and +/- inf return nan. Single and extended precisions are available with suffix f and l.

New in version 1.4.0.

void **npy_set_floatstatus_divbyzero**()
Set the divide by zero floating point exception

New in version 1.6.0.

void **npy_set_floatstatus_overflow**()
Set the overflow floating point exception

New in version 1.6.0.

void **npy_set_floatstatus_underflow**()
Set the underflow floating point exception

New in version 1.6.0.

void **npy_set_floatstatus_invalid**()
Set the invalid floating point exception

New in version 1.6.0.

int **npy_get_floatstatus**()
Get floating point status. Returns a bitmask with following possible flags:

- NPY_FPE_DIVIDEBYZERO
- NPY_FPE_OVERFLOW
- NPY_FPE_UNDERFLOW
- NPY_FPE_INVALID

New in version 1.9.0.

int **npy_clear_floatstatus**()
Clears the floating point status. Returns the previous status mask.

New in version 1.9.0.

Complex functions

New in version 1.4.0.

C99-like complex functions have been added. Those can be used if you wish to implement portable C extensions. Since we still support platforms without C99 complex type, you need to restrict to C90-compatible syntax, e.g.:

```
/* a = 1 + 2i \*/
npy_complex a = npy_cpack(1, 2);
npy_complex b;

b = npy_log(a);
```

Linking against the core math library in an extension

New in version 1.4.0.

To use the core math library in your own extension, you need to add the npymath compile and link options to your extension in your setup.py:

```
>>> from numpy.distutils.misc_util import get_info
>>> info = get_info('npymath')
>>> config.add_extension('foo', sources=['foo.c'], extra_info=info)
```

In other words, the usage of info is exactly the same as when using blas_info and co.

Half-precision functions

New in version 2.0.0.

The header file <numpy/halffloat.h> provides functions to work with IEEE 754-2008 16-bit floating point values. While this format is not typically used for numerical computations, it is useful for storing values which require floating point but do not need much precision. It can also be used as an educational tool to understand the nature of floating point round-off error.

Like for other types, NumPy includes a typedef npy_half for the 16 bit float. Unlike for most of the other types, you cannot use this as a normal type in C, since is is a typedef for npy_uint16. For example, 1.0 looks like 0x3c00 to C, and if you do an equality comparison between the different signed zeros, you will get -0.0 != 0.0 (0x8000 != 0x0000), which is incorrect.

For these reasons, NumPy provides an API to work with npy_half values accessible by including <numpy/halffloat.h> and linking to 'npymath'. For functions that are not provided directly, such as the arithmetic operations, the preferred method is to convert to float or double and back again, as in the following example.

```
npy_half sum(int n, npy_half *array) {
    float ret = 0;
    while(n--) {
        ret += npy_half_to_float(*array++);
    }
    return npy_float_to_half(ret);
}
```

External Links:

- 754-2008 IEEE Standard for Floating-Point Arithmetic
- Half-precision Float Wikipedia Article.
- OpenGL Half Float Pixel Support
- The OpenEXR image format.

NPY_HALF_ZERO
: This macro is defined to positive zero.

NPY_HALF_PZERO
: This macro is defined to positive zero.

NPY_HALF_NZERO
: This macro is defined to negative zero.

NPY_HALF_ONE
: This macro is defined to 1.0.

NPY_HALF_NEGONE
This macro is defined to -1.0.

NPY_HALF_PINF
This macro is defined to +inf.

NPY_HALF_NINF
This macro is defined to -inf.

NPY_HALF_NAN
This macro is defined to a NaN value, guaranteed to have its sign bit unset.

float **npy_half_to_float**(npy_half *h*)
Converts a half-precision float to a single-precision float.

double **npy_half_to_double**(npy_half *h*)
Converts a half-precision float to a double-precision float.

npy_half **npy_float_to_half**(float *f*)
Converts a single-precision float to a half-precision float. The value is rounded to the nearest representable half, with ties going to the nearest even. If the value is too small or too big, the system's floating point underflow or overflow bit will be set.

npy_half **npy_double_to_half**(double *d*)
Converts a double-precision float to a half-precision float. The value is rounded to the nearest representable half, with ties going to the nearest even. If the value is too small or too big, the system's floating point underflow or overflow bit will be set.

int **npy_half_eq**(npy_half *h1*, npy_half *h2*)
Compares two half-precision floats (h1 == h2).

int **npy_half_ne**(npy_half *h1*, npy_half *h2*)
Compares two half-precision floats (h1 != h2).

int **npy_half_le**(npy_half *h1*, npy_half *h2*)
Compares two half-precision floats (h1 <= h2).

int **npy_half_lt**(npy_half *h1*, npy_half *h2*)
Compares two half-precision floats (h1 < h2).

int **npy_half_ge**(npy_half *h1*, npy_half *h2*)
Compares two half-precision floats (h1 >= h2).

int **npy_half_gt**(npy_half *h1*, npy_half *h2*)
Compares two half-precision floats (h1 > h2).

int **npy_half_eq_nonan**(npy_half *h1*, npy_half *h2*)
Compares two half-precision floats that are known to not be NaN (h1 == h2). If a value is NaN, the result is undefined.

int **npy_half_lt_nonan**(npy_half *h1*, npy_half *h2*)
Compares two half-precision floats that are known to not be NaN (h1 < h2). If a value is NaN, the result is undefined.

int **npy_half_le_nonan**(npy_half *h1*, npy_half *h2*)
Compares two half-precision floats that are known to not be NaN (h1 <= h2). If a value is NaN, the result is undefined.

int **npy_half_iszero**(npy_half *h*)
Tests whether the half-precision float has a value equal to zero. This may be slightly faster than calling npy_half_eq(h, NPY_ZERO).

int **npy_half_isnan**(npy_half *h*)
Tests whether the half-precision float is a NaN.

int **npy_half_isinf**(npy_half *h*)
Tests whether the half-precision float is plus or minus Inf.

int **npy_half_isfinite**(npy_half *h*)
Tests whether the half-precision float is finite (not NaN or Inf).

int **npy_half_signbit**(npy_half *h*)
Returns 1 is h is negative, 0 otherwise.

npy_half **npy_half_copysign**(npy_half *x*, npy_half *y*)
Returns the value of x with the sign bit copied from y. Works for any value, including Inf and NaN.

npy_half **npy_half_spacing**(npy_half *h*)
This is the same for half-precision float as npy_spacing and npy_spacingf described in the low-level floating point section.

npy_half **npy_half_nextafter**(npy_half *x*, npy_half *y*)
This is the same for half-precision float as npy_nextafter and npy_nextafterf described in the low-level floating point section.

npy_uint16 **npy_floatbits_to_halfbits**(npy_uint32 *f*)
Low-level function which converts a 32-bit single-precision float, stored as a uint32, into a 16-bit half-precision float.

npy_uint16 **npy_doublebits_to_halfbits**(npy_uint64 *d*)
Low-level function which converts a 64-bit double-precision float, stored as a uint64, into a 16-bit half-precision float.

npy_uint32 **npy_halfbits_to_floatbits**(npy_uint16 *h*)
Low-level function which converts a 16-bit half-precision float into a 32-bit single-precision float, stored as a uint32.

npy_uint64 **npy_halfbits_to_doublebits**(npy_uint16 *h*)
Low-level function which converts a 16-bit half-precision float into a 64-bit double-precision float, stored as a uint64.

5.9 C API Deprecations

5.9.1 Background

The API exposed by NumPy for third-party extensions has grown over years of releases, and has allowed programmers to directly access NumPy functionality from C. This API can be best described as "organic". It has emerged from multiple competing desires and from multiple points of view over the years, strongly influenced by the desire to make it easy for users to move to NumPy from Numeric and Numarray. The core API originated with Numeric in 1995 and there are patterns such as the heavy use of macros written to mimic Python's C-API as well as account for compiler technology of the late 90's. There is also only a small group of volunteers who have had very little time to spend on improving this API.

There is an ongoing effort to improve the API. It is important in this effort to ensure that code that compiles for NumPy 1.X continues to compile for NumPy 1.X. At the same time, certain API's will be marked as deprecated so that future-looking code can avoid these API's and follow better practices.

Another important role played by deprecation markings in the C API is to move towards hiding internal details of the NumPy implementation. For those needing direct, easy, access to the data of ndarrays, this will not remove this ability. Rather, there are many potential performance optimizations which require changing the implementation details, and

NumPy developers have been unable to try them because of the high value of preserving ABI compatibility. By deprecating this direct access, we will in the future be able to improve NumPy's performance in ways we cannot presently.

5.9.2 Deprecation Mechanism NPY_NO_DEPRECATED_API

In C, there is no equivalent to the deprecation warnings that Python supports. One way to do deprecations is to flag them in the documentation and release notes, then remove or change the deprecated features in a future major version (NumPy 2.0 and beyond). Minor versions of NumPy should not have major C-API changes, however, that prevent code that worked on a previous minor release. For example, we will do our best to ensure that code that compiled and worked on NumPy 1.4 should continue to work on NumPy 1.7 (but perhaps with compiler warnings).

To use the NPY_NO_DEPRECATED_API mechanism, you need to #define it to the target API version of NumPy before #including any NumPy headers. If you want to confirm that your code is clean against 1.7, use:

```
#define NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION
```

On compilers which support a #warning mechanism, NumPy issues a compiler warning if you do not define the symbol NPY_NO_DEPRECATED_API. This way, the fact that there are deprecations will be flagged for third-party developers who may not have read the release notes closely.

CHAPTER

SIX

NUMPY INTERNALS

6.1 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 Chapter 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.

6.1.1 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 data-type array. Even if you have a contiguous chunk of memory, you cannot just assume it is safe to dereference a data- type-specific pointer to an element. Only if the *NPY_ARRAY_ALIGNED* flag is set is this 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).

6.1.2 Data-type encapsulation

The data-type is an important abstraction of the ndarray. Operations will look to the data-type 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 data-types 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 by-pass this mechanism, but the point of the data- type abstraction is to allow new data-types to be added.

One of the built-in data-types, the void data-type allows for arbitrary structured types containing 1 or more fields as elements of the array. A field is simply another data-type object along with an offset into the current structured type. In order to support arbitrarily nested fields, several recursive implementations of data-type 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 data-type object stored at the given offset. These offsets can be arbitrary numbers. Therefore, the possibility of encountering mis- aligned data must be recognized and taken into account if necessary.

6.1.3 N-D Iterators

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* (it) 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 then 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 by-passed. 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.

6.1.4 Broadcasting

In Numeric, 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 `PyMultiIter_New` 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 broad-casting rules.

6.1.5 Array Scalars

The array scalars offer a hierarchy of Python types that allow a one- to-one correspondence between the data-type 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`).

6.1.6 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 (fancy)
- boolean (single boolean array); if there is more than one boolean array as 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 which 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 to restart 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 is relatively straight forward, 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.

6.1.7 Universal Functions

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 and 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 data-types of the inputs against the available signatures. The signatures corresponding to built-in types are stored in the 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 data-type 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, mis-aligned, 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 mis-aligned) and is smaller than the buffer size, then a new output array is constructed with the special UPDATEIFCOPY flag set so that when it is DECREF'd on completion of the function, it's 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 mis-aligned or incorrect data type situations). Depending on which execution method is called for, the loop is then setup and computed.

Function call

This section describes how the basic universal function computation loop is setup 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 byte-order) 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 data-type (including being byte-swapped) 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 (mis-aligned and/or wrong data-type)
- 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 it's 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

Their are three methods of ufuncs that require calculation similar to the general-purpose ufuncs. These are reduce, accumulate, and 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 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 data-type. The buffer size and error handling is then retrieved from (per-thread) global storage. For small arrays that are mis-aligned or have incorrect data-type, 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 mis-aligned and has the correct data-type, 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 it's 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 UPDATEIFCOPY 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 the method computation is complete. 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

$$\begin{aligned} o &= i[0] \\ o &= i[k]\text{<op>}o \quad k = 1 \ldots N \end{aligned}$$

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 operations 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 data-type (including byte-order). 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 function is very similar to the reduce function 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

$$\begin{aligned} o[0] &= i[0] \\ o[k] &= i[k]\text{<op>}o[k-1] \quad k = 1 \ldots N. \end{aligned}$$

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.

Mis-aligned or a loop data-type that does not match the input and/or output data-type is handled using buffered code where-in data is copied to a temporary buffer and cast to the correct data-type 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.

6.2 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 referred 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 a 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 interpreted as C-order or Fortran-order.

This arrangement allow for very flexible use of arrays. One thing that it allows is simple changes of 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 slices. 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 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 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.

6.3 Multidimensional Array Indexing Order Issues

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 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, which 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 .flat with an 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 the next largest, and the last gets the smallest element). Since image data are normally stored by rows, this corresponds to 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.

CHAPTER

SEVEN

NUMPY AND SWIG

7.1 Numpy.i: a SWIG Interface File for NumPy

7.1.1 Introduction

The Simple Wrapper and Interface Generator (or SWIG) is a powerful tool for generating wrapper code for interfacing to a wide variety of scripting languages. SWIG can parse header files, and using only the code prototypes, create an interface to the target language. But SWIG is not omnipotent. For example, it cannot know from the prototype:

```
double rms(double* seq, int n);
```

what exactly `seq` is. Is it a single value to be altered in-place? Is it an array, and if so what is its length? Is it input-only? Output-only? Input-output? SWIG cannot determine these details, and does not attempt to do so.

If we designed `rms`, we probably made it a routine that takes an input-only array of length `n` of `double` values called `seq` and returns the root mean square. The default behavior of SWIG, however, will be to create a wrapper function that compiles, but is nearly impossible to use from the scripting language in the way the C routine was intended.

For Python, the preferred way of handling contiguous (or technically, *strided*) blocks of homogeneous data is with NumPy, which provides full object-oriented access to multidimensial arrays of data. Therefore, the most logical Python interface for the `rms` function would be (including doc string):

```
def rms(seq):
    """
    rms: return the root mean square of a sequence
    rms(numpy.ndarray) -> double
    rms(list) -> double
    rms(tuple) -> double
    """
```

where `seq` would be a NumPy array of `double` values, and its length `n` would be extracted from `seq` internally before being passed to the C routine. Even better, since NumPy supports construction of arrays from arbitrary Python sequences, `seq` itself could be a nearly arbitrary sequence (so long as each element can be converted to a `double`) and the wrapper code would internally convert it to a NumPy array before extracting its data and length.

SWIG allows these types of conversions to be defined via a mechanism called *typemaps*. This document provides information on how to use `numpy.i`, a SWIG interface file that defines a series of typemaps intended to make the type of array-related conversions described above relatively simple to implement. For example, suppose that the `rms` function prototype defined above was in a header file named `rms.h`. To obtain the Python interface discussed above, your SWIG interface file would need the following:

```
%{
#define SWIG_FILE_WITH_INIT
#include "rms.h"
%}
```

```
%include "numpy.i"

%init %{
import_array();
%}

%apply (double* IN_ARRAY1, int DIM1) {(double* seq, int n)};
%include "rms.h"
```

Typemaps are keyed off a list of one or more function arguments, either by type or by type and name. We will refer to such lists as *signatures*. One of the many typemaps defined by `numpy.i` is used above and has the signature `(double* IN_ARRAY1, int DIM1)`. The argument names are intended to suggest that the `double*` argument is an input array of one dimension and that the `int` represents the size of that dimension. This is precisely the pattern in the `rms` prototype.

Most likely, no actual prototypes to be wrapped will have the argument names `IN_ARRAY1` and `DIM1`. We use the SWIG `%apply` directive to apply the typemap for one-dimensional input arrays of type `double` to the actual prototype used by `rms`. Using `numpy.i` effectively, therefore, requires knowing what typemaps are available and what they do.

A SWIG interface file that includes the SWIG directives given above will produce wrapper code that looks something like:

```
 1 PyObject *_wrap_rms(PyObject *args) {
 2   PyObject *resultobj = 0;
 3   double *arg1 = (double *) 0 ;
 4   int arg2 ;
 5   double result;
 6   PyArrayObject *array1 = NULL ;
 7   int is_new_object1 = 0 ;
 8   PyObject * obj0 = 0 ;
 9
10   if (!PyArg_ParseTuple(args,(char *)"O:rms",&obj0)) SWIG_fail;
11   {
12     array1 = obj_to_array_contiguous_allow_conversion(
13                  obj0, NPY_DOUBLE, &is_new_object1);
14     npy_intp size[1] = {
15       -1
16     };
17     if (!array1 || !require_dimensions(array1, 1) ||
18         !require_size(array1, size, 1)) SWIG_fail;
19     arg1 = (double*) array1->data;
20     arg2 = (int) array1->dimensions[0];
21   }
22   result = (double)rms(arg1,arg2);
23   resultobj = SWIG_From_double((double)(result));
24   {
25     if (is_new_object1 && array1) Py_DECREF(array1);
26   }
27   return resultobj;
28 fail:
29   {
30     if (is_new_object1 && array1) Py_DECREF(array1);
31   }
32   return NULL;
33 }
```

The typemaps from `numpy.i` are responsible for the following lines of code: 12–20, 25 and 30. Line 10 parses

the input to the `rms` function. From the format string `"O:rms"`, we can see that the argument list is expected to be a single Python object (specified by the `O` before the colon) and whose pointer is stored in `obj0`. A number of functions, supplied by `numpy.i`, are called to make and check the (possible) conversion from a generic Python object to a NumPy array. These functions are explained in the section *Helper Functions*, but hopefully their names are self-explanatory. At line 12 we use `obj0` to construct a NumPy array. At line 17, we check the validity of the result: that it is non-null and that it has a single dimension of arbitrary length. Once these states are verified, we extract the data buffer and length in lines 19 and 20 so that we can call the underlying C function at line 22. Line 25 performs memory management for the case where we have created a new array that is no longer needed.

This code has a significant amount of error handling. Note the `SWIG_fail` is a macro for `goto fail`, refering to the label at line 28. If the user provides the wrong number of arguments, this will be caught at line 10. If construction of the NumPy array fails or produces an array with the wrong number of dimensions, these errors are caught at line 17. And finally, if an error is detected, memory is still managed correctly at line 30.

Note that if the C function signature was in a different order:

```
double rms(int n, double* seq);
```

that SWIG would not match the typemap signature given above with the argument list for `rms`. Fortunately, `numpy.i` has a set of typemaps with the data pointer given last:

```
%apply (int DIM1, double* IN_ARRAY1) {(int n, double* seq)};
```

This simply has the effect of switching the definitions of `arg1` and `arg2` in lines 3 and 4 of the generated code above, and their assignments in lines 19 and 20.

7.1.2 Using numpy.i

The `numpy.i` file is currently located in the `tools/swig` sub-directory under the `numpy` installation directory. Typically, you will want to copy it to the directory where you are developing your wrappers.

A simple module that only uses a single SWIG interface file should include the following:

```
%{
#define SWIG_FILE_WITH_INIT
%}
%include "numpy.i"
%init %{
import_array();
%}
```

Within a compiled Python module, `import_array()` should only get called once. This could be in a C/C++ file that you have written and is linked to the module. If this is the case, then none of your interface files should `#define SWIG_FILE_WITH_INIT` or call `import_array()`. Or, this initialization call could be in a wrapper file generated by SWIG from an interface file that has the `%init` block as above. If this is the case, and you have more than one SWIG interface file, then only one interface file should `#define SWIG_FILE_WITH_INIT` and call `import_array()`.

7.1.3 Available Typemaps

The typemap directives provided by `numpy.i` for arrays of different data types, say `double` and `int`, and dimensions of different types, say `int` or `long`, are identical to one another except for the C and NumPy type specifications. The typemaps are therefore implemented (typically behind the scenes) via a macro:

```
%numpy_typemaps(DATA_TYPE, DATA_TYPECODE, DIM_TYPE)
```

that can be invoked for appropriate `(DATA_TYPE, DATA_TYPECODE, DIM_TYPE)` triplets. For example:

```
%numpy_typemaps(double, NPY_DOUBLE, int)
%numpy_typemaps(int,    NPY_INT   , int)
```

The `numpy.i` interface file uses the `%numpy_typemaps` macro to implement typemaps for the following C data types and `int` dimension types:

- `signed char`
- `unsigned char`
- `short`
- `unsigned short`
- `int`
- `unsigned int`
- `long`
- `unsigned long`
- `long long`
- `unsigned long long`
- `float`
- `double`

In the following descriptions, we reference a generic `DATA_TYPE`, which could be any of the C data types listed above, and `DIM_TYPE` which should be one of the many types of integers.

The typemap signatures are largely differentiated on the name given to the buffer pointer. Names with `FARRAY` are for Fortran-ordered arrays, and names with `ARRAY` are for C-ordered (or 1D arrays).

Input Arrays

Input arrays are defined as arrays of data that are passed into a routine but are not altered in-place or returned to the user. The Python input array is therefore allowed to be almost any Python sequence (such as a list) that can be converted to the requested type of array. The input array signatures are

1D:

- `( DATA_TYPE IN_ARRAY1[ANY] )`
- `( DATA_TYPE* IN_ARRAY1, int DIM1 )`
- `( int DIM1, DATA_TYPE* IN_ARRAY1 )`

2D:

- `( DATA_TYPE IN_ARRAY2[ANY][ANY] )`
- `( DATA_TYPE* IN_ARRAY2, int DIM1, int DIM2 )`
- `( int DIM1, int DIM2, DATA_TYPE* IN_ARRAY2 )`
- `( DATA_TYPE* IN_FARRAY2, int DIM1, int DIM2 )`
- `( int DIM1, int DIM2, DATA_TYPE* IN_FARRAY2 )`

3D:

- `( DATA_TYPE IN_ARRAY3[ANY][ANY][ANY] )`

- (DATA_TYPE* IN_ARRAY3, int DIM1, int DIM2, int DIM3)
- (int DIM1, int DIM2, int DIM3, DATA_TYPE* IN_ARRAY3)
- (DATA_TYPE* IN_FARRAY3, int DIM1, int DIM2, int DIM3)
- (int DIM1, int DIM2, int DIM3, DATA_TYPE* IN_FARRAY3)

4D:

- (DATA_TYPE IN_ARRAY4[ANY][ANY][ANY][ANY])
- (DATA_TYPE* IN_ARRAY4, DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4)
- (DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, , DIM_TYPE DIM4, DATA_TYPE* IN_ARRAY4)
- (DATA_TYPE* IN_FARRAY4, DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4)
- (DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4, DATA_TYPE* IN_FARRAY4)

The first signature listed, `( DATA_TYPE IN_ARRAY[ANY] )` is for one-dimensional arrays with hard-coded dimensions. Likewise, `( DATA_TYPE IN_ARRAY2[ANY][ANY] )` is for two-dimensional arrays with hard-coded dimensions, and similarly for three-dimensional.

In-Place Arrays

In-place arrays are defined as arrays that are modified in-place. The input values may or may not be used, but the values at the time the function returns are significant. The provided Python argument must therefore be a NumPy array of the required type. The in-place signatures are

1D:

- (DATA_TYPE INPLACE_ARRAY1[ANY])
- (DATA_TYPE* INPLACE_ARRAY1, int DIM1)
- (int DIM1, DATA_TYPE* INPLACE_ARRAY1)

2D:

- (DATA_TYPE INPLACE_ARRAY2[ANY][ANY])
- (DATA_TYPE* INPLACE_ARRAY2, int DIM1, int DIM2)
- (int DIM1, int DIM2, DATA_TYPE* INPLACE_ARRAY2)
- (DATA_TYPE* INPLACE_FARRAY2, int DIM1, int DIM2)
- (int DIM1, int DIM2, DATA_TYPE* INPLACE_FARRAY2)

3D:

- (DATA_TYPE INPLACE_ARRAY3[ANY][ANY][ANY])
- (DATA_TYPE* INPLACE_ARRAY3, int DIM1, int DIM2, int DIM3)
- (int DIM1, int DIM2, int DIM3, DATA_TYPE* INPLACE_ARRAY3)
- (DATA_TYPE* INPLACE_FARRAY3, int DIM1, int DIM2, int DIM3)
- (int DIM1, int DIM2, int DIM3, DATA_TYPE* INPLACE_FARRAY3)

4D:

- (DATA_TYPE INPLACE_ARRAY4[ANY][ANY][ANY][ANY])
- (DATA_TYPE* INPLACE_ARRAY4, DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4)
- (DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, , DIM_TYPE DIM4, DATA_TYPE* INPLACE_ARRAY4)
- (DATA_TYPE* INPLACE_FARRAY4, DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4)
- (DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4, DATA_TYPE* INPLACE_FARRAY4)

These typemaps now check to make sure that the INPLACE_ARRAY arguments use native byte ordering. If not, an exception is raised.

There is also a "flat" in-place array for situations in which you would like to modify or process each element, regardless of the number of dimensions. One example is a "quantization" function that quantizes each element of an array in-place, be it 1D, 2D or whatever. This form checks for continuity but allows either C or Fortran ordering.

ND:

- (DATA_TYPE* INPLACE_ARRAY_FLAT, DIM_TYPE DIM_FLAT)

Argout Arrays

Argout arrays are arrays that appear in the input arguments in C, but are in fact output arrays. This pattern occurs often when there is more than one output variable and the single return argument is therefore not sufficient. In Python, the convential way to return multiple arguments is to pack them into a sequence (tuple, list, etc.) and return the sequence. This is what the argout typemaps do. If a wrapped function that uses these argout typemaps has more than one return argument, they are packed into a tuple or list, depending on the version of Python. The Python user does not pass these arrays in, they simply get returned. For the case where a dimension is specified, the python user must provide that dimension as an argument. The argout signatures are

1D:

- (DATA_TYPE ARGOUT_ARRAY1[ANY])
- (DATA_TYPE* ARGOUT_ARRAY1, int DIM1)
- (int DIM1, DATA_TYPE* ARGOUT_ARRAY1)

2D:

- (DATA_TYPE ARGOUT_ARRAY2[ANY][ANY])

3D:

- (DATA_TYPE ARGOUT_ARRAY3[ANY][ANY][ANY])

4D:

- (DATA_TYPE ARGOUT_ARRAY4[ANY][ANY][ANY][ANY])

These are typically used in situations where in C/C++, you would allocate a(n) array(s) on the heap, and call the function to fill the array(s) values. In Python, the arrays are allocated for you and returned as new array objects.

Note that we support DATA_TYPE* argout typemaps in 1D, but not 2D or 3D. This is because of a quirk with the SWIG typemap syntax and cannot be avoided. Note that for these types of 1D typemaps, the Python function will take a single argument representing DIM1.

Argout View Arrays

Argoutview arrays are for when your C code provides you with a view of its internal data and does not require any memory to be allocated by the user. This can be dangerous. There is almost no way to guarantee that the internal data from the C code will remain in existence for the entire lifetime of the NumPy array that encapsulates it. If the user destroys the object that provides the view of the data before destroying the NumPy array, then using that array may result in bad memory references or segmentation faults. Nevertheless, there are situations, working with large data sets, where you simply have no other choice.

The C code to be wrapped for argoutview arrays are characterized by pointers: pointers to the dimensions and double pointers to the data, so that these values can be passed back to the user. The argoutview typemap signatures are therefore

1D:

- `( DATA_TYPE** ARGOUTVIEW_ARRAY1, DIM_TYPE* DIM1 )`
- `( DIM_TYPE* DIM1, DATA_TYPE** ARGOUTVIEW_ARRAY1 )`

2D:

- `( DATA_TYPE** ARGOUTVIEW_ARRAY2, DIM_TYPE* DIM1, DIM_TYPE* DIM2 )`
- `( DIM_TYPE* DIM1, DIM_TYPE* DIM2, DATA_TYPE** ARGOUTVIEW_ARRAY2 )`
- `( DATA_TYPE** ARGOUTVIEW_FARRAY2, DIM_TYPE* DIM1, DIM_TYPE* DIM2 )`
- `( DIM_TYPE* DIM1, DIM_TYPE* DIM2, DATA_TYPE** ARGOUTVIEW_FARRAY2 )`

3D:

- `( DATA_TYPE** ARGOUTVIEW_ARRAY3, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3)`
- `( DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DATA_TYPE** ARGOUTVIEW_ARRAY3)`
- `( DATA_TYPE** ARGOUTVIEW_FARRAY3, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3)`
- `( DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DATA_TYPE** ARGOUTVIEW_FARRAY3)`

4D:

- `(DATA_TYPE** ARGOUTVIEW_ARRAY4, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4, DATA_TYPE** ARGOUTVIEW_ARRAY4)`
- `(DATA_TYPE** ARGOUTVIEW_FARRAY4, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4, DATA_TYPE** ARGOUTVIEW_FARRAY4)`

Note that arrays with hard-coded dimensions are not supported. These cannot follow the double pointer signatures of these typemaps.

Memory Managed Argout View Arrays

A recent addition to `numpy.i` are typemaps that permit argout arrays with views into memory that is managed. See the discussion here.

1D:

- `(DATA_TYPE** ARGOUTVIEWM_ARRAY1, DIM_TYPE* DIM1)`
- `(DIM_TYPE* DIM1, DATA_TYPE** ARGOUTVIEWM_ARRAY1)`

2D:

- `(DATA_TYPE** ARGOUTVIEWM_ARRAY2, DIM_TYPE* DIM1, DIM_TYPE* DIM2)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DATA_TYPE** ARGOUTVIEWM_ARRAY2)`
- `(DATA_TYPE** ARGOUTVIEWM_FARRAY2, DIM_TYPE* DIM1, DIM_TYPE* DIM2)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DATA_TYPE** ARGOUTVIEWM_FARRAY2)`

3D:

- `(DATA_TYPE** ARGOUTVIEWM_ARRAY3, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DATA_TYPE** ARGOUTVIEWM_ARRAY3)`
- `(DATA_TYPE** ARGOUTVIEWM_FARRAY3, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DATA_TYPE** ARGOUTVIEWM_FARRAY3)`

4D:

- `(DATA_TYPE** ARGOUTVIEWM_ARRAY4, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4, DATA_TYPE** ARGOUTVIEWM_ARRAY4)`
- `(DATA_TYPE** ARGOUTVIEWM_FARRAY4, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4, DATA_TYPE** ARGOUTVIEWM_FARRAY4)`

Output Arrays

The `numpy.i` interface file does not support typemaps for output arrays, for several reasons. First, C/C++ return arguments are limited to a single value. This prevents obtaining dimension information in a general way. Second, arrays with hard-coded lengths are not permitted as return arguments. In other words:

```
double[3] newVector(double x, double y, double z);
```

is not legal C/C++ syntax. Therefore, we cannot provide typemaps of the form:

```
%typemap(out) (TYPE[ANY]);
```

If you run into a situation where a function or method is returning a pointer to an array, your best bet is to write your own version of the function to be wrapped, either with `%extend` for the case of class methods or `%ignore` and `%rename` for the case of functions.

Other Common Types: bool

Note that C++ type `bool` is not supported in the list in the *Available Typemaps* section. NumPy bools are a single byte, while the C++ `bool` is four bytes (at least on my system). Therefore:

```
%numpy_typemaps(bool, NPY_BOOL, int)
```

will result in typemaps that will produce code that reference improper data lengths. You can implement the following macro expansion:

```
%numpy_typemaps(bool, NPY_UINT, int)
```

to fix the data length problem, and *Input Arrays* will work fine, but *In-Place Arrays* might fail type-checking.

Other Common Types: complex

Typemap conversions for complex floating-point types is also not supported automatically. This is because Python and NumPy are written in C, which does not have native complex types. Both Python and NumPy implement their own (essentially equivalent) `struct` definitions for complex variables:

```
/* Python */
typedef struct {double real; double imag;} Py_complex;

/* NumPy */
typedef struct {float  real, imag;} npy_cfloat;
typedef struct {double real, imag;} npy_cdouble;
```

We could have implemented:

```
%numpy_typemaps(Py_complex , NPY_CDOUBLE, int)
%numpy_typemaps(npy_cfloat , NPY_CFLOAT , int)
%numpy_typemaps(npy_cdouble, NPY_CDOUBLE, int)
```

which would have provided automatic type conversions for arrays of type `Py_complex`, `npy_cfloat` and `npy_cdouble`. However, it seemed unlikely that there would be any independent (non-Python, non-NumPy) application code that people would be using SWIG to generate a Python interface to, that also used these definitions for complex types. More likely, these application codes will define their own complex types, or in the case of C++, use `std::complex`. Assuming these data structures are compatible with Python and NumPy complex types, `%numpy_typemap` expansions as above (with the user's complex type substituted for the first argument) should work.

7.1.4 NumPy Array Scalars and SWIG

SWIG has sophisticated type checking for numerical types. For example, if your C/C++ routine expects an integer as input, the code generated by SWIG will check for both Python integers and Python long integers, and raise an overflow error if the provided Python integer is too big to cast down to a C integer. With the introduction of NumPy scalar arrays into your Python code, you might conceivably extract an integer from a NumPy array and attempt to pass this to a SWIG-wrapped C/C++ function that expects an `int`, but the SWIG type checking will not recognize the NumPy array scalar as an integer. (Often, this does in fact work – it depends on whether NumPy recognizes the integer type you are using as inheriting from the Python integer type on the platform you are using. Sometimes, this means that code that works on a 32-bit machine will fail on a 64-bit machine.)

If you get a Python error that looks like the following:

```
TypeError: in method 'MyClass_MyMethod', argument 2 of type 'int'
```

and the argument you are passing is an integer extracted from a NumPy array, then you have stumbled upon this problem. The solution is to modify the SWIG type conversion system to accept Numpy array scalars in addition to the standard integer types. Fortunately, this capabilitiy has been provided for you. Simply copy the file:

```
pyfragments.swg
```

to the working build directory for you project, and this problem will be fixed. It is suggested that you do this anyway, as it only increases the capabilities of your Python interface.

Why is There a Second File?

The SWIG type checking and conversion system is a complicated combination of C macros, SWIG macros, SWIG typemaps and SWIG fragments. Fragments are a way to conditionally insert code into your wrapper file if it is needed, and not insert it if not needed. If multiple typemaps require the same fragment, the fragment only gets inserted into your wrapper code once.

There is a fragment for converting a Python integer to a C `long`. There is a different fragment that converts a Python integer to a C `int`, that calls the rountine defined in the `long` fragment. We can make the changes we want here by changing the definition for the `long` fragment. SWIG determines the active definition for a fragment using a "first come, first served" system. That is, we need to define the fragment for `long` conversions prior to SWIG doing it internally. SWIG allows us to do this by putting our fragment definitions in the file `pyfragments.swg`. If we were to put the new fragment definitions in `numpy.i`, they would be ignored.

7.1.5 Helper Functions

The `numpy.i` file containes several macros and routines that it uses internally to build its typemaps. However, these functions may be useful elsewhere in your interface file. These macros and routines are implemented as fragments, which are described briefly in the previous section. If you try to use one or more of the following macros or functions, but your compiler complains that it does not recognize the symbol, then you need to force these fragments to appear in your code using:

```
%fragment("NumPy_Fragments");
```

in your SWIG interface file.

Macros

is_array(a)
: Evaluates as true if `a` is non-`NULL` and can be cast to a `PyArrayObject*`.

array_type(a)
: Evaluates to the integer data type code of `a`, assuming `a` can be cast to a `PyArrayObject*`.

array_numdims(a)
: Evaluates to the integer number of dimensions of `a`, assuming `a` can be cast to a `PyArrayObject*`.

array_dimensions(a)
: Evaluates to an array of type `npy_intp` and length `array_numdims(a)`, giving the lengths of all of the dimensions of `a`, assuming `a` can be cast to a `PyArrayObject*`.

array_size(a,i)
: Evaluates to the `i`-th dimension size of `a`, assuming `a` can be cast to a `PyArrayObject*`.

array_strides(a)
Evaluates to an array of type `npy_intp` and length `array_numdims(a)`, giving the stridess of all of the dimensions of `a`, assuming `a` can be cast to a `PyArrayObject*`. A stride is the distance in bytes between an element and its immediate neighbor along the same axis.

array_stride(a,i)
Evaluates to the `i`-th stride of `a`, assuming `a` can be cast to a `PyArrayObject*`.

array_data(a)
Evaluates to a pointer of type `void*` that points to the data buffer of `a`, assuming `a` can be cast to a `PyArrayObject*`.

array_descr(a)
Returns a borrowed reference to the dtype property (`PyArray_Descr*`) of `a`, assuming `a` can be cast to a `PyArrayObject*`.

array_flags(a)
Returns an integer representing the flags of `a`, assuming `a` can be cast to a `PyArrayObject*`.

array_enableflags(a,f)
Sets the flag represented by `f` of `a`, assuming `a` can be cast to a `PyArrayObject*`.

array_is_contiguous(a)
Evaluates as true if `a` is a contiguous array. Equivalent to `(PyArray_ISCONTIGUOUS(a))`.

array_is_native(a)
Evaluates as true if the data buffer of `a` uses native byte order. Equivalent to `(PyArray_ISNOTSWAPPED(a))`.

array_is_fortran(a)
Evaluates as true if `a` is FORTRAN ordered.

Routines

pytype_string()

Return type: `const char*`

Arguments:

- `PyObject* py_obj`, a general Python object.

Return a string describing the type of `py_obj`.

typecode_string()

Return type: `const char*`

Arguments:

- `int typecode`, a NumPy integer typecode.

Return a string describing the type corresponding to the NumPy `typecode`.

type_match()

Return type: `int`

Arguments:

- `int actual_type`, the NumPy typecode of a NumPy array.
- `int desired_type`, the desired NumPy typecode.

Make sure that `actual_type` is compatible with `desired_type`. For example, this allows character and byte types, or int and long types, to match. This is now equivalent to `PyArray_EquivTypenums()`.

obj_to_array_no_conversion()

Return type: `PyArrayObject*`

Arguments:

- `PyObject* input`, a general Python object.
- `int typecode`, the desired NumPy typecode.

Cast `input` to a `PyArrayObject*` if legal, and ensure that it is of type `typecode`. If `input` cannot be cast, or the `typecode` is wrong, set a Python error and return `NULL`.

obj_to_array_allow_conversion()

Return type: `PyArrayObject*`

Arguments:

- `PyObject* input`, a general Python object.
- `int typecode`, the desired NumPy typecode of the resulting array.
- `int* is_new_object`, returns a value of 0 if no conversion performed, else 1.

Convert `input` to a NumPy array with the given `typecode`. On success, return a valid `PyArrayObject*` with the correct type. On failure, the Python error string will be set and the routine returns `NULL`.

make_contiguous()

Return type: `PyArrayObject*`

Arguments:

- `PyArrayObject* ary`, a NumPy array.
- `int* is_new_object`, returns a value of 0 if no conversion performed, else 1.
- `int min_dims`, minimum allowable dimensions.
- `int max_dims`, maximum allowable dimensions.

Check to see if `ary` is contiguous. If so, return the input pointer and flag it as not a new object. If it is not contiguous, create a new `PyArrayObject*` using the original data, flag it as a new object and return the pointer.

make_fortran()

Return type: `PyArrayObject*`

Arguments

- `PyArrayObject* ary`, a NumPy array.
- `int* is_new_object`, returns a value of 0 if no conversion performed, else 1.

Check to see if `ary` is Fortran contiguous. If so, return the input pointer and flag it as not a new object. If it is not Fortran contiguous, create a new `PyArrayObject*` using the original data, flag it as a new object and return the pointer.

obj_to_array_contiguous_allow_conversion()

Return type: `PyArrayObject*`

Arguments:

- `PyObject* input`, a general Python object.
- `int typecode`, the desired NumPy typecode of the resulting array.
- `int* is_new_object`, returns a value of 0 if no conversion performed, else 1.

Convert `input` to a contiguous `PyArrayObject*` of the specified type. If the input object is not a contiguous `PyArrayObject*`, a new one will be created and the new object flag will be set.

obj_to_array_fortran_allow_conversion()

Return type: `PyArrayObject*`

Arguments:

- `PyObject* input`, a general Python object.
- `int typecode`, the desired NumPy typecode of the resulting array.
- `int* is_new_object`, returns a value of 0 if no conversion performed, else 1.

Convert `input` to a Fortran contiguous `PyArrayObject*` of the specified type. If the input object is not a Fortran contiguous `PyArrayObject*`, a new one will be created and the new object flag will be set.

require_contiguous()

Return type: `int`

Arguments:

- `PyArrayObject* ary`, a NumPy array.

Test whether `ary` is contiguous. If so, return 1. Otherwise, set a Python error and return 0.

require_native()

Return type: `int`

Arguments:

- `PyArray_Object* ary`, a NumPy array.

Require that `ary` is not byte-swapped. If the array is not byte-swapped, return 1. Otherwise, set a Python error and return 0.

require_dimensions()

Return type: `int`

Arguments:

- `PyArrayObject* ary`, a NumPy array.
- `int exact_dimensions`, the desired number of dimensions.

Require `ary` to have a specified number of dimensions. If the array has the specified number of dimensions, return 1. Otherwise, set a Python error and return 0.

require_dimensions_n()

Return type: `int`

Arguments:

- `PyArrayObject* ary`, a NumPy array.
- `int* exact_dimensions`, an array of integers representing acceptable numbers of dimensions.
- `int n`, the length of `exact_dimensions`.

Require `ary` to have one of a list of specified number of dimensions. If the array has one of the specified number of dimensions, return 1. Otherwise, set the Python error string and return 0.

require_size()

Return type: `int`

Arguments:

- `PyArrayObject* ary`, a NumPy array.
- `npy_int* size`, an array representing the desired lengths of each dimension.
- `int n`, the length of `size`.

Require `ary` to have a specified shape. If the array has the specified shape, return 1. Otherwise, set the Python error string and return 0.

require_fortran()

Return type: `int`

Arguments:

- `PyArrayObject* ary`, a NumPy array.

Require the given `PyArrayObject` to to be Fortran ordered. If the the `PyArrayObject` is already Fortran ordered, do nothing. Else, set the Fortran ordering flag and recompute the strides.

7.1.6 Beyond the Provided Typemaps

There are many C or C++ array/NumPy array situations not covered by a simple `%include "numpy.i"` and subsequent `%apply` directives.

A Common Example

Consider a reasonable prototype for a dot product function:

```
double dot(int len, double* vec1, double* vec2);
```

The Python interface that we want is:

```
def dot(vec1, vec2):
    """
    dot(PyObject,PyObject) -> double
    """
```

The problem here is that there is one dimension argument and two array arguments, and our typemaps are set up for dimensions that apply to a single array (in fact, SWIG does not provide a mechanism for associating `len` with `vec2` that takes two Python input arguments). The recommended solution is the following:

```
%apply (int DIM1, double* IN_ARRAY1) {(int len1, double* vec1),
                                      (int len2, double* vec2)}
%rename (dot) my_dot;
%exception my_dot {
    $action
    if (PyErr_Occurred()) SWIG_fail;
}
%inline %{
double my_dot(int len1, double* vec1, int len2, double* vec2) {
    if (len1 != len2) {
        PyErr_Format(PyExc_ValueError,
                     "Arrays of lengths (%d,%d) given",
                     len1, len2);
        return 0.0;
    }
    return dot(len1, vec1, vec2);
}
%}
```

If the header file that contains the prototype for `double dot()` also contains other prototypes that you want to wrap, so that you need to `%include` this header file, then you will also need a `%ignore dot;` directive, placed after the `%rename` and before the `%include` directives. Or, if the function in question is a class method, you will want to use `%extend` rather than `%inline` in addition to `%ignore`.

A note on error handling: Note that `my_dot` returns a `double` but that it can also raise a Python error. The resulting wrapper function will return a Python float representation of 0.0 when the vector lengths do not match. Since this is not `NULL`, the Python interpreter will not know to check for an error. For this reason, we add the `%exception` directive above for `my_dot` to get the behavior we want (note that `$action` is a macro that gets expanded to a valid call to `my_dot`). In general, you will probably want to write a SWIG macro to perform this task.

Other Situations

There are other wrapping situations in which `numpy.i` may be helpful when you encounter them.

- In some situations, it is possible that you could use the `%numpy_typemaps` macro to implement typemaps for your own types. See the *Other Common Types: bool* or *Other Common Types: complex* sections for examples. Another situation is if your dimensions are of a type other than `int` (say `long` for example):

```
%numpy_typemaps(double, NPY_DOUBLE, long)
```

- You can use the code in `numpy.i` to write your own typemaps. For example, if you had a five-dimensional array as a function argument, you could cut-and-paste the appropriate four-dimensional typemaps into your interface file. The modifications for the fourth dimension would be trivial.
- Sometimes, the best approach is to use the `%extend` directive to define new methods for your classes (or overload existing ones) that take a `PyObject*` (that either is or can be converted to a `PyArrayObject*`) instead of a pointer to a buffer. In this case, the helper routines in `numpy.i` can be very useful.
- Writing typemaps can be a bit nonintuitive. If you have specific questions about writing SWIG typemaps for NumPy, the developers of `numpy.i` do monitor the Numpy-discussion and Swig-user mail lists.

A Final Note

When you use the `%apply` directive, as is usually necessary to use `numpy.i`, it will remain in effect until you tell SWIG that it shouldn't be. If the arguments to the functions or methods that you are wrapping have common names,

such as `length` or `vector`, these typemaps may get applied in situations you do not expect or want. Therefore, it is always a good idea to add a `%clear` directive after you are done with a specific typemap:

```
%apply (double* IN_ARRAY1, int DIM1) {(double* vector, int length)}
%include "my_header.h"
%clear (double* vector, int length);
```

In general, you should target these typemap signatures specifically where you want them, and then clear them after you are done.

7.1.7 Summary

Out of the box, `numpy.i` provides typemaps that support conversion between NumPy arrays and C arrays:

- That can be one of 12 different scalar types: `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float` and `double`.
- That support 74 different argument signatures for each data type, including:
  - One-dimensional, two-dimensional, three-dimensional and four-dimensional arrays.
  - Input-only, in-place, argout, argoutview, and memory managed argoutview behavior.
  - Hard-coded dimensions, data-buffer-then-dimensions specification, and dimensions-then-data-buffer specification.
  - Both C-ordering ("last dimension fastest") or Fortran-ordering ("first dimension fastest") support for 2D, 3D and 4D arrays.

The `numpy.i` interface file also provides additional tools for wrapper developers, including:

- A SWIG macro (`%numpy_typemaps`) with three arguments for implementing the 74 argument signatures for the user's choice of (1) C data type, (2) NumPy data type (assuming they match), and (3) dimension type.
- Fourteen C macros and fifteen C functions that can be used to write specialized typemaps, extensions, or inlined functions that handle cases not covered by the provided typemaps. Note that the macros and functions are coded specifically to work with the NumPy C/API regardless of NumPy version number, both before and after the deprecation of some aspects of the API after version 1.6.

7.2 Testing the numpy.i Typemaps

7.2.1 Introduction

Writing tests for the `numpy.i` SWIG interface file is a combinatorial headache. At present, 12 different data types are supported, each with 74 different argument signatures, for a total of 888 typemaps supported "out of the box". Each of these typemaps, in turn, might require several unit tests in order to verify expected behavior for both proper and improper inputs. Currently, this results in more than 1,000 individual unit tests executed when `make test` is run in the `numpy/tools/swig` subdirectory.

To facilitate this many similar unit tests, some high-level programming techniques are employed, including C and SWIG macros, as well as Python inheritance. The purpose of this document is to describe the testing infrastructure employed to verify that the `numpy.i` typemaps are working as expected.

7.2.2 Testing Organization

There are three indepedent testing frameworks supported, for one-, two-, and three-dimensional arrays respectively. For one-dimensional arrays, there are two C++ files, a header and a source, named:

```
Vector.h
Vector.cxx
```

that contain prototypes and code for a variety of functions that have one-dimensional arrays as function arguments. The file:

```
Vector.i
```

is a SWIG interface file that defines a python module `Vector` that wraps the functions in `Vector.h` while utilizing the typemaps in `numpy.i` to correctly handle the C arrays.

The `Makefile` calls `swig` to generate `Vector.py` and `Vector_wrap.cxx`, and also executes the `setup.py` script that compiles `Vector_wrap.cxx` and links together the extension module `_Vector.so` or `_Vector.dylib`, depending on the platform. This extension module and the proxy file `Vector.py` are both placed in a subdirectory under the `build` directory.

The actual testing takes place with a Python script named:

```
testVector.py
```

that uses the standard Python library module `unittest`, which performs several tests of each function defined in `Vector.h` for each data type supported.

Two-dimensional arrays are tested in exactly the same manner. The above description applies, but with `Matrix` substituted for `Vector`. For three-dimensional tests, substitute `Tensor` for `Vector`. For four-dimensional tests, substitute `SuperTensor` for `Vector`. For flat in-place array tests, substitute `Flat` for `Vector`. For the descriptions that follow, we will reference the `Vector` tests, but the same information applies to `Matrix`, `Tensor` and `SuperTensor` tests.

The command `make test` will ensure that all of the test software is built and then run all three test scripts.

7.2.3 Testing Header Files

`Vector.h` is a C++ header file that defines a C macro called `TEST_FUNC_PROTOS` that takes two arguments: `TYPE`, which is a data type name such as `unsigned int`; and `SNAME`, which is a short name for the same data type with no spaces, e.g. `uint`. This macro defines several function prototypes that have the prefix `SNAME` and have at least one argument that is an array of type `TYPE`. Those functions that have return arguments return a `TYPE` value.

`TEST_FUNC_PROTOS` is then implemented for all of the data types supported by `numpy.i`:

- `signed char`
- `unsigned char`
- `short`
- `unsigned short`
- `int`
- `unsigned int`
- `long`
- `unsigned long`
- `long long`

- unsigned long long
- float
- double

7.2.4 Testing Source Files

Vector.cxx is a C++ source file that implements compilable code for each of the function prototypes specified in Vector.h. It defines a C macro TEST_FUNCS that has the same arguments and works in the same way as TEST_FUNC_PROTOS does in Vector.h. TEST_FUNCS is implemented for each of the 12 data types as above.

7.2.5 Testing SWIG Interface Files

Vector.i is a SWIG interface file that defines python module Vector. It follows the conventions for using numpy.i as described in this chapter. It defines a SWIG macro %apply_numpy_typemaps that has a single argument TYPE. It uses the SWIG directive %apply to apply the provided typemaps to the argument signatures found in Vector.h. This macro is then implemented for all of the data types supported by numpy.i. It then does a %include "Vector.h" to wrap all of the function prototypes in Vector.h using the typemaps in numpy.i.

7.2.6 Testing Python Scripts

After make is used to build the testing extension modules, testVector.py can be run to execute the tests. As with other scripts that use unittest to facilitate unit testing, testVector.py defines a class that inherits from unittest.TestCase:

```
class VectorTestCase(unittest.TestCase):
```

However, this class is not run directly. Rather, it serves as a base class to several other python classes, each one specific to a particular data type. The VectorTestCase class stores two strings for typing information:

self.typeStr
: A string that matches one of the SNAME prefixes used in Vector.h and Vector.cxx. For example, "double".

self.typeCode
: A short (typically single-character) string that represents a data type in numpy and corresponds to self.typeStr. For example, if self.typeStr is "double", then self.typeCode should be "d".

Each test defined by the VectorTestCase class extracts the python function it is trying to test by accessing the Vector module's dictionary:

```
length = Vector.__dict__[self.typeStr + "Length"]
```

In the case of double precision tests, this will return the python function Vector.doubleLength.

We then define a new test case class for each supported data type with a short definition such as:

```
class doubleTestCase(VectorTestCase):
    def __init__(self, methodName="runTest"):
        VectorTestCase.__init__(self, methodName)
        self.typeStr  = "double"
        self.typeCode = "d"
```

Each of these 12 classes is collected into a unittest.TestSuite, which is then executed. Errors and failures are summed together and returned as the exit argument. Any non-zero result indicates that at least one test did not pass.

CHAPTER

EIGHT

ACKNOWLEDGEMENTS

Large parts of this manual originate from Travis E. Oliphant's book Guide to Numpy (which generously entered Public Domain in August 2008). The reference documentation for many of the functions are written by numerous contributors and developers of Numpy, both prior to and during the Numpy Documentation Marathon.

Please help to improve NumPy's documentation! Instructions on how to join the ongoing documentation marathon can be found on the scipy.org website

[R32] Wikipedia, “Two’s complement”, http://en.wikipedia.org/wiki/Two’s_complement

[CT] Cooley, James W., and John W. Tukey, 1965, “An algorithm for the machine calculation of complex Fourier series,” *Math. Comput.* 19: 297-301.

[NR] Press, W., Teukolsky, S., Vetterline, W.T., and Flannery, B.P., 2007, *Numerical Recipes: The Art of Scientific Computing*, ch. 12-13. Cambridge Univ. Press, Cambridge, UK.

[R6] ISO/IEC standard 9899:1999, “Programming language C.”

[R284] M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83. http://www.math.sfu.ca/~cbm/aands/

[R285] Wikipedia, “Hyperbolic function”, http://en.wikipedia.org/wiki/Hyperbolic_function

[R4] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[R5] Wikipedia, “Inverse hyperbolic function”, http://en.wikipedia.org/wiki/Arcsinh

[R2] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[R3] Wikipedia, “Inverse hyperbolic function”, http://en.wikipedia.org/wiki/Arccosh

[R7] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[R8] Wikipedia, “Inverse hyperbolic function”, http://en.wikipedia.org/wiki/Arctanh

[R18] Wikipedia, “Exponential function”, http://en.wikipedia.org/wiki/Exponential_function

[R19] M. Abramovitz and I. A. Stegun, “Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables,” Dover, 1964, p. 69, http://www.math.sfu.ca/~cbm/aands/page_69.htm

[R44] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. http://www.math.sfu.ca/~cbm/aands/

[R45] Wikipedia, “Logarithm”. http://en.wikipedia.org/wiki/Logarithm

[R46] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. http://www.math.sfu.ca/~cbm/aands/

[R47] Wikipedia, “Logarithm”. http://en.wikipedia.org/wiki/Logarithm

[R48] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. http://www.math.sfu.ca/~cbm/aands/

[R49] Wikipedia, “Logarithm”. http://en.wikipedia.org/wiki/Logarithm

[R60] Wikipedia, “Curve fitting”, http://en.wikipedia.org/wiki/Curve_fitting

[R64] Wikipedia, “Curve fitting”, http://en.wikipedia.org/wiki/Curve_fitting

[R63] Wikipedia, “Curve fitting”, http://en.wikipedia.org/wiki/Curve_fitting

[R61] Wikipedia, “Curve fitting”, http://en.wikipedia.org/wiki/Curve_fitting

[R62] Wikipedia, “Curve fitting”, http://en.wikipedia.org/wiki/Curve_fitting

[R208] Dalgaard, Peter, “Introductory Statistics with R”, Springer-Verlag, 2002.

[R209] Glantz, Stanton A. “Primer of Biostatistics.”, McGraw-Hill, Fifth Edition, 2002.

[R210] Lentner, Marvin, “Elementary Applied Statistics”, Bogden and Quigley, 1972.

[R211] Weisstein, Eric W. “Binomial Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/BinomialDistribution.html

[R212] Wikipedia, “Binomial-distribution”, http://en.wikipedia.org/wiki/Binomial_distribution

[R213] NIST “Engineering Statistics Handbook” http://www.itl.nist.gov/div898/handbook/eda/section3/eda3666.htm

[R214] David McKay, “Information Theory, Inference and Learning Algorithms,” chapter 23, http://www.inference.phy.cam.ac.uk/mackay/

[R215] Wikipedia, “Dirichlet distribution”, http://en.wikipedia.org/wiki/Dirichlet_distribution

[R216] Peyton Z. Peebles Jr., “Probability, Random Variables and Random Signal Principles”, 4th ed, 2001, p. 57.

[R217] “Poisson Process”, Wikipedia, http://en.wikipedia.org/wiki/Poisson_process

[R218] “Exponential Distribution, Wikipedia, http://en.wikipedia.org/wiki/Exponential_distribution

[R219] Glantz, Stanton A. “Primer of Biostatistics.”, McGraw-Hill, Fifth Edition, 2002.

[R220] Wikipedia, “F-distribution”, http://en.wikipedia.org/wiki/F-distribution

[R221] Weisstein, Eric W. “Gamma Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GammaDistribution.html

[R222] Wikipedia, “Gamma-distribution”, http://en.wikipedia.org/wiki/Gamma-distribution

[R223] Gumbel, E. J., “Statistics of Extremes,” New York: Columbia University Press, 1958.

[R224] Reiss, R.-D. and Thomas, M., “Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields,” Basel: Birkhauser Verlag, 2001.

[R225] Lentner, Marvin, “Elementary Applied Statistics”, Bogden and Quigley, 1972.

[R226] Weisstein, Eric W. “Hypergeometric Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HypergeometricDistribution.html

[R227] Wikipedia, “Hypergeometric-distribution”, http://en.wikipedia.org/wiki/Hypergeometric_distribution

[R228] Abramowitz, M. and Stegun, I. A. (Eds.). “Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing,” New York: Dover, 1972.

[R229] Kotz, Samuel, et. al. “The Laplace Distribution and Generalizations, ” Birkhauser, 2001.

[R230] Weisstein, Eric W. “Laplace Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/LaplaceDistribution.html

[R231] Wikipedia, “Laplace Distribution”, http://en.wikipedia.org/wiki/Laplace_distribution

[R232] Reiss, R.-D. and Thomas M. (2001), “Statistical Analysis of Extreme Values, from Insurance, Finance, Hydrology and Other Fields,” Birkhauser Verlag, Basel, pp 132-133.

[R233] Weisstein, Eric W. "Logistic Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/LogisticDistribution.html

[R234] Wikipedia, "Logistic-distribution", http://en.wikipedia.org/wiki/Logistic_distribution

[R235] Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," BioScience, Vol. 51, No. 5, May, 2001. http://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

[R236] Reiss, R.D. and Thomas, M., "Statistical Analysis of Extreme Values," Basel: Birkhauser Verlag, 2001, pp. 31-32.

[R237] Buzas, Martin A.; Culver, Stephen J., Understanding regional species diversity through the log series distribution of occurrences: BIODIVERSITY RESEARCH Diversity & Distributions, Volume 5, Number 5, September 1999 , pp. 187-195(9).

[R238] Fisher, R.A,, A.S. Corbet, and C.B. Williams. 1943. The relation between the number of species and the number of individuals in a random sample of an animal population. Journal of Animal Ecology, 12:42-58.

[R239] D. J. Hand, F. Daly, D. Lunn, E. Ostrowski, A Handbook of Small Data Sets, CRC Press, 1994.

[R240] Wikipedia, "Logarithmic-distribution", http://en.wikipedia.org/wiki/Logarithmic-distribution

[R241] Papoulis, A., "Probability, Random Variables, and Stochastic Processes," 3rd ed., New York: McGraw-Hill, 1991.

[R242] Duda, R. O., Hart, P. E., and Stork, D. G., "Pattern Classification," 2nd ed., New York: Wiley, 2001.

[R243] Weisstein, Eric W. "Negative Binomial Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NegativeBinomialDistribution.html

[R244] Wikipedia, "Negative binomial distribution", http://en.wikipedia.org/wiki/Negative_binomial_distribution

[R245] Delhi, M.S. Holla, "On a noncentral chi-square distribution in the analysis of weapon systems effectiveness", Metrika, Volume 15, Number 1 / December, 1970.

[R246] Wikipedia, "Noncentral chi-square distribution" http://en.wikipedia.org/wiki/Noncentral_chi-square_distribution

[R247] Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

[R248] Wikipedia, "Noncentral F distribution", http://en.wikipedia.org/wiki/Noncentral_F-distribution

[R249] Wikipedia, "Normal distribution", http://en.wikipedia.org/wiki/Normal_distribution

[R250] P. R. Peebles Jr., "Central Limit Theorem" in "Probability, Random Variables and Random Signal Principles", 4th ed., 2001, pp. 51, 51, 125.

[R251] Francis Hunt and Paul Johnson, On the Pareto Distribution of Sourceforge projects.

[R252] Pareto, V. (1896). Course of Political Economy. Lausanne.

[R253] Reiss, R.D., Thomas, M.(2001), Statistical Analysis of Extreme Values, Birkhauser Verlag, Basel, pp 23-30.

[R254] Wikipedia, "Pareto distribution", http://en.wikipedia.org/wiki/Pareto_distribution

[R255] Weisstein, Eric W. "Poisson Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PoissonDistribution.html

[R256] Wikipedia, "Poisson distribution", http://en.wikipedia.org/wiki/Poisson_distribution

[R257] Christian Kleiber, Samuel Kotz, "Statistical size distributions in economics and actuarial sciences", Wiley, 2003.

[R258] Heckert, N. A. and Filliben, James J. "NIST Handbook 148: Dataplot Reference Manual, Volume 2: Let Subcommands and Library Functions", National Institute of Standards and Technology Handbook Series, June 2003. http://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/powpdf.pdf

[R259] Brighton Webs Ltd., "Rayleigh Distribution," http://www.brighton-webs.co.uk/distributions/rayleigh.asp

[R260] Wikipedia, "Rayleigh distribution" http://en.wikipedia.org/wiki/Rayleigh_distribution

[R262] NIST/SEMATECH e-Handbook of Statistical Methods, "Cauchy Distribution", http://www.itl.nist.gov/div898/handbook/eda/section3/eda3663.htm

[R263] Weisstein, Eric W. "Cauchy Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/CauchyDistribution.html

[R264] Wikipedia, "Cauchy distribution" http://en.wikipedia.org/wiki/Cauchy_distribution

[R265] Weisstein, Eric W. "Gamma Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GammaDistribution.html

[R266] Wikipedia, "Gamma-distribution", http://en.wikipedia.org/wiki/Gamma-distribution

[R267] Dalgaard, Peter, "Introductory Statistics With R", Springer, 2002.

[R268] Wikipedia, "Student's t-distribution" http://en.wikipedia.org/wiki/Student's_t-distribution

[R269] Wikipedia, "Triangular distribution" http://en.wikipedia.org/wiki/Triangular_distribution

[R270] Abramowitz, M. and Stegun, I. A. (Eds.). "Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing," New York: Dover, 1972.

[R271] von Mises, R., "Mathematical Theory of Probability and Statistics", New York: Academic Press, 1964.

[R272] Brighton Webs Ltd., Wald Distribution, http://www.brighton-webs.co.uk/distributions/wald.asp

[R273] Chhikara, Raj S., and Folks, J. Leroy, "The Inverse Gaussian Distribution: Theory : Methodology, and Applications", CRC Press, 1988.

[R274] Wikipedia, "Wald distribution" http://en.wikipedia.org/wiki/Wald_distribution

[R275] Waloddi Weibull, Royal Technical University, Stockholm, 1939 "A Statistical Theory Of The Strength Of Materials", Ingeniorsvetenskapsakademiens Handlingar Nr 151, 1939, Generalstabens Litografiska Anstalts Forlag, Stockholm.

[R276] Waloddi Weibull, "A Statistical Distribution Function of Wide Applicability", Journal Of Applied Mechanics ASME Paper 1951.

[R277] Wikipedia, "Weibull distribution", http://en.wikipedia.org/wiki/Weibull_distribution

[R278] Zipf, G. K., "Selected Studies of the Principle of Relative Frequency in Language," Cambridge, MA: Harvard Univ. Press, 1932.

[R261] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator," *ACM Trans. on Modeling and Computer Simulation*, Vol. 8, No. 1, pp. 3-30, Jan. 1998.

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z