NumPy Reference Guide

Release 1.1.0

Written by the NumPy community

May 19, 2008

CONTENTS

Contents:

CHAPTER

ONE

The N-dimensional array object

An array object represents a multidimensional, homogeneous array of fixed-size items. An associated data-type-descriptor object details the data-type in an array (including byteorder and any fields). An array is typically constructed using `array()` or `zeros()`.

1.1 Class members

1.1.1 a.all(axis=None, out=None)

Check if all of the elements of *a* are true.

Performs a logical_and over the given axis and returns the result

Parameters

axis : {None, integer}

Axis to perform the operation over. If None, perform over flattened array.

out : {None, array}, optional

Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

1.1.2 a.any(axis=None, out=None)

Check if any of the elements of *a* are true.

Performs a logical_or over the given axis and returns the result

Parameters

axis : {None, integer}

Axis to perform the operation over. If None, perform over flattened array and return a scalar.

out : {None, array}, optional

Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

1.1.3 a.argmax(axis=None, out=None)

Returns array of indices of the maximum values along the given axis.

Parameters

axis : {None, integer}

If None, the index is into the flattened array, otherwise along the specified axis

out : {None, array}, optional

Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

Returns

index_array : {integer_array}

Examples

```
>>> a = arange(6).reshape(2,3)
>>> a.argmax()
5
>>> a.argmax(0)
array([1, 1, 1])
>>> a.argmax(1)
array([2, 2])
```

1.1.4 a.argmin(axis=None, out=None)

Return array of indices to the minimum values along the given axis.

Parameters

axis : {None, integer}

If None, the index is into the flattened array, otherwise along the specified axis

out : {None, array}, optional

Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

Returns

index_array : {integer_array}

Examples

```
>>> a = arange(6).reshape(2,3)
>>> a.argmin()
0
>>> a.argmin(0)
array([0, 0, 0])
>>> a.argmin(1)
array([0, 0])
```

1.1.5 argsort()

a.argsort(axis=-1, kind='quicksort', order=None) -> indices

Perform an indirect sort along the given axis using the algorithm specified by the kind keyword. It returns an array of indices of the same shape as 'a' that index data along the given axis in sorted order.

Parameters

axis : integer

Axis to be indirectly sorted. None indicates that the flattened array should be used. Default is -1.

kind : string

Sorting algorithm to use. Possible values are 'quicksort', 'mergesort', or 'heapsort'. Default is 'quicksort'.

order : list type or None

When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

Returns

indices : integer array

Array of indices that sort 'a' along the specified axis.

Notes

The various sorts are characterized by average speed, worst case performance, need for work space, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The three available algorithms have the following properties:

kind	speed	worst case	work space	stable
'quicksort'	1	O(n^2)	0	no
'mergesort'	2	O(n*log(n))	~n/2	yes
'heapsort'	3	O(n*log(n))	0	no

All the sort algorithms make temporary copies of the data when the sort is not along the last axis. Consequently, sorts along the last axis are faster and use less space than sorts along other axis.

1.1.6 astype()

a.astype(t) -> Copy of array cast to type t.

Cast array m to type t. t can be either a string representing a typecode, or a python type object of type int, float, or complex.

1.1.7 byteswap()

a.byteswap(False) -> View or copy. Swap the bytes in the array.

Swap the bytes in the array. Return the byteswapped array. If the first argument is True, byteswap in-place and return a reference to self.

1.1.8 a.choose(choices, out=None, mode='raise')

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

Given an array of integers and a set of n choice arrays, this method will create a new array that merges each of the choice arrays. Where a value in *a* is i, the new array will have the value that choices[i] contains in the same place.

Parameters

choices : sequence of arrays

Choice arrays. The index array and all of the choices should be broadcastable to the same shape.

out : array, optional

If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype

mode : {'raise', 'wrap', 'clip'}, optional

Specifies how out-of-bounds indices will behave. 'raise' : raise an error 'wrap' : wrap around 'clip' : clip to the range

Returns

merged_array : array

Examples

```
>>> choices = [[0, 1, 2, 3], [10, 11, 12, 13],
...   [20, 21, 22, 23], [30, 31, 32, 33]]
>>> a = array([2, 3, 1, 0], dtype=int)
>>> a.choose(choices)
array([20, 31, 12,  3])
>>> a = array([2, 4, 1, 0], dtype=int)
>>> a.choose(choices, mode='clip')
array([20, 31, 12,  3])
>>> a.choose(choices, mode='wrap')
array([20,  1, 12,  3])
```

1.1.9 a.clip(a_min, a_max, out=None)

Return an array whose values are limited to [a_min, a_max].

Parameters

a_min :

Minimum value

a_max :

Maximum value

out : {None, array}, optional

Array into which the clipped values can be placed. Its type is preserved and it must be of the right shape to hold the output.

Returns

clipped_array : array

A new array whose elements are same as for a, but values < a_min are replaced with a_min, and > a_max with a_max.

1.1.10 a.compress(condition, axis=None, out=None)

Return selected slices of an array along given axis.

Parameters

condition : {array}

Boolean 1-d array selecting which entries to return. If len(condition) is less than the size of a along the axis, then output is truncated to length of condition array.

axis : {None, integer}

Axis along which to take slices. If None, work on the flattened array.

out : array, optional

Output array. Its type is preserved and it must be of the right shape to hold the output.

Returns

compressed_array : array

A copy of a, without the slices along axis for which condition is false.

Examples

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

1.1.11 a.conj()

Return an array with all complex-valued elements conjugated.

1.1.12 a.conjugate()

Return an array with all complex-valued elements conjugated.

1.1.13 a.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.

1.1.14 a.cumprod(axis=None, dtype=None, out=None)

Return the cumulative product of the elements along the given axis.

The cumulative product is taken over the flattened array by default, otherwise over the specified axis.

Parameters

axis : {None, -1, int}, optional

Axis along which the product is computed. The default (*axis* = None) is to compute over the flattened array.

dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns

cumprod : ndarray.

A new array holding the result is returned unless out is specified, in which case a reference to out is returned.

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

1.1.15 a.cumsum(axis=None, dtype=None, out=None)

Return the cumulative sum of the elements along the given axis.

The cumulative sum is calculated over the flattened array by default, otherwise over the specified axis.

Parameters

axis : {None, -1, int}, optional

Axis along which the sum is computed. The default (*axis* = None) is to compute over the flattened array.

dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns

cumsum : ndarray.

A new array holding the result is returned unless `out` is specified, in which case a reference to `out` is returned.

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

1.1.16 a.diagonal(offset=0, axis1=0, axis2=1)

If a is 2-d, return the diagonal of self with the given offset, i.e., the collection of elements of the form a[i,i+offset]. If a is n-d with n > 2, then the axes specified by axis1 and axis2 are used to determine the 2-d subarray whose diagonal is returned. The shape of the resulting array can be determined by removing axis1 and axis2 and appending an index to the right equal to the size of the resulting diagonals.

Parameters

offset : integer

Offset of the diagonal from the main diagonal. Can be both positive and negative. Defaults to main diagonal.

axis1 : integer

Axis to be used as the first axis of the 2-d subarrays from which the diagonals should be taken. Defaults to first index.

axis2 : integer

Axis to be used as the second axis of the 2-d subarrays from which the diagonals should be taken. Defaults to second index.

Returns

array_of_diagonals : same type as original array

If a is 2-d, then a 1-d array containing the diagonal is returned. If a is n-d, n > 2, then an array of diagonals is returned.

Examples

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


>>> a = arange(8).reshape(2,2,2)
>>> a
array([[[0, 1],
        [2, 3]],
<BLANKLINE>
       [[4, 5],
        [6, 7]]])
>>> a.diagonal(0,-2,-1)
array([[0, 3],
       [4, 7]])
```

1.1.17 a.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.

1.1.18 a.dumps()

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

1.1.19 a.fill(value)

Fill the array with a scalar value.

1.1.20 a.flatten([order])

Return a 1-d array (always copy)

Parameters

order : {'C', 'F'}

Whether to flatten in C or Fortran order.

Notes

a.flatten('F') == a.T.flatten('C')

1.1.21 a.getfield(dtype, offset)

Returns a field of the given array as a certain type. A field is a view of the array data with each itemsize determined by the given type and the offset into the current array.

1.1.22 a.item()

Copy the first element of array to a standard Python scalar and return it. The array must be of size one.

1.1.23 itemset()

1.1.24 a.max(axis=None, out=None)

Return the maximum along a given axis.

Parameters

axis : {None, int}, optional

Axis along which to operate. By default, `axis` is None and the flattened input is used.

out : array_like, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

Returns

amax : array_like

New array holding the result. If `out` was specified, `out` is returned.

1.1.25 mean()

a.mean(axis=None, dtype=None, out=None) -> mean

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis.

Parameters

axis : integer

Axis along which the means are computed. The default is to compute the mean of the flattened array.

dtype : type

Type to use in computing the means. For arrays of integer type the default is float32, for arrays of float types it is the same as the array type.

out : ndarray

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns

mean : The return type varies, see above.

A new array holding the result is returned unless out is specified, in which case a reference to out is returned.

Notes

The mean is the sum of the elements along the axis divided by the number of elements.

1.1.26 a.min(axis=None, out=None)

Return the minimum along a given axis.

Parameters

axis : {None, int}, optional

Axis along which to operate. By default, `axis` is None and the flattened input is used.

out : array_like, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

Returns

amin : array_like

New array holding the result. If `out` was specified, `out` is returned.

1.1.27 a.newbyteorder(byteorder)

Equivalent to a.view(a.dtype.newbytorder(byteorder))

1.1.28 a.nonzero()

Returns a tuple of arrays, one for each dimension of a, containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:

```
a[a.nonzero()]
```

To group the indices by element, rather than dimension, use:

```
transpose(a.nonzero())
```

instead. The result of this is always a 2d array, with a row for each non-zero element.

1.1.29 a.prod(axis=None, dtype=None, out=None)

Return the product of the array elements over the given axis

Parameters

axis : {None, integer}

Axis over which the product is taken. If None is used, then the product is over all the array elements.

dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns

product_along_axis : {array, scalar}, see dtype parameter above.

Returns an array whose shape is the same as a with the specified axis removed. Returns a 0d array when a is 1d or axis=None. Returns a reference to the specified output array if specified.

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> prod([1.,2.])
2.0
>>> prod([1.,2.], dtype=int32)
2
>>> prod([[1.,2.],[3.,4.]])
24.0
>>> prod([[1.,2.],[3.,4.]], axis=1)
array([  2.,  12.])
```

1.1.30 a.ptp(axis=None, out=None)

Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).

Parameters

axis : {None, int}, optional

Axis along which to find the peaks. If None (default) the flattened array is used.

out : array_like

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns

ptp : ndarray.

A new array holding the result, unless `out` was specified, in which case a reference to `out` is returned.

Examples

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

1.1.31 a.put(indices, values, mode='raise')

Set a.flat[n] = values[n] for all n in indices. If values is shorter than indices, it will repeat.

Parameters

indices : array_like

Target indices, interpreted as integers.

values : array_like

Values to place in *a* at target indices.

mode : {'raise', 'wrap', 'clip'}

Specifies how out-of-bounds indices will behave. 'raise' – raise an error 'wrap' – wrap around 'clip' – clip to the range

Notes

If v is shorter than mask it will be repeated as necessary. In particular v can be a scalar or length 1 array. The routine put is the equivalent of the following (although the loop is in C for speed):

ind = array(indices, copy=False) v = array(values, copy=False).astype(a.dtype) for i in ind: a.flat[i] = v[i]

Examples

```
>>> x = np.arange(5)
>>> x.put([0,2,4],[-1,-2,-3])
>>> print x
[-1  1 -2  3 -3]
```

1.1.32 a.ravel([order])

Return a 1d array containing the elements of a (copy only if needed).

The elements in the new array are taken in the order specified by the order keyword. The new array is a view of a if possible, otherwise it is a copy.

Parameters

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

If order is 'C' the elements are taken in row major order. If order is 'F' they are taken in column major order.

Returns

1d_array : {array}

Examples

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

1.1.33 a.repeat(repeats, axis=None)

Repeat elements of an array.

Parameters

a : {array_like}

Input array.

repeats : {integer, integer_array}

The number of repetitions for each element. If a plain integer, then it is applied to all elements. If an array, it needs to be of the same length as the chosen axis.

axis : {None, integer}, optional

The axis along which to repeat values. If None, then this method will operated on the flattened array *a* and return a similarly flat result.

Returns

repeated_array : array

Examples

```
>>> x = array([[1,2],[3,4]])
>>> x.repeat(2)
array([1, 1, 2, 2, 3, 3, 4, 4])
>>> x.repeat(3, axis=1)
array([[1, 1, 1, 2, 2, 2],
```

```
       [3, 3, 3, 4, 4, 4]])
>>> x.repeat([1, 2], axis=0)
array([[1, 2],
       [3, 4],
       [3, 4]])
```

1.1.34 a.reshape(shape, order='C')

Returns an array containing the data of a, but with a new shape.

The result is a view to the original array; if this is not possible, a ValueError is raised.

Parameters

shape : shape tuple or int

The new shape should be compatible with the original shape. If an integer, then the result will be a 1D array of that length.

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

Determines whether the array data should be viewed as in C (row-major) order or FORTRAN (column-major) order.

Returns

reshaped_array : array

A new view to the array.

1.1.35 a.resize(new_shape, refcheck=True, order=False)

Change size and shape of self inplace. Array must own its own memory and not be referenced by other arrays. Returns None.

1.1.36 a.round(decimals=0, out=None)

Return an array rounded a to the given number of decimals.

The real and imaginary parts of complex numbers are rounded separately. The result of rounding a float is a float so the type must be cast if integers are desired. Nothing is done if the input is an integer array and the decimals parameter has a value >= 0.

Parameters

decimals : {0, integer}, optional

Number of decimal places to round to. When decimals is negative it specifies the number of positions to the left of the decimal point.

out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns

rounded_array : {array}

If out=None, returns a new array of the same type as a containing the rounded values, otherwise a reference to the output array is returned.

Notes

Numpy rounds to even. Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc. Results may also be surprising due to the inexact representation of decimal fractions in IEEE floating point and the errors introduced when scaling by powers of ten.

Examples

```
>>> x = array([.5, 1.5, 2.5, 3.5, 4.5])
>>> x.round()
array([ 0.,  2.,  2.,  4.,  4.])
>>> x = array([1,2,3,11])
>>> x.round(decimals=1)
array([ 1,  2,  3, 11])
>>> x.round(decimals=-1)
array([ 0,  0,  0, 10])
```

1.1.37 a.searchsorted(v, side='left')

Find the indices into a sorted array such that if the corresponding keys in v were inserted before the indices the order of a would be preserved. If side='left', then the first such index is returned. If side='right', then the last such index is returned. If there is no such index because the key is out of bounds, then the length of a is returned, i.e., the key would need to be appended. The returned index array has the same shape as v.

Parameters

v : array or list type

Array of keys to be searched for in a.

side : string

Possible values are : 'left', 'right'. Default is 'left'. Return the first or last index where the key could be inserted.

Returns

indices : integer array

The returned array has the same shape as v.

Notes

The array a must be 1-d and is assumed to be sorted in ascending order. Searchsorted uses binary search to find the required insertion points.

1.1.38 setfield()

m.setfield(value, dtype, offset) -> None. places val into field of the given array defined by the data type and offset.

1.1.39 a.setflags(write=None, align=None, uic=None)

1.1.40 sort()

a.sort(axis=-1, kind='quicksort', order=None) -> None.

Perform an inplace sort along the given axis using the algorithm specified by the kind keyword.

Parameters

axis : integer

Axis to be sorted along. None indicates that the flattened array should be used. Default is -1.

kind : string

Sorting algorithm to use. Possible values are 'quicksort', 'mergesort', or 'heapsort'. Default is 'quicksort'.

order : list type or None

When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

Notes

The various sorts are characterized by average speed, worst case performance, need for work space, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The three available algorithms have the following properties:

kind	speed	worst case	work space	stable
'quicksort'	1	O(n^2)	0	no
'mergesort'	2	O(n*log(n))	~n/2	yes
'heapsort'	3	O(n*log(n))	0	no

All the sort algorithms make temporary copies of the data when the sort is not along the last axis. Consequently, sorts along the last axis are faster and use less space than sorts along other axis.

1.1.41 m.squeeze()

Remove single-dimensional entries from the shape of a.

Examples

```
>>> x = array([[[1,1,1],[2,2,2],[3,3,3]]])
>>> x.shape
(1, 3, 3)
>>> x.squeeze().shape
(3, 3)
```

1.1.42 a.std(axis=None, dtype=None, out=None, ddof=0)

Returns the standard deviation of the array elements, a measure of the spread of a distribution. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Parameters

axis : integer

Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

dtype : type

Type to use in computing the standard deviation. For arrays of integer type the default is float32, for arrays of float types it is the same as the array type.

out : ndarray

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

ddof : {0, integer}

Means Delta Degrees of Freedom. The divisor used in calculations is N-ddof.

Returns

standard deviation : The return type varies, see above.

A new array holding the result is returned unless out is specified, in which case a reference to out is returned.

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e. var = sqrt(mean(abs(x - x.mean())**2)). The computed standard deviation is computed by dividing by the number of elements, N-ddof. The option ddof defaults to zero, that is, a biased estimate. Note that for complex numbers std takes the absolute value before squaring, so that the result is always real and nonnegative.

1.1.43 a.sum(axis=None, dtype=None, out=None)

Return the sum of the array elements over the given axis

Parameters

axis : {None, integer}

Axis over which the sum is taken. If None is used, then the sum is over all the array elements.

dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

out : {None, array}, optional

Array into which the sum can be placed. Its type is preserved and it must be of the right shape to hold the output.

Returns

sum_along_axis : {array, scalar}, see dtype parameter above.

Returns an array whose shape is the same as a with the specified axis removed. Returns a 0d array when a is 1d or axis=None. Returns a reference to the specified output array if specified.

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> array([0.5, 1.5]).sum()
2.0
>>> array([0.5, 1.5]).sum(dtype=int32)
1
>>> array([[0, 1], [0, 5]]).sum(axis=0)
array([0, 6])
>>> array([[0, 1], [0, 5]]).sum(axis=1)
array([1, 5])
>>> ones(128, dtype=int8).sum(dtype=int8) # overflow!
-128
```

1.1.44 a.swapaxes(axis1, axis2)

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

Parameters

axis1 : int

First axis.

axis2 : int

Second axis.

Examples

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

>>> x = np.array([[[0,1],[2,3]],[[4,5],[6,7]]])
>>> x
array([[[0, 1],
        [2, 3]],
<BLANKLINE>
       [[4, 5],
        [6, 7]]])
>>> x.swapaxes(0,2)
array([[[0, 4],
        [2, 6]],
<BLANKLINE>
       [[1, 5],
        [3, 7]]])
```

1.1.45 a.take(indices, axis=None, out=None, mode='raise')

Return an array formed from the elements of a at the given indices.

This method does the same thing as "fancy" indexing; however, it can be easier to use if you need to specify a given axis.

Parameters

indices : int array

The indices of the values to extract.

axis : {None, int}, optional

The axis over which to select values. None signifies that the operation should be performed over the flattened array.

out : {None, array}, optional

If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.

mode : {'raise', 'wrap', 'clip'}, optional

Specifies how out-of-bounds indices will behave. 'raise' – raise an error 'wrap' – wrap around 'clip' – clip to the range

Returns

subarray : array

The returned array has the same type as a.

1.1.46 a.tofile(fid, sep="", format="%s")

Write the data to a file.

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

This is a convenience function for quick storage of array data. Information on endianess 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 endianess. Some of these problems can be overcome by outputting the data as text files at the expense of speed and file size.

Parameters

fid : file or string

An open file object or a string containing a filename.

sep : string

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

format : string

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

1.1.47 a.tolist()

Return the array as nested lists.

Copy the data portion of the array to a hierarchical Python list and return that list. Data items are converted to the nearest compatible Python type.

1.1.48 a.tostring(order='C')

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

Parameters

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

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

1.1.49 a.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)

Return the sum along diagonals of the array.

If a is 2-d, returns the sum along the diagonal of self with the given offset, i.e., the collection of elements of the form a[i,i+offset]. If a has more than two dimensions, then the axes specified by axis1 and axis2 are used to determine the 2-d subarray whose trace is returned. The shape of the resulting array can be determined by removing axis1 and axis2 and appending an index to the right equal to the size of the resulting diagonals.

Parameters

offset : {0, integer}, optional

Offset of the diagonal from the main diagonal. Can be both positive and negative. Defaults to main diagonal.

axis1 : {0, integer}, optional

Axis to be used as the first axis of the 2-d subarrays from which the diagonals should be taken. Defaults to first axis.

axis2 : {1, integer}, optional

Axis to be used as the second axis of the 2-d subarrays from which the diagonals should be taken. Defaults to second axis.

dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and a is of integer type of precision less than the default integer precision, then the default integer precision is used. Otherwise, the precision is the same as that of a.

out : {None, array}, optional

Array into which the sum can be placed. Its type is preserved and it must be of the right shape to hold the output.

Returns

sum_along_diagonals : array

If a is 2-d, a 0-d array containing the diagonal is returned. If a has larger dimensions, then an array of diagonals is returned.

Examples

```
>>> eye(3).trace()
3.0
>>> a = arange(8).reshape((2,2,2))
>>> a.trace()
array([6, 8])
```

1.1.50 a.transpose(*axes)

Returns a view of 'a' with axes transposed. If no axes are given, or None is passed, switches the order of the axes. For a 2-d array, this is the usual matrix transpose. If axes are given, they describe how the axes are permuted.

Examples

```
>>> a = 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]])
```

1.1.51 var()

a.var(axis=None, dtype=None, out=None, ddof=0) -> variance

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Parameters

axis : integer

Axis along which the variance is computed. The default is to compute the variance of the flattened array.

dtype : data-type

Type to use in computing the variance. For arrays of integer type the default is float32, for arrays of float types it is the same as the array type.

out : ndarray

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

ddof : {0, integer},

Means Delta Degrees of Freedom. The divisor used in calculation is N - ddof.

Returns

variance : The return type varies, see above.

A new array holding the result is returned unless out is specified, in which case a reference to out is returned.

Notes

The variance is the average of the squared deviations from the mean, i.e. var = mean(abs(x - x.mean())**2). The mean is computed by dividing by N-ddof, where N is the number of elements. The argument ddof defaults to zero; for an unbiased estimate supply ddof=1. Note that for complex numbers the absolute value is taken before squaring, so that the result is always real and nonnegative.

1.1.52 a.view(dtype=None, type=None)

New view of array with the same data.

Parameters

dtype : data-type

Data-type descriptor of the returned view, e.g. float32 or int16.

type : python type

Type of the returned view, e.g. ndarray or matrix.

Examples

```
>>> x = np.array([(1,2)],dtype=[('a',np.int8),('b',np.int8)])
>>> y = x.view(dtype=np.int16, type=np.matrix)

>>> print y.dtype
int16

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

CHAPTER

TWO

NumPy Function Reference

Description of function reference goes here.

2.1 numpy

2.1.1 absolute()

y = absolute(x) takes |x| elementwise.

2.1.2 absolute()

y = absolute(x) takes |x| elementwise.

2.1.3 add()

y = add(x1,x2) adds the arguments elementwise.

2.1.4 docstring(obj, docstring)

Add a docstring to a built-in obj if possible. If the obj already has a docstring raise a RuntimeError If this routine does not know how to add a docstring to the object raise a TypeError

2.1.5 add_newdoc(place, obj, doc)

Adds documentation to obj which is in module place.

If doc is a string add it to obj as a docstring

If doc is a tuple, then the first element is interpreted as an attribute of obj and the second as the docstring (method, docstring)

If doc is a list, then each element of the list should be a sequence of length two –> [(method1, docstring1), (method2, docstring2), ...]

This routine never raises an error.

2.1.6 alen(a)

Return the length of a Python object interpreted as an array of at least 1 dimension.

Parameters

a : array_like

Returns

alen : int

Length of the first dimension of *a*.

Examples

```
>>> z = numpy.zeros((7,4,5))
>>> z.shape[0]
7
>>> numpy.alen(z)
7
```

2.1.7 all(a, axis=None, out=None)

Check if all of the elements of *a* are true.

Performs a logical_and over the given axis and returns the result

Parameters

a : array_like

axis : {None, integer}

Axis to perform the operation over. If None, perform over flattened array and return a scalar.

out : {None, array}, optional

Array into which the product can be placed. Its type is preserved and it must be of the right shape to hold the output.

2.1.8 allclose(a, b, rtol=1.0000000000000001e-05, atol=1e-08)

Returns True if all components of a and b are equal subject to given tolerances.

The relative error rtol must be positive and << 1.0 The absolute error atol usually comes into play for those elements of b that are very small or zero; it says how small a must be also.

2.1.9 alltrue(a, axis=None, out=None)

Check if all of the elements of *a* are true.

Performs a logical_and over the given axis and returns the result

Parameters

a : array_like

axis : {None, integer}

Axis to perform the operation over. If None, perform over flattened array.

out : {None, array}, optional

Array into which the product can be placed. Its type is preserved and it must be of the right shape to hold the output.

2.1.10 alterdot()

alterdot() changes all dot functions to use blas.

2.1.11 amax(a, axis=None, out=None)

Return the maximum along a given axis.

Parameters

a : array_like

Input data.

axis : {None, int}, optional

Axis along which to operate. By default, `axis` is None and the flattened input is used.

out : array_like, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

Returns

amax : array_like

New array holding the result, unless `out` was specified.

Examples

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

2.1.12 amin(a, axis=None, out=None)

Return the minimum along a given axis.

Parameters

a : array_like

Input data.

axis : {None, int}, optional

Axis along which to operate. By default, `axis` is None and the flattened input is used.

out : array_like, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

Returns

amin : array_like

New array holding the result, unless `out` was specified.

Examples

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

2.1.13 angle(z, deg=0)

Return the angle of the complex argument z.

Examples

```
>>> numpy.angle(1+1j)           # in radians
0.78539816339744828
>>> numpy.angle(1+1j,deg=True)  # in degrees
45.0
```

2.1.14 any(a, axis=None, out=None)

Check if any of the elements of *a* are true.

Performs a logical_or over the given axis and returns the result

Parameters

a : array_like

axis : {None, integer}

Axis to perform the operation over. If None, perform over flattened array and return a scalar.

out : {None, array}, optional

Array into which the product can be placed. Its type is preserved and it must be of the right shape to hold the output.

2.1.15 append(arr, values, axis=None)

Append to the end of an array along axis (ravel first if None)

2.1.16 apply_along_axis(func1d, axis, arr, *args)

Execute func1d(arr[i],*args) where func1d takes 1-D arrays and arr is an N-d array. i varies so as to apply the function along the given axis for each 1-d subarray in arr.

2.1.17 apply_over_axes(func, a, axes)

Apply a function repeatedly over multiple axes, keeping the same shape for the resulting array.

func is called as res = func(a, axis). The result is assumed to be either the same shape as a or have one less dimension. This call is repeated for each axis in the axes sequence.

2.1.18 arange([start,] stop[, step,], dtype=None)

For integer arguments, just like range() except it returns an array whose type can be specified by the keyword argument dtype. If dtype is not specified, the type of the result is deduced from the type of the arguments.

For floating point arguments, the length of the result is ceil((stop - start)/step). This rule may result in the last element of the result being greater than stop.

2.1.19 arccos()

y = arccos(x) inverse cosine elementwise.

2.1.20 arccosh()

y = arccosh(x) inverse hyperbolic cosine elementwise.

2.1.21 arcsin()

y = arcsin(x) inverse sine elementwise.

2.1.22 arcsinh()

y = arcsinh(x) inverse hyperbolic sine elementwise.

2.1.23 arctan()

y = arctan(x) inverse tangent elementwise.

2.1.24 arctan2()

y = arctan2(x1,x2) a safe and correct arctan(x1/x2)

2.1.25 arctanh()

y = arctanh(x) inverse hyperbolic tangent elementwise.

2.1.26 argmax(a, axis=None)

Returns array of indices of the maximum values of along the given axis.

Parameters

a : {array_like}

Array to look in.

axis : {None, integer}

If None, the index is into the flattened array, otherwise along the specified axis

Returns

index_array : {integer_array}

Examples

```
>>> a = arange(6).reshape(2,3)
>>> argmax(a)
5
>>> argmax(a,0)
array([1, 1, 1])
>>> argmax(a,1)
array([2, 2])
```

2.1.27 argmin(a, axis=None)

Return array of indices to the minimum values along the given axis.

Parameters

a : {array_like}

Array to look in.

axis : {None, integer}

If None, the index is into the flattened array, otherwise along the specified axis

Returns

index_array : {integer_array}

Examples

```
>>> a = arange(6).reshape(2,3)
>>> argmin(a)
0
>>> argmin(a,0)
array([0, 0, 0])
>>> argmin(a,1)
array([0, 0])
```

2.1.28 argsort(a, axis=-1, kind='quicksort', order=None)

Returns array of indices that index 'a' in sorted order.

Perform an indirect sort along the given axis using the algorithm specified by the kind keyword. It returns an array of indices of the same shape as a that index data along the given axis in sorted order.

Parameters

a : array

Array to be sorted.

axis : {None, int} optional

Axis along which to sort. None indicates that the flattened array should be used.

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

Sorting algorithm to use.

order : {None, list type}, optional

When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

Returns

index_array : {integer_array}

Array of indices that sort 'a' along the specified axis.

Notes

The various sorts are characterized by average speed, worst case performance, need for work space, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The three available algorithms have the following properties:

kind	speed	worst case	work space	stable
quicksort	1	O(n^2)	0	no
mergesort	2	O(n*log(n))	~n/2	yes
heapsort	3	O(n*log(n))	0	no

All the sort algorithms make temporary copies of the data when the sort is not along the last axis. Consequently, sorts along the last axis are faster and use less space than sorts along other axis.

2.1.29 argwhere(a)

Return a 2-d array of shape N x a.ndim where each row is a sequence of indices into a. This sequence must be converted to a tuple in order to be used to index into a.

```
>>> from numpy import ones, argwhere
>>> argwhere(ones((2, 2)))
array([[0, 0],
       [0, 1],
       [1, 0],
       [1, 1]])
```

2.1.30 around(a, decimals=0, out=None)

Round a to the given number of decimals.

The real and imaginary parts of complex numbers are rounded separately. The result of rounding a float is a float so the type must be cast if integers are desired. Nothing is done if the input is an integer array and the decimals parameter has a value >= 0.

Parameters

a : {array_like}

Array containing numbers whose rounded values are desired. If a is not an array, a conversion is attempted.

decimals : {0, int}, optional

Number of decimal places to round to. When decimals is negative it specifies the number of positions to the left of the decimal point.

out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary. Numpy rounds floats to floats by default.

Returns

rounded_array : {array}

If out=None, returns a new array of the same type as a containing the rounded values, otherwise a reference to the output array is returned.

Notes

Numpy rounds to even. Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc. Results may also be surprising due to the inexact representation of decimal fractions in IEEE floating point and the errors introduced when scaling by powers of ten.

Examples

```
>>> around([.5, 1.5, 2.5, 3.5, 4.5])
array([ 0.,  2.,  2.,  4.,  4.])
>>> around([1,2,3,11], decimals=1)
array([ 1,  2,  3, 11])
>>> around([1,2,3,11], decimals=-1)
array([ 0,  0,  0, 10])
```

2.1.31 array(object, dtype=None, copy=1,order=None, subok=0,ndmin=0)

Return an array from object with the specified data-type.

Parameters

object : array-like

an array, any object exposing the array interface, any object whose __array__ method returns an array, or any (nested) sequence.

dtype : data-type

The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence. This argument can only be used to 'upcast' the array. For downcasting, use the .astype(t) method.

copy : bool

If true, then force a copy. Otherwise a copy will only occur if __array__ returns a copy, obj is a nested sequence, or a copy is needed to satisfy any of the other requirements

order : {'C', 'F', 'A' (None)}

Specify the order of the array. If order is 'C', then the array will be in C-contiguous order (last-index varies the fastest). If order is 'FORTRAN', then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is None, then the returned array may be in either C-, or Fortran-contiguous order or even discontiguous.

subok : bool

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

ndmin : int

Specifies the minimum number of dimensions that the resulting array should have. 1's will be pre-pended to the shape as needed to meet this requirement.

2.1.32 array2string(a, max_line_width=None, precision=None, suppress_small=None, separator=' ', prefix='', style=<built-in function repr>)

Return a string representation of an array.

Parameters **a** [ndarray] Input array.

max_line_width [int] The maximum number of columns the string should span. Newline characters splits the string appropriately after array elements.

precision [int] Floating point precision.

suppress_small [bool] Represent very small numbers as zero.

separator [string] Inserted between elements.

prefix [string] An array is typically printed as
'prefix(' + array2string(a) + ')'
The length of the prefix string is used to align the output correctly.

Definition list ends without a blank line; unexpected unindent.

style : function

Examples

```
>>> x = N.array([1e-16,1,2,3])
>>> print array2string(x,precision=2,separator=',',suppress_small=True)
[ 0., 1., 2., 3.]
```

2.1.33 array_equal(a1, a2)

Returns True if a1 and a2 have identical shapes and all elements equal and False otherwise.

2.1.34 array_equiv(a1, a2)

Returns True if a1 and a2 are shape consistent (mutually broadcastable) and have all elements equal and False otherwise.

2.1.35 array_repr(arr, max_line_width=None, precision=None, suppress_small=None)

2.1.36 array_split(ary, indices_or_sections, axis=0)

Divide an array into a list of sub-arrays.

Description: Divide ary into a list of sub-arrays along the specified axis. If indices_or_sections is an integer, ary is divided into that many equally sized arrays. If it is impossible to make an equal split, each of the leading arrays in the list have one additional member. If indices_or_sections is a list of sorted integers, its entries define the indexes where ary is split.

Arguments: **ary – N-D array.** Array to be divided into sub-arrays.

indices_or_sections – integer or 1D array. If integer, defines the number of (close to) equal sized sub-arrays. If it is a 1D array of sorted indices, it defines the indexes at which ary is divided. Any empty list results in a single sub-array equal to the original array.

axis – integer. default=0. Specifies the axis along which to split ary.

Caveats: Currently, the default for axis is 0. This means a 2D array is divided into multiple groups of rows. This seems like the appropriate default,

2.1.37 array_str(a, max_line_width=None, precision=None, suppress_small=None)

2.1.38 asanyarray(a, dtype=None, order=None)

Returns a as an array, but will pass subclasses through.

2.1.39 asarray(a, dtype=None, order=None)

Returns a as an array.

Unlike array(), no copy is performed if a is already an array. Subclasses are converted to base class ndarray.

2.1.40 asarray_chkfinite(a)

Like asarray, but check that no NaNs or Infs are present.

2.1.41 ascontiguousarray(a, dtype=None)

Return ‘a’ as an array contiguous in memory (C order).

2.1.42 asfarray(a, dtype=<type ‘numpy.float64’>)

asfarray(a,dtype=None) returns a as a float array.

2.1.43 asfortranarray(a, dtype=None)

Return ‘a’ as an array laid out in Fortran-order in memory.

2.1.44 asmatrix(data, dtype=None)

Returns ‘data’ as a matrix. Unlike matrix(), no copy is performed if ‘data’ is already a matrix or array. Equivalent to: matrix(data, copy=False)

2.1.45 asscalar(a)

Convert an array of size 1 to its scalar equivalent.

2.1.46 atleast_1d(*arys)

Force a sequence of arrays to each be at least 1D.

Description: Force an array to be at least 1D. If an array is 0D, the array is converted to a single row of values. Otherwise, the array is unaltered.

Arguments: *arys – arrays to be converted to 1 or more dimensional array.

Inline emphasis start-string without end-string.

Returns: input array converted to at least 1D array.

2.1.47 atleast_2d(*arys)

Force a sequence of arrays to each be at least 2D.

Description: Force an array to each be at least 2D. If the array is 0D or 1D, the array is converted to a single row of values. Otherwise, the array is unaltered.

Arguments: arys – arrays to be converted to 2 or more dimensional array.

Returns: input array converted to at least 2D array.

2.1.48 atleast_3d(*arys)

Force a sequence of arrays to each be at least 3D.

Description: Force an array each be at least 3D. If the array is 0D or 1D, the array is converted to a single 1xNx1 array of values where N is the orginal length of the array. If the array is 2D, the array is converted to a single MxNx1 array of values where MxN is the orginal shape of the array. Otherwise, the array is unaltered.

Arguments: arys – arrays to be converted to 3 or more dimensional array.

Returns: input array converted to at least 3D array.

2.1.49 average(a, axis=None, weights=None, returned=False)

Return the weighted average of array a over the given axis.

Parameters

a : array_like

Data to be averaged.

axis : {None, integer}, optional

Axis along which to average a. If None, averaging is done over the entire array irrespective of its shape.

weights : {None, array_like}, optional

The importance each datum has in the computation of the average. The weights array can either be 1D, in which case its length must be the size of a along the given axis, or of the same shape as a. If weights=None, all data are assumed to have weight equal to one.

returned :{False, boolean}, optional :

If True, the tuple (average, sum_of_weights) is returned, otherwise only the average is returned. Note that if weights=None, then the sum of the weights is also the number of elements averaged over.

Returns

average, [sum_of_weights] : {array_type, double}

Return the average along the specified axis. When returned is True, return a tuple with the average as the first element and the sum of the weights as the second element. The return type is Float if a is of integer type, otherwise it is of the same type as a. sum_of_weights is has the same type as the average.

Raises

ZeroDivisionError :

When all weights along axis are zero. See numpy.ma.average for a version robust to this type of error.

TypeError :

When the length of 1D weights is not the same as the shape of a along axis.

Examples

```
>>> average(range(1,11), weights=range(10,0,-1))
4.0
```

2.1.50 bartlett(M)

Return the Bartlett window.

The Bartlett window is very similar to a triangular window, except that the end points are at zero. It is often used in signal processing for tapering a signal, without generating too much ripple in the frequency domain.

Parameters

M : int

Number of points in the output window. If zero or less, an empty array is returned.

Returns

out : array

The triangular window, normalized to one (the value one appears only if the number of samples is odd), with the first and last samples equal to zero.

Notes

The Bartlett window is defined as

$$w(n) = \frac{2}{M-1}\left(\frac{M-1}{2} - \left|n - \frac{M-1}{2}\right|\right)$$

Most references to the Bartlett window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. Note that convolution with this window produces linear interpolation. It is also known as an apodization (which means"removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

References

Examples

```
>>> from numpy import bartlett
>>> bartlett(12)
array([ 0.        ,  0.18181818,  0.36363636,  0.54545455,  0.72727273,
        0.90909091,  0.90909091,  0.72727273,  0.54545455,  0.36363636,
        0.18181818,  0.        ])
```

Plot the window and its frequency response: >>> from numpy import clip, log10, array, bartlett >>> from scipy.fftpack import fft >>> from matplotlib import pyplot as plt

```
>>> window = bartlett(51)
>>> plt.plot(window)
>>> plt.title("Bartlett window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
>>> plt.show()
```

```
>>> A = fft(window, 2048) / 25.5
>>> mag = abs(fftshift(A))
>>> freq = linspace(-0.5,0.5,len(A))
>>> response = 20*log10(mag)
>>> response = clip(response,-100,100)
>>> plt.plot(freq, response)
>>> plt.title("Frequency response of Bartlett window")
>>> plt.ylabel("Magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
>>> plt.axis('tight'); plt.show()
```

2.1.51 base_repr(number, base=2, padding=0)

Return the representation of a number in the given base.

Base can't be larger than 36.

[1] M.S. Bartlett, "Periodogram Analysis and Continuous Spectra", Biometrika 37, 1-16, 1950.
[2] A.V. Oppenheim and R.W. Schafer, "Discrete-Time Signal Processing", Prentice-Hall, 1999, pp. 468-471.
[3] Wikipedia, "Window function", http://en.wikipedia.org/wiki/Window_function
[4] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, "Numerical Recipes", Cambridge University Press, 1986, page 429.

2.1.52 binary_repr(num, width=None)

Return the binary representation of the input number as a string.

This is equivalent to using base_repr with base 2, but about 25x faster.

For negative numbers, if width is not given, a - sign is added to the front. If width is given, the two's complement of the number is returned, with respect to that width.

2.1.53 bincount(x,weights=None)

Return the number of occurrences of each value in x.

x must be a list of non-negative integers. The output, b[i], represents the number of times that i is found in x. If weights is specified, every occurrence of i at a position p contributes weights[p] instead of 1.

See also: histogram, digitize, unique.

2.1.54 bitwise_and()

y = bitwise_and(x1,x2) computes x1 & x2 elementwise.

2.1.55 invert()

y = invert(x) computes ~x (bit inversion) elementwise.

2.1.56 bitwise_or()

y = bitwise_or(x1,x2) computes x1 | x2 elementwise.

2.1.57 bitwise_xor()

y = bitwise_xor(x1,x2) computes x1 ^ x2 elementwise.

2.1.58 blackman(M)

blackman(M) returns the M-point Blackman window.

2.1.59 bmat(obj, ldict=None, gdict=None)

Build a matrix object from string, nested sequence, or array.

Examples

```
>>> F = bmat('A, B; C, D')
>>> F = bmat([[A,B],[C,D]])
>>> F = bmat(r_[c_[A,B],c_[C,D]])
```

All of these produce the same matrix:

```
[ A  B ]
[ C  D ]
```

if A, B, C, and D are appropriately shaped 2-d arrays.

2.1.60 byte_bounds(a)

(low, high) are pointers to the end-points of an array

low is the first byte high is just *past* the last byte

If the array is not single-segment, then it may not actually use every byte between these bounds.

The array provided must conform to the Python-side of the array interface

2.1.61 can_cast(from=d1, to=d2)

Returns True if data type d1 can be cast to data type d2 without losing precision.

2.1.62 ceil()

y = ceil(x) elementwise smallest integer >= x.

2.1.63 choose(a, choices, out=None, mode='raise')

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

Given an array of integers and a set of n choice arrays, this function will create a new array that merges each of the choice arrays. Where a value in *a* is i, then the new array will have the value that choices[i] contains in the same place.

Parameters

a : int array

This array must contain integers in [0, n-1], where n is the number of choices.

choices : sequence of arrays

Choice arrays. The index array and all of the choices should be broadcastable to the same shape.

out : array, optional

If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype

mode : {'raise', 'wrap', 'clip'}, optional

Specifies how out-of-bounds indices will behave. 'raise' : raise an error 'wrap' : wrap around 'clip' : clip to the range

Returns

merged_array : array

Examples

```
>>> choices = [[0, 1, 2, 3], [10, 11, 12, 13],
...   [20, 21, 22, 23], [30, 31, 32, 33]]
>>> choose([2, 3, 1, 0], choices)
array([20, 31, 12,  3])
>>> choose([2, 4, 1, 0], choices, mode='clip')
array([20, 31, 12,  3])
>>> choose([2, 4, 1, 0], choices, mode='wrap')
array([20,  1, 12,  3])
```

2.1.64 clip(a, a_min, a_max, out=None)

Return an array whose values are limited to [a_min, a_max].

Parameters

a : {array_like}

Array containing elements to clip.

a_min : :

Minimum value

a_max : :

Maximum value

out : array, optional

The results will be placed in this array. It may be the input array for inplace clipping.

Returns

clipped_array : {array}

A new array whose elements are same as for a, but values < a_min are replaced with a_min, and > a_max with a_max.

Examples

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

2.1.65 column_stack(tup)

Stack 1D arrays as columns into a 2D array

Description: Take a sequence of 1D arrays and stack them as columns to make a single 2D array. All arrays in the sequence must have the same first dimension. 2D arrays are stacked as-is, just like with hstack. 1D arrays are turned into 2D columns first.

Arguments: **tup – sequence of 1D or 2D arrays. All arrays must have the same** first dimension.

Examples:

```
>>> import numpy
>>> a = array((1,2,3))
>>> b = array((2,3,4))
>>> numpy.column_stack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

2.1.66 common_type(*arrays)

Given a sequence of arrays as arguments, return the best inexact scalar type which is "most" common amongst them.

The return type will always be a inexact scalar type, even if all the arrays are integer arrays.

2.1.67 compare_chararrays()

2.1.68 compress(condition, a, axis=None, out=None)

Return selected slices of an array along given axis.

Parameters

condition : {array}

Boolean 1-d array selecting which entries to return. If len(condition) is less than the size of a along the axis, then output is truncated to length of condition array.

a : {array_type}

Array from which to extract a part.

axis : {None, integer}

Axis along which to take slices. If None, work on the flattened array.

out : array, optional

Output array. Its type is preserved and it must be of the right shape to hold the output.

Returns

compressed_array : array

A copy of a, without the slices along axis for which condition is false.

Examples

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

2.1.69 concatenate((a1, a2, ...), axis=0)

Join arrays together.

The tuple of sequences (a1, a2, ...) are joined along the given axis (default is the first one) into a single numpy array.

Examples

```
>>> concatenate( ([0,1,2], [5,6,7]) )
array([0, 1, 2, 5, 6, 7])
```

2.1.70 conjugate()

y = conjugate(x) takes the conjugate of x elementwise.

2.1.71 conjugate()

y = conjugate(x) takes the conjugate of x elementwise.

2.1.72 convolve(a, v, mode='full')

Returns the discrete, linear convolution of 1-D sequences a and v; mode can be 'valid', 'same', or 'full' to specify size of the resulting sequence.

2.1.73 copy(a)

Return an array copy of the given object.

2.1.74 corrcoef(x, y=None, rowvar=1, bias=0)

The correlation coefficients

2.1.75 correlate(a, v, mode='valid')

Return the discrete, linear correlation of 1-D sequences a and v; mode can be 'valid', 'same', or 'full' to specify the size of the resulting sequence

2.1.76 cos()

y = cos(x) cosine elementwise.

2.1.77 cosh()

y = cosh(x) hyperbolic cosine elementwise.

2.1.78 cov(m, y=None, rowvar=1, bias=0)

Estimate the covariance matrix.

If m is a vector, return the variance. For matrices return the covariance matrix.

If y is given it is treated as an additional (set of) variable(s).

Normalization is by (N-1) where N is the number of observations (unbiased estimate). If bias is 1 then normalization is by N.

If rowvar is non-zero (default), then each row is a variable with observations in the columns, otherwise each column is a variable and the observations are in the rows.

2.1.79 cross(a, b, axisa=-1, axisb=-1, axisc=-1, axis=None)

Return the cross product of two (arrays of) vectors.

The cross product is performed over the last axis of a and b by default, and can handle axes with dimensions 2 and 3. For a dimension of 2, the z-component of the equivalent three-dimensional cross product is returned.

2.1.80 cumprod(a, axis=None, dtype=None, out=None)

Return the cumulative product of the elements along the given axis.

The cumulative product is taken over the flattened array by default, otherwise over the specified axis.

Parameters

a : array-like

Input array or object that can be converted to an array.

axis : {None, -1, int}, optional

Axis along which the product is computed. The default (*axis* = *None*) is to compute over the flattened array.

dtype : {None, dtype}, optional

Type of the returned array and of the accumulator where the elements are multiplied. If *dtype* has the value *None* and the type of *a* is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the *dtype* is the same as that of *a*.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns

cumprod : ndarray.

A new array holding the result is returned unless *out* is specified, in which case a reference to out is returned.

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> a=numpy.array([[1,2,3],[4,5,6]])
>>> a=numpy.array([1,2,3])
>>> numpy.cumprod(a) # intermediate results 1, 1*2
...                  # total product 1*2*3 = 6
array([1, 2, 6])
>>> a=numpy.array([[1,2,3],[4,5,6]])
>>> numpy.cumprod(a,dtype=float) # specify type of output
array([   1.,    2.,    6.,   24.,  120.,  720.])
>>> numpy.cumprod(a,axis=0) # for each of the 3 columns:
...                         # product and intermediate results
array([[ 1,  2,  3],
       [ 4, 10, 18]])
>>> numpy.cumprod(a,axis=1) # for each of the two rows:
...                         # product and intermediate results
array([[  1,   2,   6],
       [  4,  20, 120]])
```

2.1.81 cumproduct(a, axis=None, dtype=None, out=None)

Return the cumulative product over the given axis.

2.1.82 cumsum(a, axis=None, dtype=None, out=None)

Return the cumulative sum of the elements along a given axis.

Parameters

a : array-like

Input array or object that can be converted to an array.

axis : {None, -1, int}, optional

Axis along which the sum is computed. The default (*axis* = *None*) is to compute over the flattened array.

dtype : {None, dtype}, optional

Type of the returned array and of the accumulator in which the elements are summed. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns

cumsum : ndarray.

A new array holding the result is returned unless *out* is specified, in which case a reference to *out* is returned.

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> import numpy
>>> a=numpy.array([[1,2,3],[4,5,6]])
>>> numpy.cumsum(a)      # cumulative sum = intermediate summing results & total sum. Default axis=No
array([ 1,  3,  6, 10, 15, 21])
>>> numpy.cumsum(a,dtype=float)      # specifies type of output value(s)
array([  1.,   3.,   6.,  10.,  15.,  21.])
>>> numpy.cumsum(a,axis=0)      # sum over rows for each of the 3 columns
array([[1, 2, 3],
       [5, 7, 9]])
>>> numpy.cumsum(a,axis=1)      # sum over columns for each of the 2 rows
array([[ 1,  3,  6],
       [ 4,  9, 15]])
```

2.1.83 degrees()

y = degrees(x) converts angle from radians to degrees

2.1.84 delete(arr, obj, axis=None)

Return a new array with sub-arrays along an axis deleted.

Return a new array with the sub-arrays (i.e. rows or columns) deleted along the given axis as specified by obj

obj may be a slice_object (s_[3:5:2]) or an integer or an array of integers indicated which sub-arrays to remove.

If axis is None, then ravel the array first.

Examples

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

>>> delete(arr, 1, 1)
array([[3, 5],
       [1, 3],
       [6, 8]])
>>> delete(arr, 1, 0)
array([[3, 4, 5],
       [6, 7, 8]])
```

2.1.85 deprecate(func, oldname=None, newname=None)

Deprecate old functions. Issues a DeprecationWarning, adds warning to oldname's docstring, rebinds oldname.__name__ and returns new function object.

Example: oldfunc = deprecate(newfunc, 'oldfunc', 'newfunc')

2.1.86 deprecate_with_doc(somestr)

Decorator to deprecate functions and provide detailed documentation with 'somestr' that is added to the functions docstring.

Example: depmsg = 'function scipy.foo has been merged into numpy.foobar' @deprecate_with_doc(depmsg) def foo():

Unexpected indentation.

```
pass
```

2.1.87 diag(v, k=0)

returns a copy of the the k-th diagonal if v is a 2-d array or returns a 2-d array with v as the k-th diagonal if v is a 1-d array.

2.1.88 diagflat(v, k=0)

Return a 2D array whose k'th diagonal is a flattened v and all other elements are zero.

Examples

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

```
>>> diagflat([1,2], 1)
array([[0, 1, 0],
       [0, 0, 2],
       [0, 0, 0]])
```

2.1.89 diagonal(a, offset=0, axis1=0, axis2=1)

Return specified diagonals.

If a is 2-d, returns the diagonal of self with the given offset, i.e., the collection of elements of the form a[i,i+offset]. If a has more than two dimensions, then the axes specified by axis1 and axis2 are used to determine the 2-d subarray whose diagonal is returned. The shape of the resulting array can be determined by removing axis1 and axis2 and appending an index to the right equal to the size of the resulting diagonals.

Parameters

a : {array_like}

Array from whis the diagonals are taken.

offset : {0, integer}, optional

Offset of the diagonal from the main diagonal. Can be both positive and negative. Defaults to main diagonal.

axis1 : {0, integer}, optional

Axis to be used as the first axis of the 2-d subarrays from which the diagonals should be taken. Defaults to first axis.

axis2 : {1, integer}, optional

Axis to be used as the second axis of the 2-d subarrays from which the diagonals should be taken. Defaults to second axis.

Returns

array_of_diagonals : array of same type as a

If a is 2-d, a 1-d array containing the diagonal is returned. If a has larger dimensions, then an array of diagonals is returned.

Examples

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

```
>>> a = arange(8).reshape(2,2,2)
>>> a
array([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])
>>> a.diagonal(0,-2,-1)
array([[0, 3],
       [4, 7]])
```

2.1.90 diff(a, n=1, axis=-1)

Calculate the nth order discrete difference along given axis.

2.1.91 digitize(x,bins)

Return the index of the bin to which each value of x belongs.

Each index i returned is such that bins[i-1] <= x < bins[i] if bins is monotonically increasing, or bins [i-1] > x >= bins[i] if bins is monotonically decreasing.

Beyond the bounds of the bins 0 or len(bins) is returned as appropriate.

2.1.92 disp(mesg, device=None, linefeed=True)

Display a message to the given device (default is sys.stdout) with or without a linefeed.

2.1.93 divide()

y = divide(x1,x2) divides the arguments elementwise.

2.1.94 dot()

dot(a,b) Returns the dot product of a and b for arrays of floating point types. Like the generic numpy equivalent the product sum is over the last dimension of a and the second-to-last dimension of b. NB: The first argument is not conjugated.

2.1.95 dsplit(ary, indices_or_sections)

Split ary into multiple sub-arrays along the 3rd axis (depth)

Description: Split a single array into multiple sub arrays. The array is divided into groups along the 3rd axis. If indices_or_sections is an integer, ary is divided into that many equally sized sub arrays. If it is impossible to make the sub-arrays equally sized, the operation throws a ValueError exception. See array_split and split for other options on indices_or_sections.

Arguments: **ary – N-D array.** Array to be divided into sub-arrays.

indices_or_sections – integer or 1D array. If integer, defines the number of (close to) equal sized sub-arrays. If it is a 1D array of sorted indices, it defines the indexes at which ary is divided. Any empty list results in a single sub-array equal to the original array.

Returns: sequence of sub-arrays. The returned arrays have the same number of dimensions as the input array.

Caveats: See vsplit caveats.

Related: dstack, split, array_split, hsplit, vsplit.

Examples:
```
>>> a = array([[[1,2,3,4],[1,2,3,4]]])
>>> dsplit(a,2)
[array([[[1, 2],
        [1, 2]]]), array([[[3, 4],
        [3, 4]]])]
```

2.1.96 dstack(tup)

Stack arrays in sequence depth wise (along third dimension)

Description: Take a sequence of arrays and stack them along the third axis. All arrays in the sequence must have the same shape along all but the third axis. This is a simple way to stack 2D arrays (images) into a single 3D array for processing. dstack will rebuild arrays divided by dsplit.

Arguments: **tup – sequence of arrays. All arrays must have the same** shape.

Examples:
```
>>> import numpy
>>> a = array((1,2,3))
>>> b = array((2,3,4))
>>> numpy.dstack((a,b))
array([[[1, 2],
        [2, 3],
        [3, 4]]])
>>> a = array([[1],[2],[3]])
>>> b = array([[2],[3],[4]])
>>> numpy.dstack((a,b))
array([[[1, 2]],
<BLANKLINE>
       [[2, 3]],
<BLANKLINE>
       [[3, 4]]])
```

2.1.97 ediff1d(ary, to_end=None, to_begin=None)

The differences between consecutive elements of an array, possibly with prefixed and/or appended values.

Parameters

ary : array

This array will be flattened before the difference is taken.

to_end : number, optional

If provided, this number will be tacked onto the end of the returned differences.

to_begin : number, optional

If provided, this number will be taked onto the beginning of the returned differences.

Returns

ed : array

The differences. Loosely, this will be (ary[1:] - ary[:-1]).

2.1.98 empty(shape, dtype=float, order='C')

Return a new array of given shape and type with all entries uninitialized. This can be faster than zeros.

Parameters

shape : tuple of integers

Shape of the new array

dtype : data-type

The desired data-type for the array.

order : {'C', 'F'}

Whether to store multidimensional data in C or Fortran order.

2.1.99 empty_like(a)

Return an empty (uninitialized) array of the shape and data-type of a.

Note that this does NOT initialize the returned array. If you require your array to be initialized, you should use zeros_like().

2.1.100 equal()

y = equal(x1,x2) returns elementwise x1 == x2 in a bool array

2.1.101 exp()

y = exp(x) e**x elementwise.

2.1.102 expand_dims(a, axis)

Expand the shape of a by including newaxis before given axis.

2.1.103 expm1()

y = expm1(x) e**x-1 elementwise.

2.1.104 extract(condition, arr)

Return the elements of ravel(arr) where ravel(condition) is True (in 1D).

Equivalent to compress(ravel(condition), ravel(arr)).

2.1.105 eye(N, M=None, k=0, dtype=<type 'float'>)

eye returns a N-by-M 2-d array where the k-th diagonal is all ones, and everything else is zeros.

2.1.106 fabs()

y = fabs(x) absolute values.

2.1.107 find_common_type(array_types, scalar_types)

Determine common type following standard coercion rules

Parameters

array_types : sequence

A list of dtype convertible objects representing arrays

scalar_types : sequence

A list of dtype convertible objects representing scalars

Returns

datatype : dtype

The common data-type which is the maximum of the array_types ignoring the scalar_types unless the maximum of the scalar_types is of a different kind.
If the kinds is not understood, then None is returned.

2.1.108 fix(x, y=None)

Round x to nearest integer towards zero.

2.1.109 flatnonzero(a)

Return indicies that are not-zero in flattened version of a

Equivalent to a.ravel().nonzero()[0]

```
>>> from numpy import arange, flatnonzero
>>> arange(-2, 3)
array([-2, -1,  0,  1,  2])
>>> flatnonzero(arange(-2, 3))
array([0, 1, 3, 4])
```

2.1.110 fliplr(m)

returns an array m with the rows preserved and columns flipped in the left/right direction. Works on the first two dimensions of m.

2.1.111 flipud(m)

returns an array with the columns preserved and rows flipped in the up/down direction. Works on the first dimension of m.

2.1.112 floor()

y = floor(x) elementwise largest integer <= x

2.1.113 floor_divide()

y = floor_divide(x1,x2) floor divides the arguments elementwise.

2.1.114 fmod()

y = fmod(x1,x2) computes (C-like) x1 % x2 elementwise.

2.1.115 frexp()

y1,y2 = frexp(x) Split the number, x, into a normalized fraction (y1) and exponent (y2)

2.1.116 frombuffer(buffer=, dtype=float, count=-1, offset=0)

Returns a 1-d array of data type dtype from buffer.

Parameters

buffer :

An object that exposes the buffer interface

dtype : data-type

Data type of the returned array.

count : int

Number of items to read. -1 means all data in the buffer.

offset : int

Number of bytes to jump from the start of the buffer before reading

Notes

If the buffer has data that is not in machine byte-order, then use a proper data type descriptor. The data will not be byteswapped, but the array will manage it in future operations.

2.1.117 fromfile(file=, dtype=float, count=-1, sep='')

Return an array of the given data type from a text or binary file.

Data written using the tofile() method can be conveniently recovered using this function.

Parameters

file : file or string

Open file object or string containing a file name.

dtype : data-type

Data type of the returned array. For binary files, it is also used to determine the size and order of the items in the file.

count : int

Number of items to read. -1 means all data in the whole file.

sep : string

Separator between items if file is a text file. Empty ("") separator means the file should be treated as binary.

Notes

WARNING: This function should be used sparingly as the binary files are not platform independent. In particular, they contain no endianess or datatype information. Nevertheless it can be useful for reading in simply formatted or binary data quickly.

2.1.118 fromfunction(function, shape, **kwargs)

Returns an array constructed by calling a function on a tuple of number grids.

The function should accept as many arguments as the length of shape and work on array inputs. The shape argument is a sequence of numbers indicating the length of the desired output for each axis.

The function can also accept keyword arguments (except dtype), which will be passed through fromfunction to the function itself. The dtype argument (default float) determines the data-type of the index grid passed to the function.

2.1.119 fromiter(iterable, dtype, count=-1)

Return a new 1d array initialized from iterable.

Parameters

iterable :

Iterable object from which to obtain data

dtype : data-type

Data type of the returned array.

count : int

Number of items to read. -1 means all data in the iterable.

Returns

new_array : ndarray

2.1.120 frompyfunc()

frompyfunc(func, nin, nout) take an arbitrary python function that takes nin objects as input and returns nout objects and return a universal function (ufunc). This ufunc always returns PyObject arrays

2.1.121 fromregex(file, regexp, dtype)

Construct an array from a text file, using regular-expressions parsing.

Array is constructed from all matches of the regular expression in the file. Groups in the regular expression are converted to fields.

Parameters

file : str or file

File name or file object to read.

regexp : str or regexp

Regular expression used to parse the file. Groups in the regular expression correspond to fields in the dtype.

dtype : dtype or dtype list

Dtype for the structured array

Examples

```
>>> f = open('test.dat', 'w')
>>> f.write("1312 foo\n1534  bar\n444   qux")
>>> f.close()
>>> np.fromregex('test.dat', r"(\d+)\s+(...)",
...              [('num', np.int64), ('key', 'S3')])
array([(1312L, 'foo'), (1534L, 'bar'), (444L, 'qux')],
      dtype=[('num', '<i8'), ('key', '|S3')])
```

2.1.122 fromstring(string, dtype=float, count=-1, sep='')

Return a new 1d array initialized from the raw binary data in string.

If count is positive, the new array will have count elements, otherwise its size is determined by the size of string. If sep is not empty then the string is interpreted in ASCII mode and converted to the desired number type using sep as the separator between elements (extra whitespace is ignored). ASCII integer conversions are base-10; octal and hex are not supported.

2.1.123 fv(rate, nper, pmt, pv, when='end')

future value computed by solving the equation

nper / (1 + rate*when) / nper fv + pv*(1+rate) + pmt*|————————-|*| (1+rate) - 1 | = 0
rate / /

fv + pv + pmt * nper = 0 (when rate == 0)

where (all can be scalars or sequences)

Parameters

rate : :

Rate of interest (per period) nper : Number of compounding periods pmt : Payment pv : Present value fv : Future value when : When payments are due ('begin' (1) or 'end' (0))

Example :

——— :

What is the future value after 10 years of saving $100 now, with :

an additional monthly savings of $100. Assume the interest rate is 5% (annually) compounded monthly?

>>> fv(0.05/12, 10*12, -100, -100) :

15692.928894335748 :

By convention, the negative sign represents cash flow out (i.e. money not :

available today). Thus, saving $100 a month at 5% annual interest leads to $15,692.93 available to spend in 10 years.

2.1.124 get_array_wrap(*args)

Find the wrapper for the array with the highest priority.

In case of ties, leftmost wins. If no wrapper is found, return None

2.1.125 get_include()

Return the directory in the package that contains the numpy/*.h header files.

Inline emphasis start-string without end-string.

Extension modules that need to compile against numpy should use this function to locate the appropriate include directory. Using distutils:

import numpy Extension('extension_name', ...

Unexpected indentation.

include_dirs=[numpy.get_include()])

2.1.126 get_numarray_include(type=None)

Return the directory in the package that contains the numpy/*.h header files.

Inline emphasis start-string without end-string.

Extension modules that need to compile against numpy should use this function to locate the appropriate include directory. Using distutils:

> import numpy Extension('extension_name', ...
>
> Unexpected indentation.
>
> > include_dirs=[numpy.get_numarray_include()])

2.1.127 get_numpy_include(*args, **kwds)

get_numpy_include is DEPRECATED!! – use get_include instead

Return the directory in the package that contains the numpy/*.h header Inline emphasis start-string without end-string.

files.

Extension modules that need to compile against numpy should use this function to locate the appropriate include directory. Using distutils:

> import numpy Extension('extension_name', ...
>
> Unexpected indentation.
>
> > include_dirs=[numpy.get_include()])

2.1.128 get_printoptions()

Return the current print options.

Returns dictionary of current print options with keys - precision : int - threshold : int - edgeitems : int - linewidth : int - suppress : bool - nanstr : string - infstr : string

SeeAlso

- set_printoptions : parameter descriptions

2.1.129 getbuffer(obj [,offset[, size]])

Create a buffer object from the given object referencing a slice of length size starting at offset. Default is the entire buffer. A read-write buffer is attempted followed by a read-only buffer.

2.1.130 getbufsize()

Return the size of the buffer used in ufuncs.

2.1.131 geterr()

Get the current way of handling floating-point errors.

Returns a dictionary with entries "divide", "over", "under", and "invalid", whose values are from the strings "ignore", "print", "log", "warn", "raise", and "call".

2.1.132 geterrcall()

Return the current callback function used on floating-point errors.

2.1.133 geterrobj()

Used internally by *geterr*.

Returns

errobj : list

Internal numpy buffer size, error mask, error callback function.

2.1.134 gradient(f, *varargs)

Calculate the gradient of an N-dimensional scalar function.

Uses central differences on the interior and first differences on boundaries to give the same shape.

Inputs:

f – An N-dimensional array giving samples of a scalar function
varargs – 0, 1, or N scalars giving the sample distances in each direction

Outputs:

N arrays of the same shape as f giving the derivative of f with respect to each dimension.

2.1.135 greater()

y = greater(x1,x2) returns elementwise x1 > x2 in a bool array.

2.1.136 greater_equal()

y = greater_equal(x1,x2) returns elementwise x1 >= x2 in a bool array.

2.1.137 hamming(M)

hamming(M) returns the M-point Hamming window.

2.1.138 hanning(M)

hanning(M) returns the M-point Hanning window.

2.1.139 histogram(a, bins=10, range=None, normed=False, weights=None, new=False)

Compute the histogram from a set of data.

Parameters

a : array

The data to histogram.

bins : int or sequence

If an int, then the number of equal-width bins in the given range. If new=True, bins can also be the bin edges, allowing for non-constant bin widths.

range : (float, float)

The lower and upper range of the bins. If not provided, range is simply (a.min(), a.max()). Using new=False, lower than range are ignored, and values higher than range are tallied in the rightmost bin. Using new=True, both lower and upper outliers are ignored.

normed : bool

If False, the result array will contain the number of samples in each bin. If True, the result array is the value of the probability *density* function at the bin normalized such that the *integral* over the range is 1. Note that the sum of all of the histogram values will not usually be 1; it is not a probability *mass* function.

weights : array

An array of weights, the same shape as a. If normed is False, the histogram is computed by summing the weights of the values falling into each bin. If normed is True, the weights are normalized, so that the integral of the density over the range is 1. This option is only available with new=True.

new : bool

Compatibility argument to transition from the old version (v1.1) to the new version (v1.2).

Returns

hist : array

The values of the histogram. See *normed* and *weights* for a description of the possible semantics.

bin_edges : float array

With new=False, return the left bin edges (length(hist)). With new=True, return the bin edges (length(hist)+1).

2.1.140 histogram2d(x, y, bins=10, range=None, normed=False, weights=None)

histogram2d(x,y, bins=10, range=None, normed=False) -> H, xedges, yedges

Compute the 2D histogram from samples x,y.

Parameters

- *x,y* : Sample arrays (1D).
- ***bins*** [Number of bins -or- [nbin x, nbin y] -or-] [bin edges] -or- [x bin edges, y bin edges].
- *range* : A sequence of lower and upper bin edges (default: [min, max]).
- ***normed*** [Boolean, if False, return the number of samples in each bin,] if True, returns the density.

- ***weights*** [An array of weights. The weights are normed only if normed] is True. Should weights.sum() not equal N, the total bin count will not be equal to the number of samples.

Return
- *hist* : Histogram array.
- *xedges, yedges* : Arrays defining the bin edges.

Example:

```
>>> x = random.randn(100,2)
>>> hist2d, xedges, yedges = histogram2d(x, bins = (6, 7))
```

SeeAlso histogramdd

2.1.141 histogramdd(sample, bins=10, range=None, normed=False, weights=None)

Return the N-dimensional histogram of the sample.

Parameters

sample : sequence or array

A sequence containing N arrays or an NxM array. Input data.

bins : sequence or scalar

A sequence of edge arrays, a sequence of bin counts, or a scalar which is the bin count for all dimensions. Default is 10.

range : sequence

A sequence of lower and upper bin edges. Default is [min, max].

normed : boolean

If False, return the number of samples in each bin, if True, returns the density.

weights : array

Array of weights. The weights are normed only if normed is True. Should the sum of the weights not equal N, the total bin count will not be equal to the number of samples.

Returns

hist : array

Histogram array.

edges : list

List of arrays defining the lower bin edges.

Examples

```
>>> x = random.randn(100,3)
>>> hist3d, edges = histogramdd(x, bins = (5, 6, 7))
```

2.1.142 hsplit(ary, indices_or_sections)

Split ary into multiple columns of sub-arrays

Description: Split a single array into multiple sub arrays. The array is divided into groups of columns. If indices_or_sections is an integer, ary is divided into that many equally sized sub arrays. If it is impossible to make the sub-arrays equally sized, the operation throws a ValueError exception. See array_split and split for other options on indices_or_sections.

Arguments: **ary – N-D array.** Array to be divided into sub-arrays.

indices_or_sections – integer or 1D array. If integer, defines the number of (close to) equal sized sub-arrays. If it is a 1D array of sorted indices, it defines the indexes at which ary is divided. Any empty list results in a single sub-array equal to the original array.

Returns: sequence of sub-arrays. The returned arrays have the same number of dimensions as the input array.

Related: hstack, split, array_split, vsplit, dsplit.

Examples:

```
>>> import numpy
>>> a= array((1,2,3,4))
>>> numpy.hsplit(a,2)
[array([1, 2]), array([3, 4])]
>>> a = array([[1,2,3,4],[1,2,3,4]])
>>> hsplit(a,2)
[array([[1, 2],
       [1, 2]]), array([[3, 4],
       [3, 4]])]
```

2.1.143 hstack(tup)

Stack arrays in sequence horizontally (column wise)

Description: Take a sequence of arrays and stack them horizontally to make a single array. All arrays in the sequence must have the same shape along all but the second axis. hstack will rebuild arrays divided by hsplit.

Arguments: **tup – sequence of arrays. All arrays must have the same** shape.

Examples:

```
>>> import numpy
>>> a = array((1,2,3))
>>> b = array((2,3,4))
>>> numpy.hstack((a,b))
array([1, 2, 3, 2, 3, 4])
>>> a = array([[1],[2],[3]])
>>> b = array([[2],[3],[4]])
>>> numpy.hstack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

2.1.144 hypot()

y = hypot(x1,x2) sqrt(x1**2 + x2**2) elementwise

2.1.145 i0(x)

2.1.146 identity(n, dtype=None)

Returns the identity 2-d array of shape n x n.

identity(n)[i,j] == 1 for all i == j == 0 for all i != j

2.1.147 imag(val)

Return the imaginary part of val.

Useful if val maybe a scalar or an array.

2.1.148 indices(dimensions, dtype=<type 'int'>)

Returns an array representing a grid of indices with row-only, and column-only variation.

2.1.149 info(object=None, maxwidth=76, output=<open file '<stdout>', mode 'w' at 0x16068>, toplevel='numpy')

Get help information for a function, class, or module.

Example:

```
>>> from numpy import *
>>> info(polyval) # doctest: +SKIP
```

polyval(p, x)

Evaluate the polymnomial p at x.

Description: If p is of length N, this function returns the value: p[0]*(x**N-1) + p[1]*(x**N-2) + ... + p[N-2]*x + p[N-1]

2.1.150 inner()

innerproduct(a,b) Returns the inner product of a and b for arrays of floating point types. Like the generic NumPy equivalent the product sum is over the last dimension of a and b. NB: The first argument is not conjugated.

2.1.151 insert(arr, obj, values, axis=None)

Return a new array with values inserted along the given axis before the given indices

If axis is None, then ravel the array first.

The obj argument can be an integer, a slice, or a sequence of integers.

Examples

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

```
>>> insert(a, [1,2], [[4],[5]], axis=0)
array([[1, 2, 3],
       [4, 4, 4],
       [4, 5, 6],
       [5, 5, 5],
       [7, 8, 9]])
```

2.1.152 int_asbuffer()

2.1.153 interp(x, xp, fp, left=None, right=None)

Return the value of a piecewise-linear function at each value in x.

The piecewise-linear function, f, is defined by the known data-points fp=f(xp). The xp points must be sorted in increasing order but this is not checked.

For values of x < xp[0] return the value given by left. If left is None, then return fp[0]. For values of x > xp[-1] return the value given by right. If right is None, then return fp[-1].

2.1.154 intersect1d(ar1, ar2)

Intersection of 1D arrays with unique elements.

Use unique1d() to generate arrays with only unique elements to use as inputs to this function. Alternatively, use intersect1d_nu() which will find the unique values for you.

Parameters

ar1 : array

ar2 : array

Returns

intersection : array

2.1.155 intersect1d_nu(ar1, ar2)

Intersection of 1D arrays with any elements.

The input arrays do not have unique elements like intersect1d() requires.

Parameters

ar1 : array

ar2 : array

Returns

intersection : array

2.1.156 invert()

y = invert(x) computes ~x (bit inversion) elementwise.

2.1.157 ipmt(rate, per, nper, pv, fv=0.0, when='end')

2.1.158 irr(values)

Internal Rate of Return

This is the rate of return that gives a net present value of 0.0

npv(irr(values), values) == 0.0

2.1.159 iscomplex(x)

Return a boolean array where elements are True if that element is complex (has non-zero imaginary part).

For scalars, return a boolean.

2.1.160 iscomplexobj(x)

Return True if x is a complex type or an array of complex numbers.

Unlike iscomplex(x), complex(3.0) is considered a complex object.

2.1.161 isfinite()

y = isfinite(x) returns True where x is finite

2.1.162 isfortran(a)

Returns True if 'a' is arranged in Fortran-order in memory with a.ndim > 1

2.1.163 isinf()

y = isinf(x) returns True where x is +inf or -inf

2.1.164 isnan()

y = isnan(x) returns True where x is Not-A-Number

2.1.165 isneginf(x, y=None)

Return a boolean array y with y[i] True for x[i] = -Inf.

If y is an array, the result replaces the contents of y.

2.1.166 isposinf(x, y=None)

Return a boolean array y with y[i] True for x[i] = +Inf.

If y is an array, the result replaces the contents of y.

2.1.167 isreal(x)

Return a boolean array where elements are True if that element is real (has zero imaginary part)

For scalars, return a boolean.

2.1.168 isrealobj(x)

Return True if x is not a complex type.

Unlike isreal(x), complex(3.0) is considered a complex object.

2.1.169 isscalar(num)

Returns True if the type of num is a scalar type.

2.1.170 issctype(rep)

Determines whether the given object represents a numeric array type.

2.1.171 issubclass_(arg1, arg2)

2.1.172 issubdtype(arg1, arg2)

2.1.173 issubsctype(arg1, arg2)

2.1.174 iterable(y)

2.1.175 ix_(*args)

Construct an open mesh from multiple sequences.

This function takes n 1-d sequences and returns n outputs with n dimensions each such that the shape is 1 in all but one dimension and the dimension with the non-unit shape value cycles through all n dimensions.

Using ix_() one can quickly construct index arrays that will index the cross product.

a[ix_([1,3,7],[2,5,8])] returns the array

a[1,2] a[1,5] a[1,8] a[3,2] a[3,5] a[3,8] a[7,2] a[7,5] a[7,8]

2.1.176 kaiser(M, beta)

kaiser(M, beta) returns a Kaiser window of length M with shape parameter beta.

2.1.177 kron(a, b)

kronecker product of a and b

Kronecker product of two arrays is block array [[a[0 ,0]*b, a[0 ,1]*b, ... , a[0 ,n-1]*b],

Unexpected indentation.

[... ...], [a[m-1,0]*b, a[m-1,1]*b, ... , a[m-1,n-1]*b]]

2.1.178 ldexp()

y = ldexp(x1,x2) Compute y = x1 * 2**x2.

2.1.179 left_shift()

y = left_shift(x1,x2) computes x1 << x2 (x1 shifted to left by x2 bits) elementwise.

2.1.180 less()

y = less(x1,x2) returns elementwise x1 < x2 in a bool array.

2.1.181 less_equal()

y = less_equal(x1,x2) returns elementwise x1 <= x2 in a bool array

2.1.182 lexsort()

lexsort(keys=, axis=-1) -> array of indices. Argsort with list of keys.

Perform an indirect sort using a list of keys. The first key is sorted, then the second, and so on through the list of keys. At each step the previous order is preserved when equal keys are encountered. The result is a sort on multiple keys. If the keys represented columns of a spreadsheet, for example, this would sort using multiple columns (the last key being used for the primary sort order, the second-to-last key for the secondary sort order, and so on).

Parameters

keys : (k,N) array or tuple of (N,) sequences

Array containing values that the returned indices should sort, or a sequence of things that can be converted to arrays of the same shape.

axis : integer

Axis to be indirectly sorted. Default is -1 (i.e. last axis).

Returns

indices : (N,) integer array

Array of indices that sort the keys along the specified axis.

Examples

```
>>> a = [1,5,1,4,3,6,7]
>>> b = [9,4,0,4,0,4,3]
>>> ind = lexsort((b,a))
>>> print ind
[2 0 4 3 1 5 6]
>>> print take(a,ind)
[1 1 3 4 5 6 7]
>>> print take(b,ind)
[0 9 0 4 4 4 3]
```

2.1.183 linspace(start, stop, num=50, endpoint=True, retstep=False)

Return evenly spaced numbers.

Return num evenly spaced samples from start to stop. If endpoint is True, the last sample is stop. If retstep is True then return (seq, step_value), where step_value used.

Parameters

start : {float}

The value the sequence starts at.

stop : {float}

The value the sequence stops at. If `endpoint` is false, then this is not included in the sequence. Otherwise it is guaranteed to be the last value.

num : {integer}

Number of samples to generate. Default is 50.

endpoint : {boolean}

If true, `stop` is the last sample. Otherwise, it is not included. Default is true.

retstep : {boolean}

If true, return '`(samples, step)`', where `step` is the spacing used in generating the samples.

Returns

samples : {array}

`num` equally spaced samples from the range [start, stop] or [start, stop).

step : {float} (Only if `retstep` is true)

Size of spacing between samples.

2.1.184 load(file, memmap=False)

Load a binary file.

Read a binary file (either a pickle, or a binary .npy/.npz file) and return the result.

Parameters

file : file-like object or string

the file to read. It must support seek and read methods

memmap : bool

If true, then memory-map the .npy file or unzip the .npz file into a temporary directory and memory-map each component This has no effect for a pickle.

Returns

result : array, tuple, dict, etc.

data stored in the file. If file contains pickle data, then whatever is stored in the pickle is returned. If the file is .npy file, then an array is returned. If the file is .npz file, then a dictionary-like object is returned which has a filename:array key:value pair for every file in the zip.

Raises

IOError :

2.1.185 loads()

loads(string) – Load a pickle from the given string

2.1.186 loadtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False)

Load ASCII data from fname into an array and return the array.

The data must be regular, same number of values in every row

Parameters

fname : filename or a file handle.

Support for gzipped files is automatic, if the filename ends in .gz

dtype : data-type

Data type of the resulting array. If this is a record data-type, the resulting array will be 1-d and each row will be interpreted as an element of the array. The number of columns used must match the number of fields in the data-type in this case.

comments : str

The character used to indicate the start of a comment in the file.

delimiter : str

A string-like character used to separate values in the file. If delimiter is unspecified or none, any whitespace string is a separator.

converters : {}

A dictionary mapping column number to a function that will convert that column to a float. Eg, if column 0 is a date string: converters={0:datestr2num}. Converters can also be used to provide a default value for missing data: converters={3:lambda s: float(s or 0)}.

skiprows : int

The number of rows from the top to skip.

usecols : sequence

A sequence of integer column indexes to extract where 0 is the first column, eg. usecols=(1,4,5) will extract the 2nd, 5th and 6th columns.

unpack : bool

If True, will transpose the matrix allowing you to unpack into named arguments on the left hand side.

Examples

```
>>> X = loadtxt('test.dat')  # data in two columns
>>> x,y,z = load('somefile.dat', usecols=(3,5,7), unpack=True)
>>> r = np.loadtxt('record.dat', dtype={'names':('gender','age','weight'),
          'formats': ('S1','i4', 'f4')})
```

SeeAlso: scipy.io.loadmat to read and write matfiles.

2.1.187 log()

y = log(x) logarithm base e elementwise.

2.1.188 log10()

y = log10(x) logarithm base 10 elementwise.

2.1.189 log1p()

y = log1p(x) log(1+x) to base e elementwise.

2.1.190 log2(x, y=None)

Returns the base 2 logarithm of x

If y is an array, the result replaces the contents of y.

2.1.191 logical_and()

y = logical_and(x1,x2) returns x1 and x2 elementwise.

2.1.192 logical_not()

y = logical_not(x) returns not x elementwise.

2.1.193 logical_or()

y = logical_or(x1,x2) returns x1 or x2 elementwise.

2.1.194 logical_xor()

y = logical_xor(x1,x2) returns x1 xor x2 elementwise.

2.1.195 logspace(start, stop, num=50, endpoint=True, base=10.0)

Evenly spaced numbers on a logarithmic scale.

Computes int(num) evenly spaced exponents from base**start to base**stop. If endpoint=True, then last number is base**stop

2.1.196 lookfor(what, module=None, import_modules=True, regenerate=False)

Search for objects whose documentation contains all given words. Shows a summary of matching objects, sorted roughly by relevance.

Parameters

what : str

String containing words to look for.

module : str, module

Module whose docstrings to go through.

import_modules : bool

Whether to import sub-modules in packages. Will import only modules in __all__

regenerate: bool :

Re-generate the docstring cache

2.1.197 asmatrix(data, dtype=None)

Returns ‘data’ as a matrix. Unlike matrix(), no copy is performed if ‘data’ is already a matrix or array. Equivalent to: matrix(data, copy=False)

2.1.198 amax(a, axis=None, out=None)

Return the maximum along a given axis.

Parameters

a : array_like

Input data.

axis : {None, int}, optional

Axis along which to operate. By default, `axis` is None and the flattened input is used.

out : array_like, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

Returns

amax : array_like

New array holding the result, unless `out` was specified.

Examples

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

2.1.199 maximum()

y = maximum(x1,x2) returns maximum (if x1 > x2: x1; else: x2) elementwise.

2.1.200 maximum_sctype(t)

returns the sctype of highest precision of the same general kind as 't'

2.1.201 may_share_memory(a, b)

Determine if two arrays can share memory

The memory-bounds of a and b are computed. If they overlap then this function returns True. Otherwise, it returns False.

A return of True does not necessarily mean that the two arrays share any element. It just means that they *might*.

2.1.202 mean(a, axis=None, dtype=None, out=None)

Compute the mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. The dtype returned for integer type arrays is float.

Parameters

a : {array_like}

Array containing numbers whose mean is desired. If a is not an array, a conversion is attempted.

axis : {None, integer}, optional

Axis along which the means are computed. The default is to compute the mean of the flattened array.

dtype : {None, dtype}, optional

Type to use in computing the mean. For arrays of integer type the default is float32, for arrays of float types it is the same as the array type.

out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns

mean : {array, scalar}, see dtype parameter above

If out=None, returns a new array containing the mean values, otherwise a reference to the output array is returned.

Notes

The mean is the sum of the elements along the axis divided by the number of elements.

Examples

```
>>> a = array([[1,2],[3,4]])
>>> mean(a)
2.5
>>> mean(a,0)
array([ 2.,  3.])
>>> mean(a,1)
array([ 1.5,  3.5])
```

2.1.203 median(a, axis=0, out=None, overwrite_input=False)

Compute the median along the specified axis.

Returns the median of the array elements. The median is taken over the first axis of the array by default, otherwise over the specified axis.

Parameters

a : array-like

Input array or object that can be converted to an array

axis : {int, None}, optional

Axis along which the medians are computed. The default is to compute the median along the first dimension. axis=None returns the median of the flattened array

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

overwrite_input : {False, True}, optional

If True, then allow use of memory of input array (a) for calculations. The input array will be modified by the call to median. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is False. Note that, if overwrite_input is true, and the input is not already an ndarray, an error will be raised.

Returns

median : ndarray.

A new array holding the result is returned unless out is specified, in which case a reference to out is returned. Return datatype is float64 for ints and floats smaller than float64, or the input datatype otherwise.

See Also :

———- :

mean :

Notes

Given a vector V length N, the median of V is the middle value of a sorted copy of V (Vs) - i.e. Vs[(N-1)/2], when N is odd. It is the mean of the two middle values of Vs, when N is even.

Examples

```
>>> import numpy as np
>>> from numpy import median
>>> a = np.array([[10, 7, 4], [3, 2, 1]])
>>> a
array([[10,  7,  4],
       [ 3,  2,  1]])
>>> median(a)
array([ 6.5,  4.5,  2.5])
>>> median(a, axis=None)
3.5
>>> median(a, axis=1)
array([ 7.,  2.])
```

```
>>> m = median(a)
>>> out = np.zeros_like(m)
>>> median(a, out=m)
array([ 6.5,  4.5,  2.5])
>>> m
array([ 6.5,  4.5,  2.5])
>>> b = a.copy()
>>> median(b, axis=1, overwrite_input=True)
array([ 7.,  2.])
>>> assert not np.all(a==b)
>>> b = a.copy()
>>> median(b, axis=None, overwrite_input=True)
3.5
>>> assert not np.all(a==b)
```

2.1.204 meshgrid(x, y)

For vectors x, y with lengths Nx=len(x) and Ny=len(y), return X, Y where X and Y are (Ny, Nx) shaped arrays with the elements of x and y repeated to fill the matrix

EG,

[X, Y] = meshgrid([1,2,3], [4,5,6,7])

X = 1 2 3 1 2 3 1 2 3 1 2 3
Y = 4 4 4 5 5 5 6 6 6 7 7 7

2.1.205 amin(a, axis=None, out=None)

Return the minimum along a given axis.

Parameters

a : array_like

Input data.

axis : {None, int}, optional

Axis along which to operate. By default, `axis` is None and the flattened input is used.

out : array_like, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

Returns

amin : array_like

New array holding the result, unless `out` was specified.

Examples

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

2.1.206 minimum()

y = minimum(x1,x2) returns minimum (if x1 < x2: x1; else: x2) elementwise

2.1.207 mintypecode(typechars, typeset='GDFgdf', default='d')

Return a minimum data type character from typeset that handles all typechars given

The returned type character must be the smallest size such that an array of the returned type can handle the data from an array of type t for each t in typechars (or if typechars is an array, then its dtype.char).

If the typechars does not intersect with the typeset, then default is returned.

If t in typechars is not a string then t=asarray(t).dtype.char is applied.

2.1.208 mirr(values, finance_rate, reinvest_rate)

Modified internal rate of return

Parameters

values: :

Cash flows (must contain at least one positive and one negative value) or nan is returned.

finance_rate : :

Interest rate paid on the cash flows

reinvest_rate : :

Interest rate received on the cash flows upon reinvestment

2.1.209 remainder()

y = remainder(x1,x2) computes x1-n*x2 where n is floor(x1 / x2)

2.1.210 modf()

y1,y2 = modf(x) breaks x into fractional (y1) and integral (y2) parts.

Each output has the same sign as the input.

2.1.211 msort(a)

2.1.212 multiply()

y = multiply(x1,x2) multiplies the arguments elementwise.

2.1.213 nan_to_num(x)

Returns a copy of replacing NaN's with 0 and Infs with large numbers

The following mappings are applied: NaN -> 0 Inf -> limits.double_max

Block quote ends without a blank line; unexpected unindent.

-Inf -> limits.double_min

2.1.214 nanargmax(a, axis=None)

Find the maximum over the given axis ignoring NaNs.

2.1.215 nanargmin(a, axis=None)

Find the indices of the minimium over the given axis ignoring NaNs.

2.1.216 nanmax(a, axis=None)

Find the maximum over the given axis ignoring NaNs.

2.1.217 nanmin(a, axis=None)

Find the minimium over the given axis, ignoring NaNs.

2.1.218 nansum(a, axis=None)

Sum the array over the given axis, treating NaNs as 0.

2.1.219 ndim(a)

Return the number of dimensions of a.

If a is not already an array, a conversion is attempted. Scalars are zero dimensional.

Parameters

a : {array_like}

Array whose number of dimensions are desired. If a is not an array, a conversion is attempted.

Returns

number_of_dimensions : {integer}

Returns the number of dimensions.

Examples

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

2.1.220 negative()

y = negative(x) determines -x elementwise

2.1.221 newbuffer(size)

Return a new uninitialized buffer object of size bytes

2.1.222 nonzero(a)

Return the indices of the elements of a which are not zero.

Parameters

a : {array_like}

Returns

tuple_of_arrays : {tuple}

Examples

```
>>> eye(3)[nonzero(eye(3))]
array([ 1.,  1.,  1.])
>>> nonzero(eye(3))
(array([0, 1, 2]), array([0, 1, 2]))
>>> eye(3)[nonzero(eye(3))]
array([ 1.,  1.,  1.])
```

2.1.223 not_equal()

y = not_equal(x1,x2) returns elementwise x1 != x2

Inline substitution_reference start-string without end-string.

2.1.224 nper(rate, pmt, pv, fv=0, when='end')

Number of periods found by solving the equation

nper / (1 + rate*when) / nper fv + pv*(1+rate) + pmt*|————————-|*| (1+rate) - 1 | = 0
rate / /

fv + pv + pmt * nper = 0 (when rate == 0)

where (all can be scalars or sequences)

Parameters

rate : :

Rate of interest (per period) nper : Number of compounding periods pmt : Payment pv : Present value fv : Future value when : When payments are due ('begin' (1) or 'end' (0))

Examples

If you only had $150 to spend as payment, how long would it take to pay-off a loan of $8,000 at 7% annual interest?

```
>>> nper(0.07/12, -150, 8000)
64.073348770661852
```

So, over 64 months would be required to pay off the loan.

The same analysis could be done with several different interest rates and/or payments and/or total amounts to produce an entire table.

```
>>> nper(*(ogrid[0.06/12:0.071/12:0.01/12, -200:-99:100, 6000:7001:1000]))
array([[[ 32.58497782,  38.57048452],
        [ 71.51317802,  86.37179563]],
```

[[33.07413144, 39.26244268], [74.06368256, 90.22989997]]])

2.1.225 npv(rate, values)

Net Present Value

sum (values_k / (1+rate)**k, k = 1..n)

2.1.226 obj2sctype(rep, default=None)

2.1.227 ones(shape, dtype=None, order='C')

Returns an array of the given dimensions which is initialized to all ones.

2.1.228 ones_like()

y = ones_like(x) returns an array of ones of the shape and typecode of x.

2.1.229 outer(a, b)

Returns the outer product of two vectors.

result[i,j] = a[i]*b[j] when a and b are vectors. Will accept any arguments that can be made into vectors.

2.1.230 packbits()

out = numpy.packbits(myarray, axis=None)

myarray : an integer type array whose elements should be packed to bits

> This routine packs the elements of a binary-valued dataset into a NumPy array of type uint8 ('B') whose bits correspond to the logical (0 or nonzero) value of the input elements. The dimension over-which bit-packing is done is given by axis. The shape of the output has the same number of dimensions as the input (unless axis is None, in which case the output is 1-d).
>
> > Example: >>> a = array([[[1,0,1], ... [0,1,0]], ... [[1,1,0], ... [0,0,1]]]) >>> b = numpy.packbits(a,axis=-1) >>> b array([[[160],[64]],[[192],[32]]], dtype=uint8)
> >
> > **Note that 160 = 128 + 32** 192 = 128 + 64

2.1.231 piecewise(x, condlist, funclist, *args, **kw)

Return a piecewise-defined function.

x is the domain

condlist is a list of boolean arrays or a single boolean array The length of the condition list must be n2 or n2-1 where n2 is the length of the function list. If len(condlist)==n2-1, then an 'otherwise' condition is formed by |'ing all the conditions and inverting.

Inline substitution_reference start-string without end-string.

funclist is a list of functions to call of length (n2). Each function should return an array output for an array input Each function can take (the same set) of extra arguments and keyword arguments which are passed in after the function list. A constant may be used in funclist for a function that returns a constant (e.g. val and lambda x: val are equivalent in a funclist).

The output is the same shape and type as x and is found by calling the functions on the appropriate portions of x.

Note: This is similar to choose or select, except the the functions are only evaluated on elements of x that satisfy the corresponding condition.

The result is |– | f1(x) for condition1

Inline substitution_reference start-string without end-string.

Block quote ends without a blank line; unexpected unindent.

y = –| f2(x) for condition2 ...

fn(x) for conditionn

Line block ends without a blank line.

|–

Inline substitution_reference start-string without end-string.

2.1.232 pkgload(*packages, **options)

Load one or more packages into parent package top-level namespace.

This function is intended to shorten the need to import many subpackages, say of scipy, constantly with statements such as

    import scipy.linalg, scipy.fftpack, scipy.etc...

Instead, you can say:

    import scipy scipy.pkgload('linalg','fftpack',...)

or

    scipy.pkgload()

to load all of them in one call.

If a name which doesn't exist in scipy's namespace is given, a warning is shown.

Parameters

***packges** : arg-tuple

the names (one or more strings) of all the modules one wishes to load into the top-level namespace. verbose= : integer verbosity level [default: -1]. verbose=-1 will suspend also warnings. force= : bool when True, force reloading loaded packages [default: False]. postpone= : bool when True, don't load packages [default: False]

2.1.233 place(arr, mask, vals)

Similar to putmask arr[mask] = vals but the 1D array vals has the same number of elements as the non-zero values of mask. Inverse of extract.

2.1.234 pmt(rate, nper, pv, fv=0, when='end')

Payment computed by solving the equation

nper / (1 + rate*when) / nper fv + pv*(1+rate) + pmt*|————————-|*| (1+rate) - 1 | = 0
rate / /

fv + pv + pmt * nper = 0 (when rate == 0)

where (all can be scalars or sequences)

Parameters

rate : :

Rate of interest (per period) nper : Number of compounding periods pmt : Payment pv : Present value fv : Future value when : When payments are due ('begin' (1) or 'end' (0))

Examples

What would the monthly payment need to be to pay off a $200,000 loan in 15 years at an annual interest rate of 7.5%?

```
>>> pmt(0.075/12, 12*15, 200000)
-1854.0247200054619
```

In order to pay-off (i.e. have a future-value of 0) the $200,000 obtained today, a monthly payment of $1,854.02 would be required.

2.1.235 poly(seq_of_zeros)

Return a sequence representing a polynomial given a sequence of roots.

If the input is a matrix, return the characteristic polynomial.

Example:

```
>>> b = roots([1,3,1,5,6])
>>> poly(b)
array([ 1.,  3.,  1.,  5.,  6.])
```

2.1.236 polyadd(a1, a2)

Adds two polynomials represented as sequences

2.1.237 polyder(p, m=1)

Return the mth derivative of the polynomial p.

2.1.238 polydiv(u, v)

Computes q and r polynomials so that u(s) = q(s)*v(s) + r(s) and deg r < deg v.

2.1.239 polyfit(x, y, deg, rcond=None, full=False)

Least squares polynomial fit.

Do a best fit polynomial of degree 'deg' of 'x' to 'y'. Return value is a vector of polynomial coefficients [pk ... p1 p0]. Eg, for n=2

p2*x0^2 + p1*x0 + p0 = y1 p2*x1^2 + p1*x1 + p0 = y1 p2*x2^2 + p1*x2 + p0 = y2 ..... p2*xk^2 + p1*xk + p0 = yk

Parameters

x : array_like

1D vector of sample points.

y : array_like

1D vector or 2D array of values to fit. The values should run down the columes in the 2D case.

deg : integer

Degree of the fitting polynomial

rcond: {None, float}, optional :

Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The defaul value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

full : {False, boolean}, optional

Switch determining nature of return value. When it is False just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

Returns

coefficients, [residuals, rank, singular_values, rcond] : variable

When full=False, only the coefficients are returned, running down the appropriate colume when y is a 2D array. When full=True, the rank of the scaled Vandermonde matrix, it's effective rank in light of the rcond value, its singular values, and the specified value of rcond are also returned.

Notes

If X is a the Vandermonde Matrix computed from x (see http://mathworld.wolfram.com/VandermondeMatrix.html), then the polynomial least squares solution is given by the 'p' in

X*p = y

where X.shape is a matrix of dimensions (len(x), deg + 1), p is a vector of dimensions (deg + 1, 1), and y is a vector of dimensions (len(x), 1).

This equation can be solved as

p = (XT*X)^-1 * XT * y

where XT is the transpose of X and -1 denotes the inverse. However, this method is susceptible to rounding errors and generally the singular value decomposition of the matrix X is preferred and that is what is done here. The singular value method takes a paramenter, 'rcond', which sets a limit on the relative size of the smallest singular value to be used in solving the equation. This may result in lowering the rank of the Vandermonde matrix, in which case a RankWarning is issued. If polyfit issues a RankWarning, try a fit of lower degree or replace x by x - x.mean(), both of which will generally improve the condition number. The routine already normalizes the vector x by its maximum absolute value to help in this regard. The rcond parameter can be set to a value smaller than its default, but the resulting fit may be spurious. The current default value of rcond is len(x)*eps, where eps is the relative precision of the floating type being used, generally around 1e-7 and 2e-16 for IEEE single and double precision respectively. This value of rcond is fairly conservative but works pretty well when x - x.mean() is used in place of x.

DISCLAIMER: Power series fits are full of pitfalls for the unwary once the degree of the fit becomes large or the interval of sample points is badly centered. The problem is that the powers x**n are generally a poor basis for the polynomial functions on the sample interval, resulting in a Vandermonde matrix is ill conditioned and coefficients sensitive to rounding erros. The computation of the polynomial values

will also sensitive to rounding errors. Consequently, the quality of the polynomial fit should be checked against the data whenever the condition number is large. The quality of polynomial fits *can not* be taken for granted. If all you want to do is draw a smooth curve through the y values and polyfit is not doing the job, try centering the sample range or look into scipy.interpolate, which includes some nice spline fitting functions that may be of use.

For more info, see http://mathworld.wolfram.com/LeastSquaresFittingPolynomial.html, but note that the k's and n's in the superscripts and subscripts on that page. The linear algebra is correct, however.

2.1.240 polyint(p, m=1, k=None)

Return the mth analytical integral of the polynomial p.

If k is None, then zero-valued constants of integration are used. otherwise, k should be a list of length m (or a scalar if m=1) to represent the constants of integration to use for each integration (starting with k[0])

2.1.241 polymul(a1, a2)

Multiplies two polynomials represented as sequences.

2.1.242 polysub(a1, a2)

Subtracts two polynomials represented as sequences

2.1.243 polyval(p, x)

Evaluate the polynomial p at x.

If p is of length N, this function returns the value:

p[0]*(x**N-1) + p[1]*(x**N-2) + ... + p[N-2]*x + p[N-1]

If x is a sequence then p(x) will be returned for all elements of x. If x is another polynomial then the composite polynomial p(x) will be returned.

Parameters

p : {array_like, poly1d}

1D array of polynomial coefficients from highest degree to zero or an instance of poly1d.

x : {array_like, poly1d}

A number, a 1D array of numbers, or an instance of poly1d.

Returns

values : {array, poly1d}

If either p or x is an instance of poly1d, then an instance of poly1d is returned, otherwise a 1D array is returned. In the case where x is a poly1d, the result is the composition of the two polynomials, i.e., substitution is used.

Notes

Horners method is used to evaluate the polynomial. Even so, for polynomial if high degree the values may be inaccurate due to rounding errors. Use carefully.

2.1.244 power()

y = power(x1,x2) computes x1**x2 elementwise.

2.1.245 ppmt(rate, per, nper, pv, fv=0.0, when='end')

2.1.246 prod(a, axis=None, dtype=None, out=None)

Return the product of the array elements over the given axis

Parameters

a : {array_like}

Array containing elements whose product is desired. If a is not an array, a conversion is attempted.

axis : {None, integer}

Axis over which the product is taken. If None is used, then the product is over all the array elements.

dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns

product_along_axis : {array, scalar}, see dtype parameter above.

Returns an array whose shape is the same as a with the specified axis removed. Returns a 0d array when a is 1d or axis=None. Returns a reference to the specified output array if specified.

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> prod([1.,2.])
2.0
>>> prod([1.,2.], dtype=int32)
2
>>> prod([[1.,2.],[3.,4.]])
24.0
>>> prod([[1.,2.],[3.,4.]], axis=1)
array([  2.,  12.])
```

2.1.247 product(a, axis=None, dtype=None, out=None)

Return the product of the array elements over the given axis

Parameters

a : {array_like}

Array containing elements whose product is desired. If a is not an array, a conversion is attempted.

axis : {None, integer}

Axis over which the product is taken. If None is used, then the product is over all the array elements.

dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns

product_along_axis : {array, scalar}, see dtype parameter above.

Returns an array whose shape is the same as a with the specified axis removed. Returns a 0d array when a is 1d or axis=None. Returns a reference to the specified output array if specified.

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> product([1.,2.])
2.0
>>> product([1.,2.], dtype=int32)
2
>>> product([[1.,2.],[3.,4.]])
24.0
>>> product([[1.,2.],[3.,4.]], axis=1)
array([  2.,  12.])
```

2.1.248 ptp(a, axis=None, out=None)

Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).

Parameters

a : array_like

Input values.

axis : {None, int}, optional

Axis along which to find the peaks. If None (default) the flattened array is used.

out : array_like

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns

ptp : ndarray.

A new array holding the result, unless `out` was specified, in which case a reference to `out` is returned.

Examples

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

2.1.249 put(a, ind, v, mode='raise')

Set a.flat[n] = v[n] for all n in ind. If v is shorter than ind, it will repeat.

Parameters

a : array_like (contiguous)

Target array.

ind : array_like

Target indices, interpreted as integers.

v : array_like

Values to place in *a* at target indices.

mode : {'raise', 'wrap', 'clip'}, optional

Specifies how out-of-bounds indices will behave. 'raise' – raise an error 'wrap' – wrap around 'clip' – clip to the range

Notes

If v is shorter than mask it will be repeated as necessary. In particular v can be a scalar or length 1 array. The routine put is the equivalent of the following (although the loop is in C for speed):

ind = array(indices, copy=False) v = array(values, copy=False).astype(a.dtype) for i in ind: a.flat[i] = v[i]

Examples

```
>>> x = np.arange(5)
>>> np.put(x,[0,2,4],[-1,-2,-3])
>>> print x
[-1  1 -2  3 -3]
```

2.1.250 putmask(a, mask, values)

Sets a.flat[n] = values[n] for each n where mask.flat[n] is true.

If values is not the same size as *a* and *mask* then it will repeat. This gives behavior different from a[mask] = values.

Parameters

a : {array_like}

Array to put data into

mask : {array_like}

Boolean mask array

values : {array_like}

Values to put

2.1.251 pv(rate, nper, pmt, fv=0.0, when='end')

Number of periods found by solving the equation

nper / (1 + rate*when) / nper fv + pv*(1+rate) + pmt*|————————-|*| (1+rate) - 1 | = 0
rate / /

fv + pv + pmt * nper = 0 (when rate == 0)

where (all can be scalars or sequences)

Parameters

rate : :

Rate of interest (per period) nper : Number of compounding periods pmt : Payment pv : Present value fv : Future value when : When payments are due ('begin' (1) or 'end' (0))

2.1.252 radians()

y = radians(x) converts angle from degrees to radians

2.1.253 rank(a)

Return the number of dimensions of a.

In old Numeric, rank was the term used for the number of dimensions. If a is not already an array, a conversion is attempted. Scalars are zero dimensional.

Parameters

a : {array_like}

Array whose number of dimensions is desired. If a is not an array, a conversion is attempted.

Returns

number_of_dimensions : {integer}

Returns the number of dimensions.

Examples

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

2.1.254 rate(nper, pmt, pv, fv, when='end', guess=0.10000000000000001, tol=9.9999999999999995e-07, maxiter=100)

Number of periods found by solving the equation

nper / (1 + rate*when) / nper fv + pv*(1+rate) + pmt*|————————-|*| (1+rate) - 1 | = 0
rate / /

fv + pv + pmt * nper = 0 (when rate == 0)

where (all can be scalars or sequences)

Parameters

rate : :

Rate of interest (per period) nper : Number of compounding periods pmt : Payment pv : Present value fv : Future value when : When payments are due ('begin' (1) or 'end' (0))

2.1.255 ravel(a, order='C')

Return a 1d array containing the elements of a (copy only if needed).

Returns the elements of a as a 1d array. The elements in the new array are taken in the order specified by the order keyword. The new array is a view of a if possible, otherwise it is a copy.

Parameters

a : {array_like}

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

If order is 'C' the elements are taken in row major order. If order is 'F' they are taken in column major order.

Returns

1d_array : {array}

Examples

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

2.1.256 real(val)

Return the real part of val.

Useful if val maybe a scalar or an array.

2.1.257 real_if_close(a, tol=100)

If a is a complex array, return it as a real array if the imaginary part is close enough to zero.

"Close enough" is defined as tol*(machine epsilon of a's element type).

2.1.258 reciprocal()

y = reciprocal(x) compute 1/x

2.1.259 remainder()

y = remainder(x1,x2) computes x1-n*x2 where n is floor(x1 / x2)

2.1.260 repeat(a, repeats, axis=None)

Repeat elements of an array.

Parameters

a : {array_like}

Input array.

repeats : {integer, integer_array}

The number of repetitions for each element. If a plain integer, then it is applied to all elements. If an array, it needs to be of the same length as the chosen axis.

axis : {None, integer}, optional

The axis along which to repeat values. If None, then this function will operated on the flattened array *a* and return a similarly flat result.

Returns

repeated_array : array

Examples

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

2.1.261 require(a, dtype=None, requirements=None)

Return an ndarray of the provided type that satisfies requirements.

This function is useful to be sure that an array with the correct flags is returned for passing to compiled code (perhaps through ctypes).

Parameters

a : array-like

The object to be converted to a type-and-requirement satisfying array dtype : data-type The required data-type (None is the default data-type – float64) requirements : list of strings The requirements list can be any of the 'ENSUREARRAY' ('E') - ensure that a base-class ndarray 'F_CONTIGUOUS' ('F') - ensure a Fortran-contiguous array 'C_CONTIGUOUS' ('C') - ensure a C-contiguous array 'ALIGNED' ('A') - ensure a data-type aligned array 'WRITEABLE' ('W') - ensure a writeable array 'OWNDATA' ('O') - ensure an array that owns its own data

The returned array will be guaranteed to have the listed requirements by making a copy if needed.

2.1.262 reshape(a, newshape, order='C')

Returns an array containing the data of a, but with a new shape.

Parameters

a : array

Array to be reshaped.

newshape : shape tuple or int

The new shape should be compatible with the original shape. If an integer, then the result will be a 1D array of that length.

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

Determines whether the array data should be viewed as in C (row-major) order or FORTRAN (column-major) order.

Returns

reshaped_array : array

This will be a new view object if possible; otherwise, it will be a copy.

2.1.263 resize(a, new_shape)

Return a new array with the specified shape.

The original array's total size can be any size. The new array is filled with repeated copies of a.

Note that a.resize(new_shape) will fill the array with 0's beyond current definition of a.

Parameters

a : {array_like}

Array to be reshaped.

new_shape : {tuple}

Shape of reshaped array.

Returns

reshaped_array : {array}

The new array is formed from the data in the old array, repeated if necessary to fill out the required number of elements, with the new shape.

2.1.264 restoredot()

restoredot() restores dots to defaults.

2.1.265 right_shift()

y = right_shift(x1,x2) computes x1 >> x2 (x1 shifted to right by x2 bits) elementwise.

2.1.266 rint()

y = rint(x) round x elementwise to the nearest integer, round halfway cases away from zero

2.1.267 roll(a, shift, axis=None)

Roll the elements in the array by 'shift' positions along the given axis.

```
>>> from numpy import roll
>>> arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> roll(arange(10), 2)
array([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])
```

2.1.268 rollaxis(a, axis, start=0)

Return transposed array so that axis is rolled before start.

```
>>> from numpy import ones, rollaxis
>>> a = ones((3,4,5,6))
>>> rollaxis(a, 3, 1).shape
(3, 6, 4, 5)
>>> rollaxis(a, 2, 0).shape
(5, 3, 4, 6)
>>> rollaxis(a, 1, 4).shape
(3, 5, 6, 4)
```

2.1.269 roots(p)

Return the roots of the polynomial coefficients in p.

The values in the rank-1 array p are coefficients of a polynomial. If the length of p is n+1 then the polynomial is p[0] * x**n + p[1] * x**(n-1) + ... + p[n-1]*x + p[n]

2.1.270 rot90(m, k=1)

returns the array found by rotating m by k*90 degrees in the counterclockwise direction. Works on the first two dimensions of m.

2.1.271 round_(a, decimals=0, out=None)

Round a to the given number of decimals.

The real and imaginary parts of complex numbers are rounded separately. The result of rounding a float is a float so the type must be cast if integers are desired. Nothing is done if the input is an integer array and the decimals parameter has a value >= 0.

Parameters

a : {array_like}

Array containing numbers whose rounded values are desired. If a is not an array, a conversion is attempted.

decimals : {0, integer}, optional

Number of decimal places to round to. When decimals is negative it specifies the number of positions to the left of the decimal point.

out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns

rounded_array : {array}

If out=None, returns a new array of the same type as a containing the rounded values, otherwise a reference to the output array is returned.

Notes

Numpy rounds to even. Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc. Results may also be surprising due to the inexact representation of decimal fractions in IEEE floating point and the errors introduced when scaling by powers of ten.

Examples

```
>>> round_([.5, 1.5, 2.5, 3.5, 4.5])
array([ 0.,  2.,  2.,  4.,  4.])
>>> round_([1,2,3,11], decimals=1)
array([ 1,  2,  3, 11])
>>> round_([1,2,3,11], decimals=-1)
array([ 0,  0,  0, 10])
```

2.1.272 round_(a, decimals=0, out=None)

Round a to the given number of decimals.

The real and imaginary parts of complex numbers are rounded separately. The result of rounding a float is a float so the type must be cast if integers are desired. Nothing is done if the input is an integer array and the decimals parameter has a value >= 0.

Parameters

a : {array_like}

Array containing numbers whose rounded values are desired. If a is not an array, a conversion is attempted.

decimals : {0, integer}, optional

Number of decimal places to round to. When decimals is negative it specifies the number of positions to the left of the decimal point.

out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns

rounded_array : {array}

If out=None, returns a new array of the same type as a containing the rounded values, otherwise a reference to the output array is returned.

Notes

Numpy rounds to even. Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc. Results may also be surprising due to the inexact representation of decimal fractions in IEEE floating point and the errors introduced when scaling by powers of ten.

Examples

```
>>> round_([.5, 1.5, 2.5, 3.5, 4.5])
array([ 0.,  2.,  2.,  4.,  4.])
>>> round_([1,2,3,11], decimals=1)
```

```
array([ 1,  2,  3, 11])
>>> round_([1,2,3,11], decimals=-1)
array([ 0,  0,  0, 10])
```

2.1.273 vstack(tup)

Stack arrays in sequence vertically (row wise)

Description: Take a sequence of arrays and stack them vertically to make a single array. All arrays in the sequence must have the same shape along all but the first axis. vstack will rebuild arrays divided by vsplit.

Arguments: **tup – sequence of arrays. All arrays must have the same** shape.

Examples:

```
>>> import numpy
>>> a = array((1,2,3))
>>> b = array((2,3,4))
>>> numpy.vstack((a,b))
array([[1, 2, 3],
       [2, 3, 4]])
>>> a = array([[1],[2],[3]])
>>> b = array([[2],[3],[4]])
>>> numpy.vstack((a,b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```

2.1.274 safe_eval(source)

Evaluate a string containing a Python literal expression without allowing the execution of arbitrary non-literal code.

Parameters

source : str

Returns

obj : object

Raises

SyntaxError if the code is invalid Python expression syntax or if it :

contains non-literal code. :

Examples

```
>>> from numpy.lib.utils import safe_eval
>>> safe_eval('1')
1
>>> safe_eval('[1, 2, 3]')
[1, 2, 3]
>>> safe_eval('{"foo": ("bar", 10.0)}')
{'foo': ('bar', 10.0)}
>>> safe_eval('import os')
Traceback (most recent call last):
...
SyntaxError: invalid syntax
>>> safe_eval('open("/home/user/.ssh/id_dsa").read()')
Traceback (most recent call last):
...
SyntaxError: Unsupported source construct: compiler.ast.CallFunc
>>> safe_eval('dict')
Traceback (most recent call last):
...
SyntaxError: Unknown name: dict
```

2.1.275 save(file, arr)

Save an array to a binary file (a string or file-like object).

If the file is a string, then if it does not have the .npy extension, it is appended and a file open.

Data is saved to the open file in NumPy-array format

Examples

import numpy as np ... np.save('myfile', a) a = np.load('myfile.npy')

2.1.276 savetxt(fname, X, fmt='%.18e', delimiter=' ')

Save the data in X to file fname using fmt string to convert the data to strings

Parameters

fname : filename or a file handle

If the filename ends in .gz, the file is automatically saved in compressed gzip format. The load() command understands gzipped files transparently.

X : array or sequence

Data to write to file.

fmt : string or sequence of strings

A single format (%10.5f), a sequence of formats, or a multi-format string, e.g. 'Iteration %d – %10.5f', in which case delimiter is ignored.

delimiter : str

Character separating columns.

Examples

```
>>> savetxt('test.out', x, delimiter=',') # X is an array
>>> savetxt('test.out', (x,y,z)) # x,y,z equal sized 1D arrays
>>> savetxt('test.out', x, fmt='%1.4e') # use exponential notation
```

2.1.277 savez(file, *args, **kwds)

Save several arrays into an .npz file format which is a zipped-archive of arrays

If keyword arguments are given, then filenames are taken from the keywords. If arguments are passed in with no keywords, then stored file names are arr_0, arr_1, etc.

2.1.278 sctype2char(sctype)

2.1.279 searchsorted(a, v, side='left')

Return indices where keys in v should be inserted to maintain order.

Find the indices into a sorted array such that if the corresponding keys in v were inserted before the indices the order of a would be preserved. If side='left', then the first such index is returned. If side='right', then the last such index is returned. If there is no such index because the key is out of bounds, then the length of a is returned, i.e., the key would need to be appended. The returned index array has the same shape as v.

Parameters

a : 1-d array

Array must be sorted in ascending order.

v : array or list type

Array of keys to be searched for in a.

side : {'left', 'right'}, optional

If 'left', the index of the first location where the key could be inserted is found, if 'right', the index of the last such element is returned. If the is no such element, then either 0 or N is returned, where N is the size of the array.

Returns

indices : integer array

Array of insertion points with the same shape as v.

Notes

The array a must be 1-d and is assumed to be sorted in ascending order. Searchsorted uses binary search to find the required insertion points.

Examples

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

2.1.280 select(condlist, choicelist, default=0)

Return an array composed of different elements in choicelist, depending on the list of conditions.

Parameters **condlist** [list of N boolean arrays of length M] The conditions C_0 through C_(N-1) which determine from which vector the output elements are taken.

choicelist [list of N arrays of length M] Th vectors V_0 through V_(N-1), from which the output elements are chosen.

Returns **output** [1-dimensional array of length M] The output at position m is the m-th element of the first vector V_n for which C_n[m] is non-zero. Note that the output depends on the order of conditions, since the first satisfied condition is used.
Equivalent to:
output = [] for m in range(M):
Unexpected indentation.
output += [V[m] for V,C in zip(values,cond) if C[m]] or [default]

2.1.281 set_numeric_ops(op=func, ...)

Set some or all of the number methods for all array objects. Do not forget **dict can be used as the argument list. Return the functions that were replaced, which can be stored and set later.

Inline strong start-string without end-string.

2.1.282 set_printoptions(precision=None, threshold=None, edgeitems=None, linewidth=None, suppress=None, nanstr=None, infstr=None)

Set options associated with printing.

Parameters **precision** [int] Number of digits of precision for floating point output (default 8).

threshold [int] Total number of array elements which trigger summarization rather than full repr (default 1000).

edgeitems [int] Number of array items in summary at beginning and end of each dimension (default 3).

linewidth [int] The number of characters per line for the purpose of inserting line breaks (default 75).

suppress [bool] Whether or not suppress printing of small floating point values using scientific notation (default False).

nanstr [string] String representation of floating point not-a-number (default nan).

infstr [string] String representation of floating point infinity (default inf).

2.1.283 set_string_function(f, repr=1)

Set the python function f to be the function used to obtain a pretty printable string version of an array whenever an array is printed. f(M) should expect an array argument M, and should return a string consisting of the desired representation of M for printing.

2.1.284 setbufsize(size)

Set the size of the buffer used in ufuncs.

2.1.285 setdiff1d(ar1, ar2)

Set difference of 1D arrays with unique elements.

Use unique1d() to generate arrays with only unique elements to use as inputs to this function.

Parameters

ar1 : array

ar2 : array

Returns

difference : array

The values in ar1 that are not in ar2.

2.1.286 seterr(all=None, divide=None, over=None, under=None, invalid=None)

Set how floating-point errors are handled.

Valid values for each type of error are the strings “ignore”, “warn”, “raise”, and “call”. Returns the old settings. If ‘all’ is specified, values that are not otherwise specified will be set to ‘all’, otherwise they will retain their old values.

Note that operations on integer scalar types (such as int16) are handled like floating point, and are affected by these settings.

Example:

```
>>> seterr(over='raise') # doctest: +SKIP
{'over': 'ignore', 'divide': 'ignore', 'invalid': 'ignore', 'under': 'ignore'}


>>> seterr(all='warn', over='raise') # doctest: +SKIP
{'over': 'raise', 'divide': 'ignore', 'invalid': 'ignore', 'under': 'ignore'}


>>> int16(32000) * int16(3) # doctest: +SKIP
Traceback (most recent call last):
      File "<stdin>", line 1, in ?
FloatingPointError: overflow encountered in short_scalars
>>> seterr(all='ignore') # doctest: +SKIP
{'over': 'ignore', 'divide': 'ignore', 'invalid': 'ignore', 'under': 'ignore'}
```

2.1.287 seterrcall(func)

Set the callback function used when a floating-point error handler is set to 'call' or the object with a write method for use when the floating-point error handler is set to 'log'

'func' should be a function that takes two arguments. The first is type of error ("divide", "over", "under", or "invalid"), and the second is the status flag (= divide + 2*over + 4*under + 8*invalid).

Returns the old handler.

2.1.288 seterrobj()

Used internally by *seterr*.

Parameters

errobj : list

[buffer_size, error_mask, callback_func]

2.1.289 setmember1d(ar1, ar2)

Return a boolean array of shape of ar1 containing True where the elements of ar1 are in ar2 and False otherwise.

Use unique1d() to generate arrays with only unique elements to use as inputs to this function.

Parameters

ar1 : array

ar2 : array

Returns

mask : bool array

The values ar1[mask] are in ar2.

2.1.290 setxor1d(ar1, ar2)

Set exclusive-or of 1D arrays with unique elements.

Use unique1d() to generate arrays with only unique elements to use as inputs to this function.

Parameters

ar1 : array

ar2 : array

Returns

xor : array

The values that are only in one, but not both, of the input arrays.

2.1.291 shape(a)

Return the shape of a.

Parameters

a : {array_like}

Array whose shape is desired. If a is not an array, a conversion is attempted.

Returns

tuple_of_integers : :

The elements of the tuple are the length of the corresponding array dimension.

Examples

```
>>> shape(eye(3))
(3, 3)
>>> shape([[1,2]])
(1, 2)
```

2.1.292 show()

2.1.293 sign()

y = sign(x) returns -1 if x < 0 and 0 if x==0 and 1 if x > 0

2.1.294 signbit()

y = signbit(x) returns True where signbit of x is set (x<0).

2.1.295 sin()

y = sin(x) sine elementwise.

2.1.296 sinc(x)

sinc(x) returns sin(pi*x)/(pi*x) at all points of array x.

2.1.297 sinh()

y = sinh(x) hyperbolic sine elementwise.

2.1.298 size(a, axis=None)

Return the number of elements along given axis.

Parameters

a : {array_like}

Array whose axis size is desired. If a is not an array, a conversion is attempted.

axis : {None, integer}, optional

Axis along which the elements are counted. None means all elements in the array.

Returns

element_count : {integer}

Count of elements along specified axis.

Examples

```
>>> a = array([[1,2,3],[4,5,6]])
>>> size(a)
6
>>> size(a,1)
3
>>> size(a,0)
2
```

2.1.299 sometrue(a, axis=None, out=None)

Assert whether some values are true.

sometrue performs a logical_or over the given axis.

Parameters

a : array_like

Array on which to operate.

axis : {None, integer}

Axis to perform the operation over. If *None* (default), perform over flattened array.

out : {None, array}, optional

Array into which the product can be placed. Its type is preserved and it must be of the right shape to hold the output.

Examples

```
>>> b = numpy.array([True, False, True, True])
>>> numpy.sometrue(b)
True
>>> a = numpy.array([1, 5, 2, 7])
>>> numpy.sometrue(a >= 5)
True
```

2.1.300 sort(a, axis=-1, kind='quicksort', order=None)

Return copy of 'a' sorted along the given axis.

Perform an inplace sort along the given axis using the algorithm specified by the kind keyword.

Parameters

a : array

Array to be sorted.

axis : {None, int} optional

Axis along which to sort. None indicates that the flattened array should be used.

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

Sorting algorithm to use.

order : {None, list type}, optional

When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

Returns

sorted_array : array of same type as a

Notes

The various sorts are characterized by average speed, worst case performance, need for work space, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The three available algorithms have the following properties:

kind	speed	worst case	work space	stable
'quicksort'	1	O(n^2)	0	no
'mergesort'	2	O(n*log(n))	~n/2	yes
'heapsort'	3	O(n*log(n))	0	no

All the sort algorithms make temporary copies of the data when the sort is not along the last axis. Consequently, sorts along the last axis are faster and use less space than sorts along other axis.

2.1.301 sort_complex(a)

Sort 'a' as a complex array using the real part first and then the imaginary part if the real part is equal (the default sort order for complex arrays). This function is a wrapper ensuring a complex return type.

2.1.302 source(object, output=<open file '<stdout>', mode 'w' at 0x16068>)

Write source for this object to output.

2.1.303 split(ary, indices_or_sections, axis=0)

Divide an array into a list of sub-arrays.

Description: Divide ary into a list of sub-arrays along the specified axis. If indices_or_sections is an integer, ary is divided into that many equally sized arrays. If it is impossible to make an equal split, an error is raised. This is the only way this function differs from the array_split() function. If indices_or_sections is a list of sorted integers, its entries define the indexes where ary is split.

Arguments: **ary – N-D array.** Array to be divided into sub-arrays.

indices_or_sections – integer or 1D array. If integer, defines the number of (close to) equal sized sub-arrays. If it is a 1D array of sorted indices, it defines the indexes at which ary is divided. Any empty list results in a single sub-array equal to the original array.

axis – integer. default=0. Specifies the axis along which to split ary.

Caveats: Currently, the default for axis is 0. This means a 2D array is divided into multiple groups of rows. This seems like the appropriate default

2.1.304 sqrt()

y = sqrt(x) square-root elementwise. For real x, the domain is restricted to x>=0.

2.1.305 square()

y = square(x) compute x**2.

2.1.306 squeeze(a)

Remove single-dimensional entries from the shape of a.

Examples

```
>>> x = array([[[1,1,1],[2,2,2],[3,3,3]]])
>>> x.shape
(1, 3, 3)
>>> squeeze(x).shape
(3, 3)
```

2.1.307 std(a, axis=None, dtype=None, out=None, ddof=0)

Compute the standard deviation along the specified axis.

Returns the standard deviation of the array elements, a measure of the spread of a distribution. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Parameters

a : {array_like}

Array containing numbers whose standard deviation is desired. If a is not an array, a conversion is attempted.

axis : {None, integer}, optional

Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

dtype : {None, dtype}, optional

Type to use in computing the standard deviation. For arrays of integer type the default is float32, for arrays of float types it is the same as the array type.

out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

ddof : {0, integer}

Means Delta Degrees of Freedom. The divisor used in calculations is N-ddof.

Returns

standard_deviation : {array, scalar}, see dtype parameter above.

If out=None, returns a new array containing the standard deviation, otherwise a reference to the output array is returned.

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e. var = sqrt(mean(abs(x - x.mean())**2)). The computed standard deviation is computed by dividing by the number of elements, N-ddof. The option ddof defaults to zero, that is, a biased estimate. Note that for complex numbers std takes the absolute value before squaring, so that the result is always real and nonnegative.

Examples

```
>>> a = array([[1,2],[3,4]])
>>> std(a)
1.1180339887498949
>>> std(a,0)
array([ 1.,  1.])
```

```
>>> std(a,1)
array([ 0.5,  0.5])
```

2.1.308 subtract()

y = subtract(x1,x2) subtracts the arguments elementwise.

2.1.309 sum(a, axis=None, dtype=None, out=None)

Return the sum of the array elements over the given axis

Parameters

a : {array_type}

Array containing elements whose sum is desired. If a is not an array, a conversion is attempted.

axis : {None, integer}

Axis over which the sum is taken. If None is used, then the sum is over all the array elements.

dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

out : {None, array}, optional

Array into which the sum can be placed. Its type is preserved and it must be of the right shape to hold the output.

Returns

sum_along_axis : {array, scalar}, see dtype parameter above.

Returns an array whose shape is the same as a with the specified axis removed. Returns a 0d array when a is 1d or axis=None. Returns a reference to the specified output array if specified.

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> sum([0.5, 1.5])
2.0
>>> sum([0.5, 1.5], dtype=N.int32)
1
>>> sum([[0, 1], [0, 5]])
```

```
6
>>> sum([[0, 1], [0, 5]], axis=1)
array([1, 5])
>>> ones(128, dtype=int8).sum(dtype=int8) # overflow!
-128
```

2.1.310 swapaxes(a, axis1, axis2)

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

Parameters

a : array_like

Input array.

axis1 : int

First axis.

axis2 : int

Second axis.

Examples

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

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

[[4, 5], [6, 7]]])

Block quote ends without a blank line; unexpected unindent.

```
>>> np.swapaxes(x,0,2)
array([[[0, 4],
        [2, 6]],
```

[[1, 5], [3, 7]]])

2.1.311 take(a, indices, axis=None, out=None, mode='raise')

Return an array formed from the elements of a at the given indices.

This function does the same thing as "fancy" indexing; however, it can be easier to use if you need to specify a given axis.

Parameters

a : array

The source array

indices : int array

The indices of the values to extract.

axis : {None, int}, optional

The axis over which to select values. None signifies that the operation should be performed over the flattened array.

out : {None, array}, optional

If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.

mode : {'raise', 'wrap', 'clip'}, optional

Specifies how out-of-bounds indices will behave. 'raise' – raise an error 'wrap' – wrap around 'clip' – clip to the range

Returns

subarray : array

The returned array has the same type as a.

2.1.312 tan()

y = tan(x) tangent elementwise.

2.1.313 tanh()

y = tanh(x) hyperbolic tangent elementwise.

2.1.314 tensordot(a, b, axes=2)

tensordot returns the product for any (ndim >= 1) arrays.

r_{xxx, yyy} = sum_k a_{xxx,k} b_{k,yyy} where

the axes to be summed over are given by the axes argument. the first element of the sequence determines the axis or axes in arr1 to sum over, and the second element in axes argument sequence determines the axis or axes in arr2 to sum over.

When there is more than one axis to sum over, the corresponding arguments to axes should be sequences of the same length with the first axis to sum over given first in both sequences, the second axis second, and so forth.

If the axes argument is an integer, N, then the last N dimensions of a and first N dimensions of b are summed over.

2.1.315 test(*args, **kw)

Run Numpy module test suite with level and verbosity.

level: None — do nothing, return None < 0 — scan for tests of level=abs(level),
Unexpected indentation.

don't run them, return TestSuite-list

Block quote ends without a blank line; unexpected unindent.

> 0 — scan for tests of level, run them, return TestRunner
> 10 — run all tests (same as specifying all=True). (backward compatibility).

verbosity: >= 0 — show information messages > 1 — show warnings on missing tests

all: True — run all test files (like self.testall()) False (default) — only run test files associated with a module

sys_argv — replacement of sys.argv[1:] during running tests.

testcase_pattern — run only tests that match given pattern.

It is assumed (when all=False) that package tests suite follows the following convention: for each package module, there exists file <packagepath>/tests/test_<modulename>.py that defines TestCase classes (with names having prefix '**test_**') with methods (with names having prefixes '**check_**' or '**bench_**'); each of these methods are called when running unit tests.

2.1.316 tile(A, reps)

Repeat an array the number of times given in the integer tuple, reps.

If reps has length d, the result will have dimension of max(d, A.ndim). If reps is scalar it is treated as a 1-tuple.

If A.ndim < d, A is promoted to be d-dimensional by prepending new axes. So a shape (3,) array is promoted to (1,3) for 2-D replication, or shape (1,1,3) for 3-D replication. If this is not the desired behavior, promote A to d-dimensions manually before calling this function.

If d < A.ndim, tup is promoted to A.ndim by pre-pending 1's to it. Thus for an A.shape of (2,3,4,5), a tup of (2,2) is treated as (1,1,2,2)

Examples: >>> a = array([0,1,2]) >>> tile(a,2) array([0, 1, 2, 0, 1, 2]) >>> tile(a,(1,2)) array([[0, 1, 2, 0, 1, 2]]) >>> tile(a,(2,2)) array([[0, 1, 2, 0, 1, 2],

Unexpected indentation.

[0, 1, 2, 0, 1, 2]])

Block quote ends without a blank line; unexpected unindent.

```
>>> tile(a,(2,1,2))
array([[[0, 1, 2, 0, 1, 2]],
<BLANKLINE>
       [[0, 1, 2, 0, 1, 2]]])
```

See Also: repeat

2.1.317 trace(a, offset=0, axis1=0, axis2=1, dtype=None, out=None)

Return the sum along diagonals of the array.

If a is 2-d, returns the sum along the diagonal of self with the given offset, i.e., the collection of elements of the form a[i,i+offset]. If a has more than two dimensions, then the axes specified by axis1 and axis2 are used to determine the 2-d subarray whose trace is returned. The shape of the resulting array can be determined by removing axis1 and axis2 and appending an index to the right equal to the size of the resulting diagonals.

Parameters

a : {array_like}

Array from whis the diagonals are taken.

offset : {0, integer}, optional

Offset of the diagonal from the main diagonal. Can be both positive and negative. Defaults to main diagonal.

axis1 : {0, integer}, optional

Axis to be used as the first axis of the 2-d subarrays from which the diagonals should be taken. Defaults to first axis.

axis2 : {1, integer}, optional

Axis to be used as the second axis of the 2-d subarrays from which the diagonals should be taken. Defaults to second axis.

dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and a is of integer type of precision less than the default integer precision, then the default integer precision is used. Otherwise, the precision is the same as that of a.

out : {None, array}, optional

Array into which the sum can be placed. Its type is preserved and it must be of the right shape to hold the output.

Returns

sum_along_diagonals : array

If a is 2-d, a 0-d array containing the diagonal is returned. If a has larger dimensions, then an array of diagonals is returned.

Examples

```
>>> trace(eye(3))
3.0
>>> a = arange(8).reshape((2,2,2))
>>> trace(a)
array([6, 8])
```

2.1.318 transpose(a, axes=None)

Return a view of the array with dimensions permuted.

Parameters

a : array_like

Input array.

axes : {None, list of int}, optional

If None (the default), reverse dimensions, otherwise permute axes according to the values given.

Examples

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

>>> np.transpose(x)
array([[0, 2],
       [1, 3]])

>>> np.transpose(x,(0,1)) # no change, axes are kept in current order
array([[0, 1],
       [2, 3]])
```

2.1.319 trapz(y, x=None, dx=1.0, axis=-1)

Integrate y(x) using samples along the given axis and the composite trapezoidal rule. If x is None, spacing given by dx is assumed.

2.1.320 tri(N, M=None, k=0, dtype=<type 'float'>)

returns a N-by-M array where all the diagonals starting from lower left corner up to the k-th are all ones.

2.1.321 tril(m, k=0)

returns the elements on and below the k-th diagonal of m. k=0 is the main diagonal, k > 0 is above and k < 0 is below the main diagonal.

2.1.322 trim_zeros(filt, trim='fb')

Trim the leading and trailing zeros from a 1D array.

Examples

```
>>> import numpy
>>> a = array((0, 0, 0, 1, 2, 3, 2, 1, 0))
>>> numpy.trim_zeros(a)
array([1, 2, 3, 2, 1])
```

2.1.323 triu(m, k=0)

returns the elements on and above the k-th diagonal of m. k=0 is the main diagonal, k > 0 is above and k < 0 is below the main diagonal.

2.1.324 true_divide()

y = true_divide(x1,x2) true divides the arguments elementwise.

2.1.325 typename(char)

Return an english description for the given data type character.

2.1.326 union1d(ar1, ar2)

Union of 1D arrays with unique elements.

Use unique1d() to generate arrays with only unique elements to use as inputs to this function.

Parameters

ar1 : array

ar2 : array

Returns

union : array

2.1.327 unique(x)

Return sorted unique items from an array or sequence.

Examples

```
>>> numpy.unique([5,2,4,0,4,4,2,2,1])
array([0, 1, 2, 4, 5])
```

2.1.328 unique1d(ar1, return_index=False)

Find the unique elements of 1D array.

Most of the other array set operations operate on the unique arrays generated by this function.

Parameters

ar1 : array

This array will be flattened if it is not already 1D.

return_index : bool, optional

If True, also return the indices against ar1 that result in the unique array.

Returns

unique : array

The unique values.

unique_indices : int array, optional

The indices of the unique values. Only provided if return_index is True.

2.1.329 unpackbits()

out = numpy.unpackbits(myarray, axis=None)

myarray - array of uint8 type where each element represents a bit-field that should be unpacked into a boolean output array

The shape of the output array is either 1-d (if axis is None) or the same shape as the input array with unpacking done along the axis specified.

2.1.330 unravel_index(x, dims)

Convert a flat index into an index tuple for an array of given shape.

e.g. for a 2x2 array, unravel_index(2,(2,2)) returns (1,0).

Example usage: p = x.argmax() idx = unravel_index(p,x.shape) x[idx] == x.max()

Note: x.flat[p] == x.max()

Thus, it may be easier to use flattened indexing than to re-map the index to a tuple.

2.1.331 unwrap(p, discont=3.1415926535897931, axis=-1)

Unwrap radian phase p by changing absolute jumps greater than 'discont' to their 2*pi complement along the given axis.

2.1.332 vander(x, N=None)

Generate the Vandermonde matrix of vector x.

The i-th column of X is the the (N-i)-1-th power of x. N is the maximum power to compute; if N is None it defaults to len(x).

2.1.333 var(a, axis=None, dtype=None, out=None, ddof=0)

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Parameters

a : {array_like}

Array containing numbers whose variance is desired. If a is not an array, a conversion is attempted.

axis : {None, integer}, optional

Axis along which the variance is computed. The default is to compute the variance of the flattened array.

dtype : {None, dtype}, optional

Type to use in computing the variance. For arrays of integer type the default is float32, for arrays of float types it is the same as the array type.

out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

ddof : {0, integer},

Means Delta Degrees of Freedom. The divisor used in calculation is N - ddof.

Returns

variance : {array, scalar}, see dtype parameter above

If out=None, returns a new array containing the variance, otherwise a reference to the output array is returned.

Notes

The variance is the average of the squared deviations from the mean, i.e. var = mean(abs(x - x.mean())**2). The computed variance is biased, i.e., the mean is computed by dividing by the number of elements, N, rather than by N-1. Note that for complex numbers the absolute value is taken before squaring, so that the result is always real and nonnegative.

Examples

```
>>> a = array([[1,2],[3,4]])
>>> var(a)
1.25
>>> var(a,0)
array([ 1.,  1.])
>>> var(a,1)
array([ 0.25,  0.25])
```

2.1.334 vdot()

vdot(a,b) Returns the dot product of a and b for scalars and vectors of floating point and complex types. The first argument, a, is conjugated.

2.1.335 vsplit(ary, indices_or_sections)

Split ary into multiple rows of sub-arrays

Description: Split a single array into multiple sub arrays. The array is divided into groups of rows. If indices_or_sections is an integer, ary is divided into that many equally sized sub arrays. If it is impossible to make the sub-arrays equally sized, the operation throws a ValueError exception. See array_split and split for other options on indices_or_sections.

Arguments: **ary – N-D array.** Array to be divided into sub-arrays.

indices_or_sections – integer or 1D array. If integer, defines the number of (close to) equal sized sub-arrays. If it is a 1D array of sorted indices, it defines the indexes at which ary is divided. Any empty list results in a single sub-array equal to the original array.

Returns: sequence of sub-arrays. The returned arrays have the same number of dimensions as the input array.

Caveats: How should we handle 1D arrays here? I am currently raising an error when I encounter them. Any better approach?

Should we reduce the returned array to their minium dimensions by getting rid of any dimensions that are 1?

Related: vstack, split, array_split, hsplit, dsplit.

Examples: import numpy >>> a = array([[1,2,3,4], ... [1,2,3,4]]) >>> numpy.vsplit(a,2) [array([[1, 2, 3, 4]]), array([[1, 2, 3, 4]])]

2.1.336 vstack(tup)

Stack arrays in sequence vertically (row wise)

Description: Take a sequence of arrays and stack them vertically to make a single array. All arrays in the sequence must have the same shape along all but the first axis. vstack will rebuild arrays divided by vsplit.

Arguments: **tup – sequence of arrays. All arrays must have the same** shape.

Examples:
```
>>> import numpy
>>> a = array((1,2,3))
>>> b = array((2,3,4))
```

```
>>> numpy.vstack((a,b))
array([[1, 2, 3],
       [2, 3, 4]])
>>> a = array([[1],[2],[3]])
>>> b = array([[2],[3],[4]])
>>> numpy.vstack((a,b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```

2.1.337 where(condition, x, y) or where(condition)

Return elements from *x* or *y*, depending on *condition*.

Parameters

condition : array of bool

When True, yield x, otherwise yield y.

x,y : 1-dimensional arrays

Values from which to choose.

Notes

This is equivalent to

[xv if c else yv for (c,xv,yv) in zip(condition,x,y)]

The result is shaped like *condition* and has elements of *x* or *y* where *condition* is respectively True or False.

In the special case, where only *condition* is given, the tuple condition.nonzero() is returned, instead.

Examples

```
>>> where([True,False,True],[1,2,3],[4,5,6])
array([1, 5, 3])
```

2.1.338 who(vardict=None)

Print the Numpy arrays in the given dictionary (or globals() if None).

2.1.339 zeros(shape, dtype=float, order='C')

Return a new array of given shape and type, filled zeros.

Parameters

shape : tuple of integers

Shape of the new array

dtype : data-type

The desired data-type for the array.

order : {'C', 'F'}

Whether to store multidimensional data in C or Fortran order.

2.1.340 zeros_like(a)

Return an array of zeros of the shape and data-type of a.

If you don't explicitly need the array to be zeroed, you should instead use empty_like(), which is a bit faster as it only allocates memory.

CHAPTER

THREE

Submodules

3.1 C-types Foreign Function Interface

3.1.1 numpy.ctypeslib

array(object, dtype=None, copy=1,order=None, subok=0,ndmin=0)

Return an array from object with the specified data-type.

Parameters

object : array-like

an array, any object exposing the array interface, any object whose __array__ method returns an array, or any (nested) sequence.

dtype : data-type

The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence. This argument can only be used to 'upcast' the array. For downcasting, use the .astype(t) method.

copy : bool

If true, then force a copy. Otherwise a copy will only occur if __array__ returns a copy, obj is a nested sequence, or a copy is needed to satisfy any of the other requirements

order : {'C', 'F', 'A' (None)}

Specify the order of the array. If order is 'C', then the array will be in C-contiguous order (last-index varies the fastest). If order is 'FORTRAN', then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is None, then the returned array may be in either C-, or Fortran-contiguous order or even discontiguous.

subok : bool

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

ndmin : int

Specifies the minimum number of dimensions that the resulting array should have. 1's will be pre-pended to the shape as needed to meet this requirement.

as_array(obj)

Create a numpy array from a ctypes array. The numpy array shares the memory with the ctypes object.

as_ctypes(obj)

Create and return a ctypes object from a numpy array. Actually anything that exposes the __array_interface__ is accepted.

ctypes_load_library(*args, **kwds)

ctypes_load_library is DEPRECATED!! – use load_library instead

deprecate(func, oldname=None, newname=None)

Deprecate old functions. Issues a DeprecationWarning, adds warning to oldname's docstring, rebinds oldname.__name__ and returns new function object.

Example: oldfunc = deprecate(newfunc, 'oldfunc', 'newfunc')

load_library(libname, loader_path)

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 (dtype), number of dimensions (ndim), shape and flags (e.g. 'C_CONTIGUOUS' or 'F_CONTIGUOUS'). If a given array does not satisfy the specified restrictions, a TypeError is raised.

Example:

> **clib.somefunc.argtypes = [ndpointer(dtype=float64,** ndim=1, flags='C_CONTIGUOUS')]
>
> Definition list ends without a blank line; unexpected unindent.
>
> clib.somefunc(array([1,2,3],dtype=float64))

prep_array(array_type)

Given a ctypes array type, construct and attach an __array_interface__ property to it if it does not yet have one.

prep_simple(simple_type, typestr)

Given a ctypes simple type, construct and attach an __array_interface__ property to it if it does not yet have one.

test(level=1, verbosity=1)

3.2 Fast Fourier Transform

3.2.1 numpy.fft

fft(a, n=None, axis=-1)

Return the n point discrete Fourier transform of a. n defaults to the length of a. If n is larger than the length of a, then a will be zero-padded to make up the difference. If n is smaller than the length of a, only the first n items in a will be used.

The packing of the result is "standard": If A = fft(a, n), then A[0] contains the zero-frequency term, A[1:n/2+1] contains the positive-frequency terms, and A[n/2+1:] contains the negative-frequency terms, in order of decreasingly negative frequency. So for an 8-point transform, the frequencies of the result are [0, 1, 2, 3, 4, -3, -2, -1].

This is most efficient for n a power of two. This also stores a cache of working memory for different sizes of fft's, so you could theoretically run into memory problems if you call this too many times with too many different n's.

fft2(a, s=None, axes=(-2,-1))

The 2d fft of a. This is really just fftn with different default behavior.

fftfreq(n, d=1.0)

fftfreq(n, d=1.0) -> f

DFT sample frequencies

The returned float array contains the frequency bins in cycles/unit (with zero at the start) given a window length n and a sample spacing d:

f = [0,1,...,n/2-1,-n/2,...,-1]/(d*n) if n is even f = [0,1,...,(n-1)/2,-(n-1)/2,...,-1]/(d*n) if n is odd

fftn(a, s=None, axes=None)

The n-dimensional fft of a. s is a sequence giving the shape of the input an result along the transformed axes, as n for fft. Results are packed analogously to fft: the term for zero frequency in all axes is in the low-order corner, while the term for the Nyquist frequency in all axes is in the middle.

If neither s nor axes is specified, the transform is taken along all axes. If s is specified and axes is not, the last len(s) axes are used. If axes are specified and s is not, the input shape along the specified axes is used. If s and axes are both specified and are not the same length, an exception is raised.

fftshift(x, axes=None)

fftshift(x, axes=None) -> y

Shift zero-frequency component to center of spectrum.

This function swaps half-spaces for all axes listed (defaults to all).

Notes: If len(x) is even then the Nyquist component is y[0].

hfft(a, n=None, axis=-1)

hfft(a, n=None, axis=-1) ihfft(a, n=None, axis=-1)

These are a pair analogous to rfft/irfft, but for the opposite case: here the signal is real in the frequency domain and has Hermite symmetry in the time domain. So here it's hermite_fft for which you must supply the length of the result if it is to be odd.

ihfft(hfft(a), len(a)) == a within numerical accuracy.

ifft(a, n=None, axis=-1)

Return the n point inverse discrete Fourier transform of a. n defaults to the length of a. If n is larger than the length of a, then a will be zero-padded to make up the difference. If n is smaller than the length of a, then a will be truncated to reduce its size.

The input array is expected to be packed the same way as the output of fft, as discussed in it's documentation.

This is the inverse of fft: ifft(fft(a)) == a within numerical accuracy.

This is most efficient for n a power of two. This also stores a cache of working memory for different sizes of fft's, so you could theoretically run into memory problems if you call this too many times with too many different n's.

ifft2(a, s=None, axes=(-2, -1))

The inverse of fft2d. This is really just ifftn with different default behavior.

ifftn(a, s=None, axes=None)

The inverse of fftn.

ifftshift(x, axes=None)

ifftshift(x,axes=None) - > y

Inverse of fftshift.

ihfft(a, n=None, axis=-1)

hfft(a, n=None, axis=-1) ihfft(a, n=None, axis=-1)

These are a pair analogous to rfft/irfft, but for the opposite case: here the signal is real in the frequency domain and has Hermite symmetry in the time domain. So here it's hfft for which you must supply the length of the result if it is to be odd.

ihfft(hfft(a), len(a)) == a within numerical accuracy.

irefft(*args, **kwds)

irefft is DEPRECATED!! – use irfft instead

irfft(a, n=None, axis=-1)

Return the real valued n point inverse discrete Fourier transform of a, where a contains the nonnegative frequency terms of a Hermite-symmetric sequence. n is the length of the result, not the input. If n is not supplied, the default is 2*(len(a)-1). If you want the length of the result to be odd, you have to say so.

If you specify an n such that a must be zero-padded or truncated, the extra/removed values will be added/removed at high frequencies. One can thus resample a series to m points via Fourier interpolation by: a_resamp = irfft(rfft(a), m).

This is the inverse of rfft: irfft(rfft(a), len(a)) == a within numerical accuracy.

irefft2(*args, **kwds)

irefft2 is DEPRECATED!! – use irfft2 instead

irfft2(a, s=None, axes=(-2, -1))

The inverse of rfft2. This is really just irfftn with different default behavior.

irefftn(*args, **kwds)

irefftn is DEPRECATED!! – use irfftn instead

irfftn(a, s=None, axes=None)

The inverse of rfftn. The transform implemented in ifft is applied along all axes but the last, then the transform implemented in irfft is performed along the last axis. As with irfft, the length of the result along that axis must be specified if it is to be odd.

irfft(a, n=None, axis=-1)

Return the real valued n point inverse discrete Fourier transform of a, where a contains the nonnegative frequency terms of a Hermite-symmetric sequence. n is the length of the result, not the input. If n is not supplied, the default is 2*(len(a)-1). If you want the length of the result to be odd, you have to say so.

If you specify an n such that a must be zero-padded or truncated, the extra/removed values will be added/removed at high frequencies. One can thus resample a series to m points via Fourier interpolation by: a_resamp = irfft(rfft(a), m).

This is the inverse of rfft: irfft(rfft(a), len(a)) == a within numerical accuracy.

irfft2(a, s=None, axes=(-2, -1))

The inverse of rfft2. This is really just irfftn with different default behavior.

irfftn(a, s=None, axes=None)

The inverse of rfftn. The transform implemented in ifft is applied along all axes but the last, then the transform implemented in irfft is performed along the last axis. As with irfft, the length of the result along that axis must be specified if it is to be odd.

refft(*args, **kwds)

refft is DEPRECATED!! – use rfft instead

rfft(a, n=None, axis=-1)

Return the n point discrete Fourier transform of the real valued array a. n defaults to the length of a. n is the length of the input, not the output.

The returned array will be the nonnegative frequency terms of the Hermite-symmetric, complex transform of the real array. So for an 8-point transform, the frequencies in the result are [0, 1, 2, 3, 4]. The first term will be real, as will the last if n is even. The negative frequency terms are not needed because they are the complex conjugates of the positive frequency terms. (This is what I mean when I say Hermite-symmetric.)

This is most efficient for n a power of two.

refft2(*args, **kwds)

refft2 is DEPRECATED!! – use rfft2 instead

rfft2(a, s=None, axes=(-2,-1))

The 2d fft of the real valued array a. This is really just rfftn with different default behavior.

refftn(*args, **kwds)

refftn is DEPRECATED!! – use rfftn instead

rfftn(a, s=None, axes=None)

The n-dimensional discrete Fourier transform of a real array a. A real transform as rfft is performed along the axis specified by the last element of axes, then complex transforms as fft are performed along the other axes.

rfft(a, n=None, axis=-1)

Return the n point discrete Fourier transform of the real valued array a. n defaults to the length of a. n is the length of the input, not the output.

The returned array will be the nonnegative frequency terms of the Hermite-symmetric, complex transform of the real array. So for an 8-point transform, the frequencies in the result are [0, 1, 2, 3, 4]. The first term will be real, as will the last if n is even. The negative frequency terms are not needed because they are the complex conjugates of the positive frequency terms. (This is what I mean when I say Hermite-symmetric.)

This is most efficient for n a power of two.

rfft2(a, s=None, axes=(-2,-1))

The 2d fft of the real valued array a. This is really just rfftn with different default behavior.

rfftn(a, s=None, axes=None)

The n-dimensional discrete Fourier transform of a real array a. A real transform as rfft is performed along the axis specified by the last element of axes, then complex transforms as fft are performed along the other axes.

test(level=1, verbosity=1)

3.3 Linear Algebra

3.3.1 numpy.linalg

cholesky(a)

Compute the Cholesky decomposition of a matrix.

Returns the Cholesky decomposition, $A = LL^*$ of a Hermitian positive-definite matrix A.

Parameters

a : array-like, shape (M, M)

Matrix to be decomposed

Returns

L : array-like, shape (M, M)

Lower-triangular Cholesky factor of A

Raises LinAlgError if decomposition fails :

Examples

```
>>> A = np.array([[1,-2j],[2j,5]])
>>> L = np.linalg.cholesky(A)
>>> L
array([[ 1.+0.j,  0.+0.j],
       [ 0.+2.j,  1.+0.j]])
>>> dot(L, L.T.conj())
array([[ 1.+0.j,  0.-2.j],
       [ 0.+2.j,  5.+0.j]])
```

cond(x, p=None)

Compute the condition number of a matrix.

The condition number of x is the norm of x times the norm of the inverse of x. The norm can be the usual L2 (root-of-sum-of-squares) norm or a number of other matrix norms.

Parameters

x : array-like, shape (M, N)

The matrix whose condition number is sought.

p : {None, 1, -1, 2, -2, inf, -inf, 'fro'}

Order of the norm:

p norm for matrices ===== ============================ None 2-norm, computed directly using the SVD 'fro' Frobenius norm inf max(sum(abs(x), axis=1)) -inf

min(sum(abs(x), axis=1)) 1 max(sum(abs(x), axis=0)) -1 min(sum(abs(x), axis=0)) 2 2-norm (largest sing. value) -2 smallest singular value ===== ============================

Returns

c : float

The condition number of the matrix. May be infinite.

det(a)

Compute the determinant of a matrix

Parameters

a : array-like, shape (M, M)

Returns

det : float or complex

Determinant of a

Notes

The determinant is computed via LU factorization, LAPACK routine z/dgetrf.

eig(a)

Compute eigenvalues and right eigenvectors of a general matrix.

Parameters

a : array-like, shape (M, M)

A complex or real 2-d array whose eigenvalues and eigenvectors will be computed.

Returns

w : double or complex array, shape (M,)

The eigenvalues, each repeated according to its multiplicity. The eigenvalues are not necessarily ordered, nor are they necessarily real for real matrices.

v : double or complex array, shape (M, M)

The normalized eigenvector corresponding to the eigenvalue w[i] is the column v[:,i].

If a is a matrix, so are all the return values. :

Raises LinAlgError if eigenvalue computation does not converge :

Notes

This is a simple interface to the LAPACK routines dgeev and zgeev that compute the eigenvalues and eigenvectors of general real and complex arrays respectively.

The number w is an eigenvalue of a if there exists a vector v satisfying the equation dot(a,v) = w*v. Alternately, if w is a root of the characteristic equation det(a - w[i]*I) = 0, where det is the determinant and I is the identity matrix. The arrays a, w, and v satisfy the equation dot(a,v[i]) = w[i]*v[:,i].

The array v of eigenvectors may not be of maximum rank, that is, some of the columns may be dependent, although roundoff error may obscure that fact. If the eigenvalues are all different, then theoretically the eigenvectors are independent. Likewise, the matrix of eigenvectors is unitary if the matrix a is normal, i.e., if dot(a, a.H) = dot(a.H, a).

The left and right eigenvectors are not necessarily the (Hermitian) transposes of each other.

eigh(a, UPLO='L')

Compute eigenvalues for a Hermitian or real symmetric matrix.

Parameters

a : array-like, shape (M, M)

A complex Hermitian or symmetric real matrix whose eigenvalues and eigenvectors will be computed.

UPLO : {'L', 'U'}

Specifies whether the pertinent array date is taken from the upper or lower triangular part of a. Possible values are 'L', and 'U'. Default is 'L'.

Returns

w : double array, shape (M,)

The eigenvalues. The eigenvalues are not necessarily ordered.

v : double or complex double array, shape (M, M)

The normalized eigenvector corresponding to the eigenvalue w[i] is the column v[:,i].

If a is a matrix, then so are the return values. :

Raises LinAlgError if eigenvalue computation does not converge :

Notes

A simple interface to the LAPACK routines dsyevd and zheevd that compute the eigenvalues and eigenvectors of real symmetric and complex Hermitian arrays respectively.

The number w is an eigenvalue of a if there exists a vector v satisfying the equation dot(a,v) = w*v. Alternately, if w is a root of the characteristic equation det(a - w[i]*I) = 0, where det is the determinant and I is the identity matrix. The eigenvalues of real symmetric or complex Hermitean matrices are always real. The array v of eigenvectors is unitary and a, w, and v satisfy the equation dot(a,v[i]) = w[i]*v[:,i].

eigvals(a)

Compute the eigenvalues of a general matrix.

Parameters

a : array-like, shape (M, M)

A complex or real matrix whose eigenvalues and eigenvectors will be computed.

Returns

w : double or complex array, shape (M,)

The eigenvalues, each repeated according to its multiplicity. They are not necessarily ordered, nor are they necessarily real for real matrices.

If a is a matrix, so is w. :

Raises LinAlgError if eigenvalue computation does not converge :

Notes

This is a simple interface to the LAPACK routines dgeev and zgeev that sets the flags to return only the eigenvalues of general real and complex arrays respectively.

The number w is an eigenvalue of a if there exists a vector v satisfying the equation dot(a,v) = w*v. Alternately, if w is a root of the characteristic equation det(a - w[i]*I) = 0, where det is the determinant and I is the identity matrix.

eigvalsh(a, UPLO='L')

Compute the eigenvalues of a Hermitean or real symmetric matrix.

Parameters

a : array-like, shape (M, M)

A complex or real matrix whose eigenvalues and eigenvectors will be computed.

UPLO : {'L', 'U'}

Specifies whether the pertinent array data is taken from the upper or lower triangular part of a. Possible values are 'L', and 'U' for upper and lower respectively. Default is 'L'.

Returns

w : double array, shape (M,)

The eigenvalues, each repeated according to its multiplicity. They are not necessarily ordered.

Raises LinAlgError if eigenvalue computation does not converge :

Notes

This is a simple interface to the LAPACK routines dsyevd and zheevd that sets the flags to return only the eigenvalues of real symmetric and complex Hermetian arrays respectively.

The number w is an eigenvalue of a if there exists a vector v satisfying the equation dot(a,v) = w*v. Alternately, if w is a root of the characteristic equation det(a - w[i]*I) = 0, where det is the determinant and I is the identity matrix.

inv(a)

Compute the inverse of a matrix.

Parameters

a : array-like, shape (M, M)

Matrix to be inverted

Returns

ainv : array-like, shape (M, M)

Inverse of the matrix a

Raises LinAlgError if a is singular or not square :

Examples

```
>>> from numpy import array, inv, dot
>>> a = array([[1., 2.], [3., 4.]])
>>> inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> dot(a, inv(a))
array([[ 1.,  0.],
       [ 0.,  1.]])
```

lstsq(a, b, rcond=-1)

Compute least-squares solution to equation $ax = b$

Compute a vector x such that the 2-norm $|b - ax|$ is minimised.

Parameters

a : array-like, shape (M, N)

b : array-like, shape (M,) or (M, K)

rcond : float

Cutoff for 'small' singular values. Singular values smaller than rcond*largest_singular_value are considered zero.

Raises LinAlgError if computation does not converge :

Returns

x : array, shape (N,) or (N, K) depending on shape of b

Least-squares solution

residues : array, shape () or (1,) or (K,)

Sums of residues, squared 2-norm for each column in $b - ax$ If rank of matrix a is < N or > M this is an empty array. If b was 1-d, this is an (1,) shape array, otherwise the shape is (K,)

rank : integer

Rank of matrix a

s : array, shape (min(M,N),)

Singular values of a

If b is a matrix, then all results except the rank are also returned as :

matrices. :

matrix_power(M, n)

Raise a square matrix to the (integer) power n.

For positive integers n, the power is computed by repeated matrix squarings and matrix multiplications. If n=0, the identity matrix of the same type as M is returned. If n<0, the inverse is computed and raised to the exponent.

Parameters

M : array-like

Must be a square array (that is, of dimension two and with equal sizes).

n : integer

The exponent can be any integer or long integer, positive negative or zero.

Returns

M to the power n :

The return value is a an array the same shape and size as M; if the exponent was positive or zero then the type of the elements is the same as those of M. If the exponent was negative the elements are floating-point.

Raises

LinAlgException :

If the matrix is not numerically invertible, an exception is raised.

Examples

```
>>> matrix_power(array([[0,1],[-1,0]]),10)
array([[-1,  0],
       [ 0, -1]])
```

norm(x, ord=None)

Matrix or vector norm.

Parameters

x : array-like, shape (M,) or (M, N)

ord : number, or {None, 1, -1, 2, -2, inf, -inf, 'fro'}

Order of the norm:
ord norm for matrices norm for vectors ===== ============================ ========================== None Frobenius norm 2-norm 'fro' Frobenius norm - inf max(sum(abs(x), axis=1)) max(abs(x)) -inf min(sum(abs(x), axis=1)) min(abs(x)) 1 max(sum(abs(x), axis=0)) as below -1 min(sum(abs(x), axis=0)) as below 2 2-norm (largest sing. value) as below -2 smallest singular value as below other - sum(abs(x)**ord)**(1./ord) ===== ============================ ==========================

Returns

n : float

Norm of the matrix or vector

Notes

For values ord < 0, the result is, strictly speaking, not a mathematical 'norm', but it may still be useful for numerical purposes.

pinv(a, rcond=1.0000000000000001e-15)

Compute the (Moore-Penrose) pseudo-inverse of a matrix.

Calculate a generalized inverse of a matrix using its singular-value decomposition and including all 'large' singular values.

Parameters

a : array-like, shape (M, N)

Matrix to be pseudo-inverted

rcond : float

Cutoff for 'small' singular values. Singular values smaller than rcond*largest_singular_value are considered zero.

Returns

B : array, shape (N, M)

If a is a matrix, then so is B.

Raises LinAlgError if SVD computation does not converge :

Examples

```
>>> from numpy import *
>>> a = random.randn(9, 6)
>>> B = linalg.pinv(a)
>>> allclose(a, dot(a, dot(B, a)))
True
>>> allclose(B, dot(B, dot(a, B)))
True
```

qr(a, mode='full')

Compute QR decomposition of a matrix.

Calculate the decomposition $A = QR$ where Q is orthonormal and R upper triangular.

Parameters

a : array-like, shape (M, N)

Matrix to be decomposed

mode : {'full', 'r', 'economic'}

Determines what information is to be returned. 'full' is the default. Economic mode is slightly faster if only R is needed.

Returns

mode = 'full' :

Q : double or complex array, shape (M, K)

R : double or complex array, shape (K, N)

Size K = min(M, N)

mode = 'r' :

R : double or complex array, shape (K, N)

mode = 'economic' :

A2 : double or complex array, shape (M, N)

The diagonal and the upper triangle of A2 contains R, while the rest of the matrix is undefined.

If a is a matrix, so are all the return values. :

Raises LinAlgError if decomposition fails :

Notes

This is an interface to the LAPACK routines dgeqrf, zgeqrf, dorgqr, and zungqr.

Examples

```
>>> from numpy import *
>>> a = random.randn(9, 6)
>>> q, r = linalg.qr(a)
>>> allclose(a, dot(q, r))
True
>>> r2 = linalg.qr(a, mode='r')
>>> r3 = linalg.qr(a, mode='economic')
>>> allclose(r, r2)
True
>>> allclose(r, triu(r3[:6,:6], k=0))
True
```

solve(a, b)

Solve the equation a x = b

Parameters

a : array-like, shape (M, M)

b : array-like, shape (M,)

Returns

x : array, shape (M,)

Raises LinAlgError if a is singular or not square :

svd(a, full_matrices=1, compute_uv=1)

Singular Value Decomposition.

Factorizes the matrix a into two unitary matrices U and Vh and an 1d-array s of singular values (real, non-negative) such that a == U S Vh if S is an suitably shaped matrix of zeros whose main diagonal is s.

Parameters

a : array-like, shape (M, N)

Matrix to decompose

full_matrices : boolean

If true, U, Vh are shaped (M,M), (N,N) If false, the shapes are (M,K), (K,N) where K = min(M,N)

compute_uv : boolean

Whether to compute U and Vh in addition to s

Returns

U: array, shape (M,M) or (M,K) depending on full_matrices :

s: array, shape (K,) :

The singular values, sorted so that s[i] >= s[i+1] K = min(M, N)

Vh: array, shape (N,N) or (K,N) depending on full_matrices :

If a is a matrix, so are all the return values. :

Raises LinAlgError if SVD computation does not converge :

Examples

```
>>> a = random.randn(9, 6) + 1j*random.randn(9, 6)
>>> U, s, Vh = linalg.svd(a)
>>> U.shape, Vh.shape, s.shape
((9, 9), (6, 6), (6,))

>>> U, s, Vh = linalg.svd(a, full_matrices=False)
>>> U.shape, Vh.shape, s.shape
((9, 6), (6, 6), (6,))
>>> S = diag(s)
>>> allclose(a, dot(U, dot(S, Vh)))
True

>>> s2 = linalg.svd(a, compute_uv=False)
>>> allclose(s, s2)
True
```

tensorinv(a, ind=2)

Find the 'inverse' of a N-d array

The result is an inverse corresponding to the operation tensordot(a, b, ind), ie.,

x == tensordot(tensordot(tensorinv(a), a, ind), x, ind) == tensordot(tensordot(a, tensorinv(a), ind), x, ind)

for all x (up to floating-point accuracy).

Parameters

a : array-like

Tensor to 'invert'. Its shape must 'square', ie., prod(a.shape[:ind]) == prod(a.shape[ind:])

ind : integer > 0

How many of the first indices are involved in the inverse sum.

Returns

b : array, shape a.shape[:ind]+a.shape[ind:]

Raises LinAlgError if a is singular or not square :

Examples

```
>>> from numpy import *
>>> a = eye(4*6)
>>> a.shape = (4,6,8,3)
>>> ainv = linalg.tensorinv(a, ind=2)
>>> ainv.shape
(8, 3, 4, 6)
>>> b = random.randn(4,6)
>>> allclose(tensordot(ainv, b), linalg.tensorsolve(a, b))
True

>>> a = eye(4*6)
>>> a.shape = (24,8,3)
>>> ainv = linalg.tensorinv(a, ind=1)
>>> ainv.shape
(8, 3, 24)
>>> b = random.randn(24)
>>> allclose(tensordot(ainv, b, 1), linalg.tensorsolve(a, b))
True
```

tensorsolve(a, b, axes=None)

Solve the tensor equation a x = b for x

It is assumed that all indices of x are summed over in the product, together with the rightmost indices of a, similarly as in tensordot(a, x, axes=len(b.shape)).

Parameters

a : array-like, shape b.shape+Q

Coefficient tensor. Shape Q of the rightmost indices of a must be such that a is 'square', ie., prod(Q) == prod(b.shape).

b : array-like, any shape

Right-hand tensor.

axes : tuple of integers

Axes in a to reorder to the right, before inversion. If None (default), no reordering is done.

Returns

x : array, shape Q

Examples

```
>>> from numpy import *
>>> a = eye(2*3*4)
>>> a.shape = (2*3,4,  2,3,4)
>>> b = random.randn(2*3,4)
>>> x = linalg.tensorsolve(a, b)
>>> x.shape
(2, 3, 4)
>>> allclose(tensordot(a, x, axes=3), b)
True
```

test(level=1, verbosity=1)

3.4 Masked Arrays

3.4.1 numpy.ma

absolute()

y = absolute(x) takes |x| elementwise.

absolute()

y = absolute(x) takes |x| elementwise.

add()

y = add(x1,x2) adds the arguments elementwise.

allclose(a, b, fill_value=True, rtol=1.0000000000000001e-05, atol=1e-08)

Return True if all elements of a and b are equal subject to given tolerances.

If fill_value is True, masked values are considered equal. If fill_value is False, masked values considered unequal. The relative error rtol should be positive and << 1.0 The absolute error atol comes into play for those elements of b that are very small or zero; it says how small *a* must be also.

allequal(a, b, fill_value=True)

Return True if all entries of a and b are equal, using fill_value as a truth value where either or both are masked.

reduce(self, target, axis=0, dtype=None)

Reduce *target* along the given *axis*.

amax(a, axis=None, out=None)

Return the maximum along a given axis.

Parameters

a : array_like

Input data.

axis : {None, int}, optional

Axis along which to operate. By default, `axis` is None and the flattened input is used.

out : array_like, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

Returns

amax : array_like

New array holding the result, unless `out` was specified.

Examples

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

amin(a, axis=None, out=None)

Return the minimum along a given axis.

Parameters

a : array_like

Input data.

axis : {None, int}, optional

Axis along which to operate. By default, `axis` is None and the flattened input is used.

out : array_like, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

Returns

amin : array_like

New array holding the result, unless `out` was specified.

Examples

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

apply_along_axis(func1d, axis, arr, *args, **kwargs)

Execute func1d(arr[i],*args) where func1d takes 1-D arrays and arr is an N-d array. i varies so as to apply the function along the given axis for each 1-d subarray in arr.

arange([start,] stop[, step,], dtype=None)

For integer arguments, just like range() except it returns an array whose type can be specified by the keyword argument dtype. If dtype is not specified, the type of the result is deduced from the type of the arguments.

For floating point arguments, the length of the result is ceil((stop - start)/step). This rule may result in the last element of the result being greater than stop.

arccos()

y = arccos(x) inverse cosine elementwise.

arccosh()

y = arccosh(x) inverse hyperbolic cosine elementwise.

arcsin()

y = arcsin(x) inverse sine elementwise.

arcsinh()

y = arcsinh(x) inverse hyperbolic sine elementwise.

arctan()

y = arctan(x) inverse tangent elementwise.

arctan2()

y = arctan2(x1,x2) a safe and correct arctan(x1/x2)

arctanh()

y = arctanh(x) inverse hyperbolic tangent elementwise.

argmax(a, axis=None, fill_value=None)

Function version of the eponymous method.

argmin(a, axis=None, fill_value=None)

Returns the array of indices for the maximum values of *a* along the specified axis.

Masked values are treated as if they had the value fill_value.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

fill_value : {var}, optional

Value used to fill in the masked values. If None, the output of maximum_fill_value(self._data) is used.

argsort(a, axis=None, kind='quicksort', order=None, fill_value=None)

Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to fill_value.

Parameters

axis : int, optional

Axis to be indirectly sorted. If not given, uses a flatten version of the array.

fill_value : {var}

Value used to fill in the masked values. If not given, self.fill_value is used instead.

kind : {string}

Sorting algorithm (default 'quicksort') Possible values: 'quicksort', 'mergesort', or 'heapsort'

Notes

This method executes an indirect sort along the given axis using the algorithm specified by the kind keyword. It returns an array of indices of the same shape as 'a' that index data along the given axis in sorted order.

The various sorts are characterized by average speed, worst case performance need for work space, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The three available algorithms have the following properties:

|——————————————————| | kind | speed | worst case | work space | stable| |————————————————————| |**'quicksort'**| 1 | O(n^2) | 0 | no | |**'mergesort'**| 2 | O(n*log(n)) | ~n/2 | yes | |**'heapsort' | 3 | O(n*log(n)) | 0 | no | |——————————————————|

All the sort algorithms make temporary copies of the data when the sort is not along the last axis. Consequently, sorts along the last axis are faster and use less space than sorts along other axis.

round_()

Round a to the given number of decimals.

The real and imaginary parts of complex numbers are rounded separately. The result of rounding a float is a float so the type must be cast if integers are desired. Nothing is done if the input is an integer array and the decimals parameter has a value >= 0.

Parameters

a : {array_like}

Array containing numbers whose rounded values are desired. If a is not an array, a conversion is attempted.

decimals : {0, integer}, optional

Number of decimal places to round to. When decimals is negative it specifies the number of positions to the left of the decimal point.

out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns

rounded_array : {array}

If out=None, returns a new array of the same type as a containing the rounded values, otherwise a reference to the output array is returned.

Notes

Numpy rounds to even. Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc. Results may also be surprising due to the inexact representation of decimal fractions in IEEE floating point and the errors introduced when scaling by powers of ten.

Examples

```
>>> round_([.5, 1.5, 2.5, 3.5, 4.5])
array([ 0.,  2.,  2.,  4.,  4.])
>>> round_([1,2,3,11], decimals=1)
array([ 1,  2,  3, 11])
>>> round_([1,2,3,11], decimals=-1)
array([ 0,  0,  0, 10])
```

array(data, dtype=None, copy=False, order=False, mask=False, fill_value=None, keep_mask=True, hard_mask=False, shrink=True, subok=True, ndmin=0)

Arrays with possibly masked values. Masked values of True exclude the corresponding element from any computation.

Construction: x = MaskedArray(data, mask=nomask, dtype=None, copy=True, fill_value=None, keep_mask=True, hard_mask=False, shrink=True)

Parameters

data : {var}

Input data.

mask : {nomask, sequence}

Mask. Must be convertible to an array of booleans with the same shape as data: True indicates a masked (eg., invalid) data.

dtype : dtype

Data type of the output. If None, the type of the data argument is used. If dtype is not None and different from data.dtype, a copy is performed.

copy : bool

Whether to copy the input data (True), or to use a reference instead. Note: data are NOT copied by default.

subok : {True, boolean}

Whether to return a subclass of MaskedArray (if possible) or a plain MaskedArray.

ndmin : {0, int}

Minimum number of dimensions

fill_value : {var}

Value used to fill in the masked values when necessary. If None, a default based on the datatype is used.

keep_mask : {True, boolean}

Whether to combine mask with the mask of the input data, if any (True), or to use only mask for the output (False).

hard_mask : {False, boolean}

Whether to use a hard mask or not. With a hard mask, masked values cannot be unmasked.

shrink : {True, boolean}

Whether to force compression of an empty mask.

asanyarray(data, dtype) = array(data, dtype, copy=0, subok=1)

Return a as an masked array. If dtype is not given or None, is is set to the dtype of a. No copy is performed if a is already an array. Subclasses are conserved.

asarray(data, dtype) = array(data, dtype, copy=0, subok=0)

Return a as a MaskedArray object of the given dtype. If dtype is not given or None, is is set to the dtype of a. No copy is performed if a is already an array. Subclasses are converted to the base class MaskedArray.

average(a, axis=None, weights=None, returned=False)

Average the array over the given axis.

Parameters

axis : {None,int}, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

weights : {None, sequence}, optional

Sequence of weights. The weights must have the shape of a, or be 1D with length the size of a along the given axis. If no weights are given, weights are assumed to be 1.

returned : {False, True}, optional

Flag indicating whether a tuple (result, sum of weights/counts) should be returned as output (True), or just the result (False).

bitwise_and()

y = bitwise_and(x1,x2) computes x1 & x2 elementwise.

bitwise_or()

y = bitwise_or(x1,x2) computes x1 | x2 elementwise.

bitwise_xor()

y = bitwise_xor(x1,x2) computes x1 ^ x2 elementwise.

ceil()

y = ceil(x) elementwise smallest integer >= x.

choose(indices, t, out=None, mode='raise')

Return array shaped like indices with elements chosen from t

clip(a, a_min, a_max, out=None)

Return an array whose values are limited to [a_min, a_max].

Parameters

a : {array_like}

Array containing elements to clip.

a_min : :

Minimum value

a_max : :

Maximum value

out : array, optional

The results will be placed in this array. It may be the input array for inplace clipping.

Returns

clipped_array : {array}

A new array whose elements are same as for a, but values < a_min are replaced with a_min, and > a_max with a_max.

Examples

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

common_fill_value(a, b)

Return the common filling value of a and b, if any. If a and b have different filling values, returns None.

compress_cols(a)

Suppress whole columnss of a 2D array that contain masked values.

compress_rowcols(x, axis=None)

Suppress the rows and/or columns of a 2D array that contains masked values.

The suppression behavior is selected with the ‘axis‘parameter. Inline interpreted text or phrase reference start-string without end-string.

- If axis is None, rows and columns are suppressed.
- If axis is 0, only rows are suppressed.
- If axis is 1 or -1, only columns are suppressed.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

Returns

compressed_array : an ndarray.

compress_rows(a)

Suppress whole rows of a 2D array that contain masked values.

compressed(x)

Return a 1-D array of all the non-masked data.

concatenate(arrays, axis=0)

Concatenate the arrays along the given axis.

cos()

y = cos(x) cosine elementwise.

cosh()

y = cosh(x) hyperbolic cosine elementwise.

count(a, axis=None)

Count the non-masked elements of the array along the given axis.

Parameters

axis : int, optional

Axis along which to count the non-masked elements. If not given, all the non masked elements are counted.

Returns

A masked array where the mask is True where all data are :

masked. If axis is None, returns either a scalar ot the :

masked singleton if all values are masked. :

count_masked(arr, axis=None)

Count the number of masked elements along the given axis.

Parameters

axis : int, optional

Axis along which to count. If None (default), a flattened version of the array is used.

default_fill_value(obj)

Calculate the default fill value for the argument object.

divide()

y = divide(x1,x2) divides the arguments elementwise.

dot(a, b, strict=False)

Return the dot product of two 2D masked arrays a and b.

Like the generic numpy equivalent, the product sum is over the last dimension of a and the second-to-last dimension of b. If strict is True, masked values are propagated: if a masked value appears in a row or column, the whole row or column is considered masked.

Parameters

strict : {boolean}

Whether masked data are propagated (True) or set to 0 for the computation.

Notes

The first argument is not conjugated.

dump(a, F)

Pickle the MaskedArray *a* to the file *F*. *F* can either be the handle of an exiting file, or a string representing a file name.

dumps(a)

Return a string corresponding to the pickling of the MaskedArray.

ediff1d(array, to_end=None, to_begin=None)

Return the differences between consecutive elements of an array, possibly with prefixed and/or appended values.

Parameters

array : {array}

Input array, will be flattened before the difference is taken. to_end : {number}, optional If provided, this number will be tacked onto the end of the returned differences. to_begin : {number}, optional If provided, this number will be taked onto the beginning of the returned differences.

Returns

ed : {array}

The differences. Loosely, this will be (ary[1:] - ary[:-1]).

empty(shape, dtype=float, order='C')

Return a new array of given shape and type with all entries uninitialized. This can be faster than zeros.

Parameters

shape : tuple of integers

Shape of the new array

dtype : data-type

The desired data-type for the array.

order : {'C', 'F'}

Whether to store multidimensional data in C or Fortran order.

empty_like(a)

Return an empty (uninitialized) array of the shape and data-type of a.

Note that this does NOT initialize the returned array. If you require your array to be initialized, you should use zeros_like().

equal()

y = equal(x1,x2) returns elementwise x1 == x2 in a bool array

exp()

y = exp(x) e**x elementwise.

expand_dims(a, axis)

Expands the shape of a by including newaxis before axis.

fabs()

y = fabs(x) absolute values.

filled(a, value=None)

Return a as an array with masked data replaced by value. If value is None, get_fill_value(a) is used instead. If a is already a ndarray, a itself is returned.

Parameters

a : maskedarray or array_like

An input object.

value : {var}, optional

Filling value. If not given, the output of get_fill_value(a) is used instead.

Returns

a : array_like

fix_invalid(a, copy=True, fill_value=None)

Return (a copy of) a where invalid data (nan/inf) are masked and replaced by fill_value.

Note that a copy is performed by default (just in case...).

Parameters

a : array_like

A (subclass of) ndarray.

copy : bool

Whether to use a copy of a (True) or to fix a in place (False).

fill_value : {var}, optional

Value used for fixing invalid data. If not given, the output of get_fill_value(a) is used instead.

Returns

b : MaskedArray

flatnotmasked_contiguous(a)

Find contiguous unmasked data in a flattened masked array.

Return a sorted sequence of slices (start index, end index).

flatnotmasked_edges(a)

Find the indices of the first and last not masked values in a 1D masked array. If all values are masked, returns None.

floor()

y = floor(x) elementwise largest integer <= x

floor_divide()

y = floor_divide(x1,x2) floor divides the arguments elementwise.

fmod()

y = fmod(x1,x2) computes (C-like) x1 % x2 elementwise.

get_data(a, subok=True)

Return the _data part of a (if any), or a as a ndarray.

Parameters

a : array_like

A ndarray or a subclass of.

subok : bool

Whether to force the output to a 'pure' ndarray (False) or to return a subclass of ndarray if approriate (True).

get_mask(a)

Return the mask of a, if any, or nomask.

To get a full array of booleans of the same shape as a, use getmaskarray.

getmaskarray(a)

Return the mask of a, if any, or a boolean array of the shape of a, full of False.

greater()

y = greater(x1,x2) returns elementwise x1 > x2 in a bool array.

greater_equal()

y = greater_equal(x1,x2) returns elementwise x1 >= x2 in a bool array.

hypot()

y = hypot(x1,x2) sqrt(x1**2 + x2**2) elementwise

indices(dimensions, dtype=<type 'int'>)

Returns an array representing a grid of indices with row-only, and column-only variation.

inner(a, b)

innerproduct(a,b) Returns the inner product of a and b for arrays of floating point types. Like the generic NumPy equivalent the product sum is over the last dimension of a and b. NB: The first argument is not conjugated. Notes —— Masked values are replaced by 0.

inner(a, b)

innerproduct(a,b) Returns the inner product of a and b for arrays of floating point types. Like the generic NumPy equivalent the product sum is over the last dimension of a and b. NB: The first argument is not conjugated. Notes —— Masked values are replaced by 0.

isMaskedArray(x)

Is x a masked array, that is, an instance of MaskedArray?

isMaskedArray(x)

Is x a masked array, that is, an instance of MaskedArray?

is_mask(m)

Return True if m is a legal mask.

Does not check contents, only type.

is_masked(x)

Does x have masked values?

isMaskedArray(x)

Is x a masked array, that is, an instance of MaskedArray?

left_shift(a, n)

Left shift n bits.

less()

y = less(x1,x2) returns elementwise x1 < x2 in a bool array.

less_equal()

y = less_equal(x1,x2) returns elementwise x1 <= x2 in a bool array

load(F)

Wrapper around `cPickle.load` which accepts either a file-like object or a filename.

loads(strg)

Load a pickle from the current string.

log()

y = log(x) logarithm base e elementwise.

log10()

y = log10(x) logarithm base 10 elementwise.

logical_and()

y = logical_and(x1,x2) returns x1 and x2 elementwise.

logical_not()

y = logical_not(x) returns not x elementwise.

logical_or()

y = logical_or(x1,x2) returns x1 or x2 elementwise.

logical_xor()

y = logical_xor(x1,x2) returns x1 xor x2 elementwise.

make_mask(m, copy=False, shrink=True, flag=None)

Return m as a mask, creating a copy if necessary or requested.

The function can accept any sequence of integers or nomask. Does not check that contents must be 0s and 1s.

Parameters

m : array_like

Potential mask.

copy : bool

Whether to return a copy of m (True) or m itself (False).

shrink : bool

Whether to shrink m to nomask if all its values are False.

make_mask_none(s)

Return a mask of shape s, filled with False.

Parameters

s : tuple

A tuple indicating the shape of the final mask.

mask_cols(a, axis=None)

Mask whole columns of a 2D array that contain masked values.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

mask_or(m1, m2, copy=False, shrink=True)

Return the combination of two masks m1 and m2.

The masks are combined with the *logical_or* operator, treating nomask as False. The result may equal m1 or m2 if the other is nomask.

Parameters

m1 : array_like

First mask.

m2 : array_like

Second mask

copy : bool

Whether to return a copy.

shrink : bool

Whether to shrink m to nomask if all its values are False.

mask_rowcols(a, axis=None)

Mask whole rows and/or columns of a 2D array that contain masked values. The masking behavior is selected with the ‘axis‘parameter.

Inline interpreted text or phrase reference start-string without end-string.

- If axis is None, rows and columns are masked.
- If axis is 0, only rows are masked.
- If axis is 1 or -1, only columns are masked.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

Returns

a *pure* ndarray. :

mask_rows(a, axis=None)

Mask whole rows of a 2D array that contain masked values.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

masked_all(shape, dtype=<type 'float'>)

Return an empty masked array of the given shape and dtype, where all the data are masked.

Parameters

dtype : dtype, optional

Data type of the output.

masked_all_like(arr)

Return an empty masked array of the same shape and dtype as the array *a*, where all the data are masked.

masked_equal(x, value, copy=True)

Shortcut to masked_where, with condition = (x == value). For floating point, consider *masked_values(x, value)* instead.

masked_greater(x, value, copy=True)

Shortcut to masked_where, with condition = (x > value).

masked_greater_equal(x, value, copy=True)

Shortcut to masked_where, with condition = (x >= value).

masked_inside(x, v1, v2, copy=True)

Shortcut to masked_where, where condition is True for x inside the interval [v1,v2] (v1 <= x <= v2). The boundaries v1 and v2 can be given in either order.

Notes

The array x is prefilled with its filling value.

masked_invalid(a, copy=True)

Mask the array for invalid values (nans or infs). Any preexisting mask is conserved.

masked_less(x, value, copy=True)

Shortcut to masked_where, with condition = (x < value).

masked_less_equal(x, value, copy=True)

Shortcut to masked_where, with condition = (x <= value).

masked_not_equal(x, value, copy=True)

Shortcut to masked_where, with condition = (x != value).

masked_object(x, value, copy=True)

Mask the array x where the data are exactly equal to value.

This function is suitable only for object arrays: for floating point, please use `masked_values` instead.

Notes

The mask is set to *nomask* if posible.

masked_outside(x, v1, v2, copy=True)

Shortcut to masked_where, where condition is True for x outside the interval [v1,v2] (x < v1)|(x > v2). The boundaries v1 and v2 can be given in either order.

Notes

The array x is prefilled with its filling value.

masked_values(x, value, rtol=1.0000000000000001e-05, atol=1e-08, copy=True)

Mask the array x where the data are approximately equal in value, i.e.

(abs(x - value) <= atol+rtol*abs(value))

Suitable only for floating points. For integers, please use `masked_equal`. The mask is set to nomask if posible.

Parameters

x : array_like

Array to fill.

value : float

Masking value.

rtol : float

Tolerance parameter.

atol : float

Tolerance parameter (1e-8).

copy : bool

Whether to return a copy of x.

masked_where(condition, a, copy=True)

Return a as an array masked where condition is true.

Masked values of a or condition are kept.

Parameters

condition : array_like

Masking condition.

a : array_like

Array to mask.

copy : bool

Whether to return a copy of a (True) or modify a in place.

max(obj, axis=None, out=None)

Return the maximum/a along the given axis.

Masked values are filled with fill_value.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

fill_value : {var}, optional

Value used to fill in the masked values. If None, use the the output of maximum_fill_value().

median(a, axis=0, out=None, overwrite_input=False)

Compute the median along the specified axis.

Returns the median of the array elements. The median is taken over the first axis of the array by default, otherwise over the specified axis.

Parameters

a : array-like

Input array or object that can be converted to an array

axis : {int, None}, optional

Axis along which the medians are computed. The default is to compute the median along the first dimension. axis=None returns the median of the flattened array

out : {None, ndarray}, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

overwrite_input : {False, True}, optional

If True, then allow use of memory of input array (a) for calculations. The input array will be modified by the call to median. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is False. Note that, if overwrite_input is true, and the input is not already an ndarray, an error will be raised.

Returns

median : ndarray.

A new array holding the result is returned unless out is specified, in which case a reference to out is returned. Return datatype is float64 for ints and floats smaller than float64, or the input datatype otherwise.

See Also :

———- :

mean :

Notes

Given a vector V length N, the median of V is the middle value of a sorted copy of V (Vs) - i.e. Vs[(N-1)/2], when N is odd. It is the mean of the two middle values of Vs, when N is even.

min(array, axis=None, out=None)

Return the minimum of a along the given axis.

Masked values are filled with fill_value.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

fill_value : {var}, optional

Value used to fill in the masked values. If None, use the the output of minimum_fill_value().

multiply()

y = multiply(x1,x2) multiplies the arguments elementwise.

negative()

y = negative(x) determines -x elementwise

not_equal()

y = not_equal(x1,x2) returns elementwise x1 != x2

Inline substitution_reference start-string without end-string.

notmasked_contiguous(a, axis=None)

Find contiguous unmasked data in a masked array along the given axis.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

Returns

A sorted sequence of slices (start index, end index). :

Notes

Only accepts 2D arrays at most.

notmasked_edges(a, axis=None)

Find the indices of the first and last not masked values along the given axis in a masked array.

If all values are masked, return None. Otherwise, return a list of 2 tuples, corresponding to the indices of the first and last unmasked values respectively.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

ones(new_shape, dtype=<type 'float'>)

Returns an array of the given dimensions which is initialized to all ones.

outer(a, b)

Returns the outer product of two vectors.

result[i,j] = a[i]*b[j] when a and b are vectors. Will accept any arguments that can be made into vectors.

Notes

Masked values are replaced by 0.

outer(a, b)

Returns the outer product of two vectors.

result[i,j] = a[i]*b[j] when a and b are vectors. Will accept any arguments that can be made into vectors.

Notes

Masked values are replaced by 0.

polyfit(x, y, deg, rcond=None, full=False)

Least squares polynomial fit.

Do a best fit polynomial of degree 'deg' of 'x' to 'y'. Return value is a vector of polynomial coefficients [pk ... p1 p0]. Eg, for n=2

p2*x0^2 + p1*x0 + p0 = y1 p2*x1^2 + p1*x1 + p0 = y1 p2*x2^2 + p1*x2 + p0 = y2 ..... p2*xk^2 + p1*xk + p0 = yk

Parameters

x : array_like

1D vector of sample points.

y : array_like

1D vector or 2D array of values to fit. The values should run down the columes in the 2D case.

deg : integer

Degree of the fitting polynomial

rcond: {None, float}, optional :

Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The defaul value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

full : {False, boolean}, optional

Switch determining nature of return value. When it is False just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

Returns

coefficients, [residuals, rank, singular_values, rcond] : variable

When full=False, only the coefficients are returned, running down the appropriate colume when y is a 2D array. When full=True, the rank of the scaled Vandermonde matrix, it's effective rank in light of the rcond value, its singular values, and the specified value of rcond are also returned.

Notes

Any masked values in x is propagated in y, and vice-versa.

power(a, b, third=None)

Computes a**b elementwise.

put(a, indices, values, mode='raise')

Set storage-indexed locations to corresponding values.

Values and indices are filled if necessary.

putmask(a, mask, values)

Set a.flat[n] = values[n] for each n where mask.flat[n] is true.

If values is not the same size of a and mask then it will repeat as necessary. This gives different behavior than a[mask] = values.

Note: Using a masked array as values will NOT transform a ndarray in a maskedarray.

rank(obj)

Return the number of dimensions of a.

In old Numeric, rank was the term used for the number of dimensions. If a is not already an array, a conversion is attempted. Scalars are zero dimensional.

Parameters

a : {array_like}

Array whose number of dimensions is desired. If a is not an array, a conversion is attempted.

Returns

number_of_dimensions : {integer}

Returns the number of dimensions.

Examples

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

remainder()

y = remainder(x1,x2) computes x1-n*x2 where n is floor(x1 / x2)

reshape(a, new_shape)

Change the shape of the array a to new_shape.

resize(x, new_shape)

Return a new array with the specified shape.

The total size of the original array can be any size. The new array is filled with repeated copies of a. If a was masked, the new array will be masked, and the new mask will be a repetition of the old one.

right_shift(a, n)

Right shift n bits.

round_(a, decimals=0, out=None)

Return a copy of a, rounded to 'decimals' places.

When 'decimals' is negative, it specifies the number of positions to the left of the decimal point. The real and imaginary parts of complex numbers are rounded separately. Nothing is done if the array is not of float type and 'decimals' is greater than or equal to 0.

Parameters

decimals : int

Number of decimals to round to. May be negative.

out : array_like

Existing array to use for output. If not given, returns a default copy of a.

Notes

If out is given and does not have a mask attribute, the mask of a is lost!

set_fill_value(a, fill_value)

Set the filling value of a, if a is a masked array. Otherwise, do nothing.

Returns

None :

shape(obj)

Return the shape of a.

Parameters

a : {array_like}

Array whose shape is desired. If a is not an array, a conversion is attempted.

Returns

tuple_of_integers : :

The elements of the tuple are the length of the corresponding array dimension.

Examples

```
>>> shape(eye(3))
(3, 3)
>>> shape([[1,2]])
(1, 2)
```

sin()

y = sin(x) sine elementwise.

sinh()

y = sinh(x) hyperbolic sine elementwise.

size(obj, axis=None)

Return the number of elements along given axis.

Parameters

a : {array_like}

Array whose axis size is desired. If a is not an array, a conversion is attempted.

axis : {None, integer}, optional

Axis along which the elements are counted. None means all elements in the array.

Returns

element_count : {integer}

Count of elements along specified axis.

Examples

```
>>> a = array([[1,2,3],[4,5,6]])
>>> size(a)
6
>>> size(a,1)
3
>>> size(a,0)
2
```

reduce(self, target, axis=0, dtype=None)

Reduce *target* along the given *axis*.

sort(a, axis=-1, kind='quicksort', order=None, endwith=True, fill_value=None)

Sort along the given axis.

Parameters

axis : int

Axis to be indirectly sorted.

kind : {string}

Sorting algorithm (default 'quicksort') Possible values: 'quicksort', 'mergesort', or 'heapsort'.

order : {var}

If a has fields defined, then the order keyword can be the field name to sort on or a list (or tuple) of field names to indicate the order that fields should be used to define the sort.

fill_value : {var}

Value used to fill in the masked values. If None, use the the output of minimum_fill_value().

endwith : bool

Whether missing values (if any) should be forced in the upper indices (at the end of the array) (True) or lower indices (at the beginning).

Returns

When used as method, returns None. :

When used as a function, returns an array.

Notes

This method sorts 'a' in place along the given axis using the algorithm specified by the kind keyword.

The various sorts may characterized by average speed, worst case performance need for work space, and whether they are stable. A stable sort keeps items with the same key in the same relative order and is most useful when used w/ argsort where the key might differ from the items being sorted. The three available algorithms have the following properties:

|——————————————————| | kind | speed | worst case | work space | stable| |————————————————————| **|'quicksort'|** 1 | O(n^2) | 0 | no | **|'mergesort'|** 2 | O(n*log(n)) | ~n/2 | yes | **|'heapsort' | 3 | O(n*log(n)) | 0 | no | |**——————————————————|

sqrt()

y = sqrt(x) square-root elementwise. For real x, the domain is restricted to x>=0.

subtract()

y = subtract(x1,x2) subtracts the arguments elementwise.

tan()

y = tan(x) tangent elementwise.

tanh()

y = tanh(x) hyperbolic tangent elementwise.

transpose(a, axes=None)

Return a view of the array with dimensions permuted according to axes, as a masked array.

If `axes` is None (default), the output view has reversed dimensions compared to the original.

true_divide()

y = true_divide(x1,x2) true divides the arguments elementwise.

vander(x, n=None)

Generate the Vandermonde matrix of vector x.

The i-th column of X is the the (N-i)-1-th power of x. N is the maximum power to compute; if N is None it defaults to len(x).

Notes

Masked values in x will result in rows of zeros.

where(condition | x, y)

Returns a (subclass of) masked array, shaped like condition, where the elements are x when condition is True, and y otherwise. If neither x nor y are given, returns a tuple of indices where condition is True (a la condition.nonzero()).

Parameters

condition : {var}

The condition to meet. Must be convertible to an integer array.

x : {var}, optional

Values of the output when the condition is met

y : {var}, optional

Values of the output when the condition is not met.

zeros(shape, dtype=float, order='C')

Return a new array of given shape and type, filled zeros.

Parameters

shape : tuple of integers

Shape of the new array

dtype : data-type

The desired data-type for the array.

order : {'C', 'F'}

Whether to store multidimensional data in C or Fortran order.

3.5 Random Sample Generation

3.5.1 numpy.random

beta()

Beta distribution over [0, 1].

beta(a, b, size=None) -> random values

binomial()

Binomial distribution of n trials and p probability of success.

binomial(n, p, size=None) -> random values

bytes()

Return random bytes.

bytes(length) -> str

chisquare()

Chi^2 distribution.

chisquare(df, size=None) -> random values

dirichlet(alpha, size=None)

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 : array

Number of samples to draw.

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 **:math:‘X = \frac{1}{\sum_{i=1}^k{y_i}} (y_1, \ldot, y_n)‘** is Dirichlet distributed.

Unknown LaTeX command: ldot

References

exponential()

Exponential distribution.

exponential(scale=1.0, size=None) -> random values

f()

F distribution.

f(dfnum, dfden, size=None) -> random values

gamma()

Gamma distribution.

gamma(shape, scale=1.0, size=None) -> random values

geometric()

Geometric distribution with p being the probability of “success” on an individual trial.

geometric(p, size=None)

get_state()

Return a tuple representing the internal state of the generator.

get_state() -> (‘MT19937’, int key[624], int pos, int has_gauss, float cached_gaussian)

[1] David McKay, “Information Theory, Inference and Learning Algorithms,” chapter 23, http://www.inference.phy.cam.ac.uk/mackay/

gumbel()

Gumbel distribution.

gumbel(loc=0.0, scale=1.0, size=None)

hypergeometric()

Hypergeometric distribution.

Consider an urn with ngood "good" balls and nbad "bad" balls. If one were to draw nsample balls from the urn without replacement, then the hypergeometric distribution describes the distribution of "good" balls in the sample.

hypergeometric(ngood, nbad, nsample, size=None)

laplace()

Laplace distribution.

laplace(loc=0.0, scale=1.0, size=None)

logistic()

Logistic distribution.

logistic(loc=0.0, scale=1.0, size=None)

lognormal()

Log-normal distribution.

Note that the mean parameter is not the mean of this distribution, but the underlying normal distribution.

    lognormal(mean, sigma) <=> exp(normal(mean, sigma))

lognormal(mean=0.0, sigma=1.0, size=None)

logseries()

Logarithmic series distribution.

logseries(p, size=None)

multinomial()

Multinomial distribution.

multinomial(n, pvals, size=None) -> random values

pvals is a sequence of probabilities that should sum to 1 (however, the last element is always assumed to account for the remaining probability as long as sum(pvals[:-1]) <= 1).

multivariate_normal()

Return an array containing multivariate normally distributed random numbers with specified mean and covariance.

multivariate_normal(mean, cov) -> random values multivariate_normal(mean, cov, [m, n, ...]) -> random values

mean must be a 1 dimensional array. cov must be a square two dimensional array with the same number of rows and columns as mean has elements.

The first form returns a single 1-D array containing a multivariate normal.

The second form returns an array of shape (m, n, ..., cov.shape[0]). In this case, output[i,j,...,:] is a 1-D array containing a multivariate normal.

negative_binomial()

Negative Binomial distribution.

negative_binomial(n, p, size=None) -> random values

noncentral_chisquare()

Noncentral Chi^2 distribution.

noncentral_chisquare(df, nonc, size=None) -> random values

noncentral_f()

Noncentral F distribution.

noncentral_f(dfnum, dfden, nonc, size=None) -> random values

normal()

Normal distribution (mean=loc, stdev=scale).

normal(loc=0.0, scale=1.0, size=None) -> random values

pareto()

Pareto distribution.

pareto(a, size=None)

permutation()

Given an integer, return a shuffled sequence of integers >= 0 and < x; given a sequence, return a shuffled array copy.

permutation(x)

poisson()

Poisson distribution.

poisson(lam=1.0, size=None) -> random values

power()

Power distribution.

power(a, size=None)

rand()

Return an array of the given dimensions which is initialized to random numbers from a uniform distribution in the range [0,1).

rand(d0, d1, ..., dn) -> random values

Note: This is a convenience function. If you want an interface that takes a tuple as the first argument use numpy.random.random_sample(shape_tuple).

randint()

Return random integers x such that low <= x < high.

randint(low, high=None, size=None) -> random values

If high is None, then 0 <= x < low.

randn()

Returns zero-mean, unit-variance Gaussian random numbers in an array of shape (d0, d1, ..., dn).

randn(d0, d1, ..., dn) -> random values

Note: This is a convenience function. If you want an interface that takes a tuple as the first argument use numpy.random.standard_normal(shape_tuple).

random_sample()

Return random floats in the half-open interval [0.0, 1.0).

random_sample(size=None) -> random values

random_integers()

Return random integers x such that low <= x <= high.

random_integers(low, high=None, size=None) -> random values.

If high is None, then 1 <= x <= low.

random_sample()

Return random floats in the half-open interval [0.0, 1.0).

random_sample(size=None) -> random values

random_sample()

Return random floats in the half-open interval [0.0, 1.0).

random_sample(size=None) -> random values

rayleigh()

Rayleigh distribution.

rayleigh(scale=1.0, size=None)

random_sample()

Return random floats in the half-open interval [0.0, 1.0).

random_sample(size=None) -> random values

seed()

Seed the generator.

seed(seed=None)

seed can be an integer, an array (or other sequence) of integers of any length, or None. If seed is None, then RandomState will try to read data from /dev/urandom (or the Windows analogue) if available or seed from the clock otherwise.

set_state()

Set the state from a tuple.

state = ('MT19937', int key[624], int pos, int has_gauss, float cached_gaussian)

For backwards compatibility, the following form is also accepted although it is missing some information about the cached Gaussian value.

state = ('MT19937', int key[624], int pos)

set_state(state)

shuffle()

Modify a sequence in-place by shuffling its contents.

shuffle(x)

standard_cauchy()

Standard Cauchy with mode=0.

standard_cauchy(size=None)

standard_exponential()

Standard exponential distribution (scale=1).

standard_exponential(size=None) -> random values

standard_gamma()

Standard Gamma distribution.

standard_gamma(shape, size=None) -> random values

standard_normal()

Standard Normal distribution (mean=0, stdev=1).

standard_normal(size=None) -> random values

standard_t()

Standard Student's t distribution with df degrees of freedom.

standard_t(df, size=None)

test(level=1, verbosity=1)

triangular()

Triangular distribution starting at left, peaking at mode, and ending at right (left <= mode <= right).

triangular(left, mode, right, size=None)

uniform()

Uniform distribution over [low, high).

uniform(low=0.0, high=1.0, size=None) -> random values

vonmises()

von Mises circular distribution with mode mu and dispersion parameter kappa on [-pi, pi].

vonmises(mu, kappa, size=None)

wald()

Wald (inverse Gaussian) distribution.

wald(mean, scale, size=None)

weibull()

Weibull distribution.

weibull(a, size=None)

zipf()

Zipf distribution.

zipf(a, size=None)

CHAPTER

FOUR

Indices and tables

- *Index*
- *Module Index*
- *Search Page*

INDEX

F

N