This page contains a large database of examples demonstrating most of the Numpy functionality. Numpy_Example_List_With_Doc has these examples interleaved with the built-in documentation, but is not as regularly updated as this page. The examples here can be easily accessed from Python using the Numpy_Example_Fetcher.

This example list is incredibly useful, and we would like to get all the good examples and comments integrated in the official numpy documentation so that they are also shipped with numpy. **You can help**. The official numpy documentation can be edited on http://docs.scipy.org.

Contents

68. diagonal()
69. diff()
70. digitize()
71. dot()
72. dsplit()
73. dstack()
74. dtype()
75. empty()
76. empty_like()
77. expand_dims()
78. eye()
79. fft()
80. fftfreq()
81. fftshift()
82. fill()
83. finfo()
84. fix()
85. flat
86. flatten()
87. fliplr()
88. flipud()
89. floor()
90. fromarrays()
91. frombuffer()
92. fromfile()
93. fromfunction()
94. fromiter()
95. generic
96. gumbel()
97. histogram()
98. hsplit()
99. hstack()
100. hypot()
101. identity()
102. ifft()
103. imag
104. index_exp[]
105. indices()
106. inf
107. inner()
108. insert()
109. inv()
110. iscomplex()
111. iscomplexobj()
112. item()
113. ix_()
114. lexsort()
115. linspace()
116. loadtxt()
117. logical_and()
118. logical_not()
119. logical_or()
120. logical_xor()
121. logspace()
122. lstsq()
123. mat()
124. matrix()
125. max()
126. maximum()
127. mean()
128. median()
129. mgrid[]
130. min()
131. minimum()
132. multiply()
133. nan
134. ndenumerate()
135. ndim
136. ndindex()
137. newaxis
138. nonzero()

138. nonzero()
139. ogrid()
140. ones()
141. ones_like()
142. outer()
143. permutation()
144. piecewise()
145. pinv()
146. poisson()
147. poly1d()
148. polyfit()
149. prod()
150. ptp()
151. put()
152. putmask()
153. r_[]
154. rand()
155. randint()
156. randn()
157. random_integers()
158. random_sample()
159. ranf()
160. ravel()
161. real
162. recarray()
163. reduce()
164. repeat()
165. reshape()
166. resize()
167. rollaxis()
168. round()
169. rot90()
170. s_[]
171. sample()
172. savetxt()
173. searchsorted()
174. seed()
175. select()
176. set_printoptions()
177. shape
178. shuffle()
179. slice()
180. solve()
181. sometrue()
182. sort()
183. split()
184. squeeze()
185. std()
186. standard_normal()
187. sum()
188. svd()
189. swapaxes()
190. T
191. take()
192. tensordot()
193. tile()
194. tofile()
195. tolist()
196. trace()
197. transpose()
198. tri()
199. tril()
200. trim_zeros()
201. triu()
202. typeDict()
203. uniform()
204. unique()
205. unique1d()
206. vander()
207. var()
208. vdot()

208. vdot()
209. vectorize()
210. view()
211. vonmises()
212. vsplit()
213. vstack()
214. weibull()
215. where()
216. zeros()
217. zeros_like()

...

```
>>> from numpy import *
>>> a = arange(12)
>>> a = a.reshape(3,2,2)
>>> print a
[[[ 0  1]
  [ 2  3]]
 [[ 4  5]
  [ 6  7]]
 [[ 8  9]
  [10 11]]]
>>> a[...,0]                                   # same as a[:,:,0]
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
>>> a[1:,...] # same as a[1:,:,:] or just a[1:]
array([[[ 4, 5],
        [ 6, 7]],
       [[ 8, 9],
        [10, 11]]])
```

See also: [], newaxis

[]

```
>>> from numpy import *
>>> a = array([ [ 0, 1, 2, 3, 4],
... [10,11,12,13,14],
... [20,21,22,23,24],
... [30,31,32,33,34] ])
>>>
>>> a[0,0] # indices start by zero
0
>>> a[-1] # last row
array([30, 31, 32, 33, 34])
>>> a[1:3,1:4] # subarray
array([[11, 12, 13],
       [21, 22, 23]])
>>>
>>> i = array([0,1,2,1]) # array of indices for the first axis
>>> j = array([1,2,3,4]) # array of indices for the second axis
>>> a[i,j]
array([ 1, 12, 23, 14])
>>>
>>> a[a<13] # boolean indexing
array([ 0, 1, 2, 3, 4, 10, 11, 12])
>>>
>>> b1 = array( [True,False,True,False] ) # boolean row selector
>>> a[b1,:]
array([[ 0, 1, 2, 3, 4],
       [20, 21, 22, 23, 24]])
>>>
>>> b2 = array( [False,True,True,False,True] ) # boolean column selector
```

```
>>> a[:,b2]
array([[ 1,  2,  4],
       [11, 12, 14],
       [21, 22, 24],
       [31, 32, 34]])
```

See also: ..., newaxis, ix_, indices, nonzero, where, slice

abs()

```
>>> from numpy import *
>>> abs(-1)
1
>>> abs(array([-1.2, 1.2]))
array([ 1.2,  1.2])
>>> abs(1.2+1j)
1.5620499351813308
```

See also: absolute, angle

absolute()

Synonym for abs()

See abs

accumulate()

```
>>> from numpy import *
>>> add.accumulate(array([1.,2.,3.,4.])) # like reduce() but also gives
intermediate results
array([ 1.,  3.,  6., 10.])
>>> array([1., 1.+2., (1.+2.)+3., ((1.+2.)+3.)+4.]) # this is what it
computed
array([ 1.,  3.,  6., 10.])
>>> multiply.accumulate(array([1.,2.,3.,4.])) # works also with other
operands
array([ 1.,  2.,  6., 24.])
>>> array([1., 1.*2., (1.*2.)*3., ((1.*2.)*3.)*4.]) # this is what it
computed
array([ 1.,  2.,  6., 24.])
>>> add.accumulate(array([[1,2,3],[4,5,6]]), axis = 0) # accumulate
every column separately
array([[1, 2, 3],
       [5, 7, 9]])
>>> add.accumulate(array([[1,2,3],[4,5,6]]), axis = 1) # accumulate
every row separately
array([[ 1,  3,  6],
       [ 4,  9, 15]])
```

See also: reduce, cumprod, cumsum

add()

```
>>> from numpy import *
>>> add(array([-1.2, 1.2]), array([1,3]))
array([-0.2,  4.2])
```

```
>>> array([-1.2, 1.2]) + array([1,3])
array([-0.2, 4.2])
```

all()

```
>>> from numpy import *
>>> a = array([True, False, True])
>>> a.all() # if all elements of a are True: return True; otherwise
False
False
>>> all(a) # this form also exists
False
>>> a = array([1,2,3])
>>> all(a > 0) # equivalent to (a > 0).all()
True
```

See also: any, alltrue, sometrue

allclose()

```
>>> allclose(array([1e10,1e-7]), array([1.00001e10,1e-8]))
False
>>> allclose(array([1e10,1e-8]), array([1.00001e10,1e-9]))
True
>>> allclose(array([1e10,1e-8]), array([1.0001e10,1e-9]))
False
```

alltrue()

```
>>> from numpy import *
>>> b = array([True, False, True, True])
>>> alltrue(b)
False
>>> a = array([1, 5, 2, 7])
>>> alltrue(a >= 5)
False
```

See also: sometrue, all, any

angle()

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

See also: real, imag, hypot

any()

```
>>> from numpy import *
>>> a = array([True, False, True])
>>> a.any() # gives True if at least 1 element of a is True, otherwise
```

```
False
True
>>> any(a) # this form also exists
True
>>> a = array([1,2,3])
>>> (a >= 1).any() # equivalent to any(a >= 1)
True
```

See also: all, alltrue, sometrue

append()

```
>>> from numpy import *
>>> a = array([10,20,30,40])
>>> append(a,50)
array([10, 20, 30, 40, 50])
>>> append(a,[50,60])
array([10, 20, 30, 40, 50, 60])
>>> a = array([[10,20,30],[40,50,60],[70,80,90]])
>>> append(a,[[15,15,15]],axis=0)
array([[10, 20, 30],
       [40, 50, 60],
       [70, 80, 90],
       [15, 15, 15]])
>>> append(a,[[15],[15],[15]],axis=1)
array([[10, 20, 30, 15],
       [40, 50, 60, 15],
       [70, 80, 90, 15]])
```

See also: insert, delete, concatenate

apply_along_axis()

```
>>> from numpy import *
>>> def myfunc(a): # function works on a 1d arrays, takes the average of
the 1st an last element
... return (a[0]+a[-1])/2
...
>>> b = array([[1,2,3],[4,5,6],[7,8,9]])
>>> apply_along_axis(myfunc,0,b) # apply myfunc to each column (axis=0)
of b
array([4, 5, 6])
>>> apply_along_axis(myfunc,1,b) # apply myfunc to each row (axis=1) of
b
array([2, 5, 8])
```

See also: apply_over_axes, vectorize

apply_over_axes()

```
>>> from numpy import *
>>> a = arange(24).reshape(2,3,4) # a has 3 axes: 0,1 and 2
>>> a
array([[[ 0, 1, 2, 3],
        [ 4, 5, 6, 7],
        [ 8, 9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
>>> apply_over_axes(sum, a, [0,2]) # sum over all axes except axis=1,
```

```
result has same shape as original
array([[[ 60],
        [ 92],
        [124]]])
```

See also: apply_along_axis, vectorize

arange()

```
>>> from numpy import *
>>> arange(3)
array([0, 1, 2])
>>> arange(3.0)
array([ 0., 1., 2.])
>>> arange(3, dtype=float)
array([ 0., 1., 2.])
>>> arange(3,10) # start,stop
array([3, 4, 5, 6, 7, 8, 9])
>>> arange(3,10,2) # start,stop,step
array([3, 5, 7, 9])
```

See also: r_, linspace, logspace, mgrid, ogrid

arccos()

```
>>> from numpy import *
>>> arccos(array([0, 1]))
array([ 1.57079633, 0. ])
```

See also: arcsin, arccosh, arctan, arctan2

arccosh()

```
>>> from numpy import *
>>> arccosh(array([e, 10.0]))
array([ 1.65745445, 2.99322285])
```

See also: arccos, arcsinh, arctanh

arcsin()

```
>>> from numpy import *
>>> arcsin(array([0, 1]))
array([ 0. , 1.57079633])
```

See also: arccos, arctan, arcsinh

arcsinh()

```
>>> from numpy import *
>>> arcsinh(array([e, 10.0]))
array([ 1.72538256, 2.99822295])
```

See also: arccosh, arcsin, arctanh

arctan()

```
>>> from numpy import *
>>> arctan(array([0, 1]))
array([ 0. , 0.78539816])
```

See also: arccos, arcsin, arctanh

arctan2()

```
>>> from numpy import *
>>> arctan2(array([0, 1]), array([1, 0]))
array([ 0. , 1.57079633])
```

See also: arcsin, arccos, arctan, arctanh

arctanh()

```
>>> from numpy import *
>>> arctanh(array([0, -0.5]))
array([ 0. , -0.54930614])
```

See also: arcsinh, arccosh, arctan, arctan2

argmax()

```
>>> from numpy import *
>>> a = array([10,20,30])
>>> maxindex = a.argmax()
>>> a[maxindex]
30
>>> a = array([[10,50,30],[60,20,40]])
>>> maxindex = a.argmax()
>>> maxindex
3
>>> a.ravel()[maxindex]
60
>>> a.argmax(axis=0) # for each column: the row index of the maximum
value
array([1, 0, 1])
>>> a.argmax(axis=1) # for each row: the column index of the maximum
value
array([1, 0])
>>> argmax(a) # also exists, slower, default is axis=-1
array([1, 0])
```

See also: argmin, nan, min, max, maximum, minimum

argmin()

```
>>> from numpy import *
>>> a = array([10,20,30])
>>> minindex = a.argmin()
```

```
>>> a[minindex]
10
>>> a = array([[10,50,30],[60,20,40]])
>>> minindex = a.argmin()
>>> minindex
0
>>> a.ravel()[minindex]
10
>>> a.argmin(axis=0) # for each column: the row index of the minimum
value
array([0, 1, 0])
>>> a.argmin(axis=1) # for each row: the column index of the minimum
value
array([0, 1])
>>> argmin(a) # also exists, slower, default is axis=-1
array([0, 1])
```

See also: argmax, nan, min, max, maximum, minimum

argsort()

argsort(axis=-1, kind="quicksort")

```
>>> from numpy import *
>>> a = array([2,0,8,4,1])
>>> ind = a.argsort() # indices of sorted array using quicksort
(default)
>>> ind
array([1, 4, 0, 3, 2])
>>> a[ind] # same effect as a.sort()
array([0, 1, 2, 4, 8])
>>> ind = a.argsort(kind='merge') # algorithm options are 'quicksort',
'mergesort' and 'heapsort'
>>> a = array([[8,4,1],[2,0,9]])
>>> ind = a.argsort(axis=0) # sorts on columns. NOT the same as
a.sort(axis=1)
>>> ind
array([[1, 1, 0],
       [0, 0, 1]])
>>> a[ind,[[0,1,2],[0,1,2]]] # 2-D arrays need fancy indexing if you
want to sort them.
array([[2, 0, 1],
       [8, 4, 9]])
>>> ind = a.argsort(axis=1) # sort along rows. Can use
a.argsort(axis=-1) for last axis.
>>> ind
array([[2, 1, 0],
       [1, 0, 2]])
>>> a = ones(17)
>>> a.argsort() # quicksort doesn't preserve original order.
array([ 0, 14, 13, 12, 11, 10, 9, 15, 8, 6, 5, 4, 3, 2, 1, 7, 16])
>>> a.argsort(kind="mergesort") # mergesort preserves order when
possible. It is a stable sort.
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])
>>> ind = argsort(a) # there is a functional form
```

See also: lexsort, sort

array()

```
>>> from numpy import *
>>> array([1,2,3]) # conversion from a list to an array
array([1, 2, 3])
```

```
>>> array([1,2,3], dtype=complex) # output type is specified
array([ 1.+0.j, 2.+0.j, 3.+0.j])
>>> array(1, copy=0, subok=1, ndmin=1) # basically equivalent to
atleast_1d
array([1])
>>> array(1, copy=0, subok=1, ndmin=2) # basically equivalent to
atleast_2d
array([[1]])
>>> array(1, subok=1, ndmin=2) # like atleast_2d but always makes a copy
array([[1]])
>>> mydescriptor = {'names': ('gender','age','weight'), 'formats':
('S1', 'f4', 'f4')} # one way of specifying the data type
>>> a = array([('M',64.0,75.0),('F',25.0,60.0)], dtype=mydescriptor) #
recarray
>>> print a
[('M', 64.0, 75.0) ('F', 25.0, 60.0)]
>>> a['weight']
array([ 75., 60.], dtype=float32)
>>> a.dtype.names # Access to the ordered field names
('gender','age','weight')
>>> mydescriptor = [('age',int16),('Nchildren',int8),('weight',float32)]
# another way of specifying the data type
>>> a = array([(64,2,75.0),(25,0,60.0)], dtype=mydescriptor)
>>> a['Nchildren']
array([2, 0], dtype=int8)
>>> mydescriptor = dtype([('x', 'f4'),('y', 'f4'), # nested recarray
... ('nested', [('i', 'i2'),('j','i2')])])
>>> array([(1.0, 2.0, (1,2))], dtype=mydescriptor) # input one row
array([(1.0, 2.0, (1, 2))],
      dtype=[('x', '<f4'), ('y', '<f4'), ('nested', [('i', '<i2'), ('j',
'<i2')])])
>>> array([(1.0, 2.0, (1,2)), (2.1, 3.2, (3,2))], dtype=mydescriptor) #
input two rows
array([(1.0, 2.0, (1, 2)), (2.0999999046325684, 3.2000000476837158, (3,
2))],
      dtype=[('x', '<f4'), ('y', '<f4'), ('nested', [('i', '<i2'), ('j',
'<i2')])])
>>> a=array([(1.0, 2.0, (1,2)), (2.1, 3.2, (3,2))], dtype=mydescriptor)
# getting some columns
>>> a['x'] # a plain column
array([ 1. , 2.0999999], dtype=float32)
>>> a['nested'] # a nested column
array([(1, 2), (3, 2)],
      dtype=[('i', '<i2'), ('j', '<i2')])
>>> a['nested']['i'] # a plain column inside a nested column
>>> mydescriptor = dtype([('x', 'f4'),('y', 'f4'), # nested recarray
... ('nested', [('i', 'i2'),('j','i2')])])
>>> array([(1.0, 2.0, (1,2))], dtype=mydescriptor) # input one row
array([(1.0, 2.0, (1, 2))],
      dtype=[('x', '<f4'), ('y', '<f4'), ('nested', [('i', '<i2'), ('j',
'<i2')])])
>>> array([(1.0, 2.0, (1,2)), (2.1, 3.2, (3,2))], dtype=mydescriptor) #
input two rows
array([(1.0, 2.0, (1, 2)), (2.0999999046325684, 3.2000000476837158, (3,
2))],
      dtype=[('x', '<f4'), ('y', '<f4'), ('nested', [('i', '<i2'), ('j',
'<i2')])])
>>> a=array([(1.0, 2.0, (1,2)), (2.1, 3.2, (3,2))], dtype=mydescriptor)
# getting some columns
>>> a['x'] # a plain column
array([ 1. , 2.0999999], dtype=float32)
>>> a['nested'] # a nested column
array([(1, 2), (3, 2)],
      dtype=[('i', '<i2'), ('j', '<i2')])
>>> a['nested']['i'] # a plain column inside a nested column
array([1, 3], dtype=int16)
```

See also: dtype, mat, asarray

arrayrange()

Synonym for arange()

See arange

array_split()

```
>>> from numpy import *
>>> a = array([[1,2,3,4],[5,6,7,8]])
>>> array_split(a,2,axis=0) # split a in 2 parts. row-wise
[array([[1, 2, 3, 4]]), array([[5, 6, 7, 8]])]
>>> array_split(a,4,axis=1) # split a in 4 parts, column-wise
[array([[1],
       [5]]), array([[2],
       [6]]), array([[3],
       [7]]), array([[4],
       [8]])]
 >>> array_split(a,3,axis=1) # impossible to split in 3 equal parts ->
first part(s) are bigger
[array([[1, 2],
       [5, 6]]), array([[3],
       [7]]), array([[4],
       [8]])]
>>> array_split(a,[2,3],axis=1) # make a split before the 2nd and the
3rd column
[array([[1, 2],
       [5, 6]]), array([[3],
       [7]]), array([[4],
       [8]])]
```

See also: dsplit, hsplit, vsplit, split, concatenate

asarray()

```
>>> from numpy import *
>>> m = matrix('1 2; 5 8')
>>> m
matrix([[1, 2],
        [5, 8]])
>>> a = asarray(m) # a is array type with same contents as m -- data is
not copied
>>> a
array([[1, 2],
       [5, 8]])
>>> m[0,0] = -99
>>> m
matrix([[-99, 2],
        [ 5, 8]])
>>> a # no copy was made, so modifying m modifies a, and vice versa
array([[-99, 2],
       [ 5, 8]])
```

See also: asmatrix, array, matrix, mat

asanyarray()

```
>>> from numpy import *
>>> a = array([[1,2],[5,8]])
>>> a
```

```
array([[1, 2],
       [5, 8]])
>>> m = matrix('1 2; 5 8')
>>> m
matrix([[1, 2],
        [5, 8]])
>>> asanyarray(a) # the array a is returned unmodified
array([[1, 2],
       [5, 8]])
>>> asanyarray(m) # the matrix m is returned unmodified
matrix([[1, 2],
        [5, 8]])
>>> asanyarray([1,2,3]) # a new array is constructed from the list
array([1, 2, 3])
```

See also: asmatrix, asarray, array, mat

asmatrix()

```
>>> from numpy import *
>>> a = array([[1,2],[5,8]])
>>> a
array([[1, 2],
       [5, 8]])
>>> m = asmatrix(a) # m is matrix type with same contents as a -- data
is not copied
>>> m
matrix([[1, 2],
        [5, 8]])
>>> a[0,0] = -99
>>> a
array([[-99,  2],
       [  5,  8]])
>>> m # no copy was made so modifying a modifies m, and vice versa
matrix([[-99,  2],
        [  5,  8]])
```

See also: asarray, array, matrix, mat

astype()

```
>>> from numpy import *
>>> x = array([1,2,3])
>>> y = x.astype(float64) # convert from int32 to float64
>>> type(y[0])
<type 'numpy.float64'>
>>> x.astype(None) # None implies converting to the default (float64)
array([1., 2., 3.])
```

See also: cast, dtype, ceil, floor, round_, fix

atleast_1d()

```
>>> from numpy import *
>>> a = 1 # 0-d array
>>> b = array([2,3]) # 1-d array
>>> c = array([[4,5],[6,7]]) # 2-d array
>>> d = arange(8).reshape(2,2,2) # 3-d array
>>> d
array([[[0, 1],
```

```
         [2, 3]],
        [[4, 5],
         [6, 7]]])
>>> atleast_1d(a,b,c,d) # all output arrays have dim >= 1
[array([1]), array([2, 3]), array([[4, 5],
        [6, 7]]), array([[[0, 1],
         [2, 3]],
        [[4, 5],
         [6, 7]]])]
```

See also: atleast_2d, atleast_3d, newaxis, expand_dims

atleast_2d()

```
>>> from numpy import *
>>> a = 1 # 0-d array
>>> b = array([2,3]) # 1-d array
>>> c = array([[4,5],[6,7]]) # 2-d array
>>> d = arange(8).reshape(2,2,2) # 3-d array
>>> d
array([[[0, 1],
         [2, 3]],
        [[4, 5],
         [6, 7]]])
>>> atleast_2d(a,b,c,d) # all output arrays have dim >= 2
[array([[1]]), array([[2, 3]]), array([[4, 5],
        [6, 7]]), array([[[0, 1],
         [2, 3]],
        [[4, 5],
         [6, 7]]])]
```

See also: atleast_1d, atleast_3d, newaxis, expand_dims

atleast_3d()

```
>>> from numpy import *
>>> a = 1 # 0-d array
>>> b = array([2,3]) # 1-d array
>>> c = array([[4,5],[6,7]]) # 2-d array
>>> d = arange(8).reshape(2,2,2) # 3-d array
>>> d
array([[[0, 1],
         [2, 3]],
        [[4, 5],
         [6, 7]]])
>>> atleast_3d(a,b,c,d) # all output arrays have dim >= 3
[array([[[1]]]), array([[[2],
         [3]]]), array([[[4],
         [5]],
        [[6],
         [7]]]), array([[[0, 1],
         [2, 3]],
        [[4, 5],
         [6, 7]]])]
```

See also: atleast_1d, atleast_2d, newaxis, expand_dims

average()

```
>>> from numpy import *
>>> a = array([1,2,3,4,5])
>>> w = array([0.1, 0.2, 0.5, 0.2, 0.2]) # weights, not necessarily
normalized
>>> average(a) # plain mean value
3.0
>>> average(a,weights=w) # weighted average
3.1666666666666665
>>> average(a,weights=w,returned=True) # output = weighted average, sum
of weights
(3.1666666666666665, 1.2)
```

See also: mean, median

beta()

```
>>> from numpy import *
>>> from numpy.random import *
>>> beta(a=1,b=10,size=(2,2)) # Beta distribution alpha=1, beta=10
array([[ 0.02571091, 0.04973536],
       [ 0.04887027, 0.02382052]])
```

See also: seed

binary_repr()

```
>>> from numpy import *
>>> a = 25
>>> binary_repr(a) # binary representation of 25
'11001'
>>> b = float_(pi) # numpy float has extra functionality ...
>>> b.nbytes # ... like the number of bytes it takes
8
>>> binary_repr(b.view('u8')) # view float number as an 8 byte integer,
then get binary bitstring
'1010100010001000010110100011000'
```

bincount()

```
>>> from numpy import *
>>> a = array([1,1,1,1,2,2,4,4,5,6,6,6]) # doesn't need to be sorted
>>> bincount(a) # 0 occurs 0 times, 1 occurs 4 times, 2 occurs twice, 3
occurs 0 times, ...
array([0, 4, 2, 0, 2, 1, 3])
>>> a = array([5,4,4,2,2])
>>> w = array([0.1, 0.2, 0.1, 0.3, 0.5])
>>> bincount(a) # 0 & 1 don't occur, 2 occurs twice, 3 doesn't occur, 4
occurs twice, 5 once
array([0, 0, 2, 0, 2, 1])
>>> bincount(a, weights=w)
array([ 0. , 0. , 0.8, 0. , 0.3, 0.1])
>>> # 0 occurs 0 times -> result[0] = 0
>>> # 1 occurs 0 times -> result[1] = 0
>>> # 2 occurs at indices 3 & 4 -> result[2] = w[3] + w[4]
>>> # 3 occurs 0 times -> result[3] = 0
>>> # 4 occurs at indices 1 & 2 -> result[4] = w[1] + w[2]
>>> # 5 occurs at index 0 -> result[5] = w[0]
```

See also: histogram, digitize

binomial()

```
>>> from numpy import *
>>> from numpy.random import *
>>> binomial(n=100,p=0.5,size=(2,3)) # binomial distribution n trials,
p= success probability
array([[38, 50, 53],
       [56, 48, 54]])
>>> from pylab import * # histogram plot example
>>> hist(binomial(100,0.5,(1000)), 20)
```

See also: random_sample, uniform, standard_normal, seed

bitwise_and()

```
>>> from numpy import *
>>> bitwise_and(array([2,5,255]), array([4,4,4]))
array([0, 4, 4])
>>> bitwise_and(array([2,5,255,2147483647L],dtype=int32),
array([4,4,4,2147483647L],dtype=int32))
array([ 0, 4, 4, 2147483647])
```

See also: bitwise_or, bitwise_xor, logical_and

bitwise_or()

```
>>> from numpy import *
>>> bitwise_or(array([2,5,255]), array([4,4,4]))
array([ 6, 5, 255])
>>> bitwise_or(array([2,5,255,2147483647L],dtype=int32),
array([4,4,4,2147483647L],dtype=int32))
array([ 6, 5, 255, 2147483647])
```

See also: bitwise_and, bitwise_xor, logical_or

bitwise_xor()

```
>>> from numpy import *
>>> bitwise_xor(array([2,5,255]), array([4,4,4]))
array([ 6, 1, 251])
>>> bitwise_xor(array([2,5,255,2147483647L],dtype=int32),
array([4,4,4,2147483647L],dtype=int32))
array([ 6, 1, 251, 0])
```

See also: bitwise_and, bitwise_or, logical_xor

bmat()

```
>>> from numpy import *
>>> a = mat('1 2; 3 4')
>>> b = mat('5 6; 7 8')
>>> bmat('a b; b a') # all elements must be existing symbols
matrix([[1, 2, 5, 6],
```

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

See also: mat

broadcast()

```
>>> from numpy import *
>>> a = array([[1,2],[3,4]])
>>> b = array([5,6])
>>> c = broadcast(a,b)
>>> c.nd # the number of dimensions in the broadcasted result
2
>>> c.shape # the shape of the broadcasted result
(2, 2)
>>> c.size # total size of the broadcasted result
4
>>> for value in c: print value
...
(1, 5)
(2, 6)
(3, 5)
(4, 6)
>>> c.reset() # reset the iterator to the beginning
>>> c.next() # next element
(1, 5)
```

See also: ndenumerate, ndindex, flat

bytes()

```
>>> from numpy import *
>>> from numpy.random import bytes
>>> print repr(bytes(5)) # string of 5 random bytes
'o\x07\x9f\xdf\xdf'
>>> print repr(bytes(5)) # another string of 5 random bytes
'\x98\xc9KD\xe0'
```

See also: shuffle, permutation, seed

c_[]

```
>>> from numpy import *
>>> c_[1:5] # for single ranges, c_ works like r_
array([1, 2, 3, 4])
>>> c_[1:5,2:6] # for comma separated values, c_ stacks column-wise
array([[1, 2],
       [2, 3],
       [3, 4],
       [4, 5]])
>>> a = array([[1,2,3],[4,5,6]])
>>> c_[a,a] # concatenation along last (default) axis (column-wise,
that's why it's called c_)
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
>>> c_['0',a,a] # concatenation along 1st axis, equivalent to r_[a,a]
array([[1, 2, 3],
       [4, 5, 6],
```

```
        [1, 2, 3],
        [4, 5, 6]])
```

See also: r_, hstack, vstack, column_stack, concatenate, bmat, s_

cast[]()

```
>>> from numpy import *
>>> x = arange(3)
>>> x.dtype
dtype('int32')
>>> cast['int64'](x)
array([0, 1, 2], dtype=int64)
>>> cast['uint'](x)
array([0, 1, 2], dtype=uint32)
>>> cast[float128](x)
array([0.0, 1.0, 2.0], dtype=float128)
>>> cast.keys() # list dtype cast possibilities
<snip>
```

See also: astype, typeDict

ceil()

```
>>> from numpy import *
>>> a = array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7])
>>> ceil(a) # nearest integers greater-than or equal to a
array([-1., -1., -0., 1., 2., 2.])
```

See also: floor, round_, fix, astype

choose()

```
>>> from numpy import *
>>> choice0 =array([10,12,14,16]) # selector and choice arrays must be
equally sized
>>> choice1 =array([20,22,24,26])
>>> choice2 =array([30,32,34,36])
>>> selector = array([0,0,2,1]) # selector can only contain integers in
range(number_of_choice_arrays)
>>> selector.choose(choice0,choice1,choice2)
array([10, 12, 34, 26])
>>> a = arange(4)
>>> choose(a >= 2, (choice0, choice1)) # separate function also exists
array([10, 12, 24, 26])
```

See also: compress, take, where, select

clip()

```
>>> from numpy import *
>>> a = array([5,15,25,3,13])
>>> a.clip(min=10,max=20)
array([10, 15, 20, 10, 13])
>>> clip(a,10,20) # this syntax also exists
```

See also: where compress

column_stack()

```
>>> from numpy import *
>>> a = array([1,2])
>>> b = array([3,4])
>>> c = array([5,6])
>>> column_stack((a,b,c)) # a,b,c are 1-d arrays with equal length
array([[1, 3, 5],
       [2, 4, 6]])
```

See also: concatenate, dstack, hstack, vstack, c_

compress()

```
>>> from numpy import *
>>> a = array([10, 20, 30, 40])
>>> condition = (a > 15) & (a < 35)
>>> condition
array([False, True, True, False], dtype=bool)
>>> a.compress(condition)
array([20, 30])
>>> a[condition] # same effect
array([20, 30])
>>> compress(a >= 30, a) # this form also exists
array([30, 40])
>>> b = array([[10,20,30],[40,50,60]])
>>> b.compress(b.ravel() >= 22)
array([30, 40, 50, 60])
>>> x = array([3,1,2])
>>> y = array([50, 101])
>>> b.compress(x >= 2, axis=1) # illustrates the use of the axis keyword
array([[10, 30],
       [40, 60]])
>>> b.compress(y >= 100, axis=0)
array([[40, 50, 60]])
```

See also: choose, take, where, trim_zeros, unique, unique1d

concatenate()

```
>>> from numpy import *
>>> x = array([[1,2],[3,4]])
>>> y = array([[5,6],[7,8]])
>>> concatenate((x,y)) # default is axis=0
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
>>> concatenate((x,y),axis=1)
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

See also: append, column_stack, dstack, hstack, vstack, array_split

conj()

Synonym for conjugate()

See conjugate()

conjugate()

```
>>> a = array([1+2j,3-4j])
>>> a.conj() # .conj() and .conjugate() are the same
array([ 1.-2.j, 3.+4.j])
>>> a.conjugate()
array([ 1.-2.j, 3.+4.j])
>>> conj(a) # is also possible
>>> conjugate(a) # is also possible
```

See also: vdot

copy()

```
>>> from numpy import *
>>> a = array([1,2,3])
>>> a
array([1, 2, 3])
>>> b = a # b is a reference to a
>>> b[1] = 4
>>> a
array([1, 4, 3])
>>> a = array([1,2,3])
>>> b = a.copy() # b is now an independent copy of a
>>> b[1] = 4
>>> a
array([1, 2, 3])
>>> b
array([1, 4, 3])
```

See also: view

corrcoef()

```
>>> from numpy import *
>>> T = array([1.3, 4.5, 2.8, 3.9]) # temperature measurements
>>> P = array([2.7, 8.7, 4.7, 8.2]) # corresponding pressure
measurements
>>> print corrcoef([T,P]) # correlation matrix of temperature and
pressure
[[ 1. 0.98062258]
 [ 0.98062258 1. ]]
>>> rho = array([8.5, 5.2, 6.9, 6.5]) # corresponding density
measurements
>>> data = column_stack([T,P,rho])
>>> print corrcoef([T,P,rho]) # correlation matrix of T,P and rho
[[ 1. 0.98062258 -0.97090288]
 [ 0.98062258 1. -0.91538464]
 [-0.97090288 -0.91538464 1. ]]
```

See also: cov, var

cos()

```
>>> cos(array([0, pi/2, pi]))
```

```
array([ 1.00000000e+00,  6.12303177e-17, -1.00000000e+00])
```

cov()

```
>>> from numpy import *
>>> x = array([1., 3., 8., 9.])
>>> variance = cov(x) # normalized by N-1
>>> variance = cov(x, bias=1) # normalized by N
>>> T = array([1.3, 4.5, 2.8, 3.9]) # temperature measurements
>>> P = array([2.7, 8.7, 4.7, 8.2]) # corresponding pressure
measurements
>>> cov(T,P) # covariance between temperature and pressure
3.9541666666666657
>>> rho = array([8.5, 5.2, 6.9, 6.5]) # corresponding density
measurements
>>> data = column_stack([T,P,rho])
>>> print cov(data) # covariance matrix of T,P and rho
[[ 1.97583333  3.95416667 -1.85583333]
 [ 3.95416667  8.22916667 -3.57083333]
 [-1.85583333 -3.57083333  1.84916667]]
```

See also: corrcoef, std, var

cross()

```
>>> from numpy import *
>>> x = array([1,2,3])
>>> y = array([4,5,6])
>>> cross(x,y) # vector cross-product
array([-3,  6, -3])
```

See also: inner, ix_, outer

cumprod()

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

See also: accumulate, prod, cumsum

cumsum()

```
>>> from numpy import *
>>> a = array([1,2,3]) # cumulative sum = intermediate summing results &
total sum
>>> a.cumsum()
array([1, 3, 6])
>>> cumsum(a) # also exists
array([1, 3, 6])
>>> a = array([[1,2,3],[4,5,6]])
>>> a.cumsum(dtype=float) # specifies type of output value(s)
array([ 1., 3., 6., 10., 15., 21.])
>>> a.cumsum(axis=0) # sum over rows for each of the 3 columns
array([[1, 2, 3],
       [5, 7, 9]])
>>> a.cumsum(axis=1) # sum over columns for each of the 2 rows
array([[ 1, 3, 6],
       [ 4, 9, 15]])
```

See also: accumulate, sum, cumprod

delete()

```
>>> from numpy import *
>>> a = array([0, 10, 20, 30, 40])
>>> delete(a, [2,4]) # remove a[2] and a[4]
array([ 0, 10, 30])
>>> a = arange(16).reshape(4,4)
>>> a
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11],
       [12, 13, 14, 15]])
>>> delete(a, s_[1:3], axis=0) # remove rows 1 and 2
array([[ 0, 1, 2, 3],
       [12, 13, 14, 15]])
>>> delete(a, s_[1:3], axis=1) # remove columns 1 and 2
array([[ 0, 3],
       [ 4, 7],
       [ 8, 11],
       [12, 15]])
```

See also: append, insert

det()

```
>>> from numpy import *
>>> from numpy.linalg import det
>>> A = array([[1., 2.],[3., 4.]])
>>> det(A) # determinant of square matrix
-2.0
```

See also: inv

diag()

```
>>> from numpy import *
>>> a = arange(12).reshape(4,3)
>>> print a
[[ 0 1 2]
 [ 3 4 5]
```

```
  [ 6  7  8]
  [ 9 10 11]]
>>> print diag(a,k=0)
[0 4 8]
>>> print diag(a,k=1)
[1 5]
>>> print diag(array([1,4,5]),k=0)
[[1 0 0]
 [0 4 0]
 [0 0 5]]
>>> print diag(array([1,4,5]),k=1)
[[0 1 0 0]
 [0 0 4 0]
 [0 0 0 5]
 [0 0 0 0]]
```

See also: diagonal, diagflat, trace

diagflat()

```
>>> from numpy import *
>>> x = array([[5,6],[7,8]])
>>> diagflat(x) # flatten x, then put elements on diagonal
array([[5, 0, 0, 0],
       [0, 6, 0, 0],
       [0, 0, 7, 0],
       [0, 0, 0, 8]])
```

See also: diag, diagonal, flatten

diagonal()

```
>>> from numpy import *
>>> a = arange(12).reshape(3,4)
>>> print a
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> a.diagonal()
array([ 0,  5, 10])
>>> a.diagonal(offset=1)
array([ 1,  6, 11])
>>> diagonal(a) # Also this form exists
array([ 0,  5, 10])
```

See also: diag, diagflat, trace

diff()

```
>>> from numpy import *
>>> x = array([0,1,3,9,5,10])
>>> diff(x) # 1st-order differences between the elements of x
array([ 1,  2,  6, -4,  5])
>>> diff(x,n=2) # 2nd-order differences, equivalent to diff(diff(x))
array([ 1,  4, -10,  9])
>>> x = array([[1,3,6,10],[0,5,6,8]])
>>> diff(x) # 1st-order differences between the columns (default:
axis=-1)
array([[2, 3, 4],
       [5, 1, 2]])
```

```
>>> diff(x,axis=0) # 1st-order difference between the rows
array([[-1, 2, 0, -2]])
```

digitize()

```
>>> from numpy import *
>>> x = array([0.2, 6.4, 3.0, 1.6])
>>> bins = array([0.0, 1.0, 2.5, 4.0, 10.0]) # monotonically increasing
>>> d = digitize(x,bins) # in which bin falls each value of x?
>>> d
array([1, 4, 3, 2])
>>> for n in range(len(x)):
... print bins[d[n]-1], "<=", x[n], "<", bins[d[n]]
...
0.0 <= 0.2 < 1.0
4.0 <= 6.4 < 10.0
2.5 <= 3.0 < 4.0
1.0 <= 1.6 < 2.5
```

See also: bincount, histogram

dot()

```
>>> from numpy import *
>>> x = array([[1,2,3],[4,5,6]])
>>> x.shape
(2, 3)
>>> y = array([[1,2],[3,4],[5,6]])
>>> y.shape
(3, 2)
>>> dot(x,y) # matrix multiplication (2,3) x (3,2) -> (2,2)
array([[22, 28],
       [49, 64]])
>>>
>>> import numpy
>>> if id(dot) == id(numpy.core.multiarray.dot): # A way to know if you
use fast blas/lapack or not.
... print "Not using blas/lapack!"
```

See also: vdot, inner, multiply

dsplit()

```
>>> from numpy import *
>>> a = array([[1,2],[3,4]])
>>> b = dstack((a,a,a,a))
>>> b.shape # stacking in depth: for k in (0,..,3): b[:,:,k] = a
(2, 2, 4)
>>> c = dsplit(b,2) # split, depth-wise, in 2 equal parts
>>> print c[0].shape, c[1].shape # for k in (0,1): c[0][:,:,k] = a and
c[1][:,:,k] = a
(2, 2, 2) (2, 2, 2)
>>> d = dsplit(b,[1,2]) # split before [:,:,1] and before [:,:,2]
>>> print d[0].shape, d[1].shape, d[2].shape # for any of the parts:
d[.][:,:,k] = a
(2, 2, 1) (2, 2, 1) (2, 2, 2)
```

See also: split, array_split, hsplit, vsplit, dstack

dstack()

```
>>> from numpy import *
>>> a = array([[1,2],[3,4]]) # shapes of a and b can only differ in the
3rd dimension (if present)
>>> b = array([[5,6],[7,8]])
>>> dstack((a,b)) # stack arrays along a third axis (depth wise)
array([[[1, 5],
        [2, 6]],
       [[3, 7],
        [4, 8]]])
```

See also: column_stack, concatenate, hstack, vstack, dsplit

dtype()

```
>>> from numpy import *
>>> dtype('int16') # using array-scalar type
dtype('int16')
>>> dtype([('f1', 'int16')]) # record, 1 field named 'f1', containing
int16
dtype([('f1', '<i2')])
>>> dtype([('f1', [('f1', 'int16')])]) # record, 1 field named 'f1'
containing a record that has 1 field.
dtype([('f1', [('f1', '<i2')])])
>>> dtype([('f1', 'uint'), ('f2', 'int32')]) # record with 2 fields:
field 1 contains an unsigned int, 2nd field an int32
dtype([('f1', '<u4'), ('f2', '<i4')])
>>> dtype([('a','f8'),('b','S10')]) # using array-protocol type strings
dtype([('a', '<f8'), ('b', '|S10')])
>>> dtype("i4, (2,3)f8") # using comma-separated field formats. (2,3) is
the shape
dtype([('f0', '<i4'), ('f1', '<f8', (2, 3))])
>>> dtype([('hello',('int',3)),('world','void',10)]) # using tuples. int
is fixed-type: 3 is shape; void is flex-type: 10 is size.
dtype([('hello', '<i4', 3), ('world', '|V10')])
>>> dtype(('int16', {'x':('int8',0), 'y':('int8',1)})) # subdivide int16
in 2 int8, called x and y. 0 and 1 are the offsets in bytes
dtype(('<i2', [('x', '|i1'), ('y', '|i1')]))
>>> dtype({'names':['gender','age'], 'formats':['S1',uint8]}) # using
dictionaries. 2 fields named 'gender' and 'age'
dtype([('gender', '|S1'), ('age', '|u1')])
>>> dtype({'surname':('S25',0),'age':(uint8,25)}) # 0 and 25 are offsets
in bytes
dtype([('surname', '|S25'), ('age', '|u1')])
>>>
>>> a = dtype('int32')
>>> a
dtype('int32')
>>> a.type # type object
<type 'numpy.int32'>
>>> a.kind # character code (one of 'biufcSUV') to identify general type
'i'
>>> a.char # unique char code of each of the 21 built-in types
'l'
>>> a.num # unique number of each of the 21 built-in types
7
>>> a.str # array-protocol typestring
'<i4'
>>> a.name # name of this datatype
'int32'
>>> a.byteorder # '=':native, '<':little endian, '>':big endian, '|':not
applicable
'='
>>> a.itemsize # item size in bytes
```

```
4
>>> a = dtype({'surname':('S25',0),'age':(uint8,25)})
>>> a.fields.keys()
['age', 'surname']
>>> a.fields.values()
[(dtype('uint8'), 25), (dtype('|S25'), 0)]
>>> a = dtype([('x', 'f4'),('y', 'f4'), # nested field
... ('nested', [('i', 'i2'),('j','i2')])])
>>> a.fields['nested'] # access nested fields
(dtype([('i', '<i2'), ('j', '<i2')]), 8)
>>> a.fields['nested'][0].fields['i'] # access nested fields
(dtype('int16'), 0)
>>> a.fields['nested'][0].fields['i'][0].type
<type 'numpy.int16'>
```

See also: array, typeDict, astype

empty()

```
>>> from numpy import *
>>> empty(3) # uninitialized array, size=3, dtype = float
array([ 6.08581638e+000, 3.45845952e-323, 4.94065646e-324])
>>> empty((2,3),int) # uninitialized array, dtype = int
array([[1075337192, 1075337192, 135609024],
       [1084062604, 1197436517, 1129066306]])
```

See also: ones, zeros, eye, identity

empty_like()

```
>>> from numpy import *
>>> a = array([[1,2,3],[4,5,6]])
>>> empty_like(a) # uninitialized array with the same shape and datatype
as 'a'
array([[ 0, 25362433, 6571520],
       [ 21248, 136447968, 4]])
```

See also: ones_like, zeros_like

expand_dims()

```
>>> from numpy import *
>>> x = array([1,2])
>>> expand_dims(x,axis=0) # Equivalent to x[newaxis,:] or x[None] or
x[newaxis]
array([[1, 2]])
>>> expand_dims(x,axis=1) # Equivalent to x[:,newaxis]
array([[1],
       [2]])
```

See also: newaxis, atleast_1d, atleast_2d, atleast_3d

eye()

```
>>> from numpy import *
>>> eye(3,4,0,dtype=float) # a 3x4 matrix containing zeros except for
the 0th diagonal that contains ones
```

```
array([[ 1., 0., 0., 0.],
       [ 0., 1., 0., 0.],
       [ 0., 0., 1., 0.]])
>>> eye(3,4,1,dtype=float) # a 3x4 matrix containing zeros except for
the 1st diagonal that contains ones
array([[ 0., 1., 0., 0.],
       [ 0., 0., 1., 0.],
       [ 0., 0., 0., 1.]])
```

See also: ones, zeros, empty, identity

fft()

```
>>> from numpy import *
>>> from numpy.fft import *
>>> signal = array([-2., 8., -6., 4., 1., 0., 3., 5.]) # could also be
complex
>>> fourier = fft(signal)
>>> fourier
array([ 13. +0.j , 3.36396103 +4.05025253j,
         2. +1.j , -9.36396103-13.94974747j,
       -21. +0.j , -9.36396103+13.94974747j,
         2. -1.j , 3.36396103 -4.05025253j])
>>>
>>> N = len(signal)
>>> fourier = empty(N,complex)
>>> for k in range(N): # equivalent but much slower
... fourier[k] = sum(signal * exp(-1j*2*pi*k*arange(N)/N))
...
>>> timestep = 0.1 # if unit=day -> freq unit=cycles/day
>>> fftfreq(N, d=timestep) # freqs corresponding to 'fourier'
array([ 0. , 1.25, 2.5 , 3.75, -5. , -3.75, -2.5 , -1.25])
```

See also: ifft, fftfreq, fftshift

fftfreq()

```
>>> from numpy import *
>>> from numpy.fft import *
>>> signal = array([-2., 8., -6., 4., 1., 0., 3., 5.])
>>> fourier = fft(signal)
>>> N = len(signal)
>>> timestep = 0.1 # if unit=day -> freq unit=cycles/day
>>> freq = fftfreq(N, d=timestep) # freqs corresponding to 'fourier'
>>> freq
array([ 0. , 1.25, 2.5 , 3.75, -5. , -3.75, -2.5 , -1.25])
>>>
>>> fftshift(freq) # freqs in ascending order
array([-5. , -3.75, -2.5 , -1.25, 0. , 1.25, 2.5 , 3.75])
```

See also: fft, ifft, fftshift

fftshift()

```
>>> from numpy import *
>>> from numpy.fft import *
>>> signal = array([-2., 8., -6., 4., 1., 0., 3., 5.])
>>> fourier = fft(signal)
>>> N = len(signal)
>>> timestep = 0.1 # if unit=day -> freq unit=cycles/day
```

```
>>> freq = fftfreq(N, d=timestep) # freqs corresponding to 'fourier'
>>> freq
array([ 0. , 1.25, 2.5 , 3.75, -5. , -3.75, -2.5 , -1.25])
>>>
>>> freq = fftshift(freq) # freqs in ascending order
>>> freq
array([-5. , -3.75, -2.5 , -1.25, 0. , 1.25, 2.5 , 3.75])
>>> fourier = fftshift(fourier) # adjust fourier to new freq order
>>>
>>> freq = ifftshift(freq) # undo previous frequency shift
>>> fourier = ifftshift(fourier) # undo previous fourier shift
```

See also: fft, ifft, fftfreq

fill()

```
>>> from numpy import *
>>> a = arange(4, dtype=int)
>>> a
array([0, 1, 2, 3])
>>> a.fill(7) # replace all elements with the number 7
>>> a
array([7, 7, 7, 7])
>>> a.fill(6.5) # fill value is converted to dtype of a
>>> a
array([6, 6, 6, 6])
```

See also: empty, zeros, ones, repeat

finfo()

```
>>> from numpy import *
>>> f = finfo(float) # the numbers given are machine dependent
>>> f.nmant, f.nexp # nr of bits in the mantissa and in the exponent
(52, 11)
>>> f.machep # most negative n so that 1.0 + 2**n != 1.0
-52
>>> f.eps # floating point precision: 2**machep
array(2.2204460492503131e-16)
>>> f.precision # nr of precise decimal digits: int(-log10(eps))
15
>>> f.resolution # 10**(-precision)
array(1.0000000000000001e-15)
>>> f.negep # most negative n so that 1.0 - 2**n != 1.0
-53
>>> f.epsneg # floating point precision: 2**negep
array(1.1102230246251565e-16)
>>> f.minexp # most negative n so that 2**n gives normal numbers
-1022
>>> f.tiny # smallest usuable floating point nr: 2**minexp
array(2.2250738585072014e-308)
>>> f.maxexp # smallest positive n so that 2**n causes overflow
1024
>>> f.min, f.max # the most negative and most positive usuable floating
number
(-1.7976931348623157e+308, array(1.7976931348623157e+308))
```

fix()

```
>>> from numpy import *
>>> a = array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7])
>>> fix(a) # round a to nearest integer towards zero
array([-1., -1., 0., 0., 1., 1.])
```

See also: round_, ceil, floor, astype

flat

```
>>> from numpy import *
>>> a = array([[10,30],[40,60]])
>>> iter = a.flat # .flat returns an iterator
>>> iter.next() # cycle through array with .next()
10
>>> iter.next()
30
>>> iter.next()
40
```

See also: broadcast, flatten

flatten()

```
>>> from numpy import *
>>> a = array([[[1,2]],[[3,4]]])
>>> print a
[[[1 2]]
 [[3 4]]]
>>> b = a.flatten() # b is now a 1-d version of a, a new array, not a
reference
>>> print b
[1 2 3 4]
```

See also: ravel, flat

fliplr()

```
>>> from numpy import *
>>> a = arange(12).reshape(4,3)
>>> a
array([[ 0, 1, 2],
       [ 3, 4, 5],
       [ 6, 7, 8],
       [ 9, 10, 11]])
>>> fliplr(a) # flip left-right
array([[ 2, 1, 0],
       [ 5, 4, 3],
       [ 8, 7, 6],
       [11, 10, 9]])
```

See also: flipud, rot90

flipud()

```
>>> from numpy import *
>>> a = arange(12).reshape(4,3)
```

```
>>> a
array([[ 0, 1, 2],
       [ 3, 4, 5],
       [ 6, 7, 8],
       [ 9, 10, 11]])
>>> flipud(a) # flip up-down
array([[ 9, 10, 11],
       [ 6, 7, 8],
       [ 3, 4, 5],
       [ 0, 1, 2]])
```

See also: fliplr, rot90

floor()

```
>>> from numpy import *
>>> a = array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7])
>>> floor(a)
array([-2., -2., -1., 0., 1., 1.]) # nearest integer smaller-than or
equal to a # nearest integers greater-than or equal to a
```

See also: ceil, round_, fix, astype

fromarrays()

```
>>> from numpy import *
>>> x = array(['Smith','Johnson','McDonald']) # datatype is string
>>> y = array(['F','F','M'], dtype='S1') # datatype is a single
character
>>> z = array([20,25,23]) # datatype is integer
>>> data = rec.fromarrays([x,y,z], names='surname, gender, age') #
convert to record array
>>> data[0]
('Smith', 'F', 20)
>>> data.age # names are available as attributes
array([20, 25, 23])
```

See also: view

frombuffer()

```
>>> from numpy import *
>>> buffer = "\x00\x00\x00\x00\x00\x00\xf0?
\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x08\
...
@\x00\x00\x00\x00\x00\x00\x10@\x00\x00\x00\x00\x00\x00\x14@\x00\x00\x00\
x00\x00\x00\x18@"
>>> a = frombuffer(buffer, complex128)
>>> a
array([ 1.+2.j, 3.+4.j, 5.+6.j])
```

See also: fromfunction, fromfile

fromfile()

```
>>> from numpy import *
>>> y = array([2.,4.,6.,8.])
```

```
>>> y.tofile("myfile.dat") # binary format
>>> y.tofile("myfile.txt", sep='\n', format = "%e") # ascii format, one
column, exponential notation
>>> fromfile('myfile.dat', dtype=float)
array([ 2., 4., 6., 8.])
>>> fromfile('myfile.txt', dtype=float, sep='\n')
array([ 2., 4., 6., 8.])
```

See also: loadtxt, fromfunction, tofile, frombuffer, savetxt

fromfunction()

```
>>> from numpy import *
>>> def f(i,j):
... return i**2 + j**2
...
>>> fromfunction(f, (3,3)) # evaluate functiom for all combinations of
indices [0,1,2]x[0,1,2]
array([[0, 1, 4],
       [1, 2, 5],
       [4, 5, 8]])
```

See also: fromfile, frombuffer

fromiter()

```
>>> from numpy import *
>>> import itertools
>>> mydata = [[55.5, 40],[60.5, 70]] # List of lists
>>> mydescriptor = {'names': ('weight','age'), 'formats': (float32,
int32)} # Descriptor of the data
>>> myiterator = itertools.imap(tuple,mydata) # Clever way of putting
list of lists into iterator

# of tuples. E.g.: myiterator.next() == (55.5, 40.)
>>> a = fromiter(myiterator, dtype = mydescriptor)
>>> a
array([(55.5, 40), (60.5, 70)],
      dtype=[('weight', '<f4'), ('age', '<i4')])
```

See also: fromarrays, frombuffer, fromfile, fromfunction

generic

```
>>> from numpy import *
>>> numpyscalar = string_('7') # Convert to numpy scalar
>>> numpyscalar # Looks like a build-in scalar...
'7'
>>> type(numpyscalar) # ... but it isn't
<type 'numpy.string_'>
>>> buildinscalar = '7' # Build-in python scalar
>>> type(buildinscalar)
<type 'str'>
>>> isinstance(numpyscalar, generic) # Check if scalar is a NumPy one
True
>>> isinstance(buildinscalar, generic) # Example on how to recognize
NumPy scalars
False
```

gumbel()

```
>>> from numpy import *
>>> from numpy.random import *
>>> gumbel(loc=0.0,scale=1.0,size=(2,3)) # Gumbel distribution
location=0.0, scale=1.0
array([[-1.25923601, 1.68758144, 1.76620507],
       [ 1.96820048, -0.21219499, 1.83579566]])
>>> from pylab import * # histogram plot example
>>> hist(gumbel(0,1,(1000)), 50)
```

See also: random_sample, uniform, poisson, seed

histogram()

```
>>> from numpy import *
>>> x = array([0.2, 6.4, 3.0, 1.6, 0.9, 2.3, 1.6, 5.7, 8.5, 4.0, 12.8])
>>> bins = array([0.0, 1.0, 2.5, 4.0, 10.0]) # increasing monotonically
>>> N,bins = histogram(x,bins)
>>> N,bins
(array([2, 3, 1, 4]), array([ 0. , 1. , 2.5, 4. , 10. ]))
>>> for n in range(len(bins)-1):
... print "# ", N[n], "number fall into bin [", bins[n], ",", bins[n+1],
"["
...
# 2 numbers fall into bin [ 0.0 , 1.0 [
# 3 numbers fall into bin [ 1.0 , 2.5 [
# 1 numbers fall into bin [ 2.5 , 4.0 [
# 4 numbers fall into bin [ 4.0 , 10.0 [
#
>>> N,bins = histogram(x,5,range=(0.0, 10.0)) # 5 bin boundaries in the
range (0,10)
>>> N,bins
(array([4, 2, 2, 1, 2]), array([ 0., 2., 4., 6., 8.]))
>>> N,bins = histogram(x,5,range=(0.0, 10.0), normed=True) # normalize
histogram, i.e. divide by len(x)
>>> N,bins
(array([ 0.18181818, 0.09090909, 0.09090909, 0.04545455, 0.09090909]),
array([ 0., 2., 4., 6., 8.]))
```

See also: bincount, digitize

hsplit()

```
>>> from numpy import *
>>> a = array([[1,2,3,4],[5,6,7,8]])
>>> hsplit(a,2) # split, column-wise, in 2 equal parts
[array([[1, 2],
       [5, 6]]), array([[3, 4],
       [7, 8]])]
>>> hsplit(a,[1,2]) # split before column 1 and before column 2
[array([[1],
       [5]]), array([[2],
       [6]]), array([[3, 4],
       [7, 8]])]
```

See also: split, array_split, dsplit, vsplit, hstack

hstack()

```
>>> from numpy import *
>>> a =array([[1],[2]]) # 2x1 array
>>> b = array([[3,4],[5,6]]) # 2x2 array
>>> hstack((a,b,a)) # only the 2nd dimension of the arrays is allowed to
be different
array([[1, 3, 4, 1],
       [2, 5, 6, 2]])
```

See also: column_stack, concatenate, dstack, vstack, hsplit

hypot()

```
>>> from numpy import *
>>> hypot(3.,4.) # hypotenuse: sqrt(3**2 + 4**2) = 5
5.0
>>> z = array([2+3j, 3+4j])
>>> hypot(z.real, z.imag) # norm of complex numbers
array([ 3.60555128, 5. ])
```

See also: angle, abs

identity()

```
>>> from numpy import *
>>> identity(3,float)
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])
```

See also: empty, eye, ones, zeros

ifft()

```
>>> from numpy import *
>>> from numpy.fft import *
>>> signal = array([-2., 8., -6., 4., 1., 0., 3., 5.])
>>> fourier = fft(signal)
>>> ifft(fourier) # Inverse fourier transform
array([-2. +0.00000000e+00j, 8. +1.51410866e-15j, -6. +3.77475828e-15j,
        4. +2.06737026e-16j, 1. +0.00000000e+00j, 0. -1.92758271e-15j,
        3. -3.77475828e-15j, 5. +2.06737026e-16j])
>>>
>>> allclose(signal.astype(complex), ifft(fft(signal))) # ifft(fft()) =
original signal
True
>>>
>>> N = len(fourier)
>>> signal = empty(N,complex)
>>> for k in range(N): # equivalent but much slower
... signal[k] = sum(fourier * exp(+1j*2*pi*k*arange(N)/N)) / N
```

See also: fft, fftfreq, fftshift

imag

```
>>> from numpy import *
>>> a = array([1+2j,3+4j,5+6j])
>>> a.imag
array([ 2., 4., 6.])
>>> a.imag = 9
>>> a
array([ 1.+9.j, 3.+9.j, 5.+9.j])
>>> a.imag = array([9,8,7])
>>> a
array([ 1.+9.j, 3.+8.j, 5.+7.j])
```

See also: real, angle

index_exp[]

```
>>> from numpy import *
>>> myslice = index_exp[2:4,...,4,::-1] # myslice could now be passed to
a function, for example.
>>> print myslice
(slice(2, 4, None), Ellipsis, 4, slice(None, None, -1))
```

See also: slice, s_

indices()

```
>>> from numpy import *
>>> indices((2,3))
array([[[0, 0, 0],
        [1, 1, 1]],
       [[0, 1, 2],
        [0, 1, 2]]])
>>> a = array([ [ 0, 1, 2, 3, 4],
... [10,11,12,13,14],
... [20,21,22,23,24],
... [30,31,32,33,34] ])
>>> i,j = indices((2,3))
>>> a[i,j]
array([[ 0, 1, 2],
       [10, 11, 12]])
```

See also: mgrid, [], ix_, slice

inf

```
>>> from numpy import *
>>> exp(array([1000.])) # inf = infinite = number too large to
represent, machine dependent
array([ inf])
>>> x = array([2,-inf,1,inf])
>>> isfinite(x) # show which elements are not nan/inf/-inf
array([True, False, True, False], dtype=bool)
>>> isinf(x) # show which elements are inf/-inf
array([False, True, False, True], dtype=bool)
>>> isposinf(x) # show which elements are inf
array([False, False, False, True], dtype=bool)
>>> isneginf(x) # show which elements are -inf
array([False, True, False, False], dtype=bool)
>>> nan_to_num(x) # replace -inf/inf with most negative/positive
representable number
```

```
array([ 2.00000000e+000, -1.79769313e+308, 1.00000000e+000,
        1.79769313e+308])
```

See also: nan, finfo

inner()

```
>>> from numpy import *
>>> x = array([1,2,3])
>>> y = array([10,20,30])
>>> inner(x,y) # 1x10+2x20+3x30 = 140
140
```

See also: cross, outer, dot

insert()

```
>>> from numpy import *
>>> a = array([10,20,30,40])
>>> insert(a,[1,3],50) # insert value 50 before elements [1] and [3]
array([10, 50, 20, 30, 50, 40])
>>> insert(a,[1,3],[50,60]) # insert value 50 before element [1] and
value 60 before element [3]
array([10, 50, 20, 30, 60, 40])
>>> a = array([[10,20,30],[40,50,60],[70,80,90]])
>>> insert(a, [1,2], 100, axis=0) # insert row with values 100 before
row[1] and before row[2]
array([[ 10, 20, 30],
       [100, 100, 100],
       [ 40, 50, 60],
       [100, 100, 100],
       [ 70, 80, 90]])
>>> insert(a, [0,1], [[100],[200]], axis=0)
array([[100, 100, 100],
       [ 10, 20, 30],
       [200, 200, 200],
       [ 40, 50, 60],
       [ 70, 80, 90]])
>>> insert(a, [0,1], [100,200], axis=1)
array([[100, 10, 200, 20, 30],
       [100, 40, 200, 50, 60],
       [100, 70, 200, 80, 90]])
```

See also: delete, append

inv()

```
>>> from numpy import *
>>> from numpy.linalg import inv
>>> a = array([[3,1,5],[1,0,8],[2,1,4]])
>>> print a
[[3 1 5]
 [1 0 8]
 [2 1 4]]
>>> inva = inv(a) # Inverse matrix
>>> print inva
[[ 1.14285714 -0.14285714 -1.14285714]
 [-1.71428571 -0.28571429 2.71428571]
 [-0.14285714 0.14285714 0.14285714]]
>>> dot(a,inva) # Check the result, should be eye(3) within machine
```

```
precision
array([[ 1.00000000e-00, 2.77555756e-17, 3.60822483e-16],
       [ 0.00000000e+00, 1.00000000e+00, 0.00000000e+00],
       [ -1.11022302e-16, 0.00000000e+00, 1.00000000e+00]])
```

See also: solve, pinv, det

iscomplex()

```
>>> import numpy as np
>>> a = np.array([1,2,3.j])
>>> np.iscomplex(a)
array([False, False, True], dtype=bool)
```

iscomplexobj()

```
>>> import numpy as np
>>> a = np.array([1,2,3.j])
>>> np.iscomplexobj(a)
True
>>> a = np.array([1,2,3])
>>> np.iscomplexobj(a)
False
>>> a = np.array([1,2,3], dtype=np.complex)
>>> np.iscomplexobj(a)
True
```

item()

```
>>> from numpy import *
>>> a = array([5])
>>> type(a[0])
<type 'numpy.int32'>
>>> a.item() # Conversion of array of size 1 to Python scalar
5
>>> type(a.item())
<type 'int'>
>>> b = array([2,3,4])
>>> b[1].item() # Conversion of 2nd element to Python scalar
3
>>> type(b[1].item())
<type 'int'>
>>> b.item(2) # Return 3rd element converted to Python scalar
4
>>> type(b.item(2))
<type 'int'>
>>> type(b[2]) # b[2] is slower than b.item(2), and there is no
conversion
<type 'numpy.int32'>
```

See also: []

ix_()

```
>>> from numpy import *
>>> a = arange(9).reshape(3,3)
>>> print a
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
>>> indices = ix_([0,1,2],[1,2,0]) # trick to be used with array
broadcasting
>>> print indices
(array([[0],
       [1],
       [2]]), array([[1, 2, 0]]))
>>> print a[indices]
[[1 2 0]
 [4 5 3]
 [7 8 6]]
>>> # The latter array is the cross-product:
>>> # [[ a[0,1] a[0,2] a[0,0]]
... # [ a[1,1] a[1,2] a[1,0]]
... # [ a[2,1] a[2,2] a[2,0]]]
...
```

See also: [], indices, cross, outer

lexsort()

```
>>> from numpy import *
>>> serialnr = array([1023, 5202, 6230, 1671, 1682, 5241])
>>> height = array([40., 42., 60., 60., 98., 40.])
>>> width = array([50., 20., 70., 60., 15., 30.])
>>>
>>> # We want to sort the serial numbers with increasing height, _AND_
>>> # serial numbers with equal heights should be sorted with increasing
width.
>>>
>>> indices = lexsort(keys = (width, height)) # mind the order!
>>> indices
array([5, 0, 1, 3, 2, 4])
>>> for n in indices:
... print serialnr[n], height[n], width[n]
...
5241 40.0 30.0
1023 40.0 50.0
5202 42.0 20.0
1671 60.0 60.0
6230 60.0 70.0
1682 98.0 15.0
>>>
>>> a = vstack([serialnr,width,height]) # Alternatively: all data in one
big matrix
>>> print a # Mind the order of the rows!
[[ 1023. 5202. 6230. 1671. 1682. 5241.]
 [ 50. 20. 70. 60. 15. 30.]
 [ 40. 42. 60. 60. 98. 40.]]
>>> indices = lexsort(a) # Sort on last row, then on 2nd last row, etc.
>>> a.take(indices, axis=-1)
array([[ 5241., 1023., 5202., 1671., 6230., 1682.],
       [ 30., 50., 20., 60., 70., 15.],
       [ 40., 40., 42., 60., 60., 98.]])
```

See also: sort, argsort

linspace()

```
>>> from numpy import *
>>> linspace(0,5,num=6) # 6 evenly spaced numbers between 0 and 5 incl.
```

```
array([ 0., 1., 2., 3., 4., 5.])
>>> linspace(0,5,num=10) # 10 evenly spaced numbers between 0 and 5
incl.
array([ 0. , 0.55555556, 1.11111111, 1.66666667, 2.22222222,
        2.77777778, 3.33333333, 3.88888889, 4.44444444, 5. ])
>>> linspace(0,5,num=10,endpoint=False) # 10 evenly spaced numbers
between 0 and 5 EXCL.
array([ 0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
>>> stepsize = linspace(0,5,num=10,endpoint=False,retstep=True) #
besides the usual array, also return the step size
>>> stepsize
(array([ 0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5]), 0.5)
>>> myarray, stepsize = linspace(0,5,num=10,endpoint=False,retstep=True)
>>> stepsize
0.5
```

See also: arange, logspace, r_

loadtxt()

```
>>> from numpy import *
>>>
>>> data = loadtxt("myfile.txt") # myfile.txt contains 4 columns of
numbers
>>> t,z = data[:,0], data[:,3] # data is 2D numpy array
>>>
>>> t,x,y,z = loadtxt("myfile.txt", unpack=True) # to unpack all columns
>>> t,z = loadtxt("myfile.txt", usecols = (0,3), unpack=True) # to
select just a few columns
>>> data = loadtxt("myfile.txt", skiprows = 7) # to skip 7 rows from top
of file
>>> data = loadtxt("myfile.txt", comments = '!') # use '!' as comment
char instead of '#'
>>> data = loadtxt("myfile.txt", delimiter=';') # use ';' as column
separator instead of whitespace
>>> data = loadtxt("myfile.txt", dtype = int) # file contains integers
instead of floats
```

See also: savetxt, fromfile

logical_and()

```
>>> from numpy import *
>>> logical_and(array([0,0,1,1]), array([0,1,0,1]))
array([False, False, False, True], dtype=bool)
>>> logical_and(array([False,False,True,True]),
array([False,True,False,True]))
array([False, False, False, True], dtype=bool)
```

See also: logical_or, logical_not, logical_xor, bitwise_and

logical_not()

```
>>> from numpy import *
>>> logical_not(array([0,1]))
>>> logical_not(array([False,True]))
```

See also: logical_or, logical_not, logical_xor, bitwise_and

logical_or()

```
>>> from numpy import *
>>> logical_or(array([0,0,1,1]), array([0,1,0,1]))
>>> logical_or(array([False,False,True,True]),
array([False,True,False,True]))
```

See also: logical_and, logical_not, logical_xor, bitwise_or

logical_xor()

```
>>> from numpy import *
>>> logical_xor(array([0,0,1,1]), array([0,1,0,1]))
>>> logical_xor(array([False,False,True,True]),
array([False,True,False,True]))
```

See also: logical_or, logical_not, logical_or, bitwise_xor

logspace()

```
>>> from numpy import *
>>> logspace(-2, 3, num = 6) # 6 evenly spaced pts on a logarithmic
scale, from 10^{-2} to 10^3 incl.
array([ 1.00000000e-02, 1.00000000e-01, 1.00000000e+00,
        1.00000000e+01, 1.00000000e+02, 1.00000000e+03])
>>> logspace(-2, 3, num = 10) # 10 evenly spaced pts on a logarithmic
scale, from 10^{-2} to 10^3 incl.
array([ 1.00000000e-02, 3.59381366e-02, 1.29154967e-01,
        4.64158883e-01, 1.66810054e+00, 5.99484250e+00,
        2.15443469e+01, 7.74263683e+01, 2.78255940e+02,
        1.00000000e+03])
>>> logspace(-2, 3, num = 6, endpoint=False) # 6 evenly spaced pts on a
logarithmic scale, from 10^{-2} to 10^3 EXCL.
array([ 1.00000000e-02, 6.81292069e-02, 4.64158883e-01,
        3.16227766e+00, 2.15443469e+01, 1.46779927e+02])
>>> exp(linspace(log(0.01), log(1000), num=6, endpoint=False)) # for
comparison
array([ 1.00000000e-02, 6.81292069e-02, 4.64158883e-01,
        3.16227766e+00, 2.15443469e+01, 1.46779927e+02])
```

See also: arange, linspace, r_

lstsq()

lstsq() is most often used in the context of least-squares fitting of data. Suppose you obtain some noisy data y as a function of a variable t, e.g. velocity as a function of time. You can use lstsq() to fit a model to the data, if the model is linear in its parameters, that is if

$y = p_0 * f_0(t) + p_1 * f_1(t) + ... + p_{N-1} * f_{N-1}(t) + \text{noise}$

where the p_i are the parameters you want to obtain through fitting and the $f_i(t)$ are *known* functions of t. What follows is an example how you can do this.

First, for the example's sake, some data is simulated:

```
>>> from numpy import *
>>> from numpy.random import normal
>>> t = arange(0.0, 10.0, 0.05) # independent variable
>>> y = 2.0 * sin(2.*pi*t*0.6) + 2.7 * cos(2.*pi*t*0.6) + normal(0.0,
1.0, len(t))
```

We would like to fit this data with: model(t) = p_0 * sin(2.*pi*t*0.6) + p_1 * cos(2.*pi*t*0.6), where p_0 and p_1 are the unknown fit parameters. Here we go:

```
>>> from numpy.linalg import lstsq
>>> Nparam = 2 # we want to estimate 2 parameters: p_0 and p_1
>>> A = zeros((len(t),Nparam), float) # one big array with all the
f_i(t)
>>> A[:,0] = sin(2.*pi*t*0.6) # f_0(t) stored
>>> A[:,1] = cos(2.*pi*t*0.6) # f_1(t) stored
>>> (p, residuals, rank, s) = lstsq(A,y)
>>> p # our final estimate of the parameters using noisy data
array([ 1.9315685 , 2.71165171])
>>> residuals # sum of the residuals: sum((p[0] * A[:,0] + p[1] * A[:,1]
- y)**2)
array([ 217.23783374])
>>> rank # rank of the array A
2
>>> s # singular values of A
array([ 10., 10.])
```

See also: pinv, polyfit, solve

mat()

```
>>> from numpy import *
>>> mat('1 3 4; 5 6 9') # matrices are always 2-dimensional
matrix([[1, 3, 4],
        [5, 6, 9]])
>>> a = array([[1,2],[3,4]])
>>> m = mat(a) # convert 2-d array to matrix
>>> m
matrix([[1, 2],
        [3, 4]])
>>> a[0] # result is 1-dimensional
array([1, 2])
>>> m[0] # result is 2-dimensional
matrix([[1, 2]])
>>> a.ravel() # result is 1-dimensional
array([1, 2, 3, 4])
>>> m.ravel() # result is 2-dimensional
matrix([[1, 2, 3, 4]])
>>> a*a # element-by-element multiplication
array([[ 1, 4],
        [ 9, 16]])
>>> m*m # (algebraic) matrix multiplication
matrix([[ 7, 10],
        [15, 22]])
>>> a**3 # element-wise power
array([[ 1, 8],
        [27, 64]])
>>> m**3 # matrix multiplication m*m*m
matrix([[ 37, 54],
        [ 81, 118]])
>>> m.T # transpose of the matrix
matrix([[1, 3],
        [2, 4]])
```

```
>>> m.H # conjugate transpose (differs from .T for complex matrices)
matrix([[1, 3],
        [2, 4]])
>>> m.I # inverse matrix
matrix([[-2. , 1. ],
        [ 1.5, -0.5]])
```

See also: bmat, array, dot, asmatrix

matrix()

```
>>> from numpy import *
>>> matrix('1 3 4; 5 6 9') # matrix is synonymous with mat
matrix([[1, 3, 4],
        [5, 6, 9]])
```

See also: mat, asmatrix

max()

```
>>> from numpy import *
>>> a = array([10,20,30])
>>> a.max()
30
>>> a = array([[10,50,30],[60,20,40]])
>>> a.max()
60
>>> a.max(axis=0) # for each of the columns, find the maximum
array([60, 50, 40])
>>> a.max(axis=1) # for each of the rows, find the maximum
array([50, 60])
>>> max(a) # also exists, but is slower
```

See also: nan, argmax, maximum, ptp

maximum()

```
>>> from numpy import *
>>> a = array([1,0,5])
>>> b = array([3,2,4])
>>> maximum(a,b) # element-by-element comparison
array([3, 2, 5])
>>> max(a.tolist(),b.tolist()) # standard Python function does not give
the same!
[3, 2, 4]
```

See also: minimum, max, argmax

mean()

```
>>> from numpy import *
>>> a = array([1,2,7])
>>> a.mean()
3.3333333333333335
>>> a = array([[1,2,7],[4,9,6]])
>>> a.mean()
4.833333333333333
```

```
>>> a.mean(axis=0) # the mean of each of the 3 columns
array([ 2.5, 5.5, 6.5])
>>> a.mean(axis=1) # the mean of each of the 2 rows
array([ 3.33333333, 6.33333333])
```

See also: average, median, var, std, sum

median()

```
>>> from numpy import *
>>> a = array([1,2,3,4,9])
>>> median(a)
3
>>> a = array([1,2,3,4,9,0])
>>> median(a)
2.5
```

See also: average, mean, var, std

mgrid[]

```
>>> from numpy import *
>>> m = mgrid[1:3,2:5] # rectangular mesh grid with x-values [1,2] and
y-values [2,3,4]
>>> print m
[[[1 1 1]
  [2 2 2]]
 [[2 3 4]
  [2 3 4]]]
>>> m[0,1,2] # x-value of grid point with index coordinates (1,2)
2
>>> m[1,1,2] # y-value of grid point with index coordinates (1,2)
4
```

See also: indices, ogrid

min()

```
>>> from numpy import *
>>> a = array([10,20,30])
>>> a.min()
10
>>> a = array([[10,50,30],[60,20,40]])
>>> a.min()
10
>>> a.min(axis=0) # for each of the columns, find the minimum
array([10, 20, 30])
>>> a.min(axis=1) # for each of the rows, find the minimum
array([10, 20])
>>> min(a) # also exists, but is slower
```

See also: nan, max, minimum, argmin, ptp

minimum()

```
>>> from numpy import *
>>> a = array([1,0,5])
```

```
>>> b = array([3,2,4])
>>> minimum(a,b) # element-by-element comparison
array([1, 0, 4])
>>> min(a.tolist(),b.tolist()) # Standard Python function does not give
the same!
[1, 0, 5]
```

See also: min, maximum, argmin

multiply()

```
>>> from numpy import *
>>> multiply(array([3,6]), array([4,7]))
array([12, 42])
```

See also: dot

nan

```
>>> from numpy import *
>>> sqrt(array([-1.0]))
array([ nan]) # nan = NaN = Not A Number
>>> x = array([2, nan, 1])
>>> isnan(x) # show which elements are nan
array([False, True, False], dtype=bool)
>>> isfinite(x) # show which elements are not nan/inf/-inf
array([True, False, True], dtype=bool)
>>> nansum(x) # same as sum() but ignore nan elements
3.0
>>> nanmax(x) # same as max() but ignore nan elements
2.0
>>> nanmin(x) # same as min() but ignore nan elements
1.0
>>> nanargmin(x) # same as argmin() but ignore nan elements
2
>>> nanargmax(x) # same as argmax() but ignore nan elements
0
>>> nan_to_num(x) # replace all nan elements with 0.0
array([ 2., 0., 1.])
```

See also: inf

ndenumerate()

```
>>> from numpy import *
>>> a = arange(9).reshape(3,3) + 10
>>> a
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
>>> b = ndenumerate(a)
>>> for position,value in b: print position,value # position is the N-
dimensional index
...
(0, 0) 10
(0, 1) 11
(0, 2) 12
(1, 0) 13
(1, 1) 14
(1, 2) 15
```

```
(2, 0) 16
(2, 1) 17
(2, 2) 18
```

See also: broadcast, ndindex

ndim

```
>>> from numpy import *
>>> a = arange(12).reshape(3,4)
>>> a
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])
>>> a.ndim # a has 2 axes
2
>>> a.shape = (2,2,3)
array([[[ 0, 1, 2],
        [ 3, 4, 5]],
       [[ 6, 7, 8],
        [ 9, 10, 11]]])
>>> a.ndim # now a has 3 axes
3
>>> len(a.shape) # same as ndim
3
```

See also: shape

ndindex()

```
>>> for index in ndindex(4,3,2):
        print index
(0,0,0)
(0,0,1)
(0,1,0)
...
(3,1,1)
(3,2,0)
(3,2,1)
```

See also: broadcast, ndenumerate

newaxis

```
>>> from numpy import *
>>> x = arange(3)
>>> x
array([0, 1, 2])
>>> x[:,newaxis] # add a new dimension/axis
array([[0],
       [1],
       [2]])
>>> x[:,newaxis,newaxis] # add two new dimensions/axes
array([[[0]],
       [[1]],
       [[2]]])
>>> x[:,newaxis] * x
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4]])
```

```
>>> y = arange(3,6)
>>> x[:,newaxis] * y # outer product, same as outer(x,y)
array([[ 0,  0,  0],
       [ 3,  4,  5],
       [ 6,  8, 10]])
>>> x.shape
(3,)
>>> x[newaxis,:].shape # x[newaxis,:] is equivalent to x[newaxis] and
x[None]
(1,3)
>>> x[:,newaxis].shape
(3,1)
```

See also: [], atleast_1d, atleast_2d, atleast_3d, expand_dims

nonzero()

```
>>> from numpy import *
>>> x = array([1,0,2,-1,0,0,8])
>>> indices = x.nonzero() # find the indices of the nonzero elements
>>> indices
(array([0, 2, 3, 6]),)
>>> x[indices]
array([1, 2, -1, 8])
>>> y = array([[0,1,0],[2,0,3]])
>>> indices = y.nonzero()
>>> indices
(array([0, 1, 1]), array([1, 0, 2]))
>>> y[indices[0],indices[1]] # one way of doing it, explains what's in
indices[0] and indices[1]
array([1, 2, 3])
>>> y[indices] # this way is shorter
array([1, 2, 3])
>>> y = array([1,3,5,7])
>>> indices = (y >= 5).nonzero()
>>> y[indices]
array([5, 7])
>>> nonzero(y) # function also exists
(array([0, 1, 2, 3]),)
```

See also: [], where, compress, choose, take

ogrid()

```
>>> from numpy import *
>>> x,y = ogrid[0:3,0:3] # x and y are useful to use with broadcasting
rules
>>> x
array([[0],
       [1],
       [2]])
>>> y
array([[0, 1, 2]])
>>> print x*y # example how to use broadcasting rules
[[0 0 0]
 [0 1 2]
 [0 2 4]]
```

See also: mgrid

ones()

```
>>> from numpy import *
>>> ones(5)
array([ 1., 1., 1., 1., 1.])
>>> ones((2,3), int)
array([[1, 1, 1],
       [1, 1, 1]])
```

See also: ones_like, zeros, empty, eye, identity

ones_like()

```
>>> from numpy import *
>>> a = array([[1,2,3],[4,5,6]])
>>> ones_like(a) # ones initialised array with the same shape and
datatype as 'a'
array([[1, 1, 1],
       [1, 1, 1]])
```

See also: ones, zeros_like

outer()

```
>>> from numpy import *
>>> x = array([1,2,3])
>>> y = array([10,20,30])
>>> outer(x,y) # outer product
array([[10, 20, 30],
       [20, 40, 60],
       [30, 60, 90]])
```

See also: inner, cross

permutation()

```
>>> from numpy import *
>>> from numpy.random import permutation
>>> permutation(4) # permutation of integers from 0 to 3
array([0, 3, 1, 2])
>>> permutation(4) # another permutation of integers from 0 to 3
array([2, 1, 0, 3])
>>> permutation(4) # yet another permutation of integers from 0 to 3
array([3, 0, 2, 1])
```

See also: shuffle, bytes, seed

piecewise()

```
>>> from numpy import *
>>> f1 = lambda x: x*x
>>> f2 = lambda x: 2*x
>>> x = arange(-2.,3.,0.1)
>>> condition = (x>1)&(x<2) # boolean array
>>> y = piecewise(x,condition, [f1,1.]) # if condition is true, return
f1, otherwise 1.
>>> y = piecewise(x, fabs(x)<=1, [f1,0]) + piecewise(x, x>1, [f2,0]) #
0. in ]-inf,-1[, f1 in [-1,+1], f2 in ]+1,+inf[
```

```
>>> print y
<snip>
```

See also: select

pinv()

```
>>> from numpy import *
>>> from numpy.linalg import pinv,svd,lstsq
>>> A = array([[1., 3., 5.],[2., 4., 6.]])
>>> b = array([1., 3.])
>>>
>>> # Question: find x such that ||A*x-b|| is minimal
>>> # Answer: x = pinvA * b, with pinvA the pseudo-inverse of A
>>>
>>> pinvA = pinv(A)
>>> print pinvA
[[-1.33333333 1.08333333]
 [-0.33333333 0.33333333]
 [ 0.66666667 -0.41666667]]
>>> x = dot(pinvA, b)
>>> print x
[ 1.91666667 0.66666667 -0.58333333]
>>>
>>> # Relation with least-squares minimisation lstsq()
>>>
>>> x,resids,rank,s = lstsq(A,b)
>>> print x # the same solution for x as above
[ 1.91666667 0.66666667 -0.58333333]
>>>
>>> # Relation with singular-value decomposition svd()
>>>
>>> U,sigma,V = svd(A)
>>> S = zeros_like(A.transpose())
>>> for n in range(len(sigma)): S[n,n] = 1. / sigma[n]
>>> dot(V.transpose(), dot(S, U.transpose())) # = pinv(A)
array([[-1.33333333, 1.08333333],
       [-0.33333333, 0.33333333],
       [ 0.66666667, -0.41666667]])
```

See also: inv, lstsq, solve, svd

poisson()

```
>>> from numpy import *
>>> from numpy.random import *
>>> poisson(lam=0.5, size=(2,3)) # poisson distribution lambda=0.5
array([[2, 0, 0],
       [1, 1, 0]])
```

See also: random_sample, uniform, standard_normal, seed

poly1d()

```
>>> from numpy import *
>>> p1 = poly1d([2,3],r=1) # specify polynomial by its roots
>>> print p1
   2
1 x - 5 x + 6
>>> p2 = poly1d([2,3],r=0) # specify polynomial by its coefficients
```

```
>>> print p2
2 x + 3
>>> print p1+p2 # +,-,*,/ and even ** are supported
   2
1 x - 3 x + 9
>>> quotient,remainder = p1/p2 # division gives a tupple with the
quotient and remainder
>>> print quotient,remainder
0.5 x - 3
15
>>> p3 = p1*p2
>>> print p3
   3 2
2 x - 7 x - 3 x + 18
>>> p3([1,2,3,4]) # evaluate the polynomial in the values [1,2,3,4]
array([10,  0,  0, 22])
>>> p3[2] # the coefficient of x**2
-7
>>> p3.r # the roots of the polynomial
array([-1.5,  3. ,  2. ])
>>> p3.c # the coefficients of the polynomial
array([ 2, -7, -3, 18])
>>> p3.o # the order of the polynomial
3
>>> print p3.deriv(m=2) # the 2nd derivative of the polynomial
12 x - 14
>>> print p3.integ(m=2,k=[1,2]) # integrate polynomial twice and use
[1,2] as integration constants
     5 4 3 2
0.1 x - 0.5833 x - 0.5 x + 9 x + 1 x + 2
```

polyfit()

```
>>> from numpy import *
>>> x = array([1,2,3,4,5])
>>> y = array([6, 11, 18, 27, 38])
>>> polyfit(x,y,2) # fit a 2nd degree polynomial to the data, result is
x**2 + 2x + 3
array([ 1.,  2.,  3.])
>>> polyfit(x,y,1) # fit a 1st degree polynomial (straight line), result
is 8x-4
array([ 8., -4.])
```

See also: lstsq

prod()

```
>>> from numpy import *
>>> a = array([1,2,3])
>>> a.prod() # 1 * 2 * 3 = 6
6
>>> prod(a) # also exists
6
>>> a = array([[1,2,3],[4,5,6]])
>>> a.prod(dtype=float) # specify type of output
720.0
>>> a.prod(axis=0) # for each of the 3 columns: product
array([ 4, 10, 18])
>>> a.prod(axis=1) # for each of the two rows: product
array([  6, 120])
```

See also: cumprod, sum

ptp()

```
>>> from numpy import *
>>> a = array([5,15,25])
>>> a.ptp() # peak-to-peak = maximum - minimum
20
>>> a = array([[5,15,25],[3,13,33]])
>>> a.ptp()
30
>>> a.ptp(axis=0) # peak-to-peak value for each of the 3 columns
array([2, 2, 8])
>>> a.ptp(axis=1) # peak-to-peak value for each of the 2 rows
array([20, 30])
```

See also: max, min

put()

```
>>> from nump import *
>>> a = array([10,20,30,40])
>>> a.put([60,70,80], [0,3,2]) # first values, then indices
>>> a
array([60, 20, 80, 70])
>>> a[[0,3,2]] = [60,70,80] # same effect
>>> a.put([40,50], [0,3,2,1]) # if value array is too short, it is
repeated
>>> a
array([40, 50, 40, 50])
>>> put(a, [0,3], [90]) # also exists, but here FIRST indices, THEN
values
>>> a
array([90, 50, 40, 90])
```

See also: putmask, take

putmask()

```
>>> from nump import *
>>> a = array([10,20,30,40])
>>> mask = array([True,False,True,True]) # size mask = size a
>>> a.putmask([60,70,80,90], mask) # first values, then the mask
>>> a
array([60, 20, 80, 90])
>>> a = array([10,20,30,40])
>>> a[mask] # reference
array([60, 80, 90])
>>> a[mask] = array([60,70,80,90]) # NOT exactly the same as putmask
>>> a
array([60, 20, 70, 80])
>>> a.putmask([10,90], mask) # if value array is too short, it is
repeated
>>> a
array([10, 20, 10, 90])
>>> putmask(a, mask, [60,70,80,90]) # also exists, but here FIRST mask,
THEN values
```

See also: put, take

r_[]

```
>>> from numpy import *
>>> r_[1:5] # same as arange(1,5)
array([1, 2, 3, 4])
>>> r_[1:10:4] # same as arange(1,10,4)
array([1, 5, 9])
>>> r_[1:10:4j] # same as linspace(1,10,4), 4 equally-spaced elements
between 1 and 10 inclusive
array([ 1., 4., 7., 10.])
>>> r_[1:5,7,1:10:4] # sequences separated with commas are concatenated
array([1, 2, 3, 4, 7, 1, 5, 9])
>>> r_['r', 1:3] # return a matrix. If 1-d, result is a 1xN matrix
matrix([[1, 2]])
>>> r_['c',1:3] # return a matrix. If 1-d, result is a Nx1 matrix
matrix([[1],
        [2]])
>>> a = array([[1,2,3],[4,5,6]])
>>> r_[a,a] # concatenation along 1st (default) axis (row-wise, that's
why it's called r_)
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
>>> r_['-1',a,a] # concatenation along last axis, same as c_[a,a]
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

See also: c_, s_, arange, linspace, hstack, vstack, column_stack, concatenate, bmat

rand()

```
>>> from numpy import *
>>> from numpy.random import *
>>> rand(3,2)
array([[ 0.65159297, 0.78872335],
       [ 0.09385959, 0.02834748],
       [ 0.8357651 , 0.43276707]])
```

See also: random_sample, seed

randint()

Synonym for random_integers()

See random_integers

randn()

```
>>> randn(2,3)
array([[ 1.22497074, -0.29508896, -0.75040033],
       [-0.54822685, -0.98032155, -1.40467696]])
```

See also: standard_normal, poisson, seed

random_integers()

```
>>> from numpy import *
>>> from numpy.random import *
>>> random_integers(-1,5,(2,2))
```

```
array([[ 3, -1],
       [-1,  0]])
```

See also: random_sample, uniform, poisson, seed

random_sample()

```
>>> from numpy import *
>>> from numpy.random import *
>>> random_sample((3,2))
array([[ 0.76228008,  0.00210605],
       [ 0.44538719,  0.72154003],
       [ 0.22876222,  0.9452707 ]])
```

See also: ranf, sample, rand, seed

ranf()

Synonym for random_sample

See random_sample, sample

ravel()

```
>>> from numpy import *
>>> a = array([[1,2],[3,4]])
>>> a.ravel() # 1-d version of a
array([1, 2, 3, 4])
>>> b = a[:,0].ravel() # a[:,0] does not occupy a single memory segment,
thus b is a copy, not a reference
>>> b
array([1, 3])
>>> c = a[0,:].ravel() # a[0,:] occupies a single memory segment, thus c
is a reference, not a copy
>>> c
array([1, 2])
>>> b[0] = -1
>>> c[1] = -2
>>> a
array([[ 1, -2],
       [ 3,  4]])
>>> ravel(a) # also exists
```

See also: flatten

real

```
>>> from numpy import *
>>> a = array([1+2j,3+4j,5+6j])
>>> a.real
array([ 1.,  3.,  5.])
>>> a.real = 9
>>> a
array([ 9.+2.j,  9.+4.j,  9.+6.j])
>>> a.real = array([9,8,7])
>>> a
array([ 9.+2.j,  8.+4.j,  7.+6.j])
```

See also: imag, angle

recarray()

```
>>> from numpy import *
>>> num = 2
>>> a = recarray(num, formats='i4,f8,f8',names='id,x,y')
>>> a['id'] = [3,4]
>>> a['id']
array([3, 4])
>>> a = rec.fromrecords([(35,1.2,7.3),(85,9.3,3.2)], names='id,x,y') #
fromrecords is in the numpy.rec submodule
>>> a['id']
array([35, 85])
```

See also: array, dtype

reduce()

```
>>> from numpy import *
>>> add.reduce(array([1.,2.,3.,4.])) # computes ((((1.)+2.)+3.)+4.)
10.0
>>> multiply.reduce(array([1.,2.,3.,4.])) # works also with other
operands. Computes ((((1.)*2.)*3.)*4.)
24.0
>>> add.reduce(array([[1,2,3],[4,5,6]]), axis = 0) # reduce every column
separately
array([5, 7, 9])
>>> add.reduce(array([[1,2,3],[4,5,6]]), axis = 1) # reduce every row
separately
array([ 6, 15])
```

See also: accumulate, sum, prod

repeat()

```
>>> from numpy import *
>>> repeat(7., 4)
array([ 7., 7., 7., 7.])
>>> a = array([10,20])
>>> a.repeat([3,2])
array([10, 10, 10, 20, 20])
>>> repeat(a, [3,2]) # also exists
>>> a = array([[10,20],[30,40]])
>>> a.repeat([3,2,1,1])
array([10, 10, 10, 20, 20, 30, 40])
>>> a.repeat([3,2],axis=0)
array([[10, 20],
       [10, 20],
       [10, 20],
       [30, 40],
       [30, 40]])
>>> a.repeat([3,2],axis=1)
array([[10, 10, 10, 20, 20],
       [30, 30, 30, 40, 40]])
```

See also: tile

reshape()

```
>>> from numpy import *
>>> x = arange(12)
>>> x.reshape(3,4) # array with 3 rows and 4 columns. 3x4=12. Total
number of elements is always the same.
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])
>>> x.reshape(3,2,2) # 3x2x2 array; 3x2x2 = 12. x itself does _not_
change.
array([[[ 0, 1],
        [ 2, 3]],
       [[ 4, 5],
        [ 6, 7]],
       [[ 8, 9],
        [10, 11]]])
>>> x.reshape(2,-1) # 'missing' -1 value n is calculated so that 2xn=12,
so n=6
array([[ 0, 1, 2, 3, 4, 5],
       [ 6, 7, 8, 9, 10, 11]])
>>> x.reshape(12) # reshape(1,12) is not the same as reshape(12)
array([0,1,2,3,4,5,6,7,8,9,10,11])
>>> reshape(x,(2,6)) # Separate function reshape() also exists
```

See also: shape, resize

resize()

```
>>> from numpy import *
>>> a = array([1,2,3,4])
>>> a.resize(2,2) # changes shape of 'a' itself
>>> print a
[[1 2]
 [3 4]]
>>> a.resize(3,2) # reallocates memoy of 'a' to change nr of elements,
fills excess elements with 0
>>> print a
[[1 2]
 [3 4]
 [0 0]]
>>> a.resize(2,4)
>>> print a
[[1 2 3 4]
 [0 0 0 0]]
>>> a.resize(2,1) # throws away elements of 'a' to fit new shape
>>> print a
[[1]
 [2]]
```

But, there is a caveat:

```
>>> b = array([1,2,3,4])
>>> c = b # c is reference to b, it doesn't 'own' its data
>>> c.resize(2,2) # no problem, nr of elements doesn't change
>>> c.resize(2,3) # doesn't work, c is only a reference
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: cannot resize an array that has been referenced or is
referencing
another array in this way. Use the resize function
>>> b.resize(2,3) # doesn't work, b is referenced by another array
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: cannot resize an array that has been referenced or is
```

```
referencing
another array in this way. Use the resize function
```

and it's not always obvious what the reference is:

```
>>> d = arange(4)
>>> d
array([0, 1, 2, 3])
>>> d.resize(5) # doesn't work, but where's the reference?
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: cannot resize an array that has been referenced or is
referencing
another array in this way. Use the resize function
>>> _ # '_' was a reference to d!
array([0, 1, 2, 3])
>>> d = resize(d, 5) # this does work, however
>>> d
array([0, 1, 2, 3, 0])
```

See also: reshape

rollaxis()

```
>>> from numpy import *
>>> a = arange(3*4*5).reshape(3,4,5)
>>> a.shape
(3, 4, 5)
>>> b = rollaxis(a,1,0) # transpose array so that axis 1 is 'rolled'
before axis 0
>>> b.shape
(4, 3, 5)
>>> b = rollaxis(a,0,2) # transpose array so that axis 0 is 'rolled'
before axis 2
>>> b.shape
(4, 3, 5)
```

See also: swapaxes, transpose

round()

round(decimals=0, out=None) -> reference to rounded values.

```
>>> from numpy import *
>>> array([1.2345, -1.647]).round() # rounds the items. Type remains
float64.
array([ 1., -2.])
>>> array([1, -1]).round() # integer arrays stay as they are
array([ 1, -1])
>>> array([1.2345, -1.647]).round(decimals=1) # round to 1 decimal place
array([ 1.2, -1.6])
>>> array([1.2345+2.34j, -1.647-0.238j]).round() # both real and complex
parts are rounded
array([ 1.+2.j, -2.-0.j])
>>> array([0.0, 0.5, 1.0, 1.5, 2.0, 2.5]).round() # numpy rounds x.5 to
nearest even.
array([ 0., 0., 1., 2., 2., 2.])
>>> a = zeros(3, dtype=int)
>>> array([1.2345, -1.647, 3.141]).round(out=a) # different output
arrays may be specified
array([ 1, -2, 3])
>>> a # and the output is cast to the new type
```

```
array([ 1, -2, 3])
>>> round_(array([1.2345, -1.647])) # round_ is the functional form. ->
a copy.
array([ 1., -2.])
>>> around(array([1.2345, -1.647])) # around is an alias of round_.
array([ 1., -2.])
```

See also: ceil, floor, fix, astype

rot90()

```
>>> from numpy import *
>>> a = arange(12).reshape(4,3)
>>> a
array([[ 0, 1, 2],
       [ 3, 4, 5],
       [ 6, 7, 8],
       [ 9, 10, 11]])
>>> rot90(a) # 'rotate' the matrix 90 degrees
array([[ 2, 5, 8, 11],
       [ 1, 4, 7, 10],
       [ 0, 3, 6, 9]])
```

See also: fliplr, flipud

s_[]

```
>>> from numpy import *
>>> s_[1:5] # easy slice generating. See r_[] examples.
slice(1, 5, None)
>>> s_[1:10:4]
slice(1, 10, 4)
>>> s_[1:10:4j]
slice(1, 10, 4j)
>>> s_['r',1:3] # to return a matrix. If 1-d, result is a 1xN matrix
('r', slice(1, 3, None))
>>> s_['c',1:3] # to return a matrix. If 1-d, result is a Nx1 matrix
('c', slice(1, 3, None))
```

See also: r_, c_, slice, index_exp

sample()

Synonym for random_sample

See also: random_sample, ranf

savetxt()

```
>>> from numpy import *
>>> savetxt("myfile.txt", data) # data is 2D array
>>> savetxt("myfile.txt", x) # x is 1D array. 1 column in file.
>>> savetxt("myfile.txt", (x,y)) # x,y are 1D arrays. 2 rows in file.
>>> savetxt("myfile.txt", transpose((x,y))) # x,y are 1D arrays. 2
columns in file.
>>> savetxt("myfile.txt", transpose((x,y)), fmt='%6.3f') # use new
format instead of '%.18e'
```

```
>>> savetxt("myfile.txt", data, delimiter = ';') # use ';' to separate
columns instead of space
```

See also: loadtxt, tofile

searchsorted()

searchsorted(keys, side="left")

```
>>> from numpy import *
>>> a = array([1,2,2,3]) # a is 1-D and in ascending order.
>>> a.searchsorted(2) # side defaults to "left"
1 # a[1] is the first element in a >= 2
>>> a.searchsorted(2, side='right') # look for the other end of the run
of twos
3 # a[3] is the first element in a > 2
>>> a.searchsorted(4) # 4 is greater than any element in a
4 # the returned index is 1 past the end of a.
>>> a.searchsorted([[1,2],[2,3]]) # whoa, fancy keys
array([[0, 1], # the returned array has the same shape as the keys
       [1, 3]])
>>> searchsorted(a,2) # there is a functional form
1
```

See also: sort, histogram

seed()

```
>>> seed([1]) # seed the pseudo-random number generator
>>> rand(3)
array([ 0.13436424, 0.84743374, 0.76377462])
>>> seed([1])
>>> rand(3)
array([ 0.13436424, 0.84743374, 0.76377462])
>>> rand(3)
array([ 0.25506903, 0.49543509, 0.44949106])
```

select()

```
>>> from numpy import *
>>> x = array([5., -2., 1., 0., 4., -1., 3., 10.])
>>> select([x < 0, x == 0, x <= 5], [x-0.1, 0.0, x+0.2], default = 100.)
array([ 5.2, -2.1, 1.2, 0. , 4.2, -1.1, 3.2, 100. ])
>>>
>>> # This is how it works:
>>>
>>> result = zeros_like(x)
>>> for n in range(len(x)):
... if x[n] < 0: result[n] = x[n]-0.1 # The order of the conditions
matters. The first one that
... elif x[n] == 0: result[n] = 0.0 # matches, will be 'selected'.
... elif x[n] <= 5: result[n] = x[n]+0.2
... else: result[n] = 100. # The default is used when none of the
conditions match
...
>>> result
array([ 5.2, -2.1, 1.2, 0. , 4.2, -1.1, 3.2, 100. ])
```

See also: choose, piecewise

set_printoptions()

```
>>> from numpy import *
>>> x = array([pi, 1.e-200])
>>> x
array([ 3.14159265e+000, 1.00000000e-200])
>>> set_printoptions(precision=3, suppress=True) # 3 digits behind
decimal point + suppress small values
>>> x
array([ 3.142, 0. ])
>>>
>>> help(set_printoptions) # see help() for keywords
'threshold','edgeitems' and 'linewidth'
```

shape

```
>>> from numpy import *
>>> x = arange(12)
>>> x.shape
(12,)
>>> x.shape = (3,4) # array with 3 rows and 4 columns. 3x4=12. Total
number of elements is always the same.
>>> x
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])
>>> x.shape = (3,2,2) # 3x2x2 array; 3x2x2 = 12. x itself _does_ change,
unlike reshape().
>>> x
array([[[ 0, 1],
        [ 2, 3]],
       [[ 4, 5],
        [ 6, 7]],
       [[ 8, 9],
        [10, 11]]])
>>> x.shape = (2,-1) # 'missing' -1 value n is calculated so that
2xn=12, so n=6
>>> x
array([[ 0, 1, 2, 3, 4, 5],
       [ 6, 7, 8, 9, 10, 11]])
>>> x.shape = 12 # x.shape = (1,12) is not the same as x.shape = 12
>>> x
array([0,1,2,3,4,5,6,7,8,9,10,11])
```

See also: reshape

shuffle()

```
>>> from numpy import *
>>> from numpy.random import shuffle
>>> x = array([1,50,-1,3])
>>> shuffle(x) # shuffle the elements of x
>>> print x
[-1 3 50 1]
>>> x = ['a','b','c','z']
>>> shuffle(x) # works with any sequence
>>> print x
['a', 'c', 'z', 'b']
```

See also: permutation, bytes

slice()

```
>>> s = slice(3,9,2) # slice objects exist outside numpy
>>> from numpy import *
>>> a = arange(20)
>>> a[s]
array([3, 5, 7])
>>> a[3:9:2] # same thing
array([3, 5, 7])
```

See also: [], ..., newaxis, s_, ix_, indices, index_exp

solve()

```
>>> from numpy import *
>>> from numpy.linalg import solve
>>>
>>> # The system of equations we want to solve for (x0,x1,x2):
>>> # 3 * x0 + 1 * x1 + 5 * x2 = 6
>>> # 1 * x0 + 8 * x2 = 7
>>> # 2 * x0 + 1 * x1 + 4 * x2 = 8
>>>
>>> a = array([[3,1,5],[1,0,8],[2,1,4]])
>>> b = array([6,7,8])
>>> x = solve(a,b)
>>> print x # This is our solution
[-3.28571429 9.42857143 1.28571429]
>>>
>>> dot(a,x) # Just checking if we indeed obtain the righthand side
array([ 6., 7., 8.])
```

See also: inv

sometrue()

```
>>> from numpy import *
>>> b = array([True, False, True, True])
>>> sometrue(b)
True
>>> a = array([1, 5, 2, 7])
>>> sometrue(a >= 5)
True
```

See also: alltrue, all, any

sort()

sort(axis=-1, kind="quicksort")

```
>>> from numpy import *
>>> a = array([2,0,8,4,1])
>>> a.sort() # in-place sorting with quicksort (default)
>>> a
array([0, 1, 2, 4, 8])
>>> a.sort(kind='mergesort') # algorithm options are 'quicksort',
'mergesort' and 'heapsort'
>>> a = array([[8,4,1],[2,0,9]])
>>> a.sort(axis=0)
```

```
>>> a
array([[2, 0, 1],
       [8, 4, 9]])
>>> a = array([[8,4,1],[2,0,9]])
>>> a.sort(axis=1) # default axis = -1
>>> a
array([[1, 4, 8],
       [0, 2, 9]])
>>> sort(a) # there is a functional form
```

See also: argsort, lexsort

split()

```
>>> from numpy import *
>>> a = array([[1,2,3,4],[5,6,7,8]])
>>> split(a,2,axis=0) # split a in 2 parts. row-wise
array([[1, 2, 3, 4]]), array([[5, 6, 7, 8]])]
>>> split(a,4,axis=1) # split a in 4 parts, column-wise
[array([[1],
       [5]]), array([[2],
       [6]]), array([[3],
       [7]]), array([[4],
       [8]])]
>>> split(a,3,axis=1) # impossible to split in 3 equal parts -> error
(SEE: array_split)
Traceback (most recent call last):
<snip>
ValueError: array split does not result in an equal division
>>> split(a,[2,3],axis=1) # make a split before the 2nd and the 3rd
column
[array([[1, 2],
       [5, 6]]), array([[3],
       [7]]), array([[4],
       [8]])]
```

See also: dsplit, hsplit, vsplit, array_split, concatenate

squeeze()

```
>>> from numpy import *
>>> a = arange(6)
>>> a = a.reshape(1,2,1,1,3,1)
>>> a
array([[[[[[0],
           [1],
           [2]]]],
        [[[[3],
           [4],
           [5]]]]]])
>>> a.squeeze() # result has shape 2x3, all dimensions with length 1 are
removed
array([[0, 1, 2],
       [3, 4, 5]])
>>> squeeze(a) # also exists
```

std()

```
>>> from numpy import *
>>> a = array([1.,2,7])
```

```
>>> a.std() # normalized by N (not N-1)
2.6246692913372702
>>> a = array([[1.,2,7],[4,9,6]])
>>> a.std()
2.793842435706702
>>> a.std(axis=0) # standard deviation of each of the 3 columns
array([ 1.5, 3.5, 0.5])
>>> a.std(axis=1) # standard deviation of each of the 2 columns
array([ 2.62466929, 2.05480467])
```

See also: mean, var, cov

standard_normal()

```
>>> standard_normal((2,3))
array([[ 1.12557608, -0.13464922, -0.35682992],
       [-1.54090277, 1.21551589, -1.82854551]])
```

See also: randn, uniform, poisson, seed

sum()

```
>>> from numpy import *
>>> a = array([1,2,3])
>>> a.sum()
6
>>> sum(a) # also exists
>>> a = array([[1,2,3],[4,5,6]])
>>> a.sum()
21
>>> a.sum(dtype=float) # specify type of output
21.0
>>> a.sum(axis=0) # sum over rows for each of the 3 columns
array([5, 7, 9])
>>> a.sum(axis=1) # sum over columns for each of the 2 rows
array([ 6, 15])
```

See also: accumulate, nan, cumsum, prod

svd()

```
>>> from numpy import *
>>> from numpy.linalg import svd
>>> A = array([[1., 3., 5.],[2., 4., 6.]]) # A is a (2x3) matrix
>>> U,sigma,V = svd(A)
>>> print U # U is a (2x2) unitary matrix
[[-0.61962948 -0.78489445]
 [-0.78489445 0.61962948]]
>>> print sigma # non-zero diagonal elements of Sigma
[ 9.52551809 0.51430058]
>>> print V # V is a (3x3) unitary matrix
[[-0.2298477 -0.52474482 -0.81964194]
 [ 0.88346102 0.24078249 -0.40189603]
 [ 0.40824829 -0.81649658 0.40824829]]
>>> Sigma = zeros_like(A) # constructing Sigma from sigma
>>> n = min(A.shape)
>>> Sigma[:n,:n] = diag(sigma)
>>> print dot(U,dot(Sigma,V)) # A = U * Sigma * V
```

```
[[ 1. 3. 5.]
 [ 2. 4. 6.]]
```

See also: pinv

swapaxes()

```
>>> from numpy import *
>>> a = arange(30)
>>> a = a.reshape(2,3,5)
>>> a
array([[[ 0, 1, 2, 3, 4],
        [ 5, 6, 7, 8, 9],
        [10, 11, 12, 13, 14]],
       [[15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]]])
>>> b = a.swapaxes(1,2) # swap the 2nd and the 3rd axis
>>> b
array([[[ 0, 5, 10],
        [ 1, 6, 11],
        [ 2, 7, 12],
        [ 3, 8, 13],
        [ 4, 9, 14]],
       [[15, 20, 25],
        [16, 21, 26],
        [17, 22, 27],
        [18, 23, 28],
        [19, 24, 29]]])
>>> b.shape
(2, 5, 3)
>>> b[0,0,0] = -1 # be aware that b is a reference, not a copy
>>> print a[0,0,0]
```

See also: transpose, rollaxis

T

```
>>> from numpy import *
>>> x = array([[1.,2.],[3.,4.]])
>>> x
array([[ 1., 2.],
       [ 3., 4.]])
>>> x.T # shortcut for transpose()
array([[ 1., 3.],
       [ 2., 4.]])
```

See also: transpose

take()

```
>>> from numpy import *
>>> a= array([10,20,30,40])
>>> a.take([0,0,3]) # [0,0,3] is a set of indices
array([10, 10, 40])
>>> a[[0,0,3]] # the same effect
array([10, 10, 40])
>>> a.take([[0,1],[0,1]]) # shape of return array depends on shape of
indices array
array([[10, 20],
```

```
       [10, 20]])
>>> a = array([[10,20,30],[40,50,60]])
>>> a.take([0,2],axis=1)
array([[10, 30],
       [40, 60]])
>>> take(a,[0,2],axis=1) # also exists
```

See also: [], put, putmask, compress, choose

tensordot()

```
>>> from numpy import *
>>> a = arange(60.).reshape(3,4,5)
>>> b = arange(24.).reshape(4,3,2)
>>> c = tensordot(a,b, axes=([1,0],[0,1])) # sum over the 1st and 2nd
dimensions
>>> c.shape
(5,2)
>>> # A slower but equivalent way of computing the same:
>>> c = zeros((5,2))
>>> for i in range(5):
... for j in range(2):
... for k in range(3):
... for n in range(4):
... c[i,j] += a[k,n,i] * b[n,k,j]
...
```

See also: dot

tile()

```
>>> from numpy import *
>>> a = array([10,20])
>>> tile(a, (3,2)) # concatenate 3x2 copies of a together
array([[10, 20, 10, 20],
       [10, 20, 10, 20],
       [10, 20, 10, 20]])
>>> tile(42.0, (3,2)) # works for scalars, too
array([[ 42., 42.],
       [ 42., 42.],
       [ 42., 42.]])
>>> tile([[1,2],[4,8]], (2,3)) # works for 2-d arrays and list literals,
too
array([[1, 2, 1, 2, 1, 2],
       [4, 8, 4, 8, 4, 8],
       [1, 2, 1, 2, 1, 2],
       [4, 8, 4, 8, 4, 8]])
```

See also: hstack, vstack, r_, c_, concatenate, repeat

tofile()

```
>>> from numpy import *
>>> x = arange(10.)
>>> y = x**2
>>> y.tofile("myfile.dat") # binary format
>>> y.tofile("myfile.txt", sep=' ', format = "%e") # ascii format, one
row, exp notation, values separated by 1 space
```

```
>>> y.tofile("myfile.txt", sep='\n', format = "%e") # ascii format, one
column, exponential notation
```

See also: fromfile, loadtxt, savetxt

tolist()

```
>>> from numpy import *
>>> a = array([[1,2],[3,4]])
>>> a.tolist() # convert to a standard python list
[[1, 2], [3, 4]]
```

trace()

```
>>> from numpy import *
>>> a = arange(12).reshape(3,4)
>>> a
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])
>>> a.diagonal()
array([ 0, 5, 10])
>>> a.trace()
15
>>> a.diagonal(offset=1)
array([ 1, 6, 11])
>>> a.trace(offset=1)
18
```

See also: diag, diagonal

transpose()

A very simple example:

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

From this, a more elaborate example can be understood:

```
>>> a = arange(30)
>>> a = a.reshape(2,3,5)
>>> a
array([[[ 0, 1, 2, 3, 4],
        [ 5, 6, 7, 8, 9],
        [10, 11, 12, 13, 14]],
       [[15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]]])
>>> b = a.transpose()
```

```
>>> b
array([[[ 0, 15],
        [ 5, 20],
        [10, 25]],
       [[ 1, 16],
        [ 6, 21],
        [11, 26]],
       [[ 2, 17],
        [ 7, 22],
        [12, 27]],
       [[ 3, 18],
        [ 8, 23],
        [13, 28]],
       [[ 4, 19],
        [ 9, 24],
        [14, 29]]])
>>> b.shape
(5, 3, 2)
>>> b = a.transpose(1,0,2) # First axis 1, then axis 0, then axis 2
>>> b
array([[[ 0, 1, 2, 3, 4],
        [15, 16, 17, 18, 19]],
       [[ 5, 6, 7, 8, 9],
        [20, 21, 22, 23, 24]],
       [[10, 11, 12, 13, 14],
        [25, 26, 27, 28, 29]]])
>>> b.shape
(3, 2, 5)
>>> b = transpose(a, (1,0,2)) # A separate transpose() function also
exists
```

See also: T, swapaxes, rollaxis

tri()

```
>>> from numpy import *
>>> tri(3,4,k=0,dtype=float) # 3x4 matrix of Floats, triangular, the
k=0-th diagonal and below is 1, the upper part is 0
array([[ 1., 0., 0., 0.],
       [ 1., 1., 0., 0.],
       [ 1., 1., 1., 0.]])
>>> tri(3,4,k=1,dtype=int)
array([[1, 1, 0, 0],
       [1, 1, 1, 0],
       [1, 1, 1, 1]])
```

See also: tril, triu

tril()

```
>>> from numpy import *
>>> a = arange(10,100,10).reshape(3,3)
>>> a
array([[10, 20, 30],
       [40, 50, 60],
       [70, 80, 90]])
>>> tril(a,k=0)
array([[10, 0, 0],
       [40, 50, 0],
       [70, 80, 90]])
>>> tril(a,k=1)
array([[10, 20, 0],
```

```
        [40, 50, 60],
        [70, 80, 90]])
```

See also: tri, triu

trim_zeros()

```
>>> from numpy import *
>>> x = array([0, 0, 0, 1, 2, 3, 0, 0])
>>> trim_zeros(x,'f') # remove zeros at the front
array([1, 2, 3, 0, 0])
>>> trim_zeros(x,'b') # remove zeros at the back
array([0, 0, 0, 1, 2, 3])
>>> trim_zeros(x,'bf') # remove zeros at the back and the front
array([1, 2, 3])
```

See also: compress

triu()

```
>>> from numpy import *
>>> a = arange(10,100,10).reshape(3,3)
>>> a
array([[10, 20, 30],
       [40, 50, 60],
       [70, 80, 90]])
>>> triu(a,k=0)
array([[10, 20, 30],
       [ 0, 50, 60],
       [ 0, 0, 90]])
>>> triu(a,k=1)
array([[ 0, 20, 30],
       [ 0, 0, 60],
       [ 0, 0, 0]])
```

See also: tri, tril

typeDict()

```
>>> from numpy import *
>>> typeDict['short']
<type 'numpy.int16'>
>>> typeDict['uint16']
<type 'numpy.uint16'>
>>> typeDict['void']
<type 'numpy.void'>
>>> typeDict['S']
<type 'numpy.string_'>
```

See also: dtype, cast

uniform()

```
>>> from numpy import *
>>> from numpy.random import *
>>> uniform(low=0,high=10,size=(2,3)) # uniform numbers in range [0,10)
```

```
array([[ 6.66689951, 4.50623001, 4.69973967],
       [ 6.52977732, 3.24688284, 5.01917021]])
```

See also: standard_normal, poisson, seed

unique()

```
>>> from numpy import *
>>> x = array([2,3,2,1,0,3,4,0])
>>> unique(x) # remove double values
array([0, 1, 2, 3, 4])
```

See also: compress, unique1d

unique1d()

```
>>> np.unique1d([1, 1, 2, 2, 3, 3])
array([1, 2, 3])
>>> a = np.array([[1, 1], [2, 3]])
>>> np.unique1d(a)
array([1, 2, 3])
>>> np.unique1d([1,2,6,4,2,3,2], return_index=True)
(array([1, 2, 3, 4, 6]), array([0, 1, 5, 3, 2]))
>>> x = [1,2,6,4,2,3,2]
>>> u, i = np.unique1d(x, return_inverse=True)
>>> u
array([1, 2, 3, 4, 6])
>>> i
array([0, 1, 4, 3, 1, 2, 1])
>>> [u[p] for p in i]
[1, 2, 6, 4, 2, 3, 2]
```

See also: compress, unique

vander()

```
>>> from numpy import *
>>> x = array([1,2,3,5])
>>> N=3
>>> vander(x,N) # Vandermonde matrix of the vector x
array([[ 1, 1, 1],
       [ 4, 2, 1],
       [ 9, 3, 1],
       [25, 5, 1]])
>>> column_stack([x**(N-1-i) for i in range(N)]) # to understand what a
Vandermonde matrix contains
array([[ 1, 1, 1],
       [ 4, 2, 1],
       [ 9, 3, 1],
       [25, 5, 1]])
```

var()

```
>>> from numpy import *
>>> a = array([1,2,7])
>>> a.var() # normalised with N (not N-1)
6.8888888888888875
```

```
>>> a = array([[1,2,7],[4,9,6]])
>>> a.var()
7.8055555555555571
>>> a.var(axis=0) # the variance of each of the 3 columns
array([ 2.25, 12.25, 0.25])
>>> a.var(axis=1) # the variance of each of the 2 rows
array([ 6.88888889, 4.22222222])
```

See also: cov, std, mean

vdot()

```
>>> from numpy import *
>>> x = array([1+2j,3+4j])
>>> y = array([5+6j,7+8j])
>>> vdot(x,y) # conj(x) * y = (1-2j)*(5+6j)+(3-4j)*(7+8j)
(70-8j)
```

See also: dot, inner, cross, outer

vectorize()

```
>>> from numpy import *
>>> def myfunc(x):
... if x >= 0: return x**2
... else: return -x
...
>>> myfunc(2.) # works fine
4.0
>>> myfunc(array([-2,2])) # doesn't work, try it...
<snip>
>>> vecfunc = vectorize(myfunc, otypes=[float]) # declare the return
type as float
>>> vecfunc(array([-2,2])) # works fine!
array([ 2., 4.])
```

See also: apply_along_axis, apply_over_axes

view()

```
>>> from numpy import *
>>> a = array([1., 2.])
>>> a.view() # new array referring to the same data as 'a'
array([ 1., 2.])
>>> a.view(complex) # pretend that a is made up of complex numbers
array([ 1.+2.j])
>>> a.view(int) # view(type) is NOT the same as astype(type)!
array([ 0, 1072693248, 0, 1073741824])
>>>
>>> mydescr = dtype({'names': ['gender','age'], 'formats': ['S1',
'i2']})
>>> a = array([('M',25),('F',30)], dtype = mydescr) # array with records
>>> b = a.view(recarray) # convert to a record array, names are now
attributes
>>> >>> a['age'] # works with 'a' but not with 'b'
array([25, 30], dtype=int16)
>>> b.age # works with 'b' but not with 'a'
array([25, 30], dtype=int16)
```

See also: copy

vonmises()

```
>>> from numpy import *
>>> from numpy.random import *
>>> vonmises(mu=1,kappa=1,size=(2,3)) # Von Mises distribution mean=1.0,
kappa=1
array([[ 0.81960554, 1.37470839, -0.15700173],
       [ 1.2974554 , 2.94229797, 0.32462307]])
>>> from pylab import * # histogram plot example
>>> hist(vonmises(1,1,(10000)), 50)
```

See also: random_sample, uniform, standard_normal, seed

vsplit()

```
>>> from numpy import *
>>> a = array([[1,2],[3,4],[5,6],[7,8]])
>>> vsplit(a,2) # split, row-wise, in 2 equal parts
[array([[1, 2],
       [3, 4]]), array([[5, 6],
       [7, 8]])]
>>> vsplit(a,[1,2]) # split, row-wise, before row 1 and before row 2
[array([[1, 2]]), array([[3, 4]]), array([[5, 6],
       [7, 8]])]
```

See also: split, array_split, dsplit, hsplit, vstack

vstack()

```
>>> from numpy import *
>>> a =array([1,2])
>>> b = array([[3,4],[5,6]])
>>> vstack((a,b,a)) # only the first dimension of the arrays is allowed
to be different
array([[1, 2],
       [3, 4],
       [5, 6],
       [1, 2]])
```

See also: hstack, column_stack, concatenate, dstack, vsplit

weibull()

```
>>> from numpy import *
>>> from numpy.random import *
>>> weibull(a=1,size=(2,3)) # I think a is the shape parameter
array([[ 0.08303065, 3.41486412, 0.67430149],
       [ 0.41383893, 0.93577601, 0.45431195]])
>>> from pylab import * # histogram plot example
>>> hist(weibull(5, (1000)), 50)
```

See also: random_sample, uniform, standard_normal, seed

where()

```
>>> from numpy import *
>>> a = array([3,5,7,9])
>>> b = array([10,20,30,40])
>>> c = array([2,4,6,8])
>>> where(a <= 6, b, c)
array([10, 20, 6, 8])
>>> where(a <= 6, b, -1)
array([10, 20, -1, -1])
>>> indices = where(a <= 6) # returns a tuple; the array contains
indices.
>>> indices
(array([0, 1]),)
>>> b[indices]
array([10, 20])
>>> b[a <= 6] # an alternative syntax
array([10, 20])
>>> d = array([[3,5,7,9],[2,4,6,8]])
>>> where(d <= 6) # tuple with first all the row indices, then all the
column indices
(array([0, 0, 1, 1, 1]), array([0, 1, 0, 1, 2]))
```

Be aware of the difference between x[list of bools] and x[list of integers]!

```
>>> from numpy import *
>>> x = arange(5,0,-1)
>>> print x
[5 4 3 2 1]
>>> criterion = (x <= 2) | (x >= 5)
>>> criterion
array([True, False, False, True, True], dtype=bool)
>>> indices = where(criterion, 1, 0)
>>> print indices
[1 0 0 1 1]
>>> x[indices] # integers!
array([4, 5, 5, 4, 4])
>>> x[criterion] # bools!
array([5, 2, 1])
>>> indices = where(criterion)
>>> print indices
(array([0, 3, 4]),)
>>> x[indices]
array([5, 2, 1])
```

See also: [], nonzero, clip

zeros()

```
>>> from numpy import *
>>> zeros(5)
array([ 0., 0., 0., 0., 0.])
>>> zeros((2,3), int)
array([[0, 0, 0],
       [0, 0, 0]])
```

See also: zeros_like, ones, empty, eye, identity

zeros_like()

```
>>> from numpy import *
>>> a = array([[1,2,3],[4,5,6]])
```

```
>>> zeros_like(a) # with zeros initialised array with the same shape and
datatype as 'a'
array([[0, 0, 0],
       [0, 0, 0]])
```

See also: ones_like, zeros

CategoryCategory