Numpy_Example_list

11-7-23

1. ...

In [1]:
```
#1. ...
import numpy as np
a = np.arange(13)
a
```

Out[1]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

In [2]:
```
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]]]
```

In [3]:
```
b = np.arange(15)
```

In [4]:
```
a[...,0]
```

Out[4]:
```
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
```

In [5]:
```
a[:,:,0]
```

Out[5]:
```
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
```

In [6]:
```
a
```

Out[6]:
```
array([[[ 0,  1],
        [ 2,  3]],

       [[ 4,  5],
        [ 6,  7]],

       [[ 8,  9],
        [10, 11]]])
```

In [ ]:
```
a[1:]
```

Out[ ]:
```
array([[[ 4,  5],
        [ 6,  7]],

       [[ 8,  9],
        [10, 11]]])
```

In [8]:
```
a[1:,...]
```

Out[8]:
```
array([[[ 4,  5],
        [ 6,  7]],

       [[ 8,  9],
        [10, 11]]])
```

In [9]:
```
from numpy import *
b= arange(15)
b= b.reshape(3,5,1)
b
```

Out[9]:
```
array([[[ 0],
        [ 1],
        [ 2],
        [ 3],
        [ 4]],

       [[ 5],
        [ 6],
        [ 7],
        [ 8],
        [ 9]],

       [[10],
        [11],
        [12],
        [13],
        [14]]])
```

In [10]:
```
c= arange(10)
c= c.reshape(2,5,1)
c
```

Out[10]:
```
array([[[0],
        [1],
        [2],
        [3],
        [4]],

       [[5],
        [6],
        [7],
        [8],
        [9]]])
```

In [11]:
```
c
```

Out[11]:
```
array([[[0],
        [1],
        [2],
        [3],
        [4]],

       [[5],
        [6],
        [7],
        [8],
        [9]]])
```

In [12]:
```
c[1:]
```

Out[12]:
```
array([[[5],
        [6],
        [7],
        [8],
        [9]]])
```

In [13]:
```
c[1:,...]
```

Out[13]:
```
array([[[5],
        [6],
        [7],
        [8],
        [9]]])
```

In [14]:
```
c[0:,...]
```

Out[14]:
```
array([[[0],
        [1],
        [2],
        [3],
        [4]],

       [[5],
        [6],
        [7],
        [8],
        [9]]])
```

In [15]:
```
d= arange(19)
d
```

Out[15]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18])
```

In [16]:
```
x = arange(20)
x = x.reshape(2,5,2)
x
```

Out[16]:
```
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5],
        [ 6,  7],
        [ 8,  9]],

       [[10, 11],
        [12, 13],
        [14, 15],
        [16, 17],
        [18, 19]]])
```

In [17]:
```
x = x.reshape(2,10,1)
x
```

Out[17]:
```
array([[[ 0],
        [ 1],
        [ 2],
        [ 3],
        [ 4],
        [ 5],
        [ 6],
        [ 7],
        [ 8],
        [ 9]],

       [[10],
        [11],
        [12],
        [13],
        [14],
        [15],
        [16],
        [17],
        [18],
        [19]]])
```

In [18]:
```
x = x.reshape(5,2,2)
x
```

Out[18]:
```
array([[[ 0,  1],
        [ 2,  3]],

       [[ 4,  5],
        [ 6,  7]],

       [[ 8,  9],
        [10, 11]],

       [[12, 13],
        [14, 15]],

       [[16, 17],
        [18, 19]]])
```

In [19]:
```
x[...,0]     #x[:,:,0]
```

Out[19]:
```
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10],
       [12, 14],
       [16, 18]])
```

In [20]:
```
x[...,1]    #x[:,:,1]
```

Out[20]:
```
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11],
       [13, 15],
       [17, 19]])
```

In [21]:
```
x[1,...]   #x[1,:,:]
```

Out[21]:
```
array([[4, 5],
       [6, 7]])
```

In [22]:
```
x[1:,...]  #x[1:,:,:]
```

Out[22]:
```
array([[[ 4,  5],
        [ 6,  7]],

       [[ 8,  9],
        [10, 11]],

       [[12, 13],
        [14, 15]],

       [[16, 17],
        [18, 19]]])
```

In [23]:
```
x[2:,...]
```

Out[23]:
```
array([[[ 8,  9],
        [10, 11]],

       [[12, 13],
        [14, 15]],

       [[16, 17],
        [18, 19]]])
```

In [24]:
```
x[2,...]
```

Out[24]:
```
array([[ 8,  9],
       [10, 11]])
```

In [25]:
```
#x[...,2]
```

In [26]:
```
ab = arange(30)
ab = ab.reshape(3,2,5)
ab
```

Out[26]:
```
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]]])
```

In [27]:
```
ab[...,0]
```

Out[27]:
```
array([[ 0,  5],
       [10, 15],
       [20, 25]])
```

In [28]:
```
ab[0,...]
```

Out[28]:
```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

In [29]:
```
ab[1,...]
```

Out[29]:
```
array([[10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

In [30]:
```
ab[...,1]
```

Out[30]:
```
array([[ 1,  6],
       [11, 16],
       [21, 26]])
```

In [31]:
```
ab[...,4]
```

Out[31]:
```
array([[ 4,  9],
       [14, 19],
       [24, 29]])
```

In [32]:
```
ab
```

Out[32]:
```
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]]])
```

In [33]:
```
ab[1:,...]
```

Out[33]:
```
array([[[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]]])
```

In [34]:
```
ab[...,:-1]
```

Out[34]:
```
array([[[ 0,  1,  2,  3],
        [ 5,  6,  7,  8]],

       [[10, 11, 12, 13],
        [15, 16, 17, 18]],

       [[20, 21, 22, 23],
        [25, 26, 27, 28]]])
```

In [35]:
```
ab[...,2]
```

Out[35]:
```
array([[ 2,  7],
       [12, 17],
       [22, 27]])
```

In [36]:
```
ab[...,:2]
```

Out[36]:
```
array([[[ 0,  1],
        [ 5,  6]],

       [[10, 11],
        [15, 16]],

       [[20, 21],
        [25, 26]]])
```

In [37]:
```
ab
```

Out[37]:
```
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9]],

       [[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]]])
```

2. []

In [38]:
```
a = array([[0,1,2,3,4], [10,11,12,13,14], [20,21,22,23,24], [30,31,32,33,34]])
a
```

Out[38]:
```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

In [39]:
```
a[0][0]
```

Out[39]:
```
0
```

In [40]:
```
a[0][2]
```

Out[40]:
```
2
```

In [41]:
```
a[2][1]
```

Out[41]:
```
21
```

In [42]:
```
a[-1]
```

Out[42]:
```
array([30, 31, 32, 33, 34])
```

In [43]:
```
print(type(a))
```

```
<class 'numpy.ndarray'>
```

In [44]:
```
a
```

Out[44]:
```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

In [45]:
```
a[2,3]
```

Out[45]:
```
23
```

In [46]:
```
a[1:3,1:4]
```

Out[46]:
```
array([[11, 12, 13],
       [21, 22, 23]])
```

In [47]:
```
a = array([[0,1,2,3,4], [10,11,12,13,14], [20,21,22,23,24],[30,31,32,33,34]])
```

In [48]:
```
a
```

Out[48]:
```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

In [49]:
```
a[0,0]
```

Out[49]:
```
0
```

In [50]:
```
type(a)
```

Out[50]:
```
numpy.ndarray
```

In [51]:
```
a[1,5]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[51], line 1
----> 1 a[1,5]

IndexError: index 5 is out of bounds for axis 1 with size 5
```

In [ ]:
```
a[1,4]
```

In [52]:
```
a
```

Out[52]:
```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

In [53]:
```
a[0]
```

Out[53]:
```
array([0, 1, 2, 3, 4])
```

In [54]:
```
a[-1]
```

Out[54]:
```
array([30, 31, 32, 33, 34])
```

In [55]:
```
a[1:3,1:4]  #subarray
```

Out[55]:
```
array([[11, 12, 13],
       [21, 22, 23]])
```

In [56]:
```
a[1:3,2:4]
```

Out[56]:
```
array([[12, 13],
       [22, 23]])
```

In [57]:
```
'''
>>> 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])
'''
```

Out[57]:
```
'\ni = array([0,1,2,1]) # array of indices for the first axis\nj = array([1,2,3,
4]) # array of indices for the second axis\na[i,j]\narray([ 1, 12, 23, 14])\n\na[a
<13] # boolean indexing\narray([ 0, 1, 2, 3, 4, 10, 11, 12])\n'
```

In [ ]:

In [ ]:

In [58]:
```
a
```

Out[58]:
```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

In [59]:
```
b1=array([True,False,True,False])   #boolean row selector
a[b1,:]
```

Out[59]:
```
array([[ 0,  1,  2,  3,  4],
       [20, 21, 22, 23, 24]])
```

In [60]:
```
b2 = array([True,False,False,True,True])    #boolean column selector
a[:,b2]
```

Out[60]:
```
array([[ 0,  3,  4],
       [10, 13, 14],
       [20, 23, 24],
       [30, 33, 34]])
```

In [61]:
```
a[a<13]        #boolean indexing
```

Out[61]:
```
array([ 0,  1,  2,  3,  4, 10, 11, 12])
```

3. abs()

4. absolute()

In [62]:
```
abs(-3)
```

Out[62]:
```
3
```

In [63]:
```
abs(array([-1.7,-1.2]))
```

Out[63]:
```
array([1.7, 1.2])
```

In [64]:
```
abs(1+3j)
```

Out[64]:
```
3.1622776601683795
```

In [65]:
```
abs(1.5+2j)
```

Out[65]:
```
2.5
```

In [66]:
```
abs(True)
```

Out[66]:
```
1
```

In [67]:
```
abs(0b10)
```

Out[67]:
```
2
```

In [68]:
```
abs(0o234)
```

Out[68]:
```
156
```

In [69]:
```
abs(~18)
```

Out[69]:
```
19
```

In [70]:
```
abs(23^56)
```

Out[70]:
```
47
```

In [71]:
```
abs(21 & 7)
```

Out[71]:
```
5
```

5. accumulate()

In [72]:
```
add.accumulate(array([1.,2.,3.,4.]))  #Accumulate the result of applying the operat
```

Out[72]:
```
array([ 1.,  3.,  6., 10.])
```

In [73]:
```
import numpy as np
```

In [74]:
```
np.add.accumulate(array([1,2,3,4]))
```

Out[74]:
```
array([ 1,  3,  6, 10])
```

In [75]:
```
np.multiply.accumulate(array([1,2,3,4]))
```

Out[75]:
```
array([ 1,  2,  6, 24])
```

In [76]:
```
'''
2-D array examples:

>>> I = np.eye(2)
>>> I
array([[1.,  0.],
       [0.,  1.]])

Accumulate along axis 0 (rows), down columns:

>>> np.add.accumulate(I, 0)
array([[1.,  0.],
       [1.,  1.]])
>>> np.add.accumulate(I) # no axis specified = axis zero
array([[1.,  0.],
       [1.,  1.]])

Accumulate along axis 1 (columns), through rows:

>>> np.add.accumulate(I, 1)
array([[1.,  1.],
       [0.,  1.]])
```

```
Type:      builtin_function_or_method
'''
```

Out[76]:
```
'\n2-D array examples:\n\n>>> I = np.eye(2)\n>>> I\narray([[1.,  0.],\n       [0.,
1.]])\n\nAccumulate along axis 0 (rows), down columns:\n\n>>> np.add.accumulate(I,
0)\narray([[1.,  0.],\n       [1.,  1.]])\n>>> np.add.accumulate(I) # no axis spec
ified = axis zero\narray([[1.,  0.],\n       [1.,  1.]])\n\nAccumulate along axis
1 (columns), through rows:\n\n>>> np.add.accumulate(I, 1)\narray([[1.,  1.],\n
[0.,  1.]])\nType:      builtin_function_or_method\n'
```

In [77]:
```
'''
Return a 2-D array with ones on the diagonal and zeros elsewhere.
'''
I = eye(2)
I
```

Out[77]:
```
array([[1., 0.],
       [0., 1.]])
```

In [78]:
```
add.accumulate(I,0) #axis =0 (rows) , down columns
```

Out[78]:
```
array([[1., 0.],
       [1., 1.]])
```

In [79]:
```
add.accumulate(I)  #default axis =0
```

Out[79]:
```
array([[1., 0.],
       [1., 1.]])
```

In [80]:
```
add.accumulate(I,1)  #axis =1 (columns)  ,thru rows
```

Out[80]:
```
array([[1., 1.],
       [0., 1.]])
```

In [81]:
```
I = eye(3,3)
I
```

Out[81]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [82]:
```
add.accumulate(I)  #default axis 0 (rows) i.e down columns
```

Out[82]:
```
array([[1., 0., 0.],
       [1., 1., 0.],
       [1., 1., 1.]])
```

In [83]:
```
add.accumulate(I,1)
```

Out[83]:
```
array([[1., 1., 1.],
       [0., 1., 1.],
       [0., 0., 1.]])
```

In [84]:
```
multiply.accumulate(array([1.,2.,3.,4.]), dtype = int)
```

Out[84]:
```
array([ 1,  2,  6, 24])
```

In [85]:
```
divide.accumulate(array([1,2,3,4]))
```

Out[85]:
```
array([1.        , 0.5       , 0.16666667, 0.04166667])
```

In [86]:
```
add.accumulate(array([[1,2,3],[4,5,6]]),axis=0)  #accumulate every column seperate
```

Out[86]:
```
array([[1, 2, 3],
       [5, 7, 9]])
```

In [87]:
```
add.accumulate(array([[1,2,3],[4,5,6]]),axis=1)   #accumulate every row seperately
```

Out[87]:
```
array([[ 1,  3,  6],
       [ 4,  9, 15]])
```

In [88]:
```
add.accumulate(array([1,2,3,4]))
```

Out[88]:
```
array([ 1,  3,  6, 10])
```

6. add()

In [89]:
```
add(array([1.3,4]), array([-0.3,-5]))
```

Out[89]:
```
array([ 1., -1.])
```

In [90]:
```
array([1.3,4]) + array([-0.3,-5])
```

Out[90]:
```
array([ 1., -1.])
```

7. all()

In [91]:
```
a = array([True,False,True,True])
all(a)
#a.all()
```

Out[91]:
```
False
```

In [92]:
```
a = array([1,2,3])
all(a>0)
#(a>0).all()
```

Out[92]:
```
True
```

8. allclose()

In [93]:
```
'''
Signature: allclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)
Docstring:
Returns True if two arrays are element-wise equal within a tolerance.

The tolerance values are positive, typically very small numbers.  The
relative difference (`rtol` * abs(`b`)) and the absolute difference
`atol` are added together to compare against the absolute difference
between `a` and `b`.

a and b: These are the input arrays to be compared.
rtol (optional): The relative tolerance parameter. The default value is 1e-05.
atol (optional): The absolute tolerance parameter. The default value is 1e-08.
equal_nan (optional): If True, then NaN values are considered equal. The default va
'''
```

Out[93]:
```
'\nSignature: allclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)\nDocstrin
g:\nReturns True if two arrays are element-wise equal within a tolerance.\n\nThe t
olerance values are positive, typically very small numbers.  The\nrelative differe
nce (`rtol` * abs(`b`)) and the absolute difference\n`atol` are added together to
compare against the absolute difference\nbetween `a` and `b`.\n\na and b: These ar
e the input arrays to be compared.\nrtol (optional): The relative tolerance parame
ter. The default value is 1e-05.\natol (optional): The absolute tolerance paramete
r. The default value is 1e-08.\nequal_nan (optional): If True, then NaN values are
considered equal. The default value is False.\n'
```

In [94]:
```
allclose(array([1e10,1e-8]), array([1.00001e10, 1e-9]))
```

Out[94]:
```
True
```

In [95]:
```
allclose(array([1,2,3]),array([2,4,6]))
```

Out[95]:
```
False
```

1. alltrue()

In [96]:
```
a = array([True,True,True])
alltrue(a)
```

Out[96]:
```
True
```

In [97]:
```
a= array([4,5,6,7,8])
alltrue(a>=4)
```

Out[97]:
```
True
```

In [98]:
```
a= array([1,2,3,4,5,6,7,8])
alltrue(a>=4)
```

Out[98]:
```
False
```

10. angle()

In [99]:
```
'''
Signature: angle(z, deg=False)
Docstring:
Return the angle of the complex argument.

Parameters
----------
z : array_like
    A complex number or sequence of complex numbers.
deg : bool, optional
    Return angle in degrees if True, radians if False (default).
'''
```

Out[99]:
```
'\nSignature: angle(z, deg=False)\nDocstring:\nReturn the angle of the complex arg
ument.\n\nParameters\n----------\nz : array_like\n    A complex number or sequence
of complex numbers.\ndeg : bool, optional\n    Return angle in degrees if True, ra
dians if False (default).\n'
```

In [100…
```
'''
Examples
--------
```

```
>>> np.angle([1.0, 1.0j, 1+1j])               # in radians
array([ 0.        ,  1.57079633,  0.78539816]) # may vary
>>> np.angle(1+1j, deg=True)                  # in degrees
45.0
'''
```

Out[100]:
```
'\nExamples\n--------\n>>> np.angle([1.0, 1.0j, 1+1j])               # in radians
\narray([ 0.        ,  1.57079633,  0.78539816]) # may vary\n>>> np.angle(1+1j, de
g=True)                  # in degrees\n45.0\n'
```

In [101…
```
angle(1+5j)    #deg = False ,gives in radians
```

Out[101]:
```
1.373400766945016
```

In [102…
```
print(type(angle(1+5j)))
```

```
<class 'numpy.float64'>
```

In [103…
```
angle(1+5j, deg = True)  #deg = True ,gives in degrees
```

Out[103]:
```
78.69006752597979
```

In [104…
```
angle([1.0, 1.0j, 1+1j])
```

Out[104]:
```
array([0.        , 1.57079633, 0.78539816])
```

In [105…
```
angle(1+1j, deg= True)
```

Out[105]:
```
45.0
```

11. any()

In [106…
```
from numpy import *
```

In [107…
```
a1 = array([True,False,True,True])
a1.any()
```

Out[107]:
```
True
```

In [108…
```
any(a1)
```

Out[108]:
```
True
```

In [109…
```
a2 = array([1,2,3,4,5])
a2
```

Out[109]:
```
array([1, 2, 3, 4, 5])
```

In [110…
```
(a2>=2).any()
```

Out[110]:
```
True
```

In [111…
```
(a2>5).any()
```

Out[111]:
```
False
```

In [112…
```
(a2<1).any()
```

Out[112]:
```
False
```

In [113…
```
any(a2==6)
```

Out[113]:
```
False
```

12. append()

''' Signature: append(arr, values, axis=None) Docstring: Append values to the end of an array.

Examples

> np.append([1, 2, 3], [[4, 5, 6], [7, 8, 9]]) array([1, 2, 3, ..., 7, 8, 9])

When `axis` is specified, `values` must have the correct shape.

> np.append([[1, 2, 3], [4, 5, 6]], [[7, 8, 9]], axis=0) array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) np.append([[1, 2, 3], [4, 5, 6]], [7, 8, 9], axis=0) Traceback (most recent call last): ... ValueError: all the input arrays must have same number of dimensions, but the array at index 0 has 2 dimension(s) and the array at index 1 has 1 dimension(s)

'''

In [115…
```
x = array([10,20,30,40])
x
```

Out[115]:
```
array([10, 20, 30, 40])
```

In [116…
```
append(x,50)
```

Out[116]:
```
array([10, 20, 30, 40, 50])
```

In [117…
```
append(x,[50,60])
```

Out[117]:
```
array([10, 20, 30, 40, 50, 60])
```

In [118…
```
x
```

Out[118]:
```
array([10, 20, 30, 40])
```

In [119…
```
append(x,[70,10])
```

Out[119]:
```
array([10, 20, 30, 40, 70, 10])
```

In [120…
```
a = array([[1,2,3],[4,5,6],[7,8,9]])
a
```

Out[120]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [121…
```
append(a,[10,11,12])
```

Out[121]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

In [122…
```
a
```

Out[122]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [123…
```
append(a, [[10,11,12]], axis=0)
```

Out[123]:
```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

In [124…
```
append(a, [[0,1,6]], axis =1)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[124], line 1
----> 1 append(a, [[0,1,6]], axis =1)

File ~\anaconda3\conda\Lib\site-packages\numpy\lib\function_base.py:5617, in append(arr, values, axis)
   5615     values = ravel(values)
   5616     axis = arr.ndim-1
-> 5617 return concatenate((arr, values), axis=axis)

ValueError: all the input array dimensions except for the concatenation axis must match exactly, but along dimension 0, the array at index 0 has size 3 and the array at index 1 has size 1
```

In [125…
```
append(a, [[0],[1],[2]], axis=1)
```

Out[125]:
```
array([[1, 2, 3, 0],
       [4, 5, 6, 1],
       [7, 8, 9, 2]])
```

In [126…
```
append(a, [[2],[3],[4]], axis=1)
```

Out[126]:
```
array([[1, 2, 3, 2],
       [4, 5, 6, 3],
       [7, 8, 9, 4]])
```

In [127…
```
 arr1 = array([1,4,5,9])
```

In [128…
```
arr1
```

Out[128]:
```
array([1, 4, 5, 9])
```

In [129…
```
append(arr1, 90)
```

Out[129]:
```
array([ 1,  4,  5,  9, 90])
```

In [130…
```
append(arr1, [10,20])
```

Out[130]:
```
array([ 1,  4,  5,  9, 10, 20])
```

In [131…
```
append(arr1, [1,2,3], axis=0)
```

Out[131]:
```
array([1, 4, 5, 9, 1, 2, 3])
```

In [132…
```
a2 = array([[1,2,3],[2,3,4]])
```

In [133…
```
a2
```

Out[133]:
```
array([[1, 2, 3],
       [2, 3, 4]])
```

In [134…
```
a2 = array([[1,2,3],[2,3,4]])
```

In [135…
```
append(a2, [[5,6,7]], axis=0)
```

Out[135]:
```
array([[1, 2, 3],
       [2, 3, 4],
       [5, 6, 7]])
```

In [136…
```
m = arange(50).reshape(5,10)
```

In [137…
```
m
```

Out[137]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

In [138…
```
append(m, [[50,51,52,53,54,55,56,57,58,59]], axis=0)
```

Out[138]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59]])
```

13. apply_along_axis()

In [139…
```
from numpy import *
```

In [140…
```
def myfunc(a):    #function works on 1d arrays, takes average of 1st and last eleme
    return (a[0]+a[-1])/2
```

In [ ]:
```

```

In [141…
```
b = array([[1,2,3],[4,5,6],[7,8,9]])
```

In [142…
```
b
```

Out[142]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [143…
```
apply_along_axis(myfunc,0,b)   #axis 0 rows means down the column each column
```

Out[143]:
```
array([4., 5., 6.])
```

In [144…
```
apply_along_axis(myfunc,1,b) #apply myfunc to each row of b
```

Out[144]:
```
array([2., 5., 8.])
```

In [145…
```
c= array([[1,3,6,7],[2,9,0,3],[2,4,5,6]])
```

In [146…
```
c
```

Out[146]:
```
array([[1, 3, 6, 7],
       [2, 9, 0, 3],
       [2, 4, 5, 6]])
```

In [147…
```
apply_along_axis(myfunc,0,c)
```

Out[147]:
```
array([1.5, 3.5, 5.5, 6.5])
```

In [148…
```
def myfun(x):
    return x.max()
```

In [149…
```
c
```

Out[149]:
```
array([[1, 3, 6, 7],
       [2, 9, 0, 3],
       [2, 4, 5, 6]])
```

In [150…
```
apply_along_axis(myfun,0,c)
```

Out[150]:
```
array([2, 9, 6, 7])
```

In [151…
```
apply_along_axis(myfun,1,c)
```

Out[151]:
```
array([7, 9, 6])
```

14. apply_over_axes

In [152…
```
a1 = arange(24).reshape(2,3,4)
```

In [153…
```
a1
```

Out[153]:
```
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]]])
```

In [154…
```
apply_over_axes(sum,a1,[0,2])
```

Out[154]:
```
array([[[ 60],
        [ 92],
        [124]]])
```

In [155…
```
apply_over_axes(subtract,a1,[0,2])
```

Out[155]:
```
array([[[-2, -1,  0,  1],
        [ 2,  3,  4,  5],
        [ 6,  7,  8,  9]],

       [[10, 11, 12, 13],
        [14, 15, 16, 17],
        [18, 19, 20, 21]]])
```

15. arange()

In [156…
```
arange(15)
```

Out[156]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

In [157…
```
arange(5.0)
```

Out[157]:
```
array([0., 1., 2., 3., 4.])
```

In [158…
```
arange(20, dtype=int)
```

Out[158]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

In [159…
```
arange(2,10)
```

Out[159]:
```
array([2, 3, 4, 5, 6, 7, 8, 9])
```

In [160…
```
arange(10,30,5)
```

Out[160]:
```
array([10, 15, 20, 25])
```

16. arccos()

In [161…
```
'''
Trigonometric inverse cosine, element-wise.
'''
```

Out[161]:
```
'\nTrigonometric inverse cosine, element-wise.\n'
```

In [162…
```
arccos(array([1,0]))
```

Out[162]:
```
array([0.        , 1.57079633])
```

In [163…
```
arccos(array([0,1]))
```

Out[163]:
```
array([1.57079633, 0.        ])
```

17. arccosh()

In [164…
```
arccosh(array([e,10.0]))  #inverse hyperbolic cosine
```

Out[164]:
```
array([1.65745445, 2.99322285])
```

In [165…
```
arccosh(array([e,15.0]))
```
Out[165]:
```
array([1.65745445, 3.40008441])
```

18. arcsin()

In [166…
```
#inverse sine
```
In [167…
```
arcsin(array([1,0]))
```
Out[167]:
```
array([1.57079633, 0.        ])
```
In [168…
```
arcsin(array([0,1]))
```
Out[168]:
```
array([0.        , 1.57079633])
```
In [169…
```
arcsin(array([0.5,0.5]))
```
Out[169]:
```
array([0.52359878, 0.52359878])
```

19. arcsinh()

In [170…
```
#inverse hyperbolic sine
```
In [171…
```
arcsinh(array([e,1]))
```
Out[171]:
```
array([1.72538256, 0.88137359])
```
In [172…
```
arcsinh(array([e,10.0]))
```
Out[172]:
```
array([1.72538256, 2.99822295])
```
In [173…
```
arcsinh(array([e,20.0]))
```
Out[173]:
```
array([1.72538256, 3.68950387])
```

20. arctan()

In [174…
```
#Trigonometric inverse tangent, element-wise.
```
In [175…
```
arctan(array([1,0.5]))
```
Out[175]:
```
array([0.78539816, 0.46364761])
```
In [176…
```
arctan(array([1,1]))
```
Out[176]:
```
array([0.78539816, 0.78539816])
```

21. arctan2()

In [177…
```
arctan2(array([1,0]),array([0,1]))
```

Out[177]:
```
array([1.57079633, 0.        ])
```

22. arctanh()

In [178…
```
'''
Inverse hyperbolic tangent element-wise.
'''
```

Out[178]:
```
'\nInverse hyperbolic tangent element-wise.\n'
```

In [179…
```
arctanh(array([0,-0.5]))
```

Out[179]:
```
array([ 0.        , -0.54930614])
```

23. argmax()

In [180…
```
a = array([10,20,30,40,50,60])
a
```

Out[180]:
```
array([10, 20, 30, 40, 50, 60])
```

In [181…
```
maxindex = a.argmax()
```

In [182…
```
maxindex
```

Out[182]:
```
5
```

In [183…
```
a[maxindex]
```

Out[183]:
```
60
```

In [184…
```
a1 = array([[1,4,9],[23,4,5]])
```

In [185…
```
a1
```

Out[185]:
```
array([[ 1,  4,  9],
       [23,  4,  5]])
```

In [186…
```
maxindex = a1.argmax()
```

In [187…
```
maxindex
```

Out[187]:
```
3
```

In [188…
```
#a1[maxindex]
a1.ravel()[maxindex]
```

Out[188]:
```
23
```

In [189…
```
a2 = array([[10,50,30],[60,20,40]])
```

In [190…
```
maxindex = a2.argmax()
```

In [191…
```
maxindex
```

Out[191]:
```
3
```

In [192…
```
a2.ravel()[maxindex]
```

Out[192]:
```
60
```

In [193…
```
a2[maxindex]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[193], line 1
----> 1 a2[maxindex]

IndexError: index 3 is out of bounds for axis 0 with size 2
```

In [194…
```
b1 =array([[1,5,10],[29,10,2]])
```

In [195…
```
b1
```

Out[195]:
```
array([[ 1,  5, 10],
       [29, 10,  2]])
```

In [196…
```
maxindex = b1.argmax()
```

In [197…
```
maxindex
```

Out[197]:
```
3
```

In [198…
```
b1.ravel()
```

Out[198]:
```
array([ 1,  5, 10, 29, 10,  2])
```

In [199…
```
b1.ravel()[maxindex]
```

Out[199]:
```
29
```

In [200…
```
b1
```

Out[200]:
```
array([[ 1,  5, 10],
       [29, 10,  2]])
```

In [201…
```
b1.argmax(axis=0)
```

Out[201]:
```
array([1, 1, 0], dtype=int64)
```

In [202…
```
b1.argmax(axis=1)
```

Out[202]:
```
array([2, 0], dtype=int64)
```

In [203…
```
argmax(b1, axis=1)
```

Out[203]:
```
array([2, 0], dtype=int64)
```

In [ ]:

24. argmin()

In [204…
```
a
```
Out[204]:
```
array([10, 20, 30, 40, 50, 60])
```

In [205…
```
minindex = a.argmin()
```

In [206…
```
minindex
```
Out[206]:
```
0
```

In [207…
```
a[minindex]
```
Out[207]:
```
10
```

In [208…
```
a1
```
Out[208]:
```
array([[ 1,  4,  9],
       [23,  4,  5]])
```

In [209…
```
minindex = a1.argmin()
```

In [210…
```
minindex
```
Out[210]:
```
0
```

In [211…
```
a1[minindex]
```
Out[211]:
```
array([1, 4, 9])
```

In [212…
```
a1.ravel()[minindex]
```
Out[212]:
```
1
```

In [213…
```
a1
```
Out[213]:
```
array([[ 1,  4,  9],
       [23,  4,  5]])
```

In [214…
```
a1.argmin(axis=0)
```
Out[214]:
```
array([0, 0, 1], dtype=int64)
```

In [215…
```
a1.argmin(axis=1)
```
Out[215]:
```
array([0, 1], dtype=int64)
```

25. argsort()

In [216…
```
from numpy import *
```

In [217…
```
c1 = array([2,7,0,3,10])
```

In [218…
```
ind = c1.argsort()
```

In [219…
```
ind
```

Out[219]:
```
array([2, 0, 3, 1, 4], dtype=int64)
```

In [220…
```
c1[ind]
```

Out[220]:
```
array([ 0,  2,  3,  7, 10])
```

In [221…
```
c1.sort()
```

In [222…
```
c1
```

Out[222]:
```
array([ 0,  2,  3,  7, 10])
```

In [223…
```
# algorithm options are 'quicksort', 'mergesort' and 'heapsort'
ind = c1.argsort(kind = 'heap')
```

In [224…
```
c1[ind]
```

Out[224]:
```
array([ 0,  2,  3,  7, 10])
```

In [225…
```
a= array([[8,2,7],[2,0,1]])
a
```

Out[225]:
```
array([[8, 2, 7],
       [2, 0, 1]])
```

In [226…
```
ind = a.argsort()
```

In [227…
```
ind
```

Out[227]:
```
array([[1, 2, 0],
       [1, 2, 0]], dtype=int64)
```

In [228…
```
ind = a.argsort(axis =0)   #sort columns
ind
```

Out[228]:
```
array([[1, 1, 1],
       [0, 0, 0]], dtype=int64)
```

In [229…
```
 # 2-D arrays need fancy indexing if you want to sort them.
a[ind,[[0,1,2],[0,1,2]]]
```

Out[229]:
```
array([[2, 0, 1],
       [8, 2, 7]])
```

In [230…
```
ind = a.argsort(axis =1)   #sort rows
ind
```

Out[230]:
```
array([[1, 2, 0],
       [1, 2, 0]], dtype=int64)
```

In [231…
```
 # 2-D arrays need fancy indexing if you want to sort them.
a[ind,[[0,1,2],[0,1,2]]]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[231], line 2
      1 # 2-D arrays need fancy indexing if you want to sort them.
----> 2 a[ind,[[0,1,2],[0,1,2]]]

IndexError: index 2 is out of bounds for axis 0 with size 2
```

In [232…
```
a=ones(15)
a
```

Out[232]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

In [233…
```
a.argsort(axis = -1,kind='quick')  ## quicksort doesn't preserve original order.
```

Out[233]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14],
      dtype=int64)
```

In [234…
```
a.argsort(kind = 'merge')
```

Out[234]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14],
      dtype=int64)
```

In [235…
```
ind = argsort(a)
```

In [236…
```
ind
```

Out[236]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14],
      dtype=int64)
```

In [237…
```
a
```

Out[237]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

In [238…
```
c1
```

Out[238]:
```
array([ 0,  2,  3,  7, 10])
```

In [239…
```
print(c1.sort())
```

```
None
```

26. array()

In [240…
```
'''
Docstring:
array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0,
      like=None)

Create an array.

Parameters
----------
object : array_like
    An array, any object exposing the array interface, an object whose
    __array__ method returns an array, or any (nested) sequence.
    If object is a scalar, a 0-dimensional array containing object is
    returned.
dtype : data-type, optional
    The desired data-type for the array.  If not given, then the type will
    be determined as the minimum type required to hold the objects in the
```

```
    sequence.
copy : bool, optional
    If true (default), then the object is copied.  Otherwise, a copy will
    only be made if __array__ returns a copy, if obj is a nested sequence,
    or if a copy is needed to satisfy any of the other requirements
    (`dtype`, `order`, etc.).
order : {'K', 'A', 'C', 'F'}, optional
    Specify the memory layout of the array. If object is not an array, the
    newly created array will be in C order (row major) unless 'F' is
    specified, in which case it will be in Fortran order (column major).
    If object is an array the following holds.

    ===== ========= ===================================================
    order  no copy                     copy=True
    ===== ========= ===================================================
    'K'   unchanged F & C order preserved, otherwise most similar order
    'A'   unchanged F order if input is F and not C, otherwise C order
    'C'   C order   C order
    'F'   F order   F order
    ===== ========= ===================================================
subok : bool, optional
    If True, then sub-classes will be passed-through, otherwise
    the returned array will be forced to be a base-class array (default).
ndmin : int, optional
    Specifies the minimum number of dimensions that the resulting
    array should have.  Ones will be prepended to the shape as
    needed to meet this requirement.
'''
```

Out[240]:
```
"\nDocstring:\narray(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0,\n      like=None)\n\nCreate an array.\n\nParameters\n----------\nobject : array_like\n    An array, any object exposing the array interface, an object whose\n__array__ method returns an array, or any (nested) sequence.\n    If object is a scalar, a 0-dimensional array containing object is\n    returned.\ndtype : data-type, optional\n    The desired data-type for the array.  If not given, then the type will\n    be determined as the minimum type required to hold the objects in the\nsequence.\ncopy : bool, optional\n    If true (default), then the object is copied.  Otherwise, a copy will\n    only be made if __array__ returns a copy, if obj is a nested sequence,\n    or if a copy is needed to satisfy any of the other requirements\n    (`dtype`, `order`, etc.).\norder : {'K', 'A', 'C', 'F'}, optional\nSpecify the memory layout of the array. If object is not an array, the\n    newly created array will be in C order (row major) unless 'F' is\n    specified, in which case it will be in Fortran order (column major).\n    If object is an array the following holds.\n\n    ===== ========= ===================================================\n    order  no copy                     copy=True\n    ===== ========= ===================================================\n    'K'   unchanged F & C order preserved, otherwise most similar order\n    'A'   unchanged F order if input is F and not C, otherwise C order\n    'C'   C order   C order\n    'F'   F order   F order\n    ===== ========= ===================================================\nsubok : bool, optional\n    If True, then sub-classes will be passed-through, otherwise\n    the returned array will be forced to be a base-class array (default).\nndmin : int, optional\n    Specifies the minimum number of dimensions that the resulting\n    array should have.  Ones will be prepended to the shape as\n    needed to meet this requirement.\n"
```

In [241…
```
from numpy import *
```

In [242…
```
array([1,2,3,4])
```

Out[242]:
```
array([1, 2, 3, 4])
```

In [243…
```
type(array([1,2,3,4]))
```

Out[243]:
```
numpy.ndarray
```

In [244…
```
array([1,2,3,4], dtype = complex)
```

Out[244]:
```
array([1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j])
```

In [245…
```
array(1, copy=0, subok=1, ndmin=1)
```

Out[245]:
```
array([1])
```

In [246…
```
array([[1,2],[3,4]])
```

Out[246]:
```
array([[1, 2],
       [3, 4]])
```

In [247…
```
mydescriptor = {'names': ('gender','age','weight'), 'formats': ('S1','f4','f4')}
```

In [248…
```
a = array([('M','23','70'), ('F','25','75')], dtype=mydescriptor)
```

In [249…
```
print(a)
```

```
[(b'M', 23., 70.) (b'F', 25., 75.)]
```

In [250…
```
a
```

Out[250]:
```
array([(b'M', 23., 70.), (b'F', 25., 75.)],
      dtype=[('gender', 'S1'), ('age', '<f4'), ('weight', '<f4')])
```

In [251…
```
a['weight']
```

Out[251]:
```
array([70., 75.], dtype=float32)
```

In [252…
```
a['age']
```

Out[252]:
```
array([23., 25.], dtype=float32)
```

In [253…
```
a['gender']
```

Out[253]:
```
array([b'M', b'F'], dtype='|S1')
```

In [254…
```
a.dtype.names
```

Out[254]:
```
('gender', 'age', 'weight')
```

In [ ]:

27. arrayrange()

synonym for arange()

In [ ]:

28. array_split()

In [255…
```
a= array([[1,2,3,4],[5,6,7,8]])
```

In [256…
```
a
```

Out[256]:
```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

In [257…
```
array_split(a,2,axis=0) #split a in two parts along row axis
```

Out[257]:
```
[array([[1, 2, 3, 4]]), array([[5, 6, 7, 8]])]
```

In [258…
```
array_split(a,4,axis=0)
```

Out[258]:
```
[array([[1, 2, 3, 4]]),
 array([[5, 6, 7, 8]]),
 array([], shape=(0, 4), dtype=int32),
 array([], shape=(0, 4), dtype=int32)]
```

In [259…
```
array_split(a,4,axis=1)
```

Out[259]:
```
[array([[1],
        [5]]),
 array([[2],
        [6]]),
 array([[3],
        [7]]),
 array([[4],
        [8]])]
```

In [260…
```
array_split(a,3,axis=1)
```

Out[260]:
```
[array([[1, 2],
        [5, 6]]),
 array([[3],
        [7]]),
 array([[4],
        [8]])]
```

In [261…
```
array_split(a,[2,3],axis=1)
```

Out[261]:
```
[array([[1, 2],
        [5, 6]]),
 array([[3],
        [7]]),
 array([[4],
        [8]])]
```

In [ ]:
```
```

29. asarray()

In [262…
```
m= matrix('1 2; 5 8')
m
```

Out[262]:
```
matrix([[1, 2],
        [5, 8]])
```

In [263…
```
a1 = asarray(m)  # a is array type with same contents as m -- data is not copied
```

In [264…
```
a1
```

Out[264]:
```
array([[1, 2],
       [5, 8]])
```

In [265…
```
m[1,1] = -90
m
```

Out[265]:
```
matrix([[  1,   2],
        [  5, -90]])
```

In [266…
```
a1   # no copy was made, so modifying m modifies a, and vice versa
```

Out[266]:
```
array([[  1,   2],
       [  5, -90]])
```

In [267…
```
a1[0,0] = -4
a1
```

Out[267]:
```
array([[ -4,   2],
       [  5, -90]])
```

In [268…
```
m
```

Out[268]:
```
matrix([[ -4,   2],
        [  5, -90]])
```

30. asanyarray()

In [269…
```
a= array([[1,2],[3,4]])
```

In [270…
```
a
```

Out[270]:
```
array([[1, 2],
       [3, 4]])
```

In [271…
```
m = matrix('1 2; 5 8')
m
```

Out[271]:
```
matrix([[1, 2],
        [5, 8]])
```

In [272…
```
asanyarray(a)
```

Out[272]:
```
array([[1, 2],
       [3, 4]])
```

In [273…
```
asanyarray(a) is a
```

Out[273]:
```
True
```

In [274…
```
asanyarray(m)
```

Out[274]:
```
matrix([[1, 2],
        [5, 8]])
```

In [275…
```
asanyarray(m) is m
```

Out[275]:
```
True
```

In [276…
```
asanyarray([1,2,3,4])
```

Out[276]:
```
array([1, 2, 3, 4])
```

31. asmatrix()

In [277…
```
a = array([[1,2,3],[4,5,6]])
```

In [278…
```
a
```

Out[278]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [279…
```
m = asmatrix(a)    # m is matrix type with same contents as a -- data is not copie
```

In [280…
```
m
```

Out[280]:
```
matrix([[1, 2, 3],
        [4, 5, 6]])
```

In [281…
```
m[0,1]=45
```

In [282…
```
m
```

Out[282]:
```
matrix([[ 1, 45,  3],
        [ 4,  5,  6]])
```

In [283…
```
a
```

Out[283]:
```
array([[ 1, 45,  3],
       [ 4,  5,  6]])
```

In [284…
```
a[1,1]=-7
```

In [285…
```
a
```

Out[285]:
```
array([[ 1, 45,  3],
       [ 4, -7,  6]])
```

In [286…
```
m    # no copy was made so modifying a modifies m, and vice versa
```

Out[286]:
```
matrix([[ 1, 45,  3],
        [ 4, -7,  6]])
```

32. astype()

In [287…
```
x= array([0,1,23])
```

In [288…
```
y=x.astype(float64)
y
```

Out[288]:
```
array([ 0.,  1., 23.])
```

In [289…
```
print(y[0])
type(y[0])
```

Out[289]:
```
0.0
numpy.float64
```

33. atleast_1d()

In [290…
```
a = 1 #0-d array
b = array([1,2])  #1-d array
c = array([[1,2],[3,4]])  #2-d array
d = arange(8).reshape(2,2,2)  #3-d array
```

In [291…
```
print(a)
print(b)
print(c)
print(d)
```

```
1
[1 2]
[[1 2]
 [3 4]]
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

In [292…
```
atleast_1d(a,b,c,d)   # all output arrays have dim >= 1
```

Out[292]:
```
[array([1]),
 array([1, 2]),
 array([[1, 2],
        [3, 4]]),
 array([[[0, 1],
         [2, 3]],

        [[4, 5],
         [6, 7]]])]
```

34. atleast_2d()

In [293…
```
from numpy import *
```

In [294…
```
a = 1
b = array([2,3])
c = array([[4,5],[6,7]])
d = arange(8).reshape(2,2,2)
d
```

Out[294]:
```
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
```

In [295…
```
atleast_2d(a,b,c,d)  # all output arrays have dim >= 2
```

Out[295]:
```
[array([[1]]),
 array([[2, 3]]),
 array([[4, 5],
        [6, 7]]),
 array([[[0, 1],
         [2, 3]],

        [[4, 5],
         [6, 7]]])]
```

35. atleast_3d()

In [296…
```
a = 1
b = array([2,3])
c = array([[4,5],[6,7]])
d = arange(8).reshape(2,2,2)
d
```

Out[296]:
```
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
```

In [297…
```
atleast_3d(a,b,c,d)  ## all output arrays have dim >= 3
```

Out[297]:
```
[array([[[1]]]),
 array([[[2],
         [3]]]),
 array([[[4],
         [5]],

        [[6],
         [7]]]),
 array([[[0, 1],
         [2, 3]],

        [[4, 5],
         [6, 7]]])]
```

36. average()

In [298…
```
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
```

Out[298]:
```
3.0
```

In [299…
```
average(a, weights=w)  ## weighted average
```

Out[299]:
```
3.1666666666666665
```

In [300…
```
average(a, weights=w, returned=True)  ## output = weighted average, sum of weights
```

Out[300]:
```
(3.1666666666666665, 1.2)
```

37. beta()

In [301…
```
#Docstring:
#beta(a, b, size=None)

#Draw samples from a Beta distribution.

#The Beta distribution is a special case of the Dirichlet distribution,
#and is related to the Gamma distribution.  It has the probability
#distribution function
```

In [302…
```
from numpy import *
from numpy.random import *
beta(a=1,b=10,size=(2,2))
```

Out[302]:
```
array([[0.12350185, 0.13825714],
       [0.1070757 , 0.03289319]])
```

38. binary_repr()

In [303…
```
a=56
binary_repr(a)
```

Out[303]:
```
'111000'
```

In [304…
```
bin(a)
```

Out[304]:
```
'0b111000'
```

In [305…
```
b= float_(pi)  ## numpy float has extra functionality ...
b
```

Out[305]:
```
3.141592653589793
```

In [306…
```
b.nbytes  #number of bytes it takes
```

Out[306]:
```
8
```

In [307…
```
binary_repr(b.view('u8'))   ##view float number as an 8 byte integer, then get binc
```

Out[307]:
```
'1000000000010010010000111111011010101000100010000101101000110000'
```

In [308…
```
binary_repr(b.view('u8'))
```

Out[308]:
```
'1000000000010010010000111111011010101000100010000101101000110000'
```

39. bincount()

In [309…
```
a = array([1,1,2,2,2,3,4,5,5,5,6,7,7,7])
```

In [310…
```
bincount(a)
```

Out[310]:
```
array([0, 2, 3, 1, 1, 3, 1, 3], dtype=int64)
```

In [311…
```
a1 = array([6,2,4,4,1])
w = array([0.1,0.5,0.1,0.2,0.2])
```

In [312…
```
bincount(a1)
```

Out[312]:
```
array([0, 1, 1, 0, 2, 0, 1], dtype=int64)
```

In [313…
```
bincount(a1, weights =w)
```

Out[313]:
```
array([0. , 0.2, 0.5, 0. , 0.3, 0. , 0.1])
```

40. binomial()

In [314…
```
from numpy import *
from numpy.random import*
```

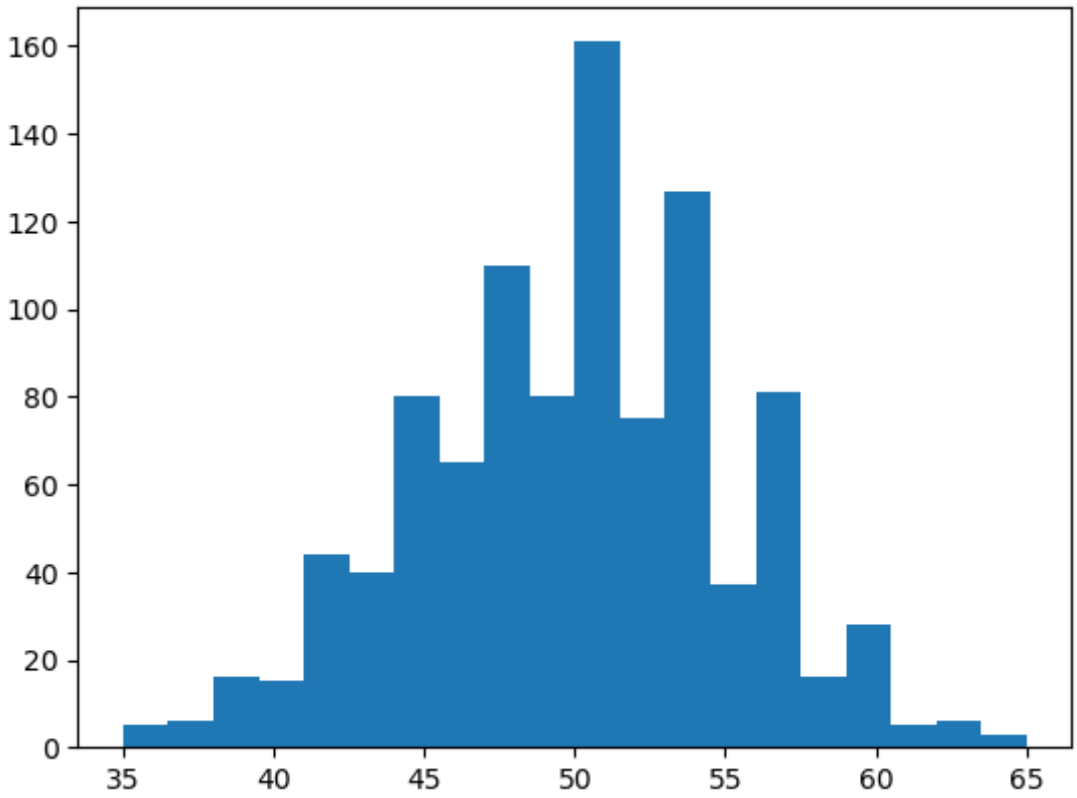
In [315…
```
binomial(n=100,p=0.5,size=(2,3))
```

Out[315]:
```
array([[48, 53, 47],
       [54, 55, 41]])
```

In [316…
```
from pylab import *
```

In [317…
```
hist(binomial(100,0.5,(1000)),20)
```

Out[317]:
```
(array([  5.,   6.,  16.,  15.,  44.,  40.,  80.,  65., 110.,  80., 161.,
         75., 127.,  37.,  81.,  16.,  28.,   5.,   6.,   3.]),
 array([35. , 36.5, 38. , 39.5, 41. , 42.5, 44. , 45.5, 47. , 48.5, 50. ,
        51.5, 53. , 54.5, 56. , 57.5, 59. , 60.5, 62. , 63.5, 65. ]),
 <BarContainer object of 20 artists>)
```



41. bitwise_and()

In [318…
```
bitwise_and(array([1,2,3]), array([2,3,4]))
```

Out[318]:
```
array([0, 2, 0])
```

In [319…
```
bitwise_and(array([2,5,255,2147483647],dtype=int32), array([4,4,4,2147483647],dtyp
```

Out[319]:
```
array([         0,          4,          4, 2147483647])
```

42. bitwise_or()

In [320…
```
bitwise_or(array([2,5,255]), array([4,4,4]))
```

Out[320]:
```
array([  6,   5, 255])
```

In [321…
```
bitwise_or(array([2,5,255,2147483647L],dtype=int32), array([4,4,4,2147483647L],dty
'''
The issue is with the "2147483647L" value in both arrays.
In Python 2, adding an 'L' to the end of an integer literal denotes a long integer
However, Python 3 does not support the 'L' suffix for long integers, and instead, a
to be long by default.
'''
```

```
  Cell In[321], line 1
    bitwise_or(array([2,5,255,2147483647L],dtype=int32), array([4,4,4,2147483647
L],dtype=int32))
                                                         ^
SyntaxError: invalid decimal literal
```

In [322…
```
bitwise_or(array([2,5,255,2147483647],dtype=int32), array([4,4,4,2147483647],dtype
```

Out[322]:
```
array([         6,          5,        255, 2147483647])
```

43. bitwise_xor()

In [323…
```
bitwise_xor(array([2,5,255]), array([4,4,4]))
```

Out[323]:
```
array([  6,   1, 251])
```

In [324…
```
bitwise_xor(array([2,5,255,2147483647],dtype=int32), array([4,4,4,2147483647],dtyp
```

Out[324]:
```
array([  6,   1, 251,   0])
```

44. bmat()

In [325…
```
a = mat('1 2;3 4')
b = mat('5 6;7 8')
bmat('a b;b a')
```

Out[325]:
```
matrix([[1, 2, 5, 6],
        [3, 4, 7, 8],
        [5, 6, 1, 2],
        [7, 8, 3, 4]])
```

45. broadcast()

In [326…
```
a = array([[1,2],[3,4]])
b = array([5,6])
c= broadcast(a,b)
print(c)
```

```
<numpy.broadcast object at 0x00000164014B4F70>
```

In [327…
```
c.nd
```

Out[327]:
```
2
```

In [328…
```
c.shape
```

Out[328]:
```
(2, 2)
```

In [329…
```
c.size
```

Out[329]:
```
4
```

In [330…
```
for i in c:
    print(i)
```

```
(1, 5)
(2, 6)
(3, 5)
(4, 6)
```

In [331…
```
c.reset()
```

In [332…
```
c.next()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[332], line 1
----> 1 c.next()

AttributeError: 'numpy.broadcast' object has no attribute 'next'
```

46. bytes()

In [333…
```
from numpy import *
from numpy.random import bytes
```

In [334…
```
print(repr(bytes(5)))  #string of five random bytes
```

```
b'\xec\xca\xd6[\x99'
```

In [335…
```
print(repr(bytes(5)))
```

```
b'\xa4V\x9f\x03\xbe'
```

47. c_[]

In [336…
```
arange(0,5)
```

Out[336]:
```
array([0, 1, 2, 3, 4])
```

In [337…
```
c_[1:5]
```

Out[337]:
```
array([[1],
       [2],
       [3],
       [4]])
```

In [338…
```
c_[1:4,5:8]
```

Out[338]:
```
array([[1, 5],
       [2, 6],
       [3, 7]])
```

In [339…
```
a = array([[1,2,3],[5,6,7]])
```

In [340…
```
c_[a,a]  # along column axis
```

Out[340]:
```
array([[1, 2, 3, 1, 2, 3],
       [5, 6, 7, 5, 6, 7]])
```

In [341…
```
c_['0',a,a]  #along row axis
```

Out[341]:
```
array([[1, 2, 3],
       [5, 6, 7],
       [1, 2, 3],
       [5, 6, 7]])
```

48. cast[] ()

In [342…
```
from numpy import *
x = arange(6)
```

In [343…
```
x
```

Out[343]:
```
array([0, 1, 2, 3, 4, 5])
```

In [344…
```
x.dtype
```

Out[344]:
```
dtype('int32')
```

In [345…
```
cast['int64'](x)
```

Out[345]:
```
array([0, 1, 2, 3, 4, 5], dtype=int64)
```

In [346…
```
cast['uint'](x)
```

Out[346]:
```
array([0, 1, 2, 3, 4, 5], dtype=uint32)
```

In [347…
```
cast[float128](x)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[347], line 1
----> 1 cast[float128](x)

NameError: name 'float128' is not defined
```

In [348…
```
cast.keys()
```

Out[348]:
```
dict_keys([<class 'numpy.timedelta64'>, <class 'numpy.clongdouble'>, <class 'nump
y.uintc'>, <class 'numpy.int64'>, <class 'numpy.void'>, <class 'numpy.int8'>, <cla
ss 'numpy.longdouble'>, <class 'numpy.uint32'>, <class 'numpy.int16'>, <class 'num
py.uint64'>, <class 'numpy.intc'>, <class 'numpy.uint8'>, <class 'numpy.float64'>,
<class 'numpy.bool_'>, <class 'numpy.float32'>, <class 'numpy.datetime64'>, <class
'numpy.str_'>, <class 'numpy.object_'>, <class 'numpy.complex64'>, <class 'numpy.u
int16'>, <class 'numpy.bytes_'>, <class 'numpy.float16'>, <class 'numpy.complex12
8'>, <class 'numpy.int32'>])
```

49. ceil()

In [349…
```
a = array([-1.4,-2.8,2,7.090])
```

In [350…
```
ceil(a)
```

Out[350]:
```
array([-1., -2.,  2.,  8.])
```

50. choose()

In [351…
```
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_o
selector.choose(choice0,choice1,choice2)
```

Out[351]:
```
array([10, 12, 34, 26])
```

In [352…
```
a = arange(4)
choose(a >= 2, (choice0, choice1)) # separate function also exists
```

Out[352]:
```
array([10, 12, 24, 26])
```

51. clip()

In [353…
```
from numpy import *
a = array([5,15,25,3,13])
a
```

Out[353]:
```
array([ 5, 15, 25,  3, 13])
```

In [354…
```
a.clip(min=10,max=20)
```

Out[354]:
```
array([10, 15, 20, 10, 13])
```

In [355…
```
clip(a,10,20)
```

Out[355]:
```
array([10, 15, 20, 10, 13])
```

In [356…
```
b = array([6,10,11,4,39])
b
```

Out[356]:
```
array([ 6, 10, 11,  4, 39])
```

In [357…
```
b.clip(min=8, max=15 )
```

Out[357]:
```
array([ 8, 10, 11,  8, 15])
```

52. column_stack()

In [358…
```
a = array([1,2,3])
b = array([4,5,6])
c = array([7,8,9])
column_stack((a,b,c))
```

Out[358]:
```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

53. compress()

In [359…
```
a = array([10,20,30,40,50])
condition = (a>25) & (a<45)
condition
```

Out[359]:
```
array([False, False,  True,  True, False])
```

In [360…
```
a.compress(condition)
```

Out[360]:
```
array([30, 40])
```

In [361…
```
a[condition]
```

Out[361]:
```
array([30, 40])
```

In [362…
```
compress(a>=30, a)
```

Out[362]:
```
array([30, 40, 50])
```

In [363…
```
b = array([[10,20,30],[40,50,60]])
b
```

Out[363]:
```
array([[10, 20, 30],
       [40, 50, 60]])
```

In [364…
```
b.compress(b.ravel() >= 22)
```

Out[364]:
```
array([30, 40, 50, 60])
```

In [365…
```
x = array([3,1,2])
y = array([50,101])
```

In [366…
```
b.compress(x>=2, axis=1)
```

Out[366]:
```
array([[10, 30],
       [40, 60]])
```

In [367…
```
b.compress(y>100, axis=0)
```

Out[367]:
```
array([[40, 50, 60]])
```

54. concatenate()

In [368…
```
x = ([[1,2],[3,4]])
y = ([[5,6],[7,8]])
```

In [369…
```
concatenate((x,y))
```

Out[369]:
```
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
```

In [370…
```
concatenate((x,y), axis=1)
```

Out[370]:
```
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

55. conj()

56. conjugate()

In [371…
```
a = array([1+2j, 5-6j])
a.conj()
```

Out[371]:
```
array([1.-2.j, 5.+6.j])
```

In [372…
```
a.conjugate()
```

Out[372]:
```
array([1.-2.j, 5.+6.j])
```

In [373…
```
conj(a)
```

Out[373]:
```
array([1.-2.j, 5.+6.j])
```

57. copy()

In [374…
```
a =array([1,2,3])
a
```

Out[374]:
```
array([1, 2, 3])
```

In [375…
```
b = a #b is reference to a
```

In [376…
```
b[1] = 4
```

In [377…
```
a
```

Out[377]:
```
array([1, 4, 3])
```

In [378…
```
a1 = array([5,6,7])
```

In [379…
```
b1 = a1.copy()
```

In [380…
```
b1
```

Out[380]:
```
array([5, 6, 7])
```

In [381…
```
b1[1]=10
b1
```

Out[381]:
```
array([ 5, 10,  7])
```

In [382…
```
a1
```

Out[382]:
```
array([5, 6, 7])
```

58. corrcoef()

In [383…
```
T = array([1.8, 2.7, 1.0, 3.7, 1.8])
P = array([2.0, 1.6, 3.1, 0.5, 1.2])
print(corrcoef([T,P]))
```

```
[[ 1.         -0.86340489]
 [-0.86340489  1.        ]]
```

In [384…
```
r = array([3.0, 7.1, 1.9, 0.7, 3.9])
data = column_stack([T,P,r])
data
```

Out[384]:
```
array([[1.8, 2. , 3. ],
       [2.7, 1.6, 7.1],
       [1. , 3.1, 1.9],
       [3.7, 0.5, 0.7],
       [1.8, 1.2, 3.9]])
```

In [385…
```
print(corrcoef([T,P,r]))
```

```
[[ 1.         -0.86340489 -0.04387512]
 [-0.86340489  1.          0.04167318]
 [-0.04387512  0.04167318  1.        ]]
```

In [386…
```
print(corrcoef(data))
```

```
[[ 1.          0.94063416  0.07381732 -0.58425442  0.9325469 ]
 [ 0.94063416  1.         -0.269061   -0.27410467  0.99973344]
 [ 0.07381732 -0.269061    1.         -0.85248451 -0.29122573]
 [-0.58425442 -0.27410467 -0.85248451  1.         -0.25182802]
 [ 0.9325469   0.99973344 -0.29122573 -0.25182802  1.        ]]
```

59. cos()

In [387…
```
cos(array([0,pi/2,pi]))
```

Out[387]:
```
array([ 1.000000e+00,  6.123234e-17, -1.000000e+00])
```

60. cov()

In [388…
```
x = array([1., 3., 7., 2.])
```

In [389…
```
variance = cov(x)  # normalised by N-1
variance
```

Out[389]:
```
array(6.91666667)
```

In [390…
```
variance = cov(x, bias =1)  # Normalised by N
variance
```

Out[390]:
```
array(5.1875)
```

In [391…
```
T = array([1.8, 2.7, 1.0, 3.7])
P = array([2.0, 1.6, 3.1, 0.5])
```

In [392…
```
cov(T,P)
```

Out[392]:
```
array([[ 1.35333333, -1.23      ],
       [-1.23      ,  1.15333333]])
```

In [393…
```
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
```

Out[393]:
```
array([[1.97583333, 3.95416667],
       [3.95416667, 8.22916667]])
```

In [394…
```
den = array([1,7,2.,3.])
```

In [395…
```
data = column_stack([T,P,den])
```

In [396…
```
data
```

Out[396]:
```
array([[1.3, 2.7, 1. ],
       [4.5, 8.7, 7. ],
       [2.8, 4.7, 2. ],
       [3.9, 8.2, 3. ]])
```

In [397…
```
cov([T,P,den])
```

Out[397]:
```
array([[1.97583333, 3.95416667, 3.15833333],
       [3.95416667, 8.22916667, 6.20833333],
       [3.15833333, 6.20833333, 6.91666667]])
```

In [398…
```
print(cov(data))
```

```
[[0.82333333 1.33666667 1.24833333 2.52166667]
 [1.33666667 4.46333333 1.76166667 4.10833333]
 [1.24833333 1.76166667 1.92333333 3.82166667]
 [2.52166667 4.10833333 3.82166667 7.72333333]]
```

61.cross()

In [399…
```
x = array([1,2,3])
y = array([4,5,6])
cross(x,y)  #vector cross product
```

Out[399]:
```
array([-3,  6, -3])
```

62. cumprod()

In [400…
```
a = array([1,2,3])
a.cumprod()
```

Out[400]:
```
array([1, 2, 6])
```

In [401…
```
a= array([[1,2,3],[4,5,6]])
```

In [402…
```
a.cumprod(dtype=float)
```

Out[402]:
```
array([  1.,   2.,   6.,  24., 120., 720.])
```

In [403…
```
cumprod(a, axis=0)
```

Out[403]:
```
array([[ 1,  2,  3],
       [ 4, 10, 18]])
```

In [404…
```
cumprod(a, axis=1)
```

Out[404]:
```
array([[  1,   2,   6],
       [  4,  20, 120]])
```

63. cumsum()

In [405…
```
a = array([1,2,3])
cumsum(a)
```

Out[405]:
```
array([1, 3, 6])
```

In [406…
```
a = array([[1,2,3],[4,5,6]])
cumsum(a)
```

Out[406]:
```
array([ 1,  3,  6, 10, 15, 21])
```

In [407…
```
cumsum(a, dtype = float)
```

Out[407]:
```
array([ 1.,  3.,  6., 10., 15., 21.])
```

In [408…
```
cumsum(a, axis=0)
```

Out[408]:
```
array([[1, 2, 3],
       [5, 7, 9]])
```

In [409…
```
cumsum(a, axis=1)
```

Out[409]:
```
array([[ 1,  3,  6],
       [ 4,  9, 15]])
```

64. delete()

In [410…
```
a = array([1,2,3,4,5])
delete(a, [2,3])
```

Out[410]:
```
array([1, 2, 5])
```

In [411…
```
a = arange(16).reshape(4,4)
a
```

Out[411]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [412…
```
delete(a, s_[1:3], axis=0)
```

Out[412]:
```
array([[ 0,  1,  2,  3],
       [12, 13, 14, 15]])
```

In [413…
```
delete(a, s_[1:3], axis=1)
```

Out[413]:
```
array([[ 0,  3],
       [ 4,  7],
       [ 8, 11],
       [12, 15]])
```

65. det()

In [414…
```
from numpy import *
from numpy.linalg import det
A = array([[1., 2.],[3., 4.]])
det(A) # determinant of square matrix
```

Out[414]:
```
-2.0000000000000004
```

In [415…
```
#3x+4y=1
#2x+y=-1
a=array([[3,4],[2,1]])
b=array([1,-1])
```

In [416…
```
from numpy.linalg import solve
solve(a,b)
```

Out[416]:
```
array([-1.,  1.])
```

66. diag()

In [417…
```
a = arange(12).reshape(4,3)
a
```

Out[417]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [418…
```
print(diag(a, k=0))
```

```
[0 4 8]
```

In [419…
```
print(diag(a, k=1))
```

```
[1 5]
```

In [420…
```
print(diag(a, k=2))
```

```
[2]
```

In [421…
```
print(diag(a, k=-1))
```

```
[ 3  7 11]
```

67. diagflat()

In [422…
```
from numpy import *
```

In [423…
```
x = array([[1,2,3],[4,5,6]])
'''Create a two-dimensional array with the flattened input as a diagonal.'''
diagflat(x)
```

Out[423]:
```
array([[1, 0, 0, 0, 0, 0],
       [0, 2, 0, 0, 0, 0],
       [0, 0, 3, 0, 0, 0],
       [0, 0, 0, 4, 0, 0],
       [0, 0, 0, 0, 5, 0],
       [0, 0, 0, 0, 0, 6]])
```

In [424…
```
print(diagflat.__doc__)
```

```
Create a two-dimensional array with the flattened input as a diagonal.

Parameters
----------
v : array_like
    Input data, which is flattened and set as the `k`-th
    diagonal of the output.
k : int, optional
    Diagonal to set; 0, the default, corresponds to the "main" diagonal,
    a positive (negative) `k` giving the number of the diagonal above
    (below) the main.

Returns
-------
out : ndarray
    The 2-D output array.

See Also
--------
diag : MATLAB work-alike for 1-D and 2-D arrays.
diagonal : Return specified diagonals.
trace : Sum along diagonals.

Examples
--------
>>> np.diagflat([[1,2], [3,4]])
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])

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

68. diagonal()

In [425…
```
a = arange(12).reshape(3,4)
```

In [426…
```
a
```

Out[426]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [427…
```
diagonal(a)
```

Out[427]:
```
array([ 0,  5, 10])
```

In [428…
```
a.diagonal(offset=1)
```

Out[428]:
```
array([ 1,  6, 11])
```

69. diff()

In [429…
```
a = array([2,7,1,9,2,3])
```

In [430…
```
diff(a)  # 1st-order differences between the elements of x
```

Out[430]:
```
array([ 5, -6,  8, -7,  1])
```

In [431…
```
# 2nd-order differences, equivalent to diff(diff(x))
diff(a,n=2)
```

Out[431]:
```
array([-11,  14, -15,   8])
```

In [432…
```
diff(a,n=3)
```

Out[432]:
```
array([ 25, -29,  23])
```

In [433…
```
q = array([[1,2,3],[7,8,9]])
```

In [434…
```
diff(q)  # 1st-order differences between the columns (default: axis=-1)
```

Out[434]:
```
array([[1, 1],
       [1, 1]])
```

In [435…
```
diff(q,n=2)
```

Out[435]:
```
array([[0],
       [0]])
```

In [436…
```
diff(q, axis=0)  # 1st-order difference between the rows
```

Out[436]:
```
array([[6, 6, 6]])
```

70. digitize()

In [437…
```
x = array([1.0,1.8,6.7,2.3,4.1])
bins = array([0.0, 0.4,1.4,5.4,10.0])
```

In [438…
```
d = digitize(x,bins)
```

In [439…
```
d
```

Out[439]:
```
array([2, 3, 4, 3, 3], dtype=int64)
```

71. dot()

In [440…
```
x = array([[1,2],[4,6]])
x.shape
```

Out[440]:
```
(2, 2)
```

In [441…
```
y = array([[9,0],[6,7]])
y.shape
```

Out[441]:
```
(2, 2)
```

In [442…
```
dot(x,y)  # matrix multiplication (2,3) x (3,2) -> (2,2)
```

Out[442]:
```
array([[21, 14],
       [72, 42]])
```

In [443…
```
a = array([[1,2,3],[1,2,3]])
a.shape
```

Out[443]:
```
(2, 3)
```

In [444…
```
b = array([[1,2],[3,4],[8,9]])
b.shape
```

Out[444]:
```
(3, 2)
```

In [445…
```
dot(a,b)
```

Out[445]:
```
array([[31, 37],
       [31, 37]])
```

In [446…
```
id(dot(a,b))
```

Out[446]:
```
1529269833136
```

In [447…
```
id(dot)
```

Out[447]:
```
1529171307184
```

In [448…
```
type(dot(a,b))
```

Out[448]:
```
numpy.ndarray
```

In [449…
```
import numpy
if id(dot) == id(numpy.core.multiarray.dot): # A way to know if you use fast blas/l
     print("Not using blas/lapack!")
```

```
Not using blas/lapack!
```

72.dsplit()

In [450…
```
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
```

Out[450]:
```
(2, 2, 4)
```

In [451…
```
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] =
```

```
(2, 2, 2) (2, 2, 2)
```

In [452…
```
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)
```

73. dstack()

In [453…
```
from numpy import *
a = array([[1,2],[3,4]]) # shapes of a and b can only differ in the 3rd dimension
b = array([[5,6],[7,8]])
dstack((a,b)) # stack arrays along a third axis (depth wise)
```

Out[453]:
```
array([[[1, 5],
        [2, 6]],

       [[3, 7],
        [4, 8]]])
```

74. dtype()

In [ ]:

In [ ]:

75. empty()

In [454…
```
from numpy import *
empty(3) # uninitialized array, size=3, dtype = float
```

Out[454]:
```
array([0.e+000, 1.e-323, 2.e-323])
```

In [455…
```
empty((2,3),int) # uninitialized array, dtype = int
```

Out[455]:
```
array([[0, 0, 2],
       [0, 4, 0]])
```

76. empty_like()

In [456…
```
from numpy import *
a = array([[1,2,3],[4,5,6]])
empty_like(a) # uninitialized array with the same shape and datatype as 'a'
```

Out[456]:
```
array([[  29228576,        356,          0],
       [         0,          1, -218103808]])
```

77. expand_dims()

In [457…
```
from numpy import *
x = array([1,2])
expand_dims(x,axis=0) # Equivalent to x[newaxis,:] or x[None] or x[newaxis]
```

Out[457]:
```
array([[1, 2]])
```

In [458…
```
expand_dims(x,axis=1) # Equivalent to x[:,newaxis]
```

Out[458]:
```
array([[1],
       [2]])
```

78. eye()

In [459…
```
from numpy import *
eye(3,4,0,dtype=float) # a 3x4 matrix containing zeros except for the 0th diagonal
```

Out[459]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.]])
```

In [460…
```
eye(3,4,1,dtype=float) # a 3x4 matrix containing zeros except for the 1st diagonal
```

Out[460]:
```
array([[0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

79. fft()

In [461…
```
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
```

Out[461]:
```
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])
```

In [462…
```
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'
```

Out[462]:
```
array([ 0.  ,  1.25,  2.5 ,  3.75, -5.  , -3.75, -2.5 , -1.25])
```

80. fftfreq()

In [463…
```
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
```

Out[463]:
```
array([ 0.  ,  1.25,  2.5 ,  3.75, -5.  , -3.75, -2.5 , -1.25])
```

In [464…
```
fftshift(freq) # freqs in ascending order
```

Out[464]:
```
array([-5.  , -3.75, -2.5 , -1.25,  0.  ,  1.25,  2.5 ,  3.75])
```

81. fftshift()

In [465…
```
from numpy import *
from numpy.fft import *
```

In [466…
```
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
```

Out[466]:
```
array([ 0.  ,  1.25,  2.5 ,  3.75, -5.  , -3.75, -2.5 , -1.25])
```

In [467…
```
freq = fftshift(freq)  # freqs in ascending order
print(freq)
'''Shift the zero-frequency component to the center of the spectrum.'''
```

Out[467]:
```
[-5.   -3.75 -2.5  -1.25  0.    1.25  2.5   3.75]
'Shift the zero-frequency component to the center of the spectrum.'
```

In [468…
```
fourier = fftshift(fourier)
```

In [469…
```
freq = ifftshift(freq) # undo previous frequency shift
fourier = ifftshift(fourier) # undo previous fourier shift
```

In [470…
```
freq
```

Out[470]:
```
array([ 0.  ,  1.25,  2.5 ,  3.75, -5.  , -3.75, -2.5 , -1.25])
```

In [471…
```
fourier
```

Out[471]:
```
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])
```

82. fill()

In [472…
```
from numpy import *
```

In [473…
```
a = arange(4, dtype=int)
a
```

Out[473]:
```
array([0, 1, 2, 3])
```

In [474…
```
a.fill(7) # replace all elements with the number 7
a
```

Out[474]:
```
array([7, 7, 7, 7])
```

In [475…
```
a.fill(6.5)
a
```

Out[475]:
```
array([6, 6, 6, 6])
```

83. finfo()

In [476…
```
f = finfo(float)
f
```

Out[476]:
```
finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.7976931348623157e+308, dtype=float64)
```

In [477…
```
f.nmant, f.nexp  #nr bits in the mantissa and in the exponent
```

Out[477]:
```
(52, 11)
```

In [478…
```
f.machep
```

Out[478]:
```
-52
```

In [479…
```
f.eps
```

Out[479]:
```
2.220446049250313e-16
```

In [480…
```
f.precision
```

Out[480]:
```
15
```

In [481…
```
f.resolution
```

Out[481]:
```
1e-15
```

In [482…
```
f.negep
```

Out[482]:
```
-53
```

In [483…
```
f.epsneg
```

Out[483]:
```
1.1102230246251565e-16
```

In [484…
```
f.minexp
```

Out[484]:
```
-1022
```

In [485…
```
f.tiny
```

Out[485]:
```
2.2250738585072014e-308
```

In [486…
```
f.maxexp
```

Out[486]:
```
1024
```

In [487…
```
f.min, f.max
```

Out[487]:
```
(-1.7976931348623157e+308, 1.7976931348623157e+308)
```

84. fix()

In [488…
```
a = array([-1.7, -1.0, -0.2, 0.2, 1.5, 1.7])
a
```

Out[488]:
```
array([-1.7, -1. , -0.2,  0.2,  1.5,  1.7])
```

In [489…
```
fix(a)
```

Out[489]:
```
array([-1., -1., -0.,  0.,  1.,  1.])
```

85. flat()

In [490…
```
a = array([[10,20],[30,40]])
a
```

Out[490]:
```
array([[10, 20],
       [30, 40]])
```

In [491…
```
iter = a.flat
```

In [492…
```
iter
```

Out[492]:
```
<numpy.flatiter at 0x1640378ce90>
```

86. flatten()

In [493…
```
a = array([[[1,2]],[[3,4]]])
```

In [494…
```
a
```

Out[494]:
```
array([[[1, 2]],

       [[3, 4]]])
```

In [495…
```
a.shape
```

Out[495]:
```
(2, 1, 2)
```

In [496…
```
type(a)
```

Out[496]:
```
numpy.ndarray
```

In [497…
```
a[0]
```

Out[497]:
```
array([[1, 2]])
```

In [498…
```
a[1]
```

Out[498]:
```
array([[3, 4]])
```

In [499…
```
a[1][0]
```

Out[499]:
```
array([3, 4])
```

In [500…
```
a[1][0][0]
```

Out[500]:
```
3
```

In [501…
```
a[1][0][1]
```

Out[501]:
```
4
```

In [502…
```
a[0][0][1]
```

Out[502]:
```
2
```

In [503…
```
print(a)
```

```
[[[1 2]]

 [[3 4]]]
```

In [504…
```
b=a.flatten()
```

In [505…
```
b
```

Out[505]:
```
array([1, 2, 3, 4])
```

87. fliplr()

In [506…
```
a = arange(12).reshape(4,3)
```

In [507…
```
a
```

Out[507]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [508…
```
fliplr(a)  #flip left right
```

Out[508]:
```
array([[ 2,  1,  0],
       [ 5,  4,  3],
       [ 8,  7,  6],
       [11, 10,  9]])
```

88. flipup()

In [509…
```
a = arange(12).reshape(4,3)
a
```

Out[509]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [510…
```
flipud(a)
```

Out[510]:
```
array([[ 9, 10, 11],
       [ 6,  7,  8],
       [ 3,  4,  5],
       [ 0,  1,  2]])
```

89. floor()

In [511…
```
a = array([-1.7, -1.2, 0.4, 1.0, 2.5, 3.7])
a
```

Out[511]:
```
array([-1.7, -1.2,  0.4,  1. ,  2.5,  3.7])
```

In [512…
```
floor(a)
```

Out[512]:
```
array([-2., -2.,  0.,  1.,  2.,  3.])
```

90. fromarrays()

In [513…
```
x = array(['Smith', 'warner', 'alexa'])   # datatype is string
y = array(['F','M','F'])  # datatype is single character
z = array([10,20,45])  #datatype is integer
data = rec.fromarrays([x,y,z], names = 'surname, gender, age')  #convert to record
data[0]
```

Out[513]:
```
('Smith', 'F', 10)
```

In [514…
```
data.age
```

Out[514]:
```
array([10, 20, 45])
```

91. frombuffer()

In [ ]:

In [ ]:

92. fromfile()

In [515…
```
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, exponer
fromfile('myfile.dat', dtype=float)
```

Out[515]:
```
array([2., 4., 6., 8.])
```

In [516…
```
fromfile('myfile.txt', dtype=float, sep='\n')
```

Out[516]:
```
array([2., 4., 6., 8.])
```

93. fromfunction()

In [517…
```
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]
```

Out[517]:
```
array([[0., 1., 4.],
       [1., 2., 5.],
       [4., 5., 8.]])
```

94. fromiter()

In [518…
```
from numpy import *
import itertools
mydata = [[55.5, 40],[60.5, 70]] # List of lists
mydescriptor = {'names': ('weight','age'), 'formats': (float32, int32)} # Descript
myiterator = itertools.imap(tuple,mydata) # Clever way of putting list of lists in
# of tuples. E.g.: myiterator.next() == (55.5, 40.)
a = fromiter(myiterator, dtype = mydescriptor)
a
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[518], line 5
      3 mydata = [[55.5, 40],[60.5, 70]] # List of lists
      4 mydescriptor = {'names': ('weight','age'), 'formats': (float32, int32)} #
Descriptor of the data
----> 5 myiterator = itertools.imap(tuple,mydata) # Clever way of putting list of
lists into iterator
      6 # of tuples. E.g.: myiterator.next() == (55.5, 40.)
      7 a = fromiter(myiterator, dtype = mydescriptor)

AttributeError: module 'itertools' has no attribute 'imap'
```

95. generic

In [519…
```
from numpy import *
numpyscalar = string_('7') # Convert to numpy scalar
numpyscalar # Looks like a build-in scalar...
```

Out[519]:
```
b'7'
```

In [520…
```
type(numpyscalar) # ... but it isn't
```

Out[520]:
```
numpy.bytes_
```

In [521…
```
buildinscalar = '7' # Build-in python scalar
type(buildinscalar)
```

Out[521]:
```
str
```

In [522…
```
isinstance(numpyscalar, generic) # Check if scalar is a NumPy one
isinstance(buildinscalar, generic) # Example on how to recognize NumPy scalars
```

Out[522]:
```
False
```

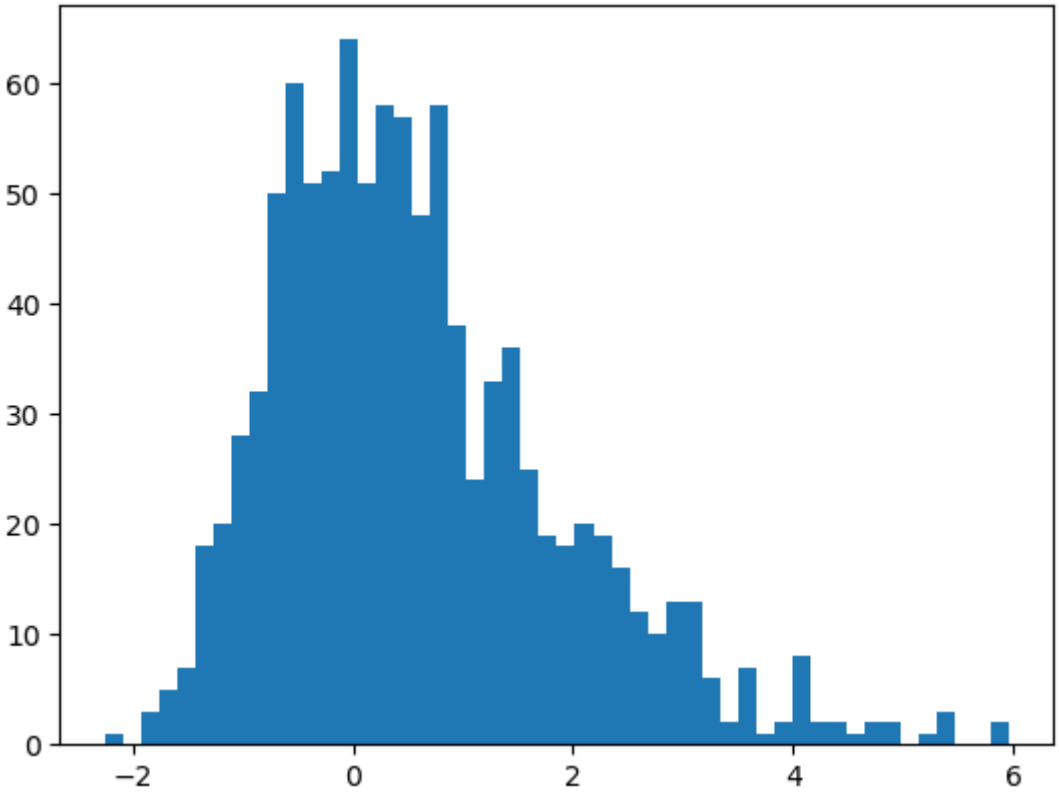
96. gumbel()

In [523…
```
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
```

Out[523]:
```
array([[-1.21577504,  1.316322  ,  1.13088299],
       [ 5.23509762,  1.08430591,  0.53347248]])
```

In [524…
```
from pylab import * # histogram plot example
hist(gumbel(0,1,(1000)), 50)
```

Out[524]:
```
(array([ 1.,  0.,  3.,  5.,  7., 18., 20., 28., 32., 50., 60., 51., 52.,
        64., 51., 58., 57., 48., 58., 38., 24., 33., 36., 25., 19., 18.,
        20., 19., 16., 12., 10., 13., 13.,  6.,  2.,  7.,  1.,  2.,  8.,
         2.,  2.,  1.,  2.,  2.,  0.,  1.,  3.,  0.,  0.,  2.]),
 array([-2.24774704, -2.08364095, -1.91953486, -1.75542876, -1.59132267,
        -1.42721658, -1.26311048, -1.09900439, -0.9348983 , -0.7707922 ,
        -0.60668611, -0.44258002, -0.27847392, -0.11436783,  0.04973826,
         0.21384436,  0.37795045,  0.54205654,  0.70616264,  0.87026873,
         1.03437482,  1.19848092,  1.36258701,  1.5266931 ,  1.6907992 ,
         1.85490529,  2.01901138,  2.18311748,  2.34722357,  2.51132966,
         2.67543576,  2.83954185,  3.00364794,  3.16775404,  3.33186013,
         3.49596622,  3.66007232,  3.82417841,  3.9882845 ,  4.1523906 ,
         4.31649669,  4.48060278,  4.64470888,  4.80881497,  4.97292106,
         5.13702716,  5.30113325,  5.46523934,  5.62934544,  5.79345153,
         5.95755762]),
 <BarContainer object of 50 artists>)
```



97. histogram()

In [525…
```
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
```

Out[525]:
```
(array([2, 3, 1, 4], dtype=int64), array([ 0. ,  1. ,  2.5,  4. , 10. ]))
```

In [526…
```
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
```

```
#  2 number fall into bin [ 0.0 , 1.0 [
#  3 number fall into bin [ 1.0 , 2.5 [
#  1 number fall into bin [ 2.5 , 4.0 [
#  4 number fall into bin [ 4.0 , 10.0 [
```

Out[526]:
```
(array([4, 2, 2, 1, 1], dtype=int64), array([ 0.,  2.,  4.,  6.,  8., 10.]))
```

In [527…
```
N,bins = histogram(x,5,range=(0.0, 10.0)) # normalize histogram, i.e. divide by le
N,bins
```

Out[527]:
```
(array([4, 2, 2, 1, 1], dtype=int64), array([ 0.,  2.,  4.,  6.,  8., 10.]))
```

98. hsplit()

In [528…
```
from numpy import *
a = array([[1,2,3,4],[5,6,7,8]])
hsplit(a,2) # split, column-wise, in 2 equal parts
```

Out[528]:
```
[array([[1, 2],
        [5, 6]]),
 array([[3, 4],
        [7, 8]])]
```

In [529…
```
hsplit(a,[1,2]) # split before column 1 and before column 2
```

Out[529]:
```
[array([[1],
        [5]]),
 array([[2],
        [6]]),
 array([[3, 4],
        [7, 8]])]
```

99. hstack()

In [530…
```
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
```

Out[530]:
```
array([[1, 3, 4, 1],
       [2, 5, 6, 2]])
```

100. hypot()

In [531…
```
from numpy import *
hypot(3.,4.) # hypotenuse: sqrt(3**2 + 4**2) = 5
```

Out[531]:
```
5.0
```

In [532…
```
z = array([2+3j, 3+4j])
hypot(z.real, z.imag) # norm of complex numbers
```

Out[532]:
```
array([3.60555128, 5.        ])
```

101. identity()

In [533…
```
from numpy import *
```

In [534…
```
identity(3, int)
```

Out[534]:
```
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
```

102. ifft()

In [535…
```
from numpy.fft import *
```

In [536…
```
signal = array([-2., 8., -6., 4., 1., 0., 3., 5.])
fourier = fft(signal)
ifft(fourier) # Inverse fourier transform
```

Out[536]:
```
array([-2.00000000e+00+0.j,  8.00000000e+00+0.j, -6.00000000e+00+0.j,
        4.00000000e+00+0.j,  1.00000000e+00+0.j, -1.77635684e-15+0.j,
        3.00000000e+00+0.j,  5.00000000e+00+0.j])
```

In [537…
```
allclose(signal.astype(complex), ifft(fft(signal))) # ifft(fft()) = original signa
```

Out[537]:
```
True
```

In [538…
```
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
```

103. imag()

In [539…
```
a = array([1+3j,4+1j,2+9j])
```

In [540…
```
a.imag
```

Out[540]:
```
array([3., 1., 9.])
```

In [541…
```
a.imag = 9
```

In [542…
```
a
```

Out[542]:
```
array([1.+9.j, 4.+9.j, 2.+9.j])
```

In [543…
```
a.imag = array([6,7,5])
```

In [544…
```
a
```

Out[544]:
```
array([1.+6.j, 4.+7.j, 2.+5.j])
```

104. index_exp[]

In [545…
```
myslice = index_exp[2:4,...,4,::-1] # myslice could now be passed to a function, fo
print(myslice)
```

```
(slice(2, 4, None), Ellipsis, 4, slice(None, None, -1))
```

In [546…
```
#A nicer way to build up index tuples for arrays.
```

105. indices[]

In [547…
```
indices((2,3))
```

Out[547]:
```
array([[[0, 0, 0],
        [1, 1, 1]],

       [[0, 1, 2],
        [0, 1, 2]]])
```

In [548…
```
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((2,2))
a[i,j]
```

Out[548]:
```
array([[ 0,  1],
       [10, 11]])
```

106. inf

In [549…
```
from numpy import *
exp(array([1000.])) # inf = infinite = number too large to represent, machine depe
```

```
C:\Users\nithi\AppData\Local\Temp\ipykernel_20556\2966194585.py:2: RuntimeWarning:
overflow encountered in exp
  exp(array([1000.])) # inf = infinite = number too large to represent, machine de
pendent
```

Out[549]:
```
array([inf])
```

In [550…
```
x = array([2,-inf,1,inf])
isfinite(x) # show which elements are not nan/inf/-inf
```

Out[550]:
```
array([ True, False,  True, False])
```

In [551…
```
isinf(x)
```

Out[551]:
```
array([False,  True, False,  True])
```

In [552…
```
isposinf(x)
```

Out[552]:
```
array([False, False, False,  True])
```

In [553…
```
isneginf(x)
```

Out[553]:
```
array([False,  True, False, False])
```

In [554…
```
nan_to_num(x)
```

Out[554]:
```
array([ 2.00000000e+000, -1.79769313e+308,  1.00000000e+000,
        1.79769313e+308])
```

107. inner()

In [555…
```
x = array([1,2,3])
y = array([4,5,6])
inner(x,y)  #1x4+2x5+3x6=32
```

Out[555]:
```
32
```

108. insert()

In [556…
```
a = array([10,20,30,40])
```

In [557…
```
insert(a,[1,3],50)
```

Out[557]:
```
array([10, 50, 20, 30, 50, 40])
```

In [558…
```
insert(a,[1,3],[50,60])
```

Out[558]:
```
array([10, 50, 20, 30, 60, 40])
```

In [559…
```
a = array([[10,20,30],[40,50,60],[70,80,90]])
```

In [560…
```
insert(a,[1,2],100,axis=1)
```

Out[560]:
```
array([[ 10, 100,  20, 100,  30],
       [ 40, 100,  50, 100,  60],
       [ 70, 100,  80, 100,  90]])
```

In [561…
```
insert(a, [1,2], [[100],[200]], axis=0)
```

Out[561]:
```
array([[ 10,  20,  30],
       [100, 100, 100],
       [ 40,  50,  60],
       [200, 200, 200],
       [ 70,  80,  90]])
```

109. inv()

In [562…
```
from numpy.linalg import inv
```

In [563…
```
a = array([[3,4,1],[6,7,8],[1,9,0]])
```

In [564…
```
print(a)
```

```
[[3 4 1]
 [6 7 8]
 [1 9 0]]
```

In [565…
```
inva = inv(a)
```

In [566…
```
inva
```

Out[566]:
```
array([[ 0.52554745, -0.06569343, -0.18248175],
       [-0.05839416,  0.00729927,  0.13138686],
       [-0.34306569,  0.16788321,  0.02189781]])
```

In [567…
```
eye(3)
```

Out[567]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [568…
```
dot(a, inva)
```

Out[568]:
```
array([[ 1.00000000e+00,  0.00000000e+00,  2.77555756e-17],
       [ 0.00000000e+00,  1.00000000e+00,  8.32667268e-17],
       [ 4.16333634e-17, -3.46944695e-18,  1.00000000e+00]])
```

110. iscomplexobj()

In [569…
```
a = array([1,2,3.j])
```

In [570…
```
iscomplexobj(a)
```

Out[570]:
```
True
```

In [571…
```
a = array([1,2,3])
```

In [572…
```
iscomplexobj(a)
```

Out[572]:
```
False
```

In [573…
```
a = array([1,2,7], dtype=complex)
```

In [574…
```
a
```

Out[574]:
```
array([1.+0.j, 2.+0.j, 7.+0.j])
```

In [575…
```
iscomplexobj(a)
```

Out[575]:
```
True
```

111. item()

In [576…
```
from numpy import *
a = array([5])
type(a[0])
```

Out[576]:
```
numpy.int32
```

In [577…
```
a.item() # Conversion of array of size 1 to Python scalar
```

Out[577]:
```
5
```

In [578…
```
type(a.item())
```

Out[578]:
```
int
```

In [579…
```
b = array([2,3,4])
b[1].item() # Conversion of 2nd element to Python scalar
```

Out[579]:
```
3
```

In [580…
```
type(b[1].item())
```

Out[580]:
```
int
```

In [581…
```
b.item(2) # Return 3rd element converted to Python scalar
```

Out[581]:
```
4
```

In [582…
```
print(type(b.item(2)))
print(type(b[2])) # b[2] is slower than b.item(2), and there is no conversion
```

```
<class 'int'>
<class 'numpy.int32'>
```

112. ix_()

In [583…
```
from numpy import *
a = arange(9).reshape(3,3)
print(a)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

In [584…
```
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]]))
```

In [585…
```
print(a[indices])
```

```
[[1 2 0]
 [4 5 3]
 [7 8 6]]
```

In [586…
```
# 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]]]
```

113. lexsort()

In [587…
```
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
```

Out[587]:
```
array([5, 0, 1, 3, 2, 4], dtype=int64)
```

In [588…
```
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
```

In [589…
```
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.]]
```

In [590…
```
indices = lexsort(a) # Sort on last row, then on 2nd last row, etc.
a.take(indices, axis=-1)
```

Out[590]:
```
array([[5241., 1023., 5202., 1671., 6230., 1682.],
       [  30.,   50.,   20.,   60.,   70.,   15.],
       [  40.,   40.,   42.,   60.,   60.,   98.]])
```

114. linspace()

In [591…
```
from numpy import *
linspace(0,5,num=6) # 6 evenly spaced numbers between 0 and 5 incl.
```

Out[591]:
```
array([0., 1., 2., 3., 4., 5.])
```

In [592…
```
linspace(0,5,num=10) # 10 evenly spaced numbers between 0 and 5 incl.
```

Out[592]:
```
array([0.        , 0.55555556, 1.11111111, 1.66666667, 2.22222222,
       2.77777778, 3.33333333, 3.88888889, 4.44444444, 5.        ])
```

In [593…
```
linspace(0,5,num=10,endpoint=False) # 10 evenly spaced numbers between 0 and 5 EXCl
```

Out[593]:
```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
```

In [594…
```
stepsize = linspace(0,5,num=10,endpoint=False,retstep=True) # besides the usual ar
stepsize
```

Out[594]:
```
(array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5]), 0.5)
```

In [595…
```
myarray, stepsize = linspace(0,5,num=10,endpoint=False,retstep=True)
stepsize
```

Out[595]:
```
0.5
```

115. loadtxt()

In [596…
```
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 c
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
data = loadtxt("myfile.txt", dtype = int) # file contains integers instead of floa
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[596], line 6
      3 #t,z = data[:,0], data[:,3] # data is 2D numpy array
      5 t,x,y,z = loadtxt("myfile.txt", unpack=True) # to unpack all columns
----> 6 t,z = loadtxt("myfile.txt", usecols = (0,3), unpack=True) # to select just
a few columns
      7 data = loadtxt("myfile.txt", skiprows = 7) # to skip 7 rows from top of fi
le
      8 data = loadtxt("myfile.txt", comments = '!') # use '!' as comment char ins
tead of '#'

File ~\anaconda3\conda\Lib\site-packages\numpy\lib\npyio.py:1373, in loadtxt(fnam
e, dtype, comments, delimiter, converters, skiprows, usecols, unpack, ndmin, encod
ing, max_rows, quotechar, like)
   1370 if isinstance(delimiter, bytes):
   1371     delimiter = delimiter.decode('latin1')
-> 1373 arr = _read(fname, dtype=dtype, comment=comment, delimiter=delimiter,
   1374             converters=converters, skiplines=skiprows, usecols=usecols,
   1375             unpack=unpack, ndmin=ndmin, encoding=encoding,
   1376             max_rows=max_rows, quote=quotechar)
   1378 return arr

File ~\anaconda3\conda\Lib\site-packages\numpy\lib\npyio.py:1016, in _read(fname,
delimiter, comment, quote, imaginary_unit, usecols, skiplines, max_rows, converter
s, ndmin, unpack, dtype, encoding)
   1013     data = _preprocess_comments(data, comments, encoding)
   1015 if read_dtype_via_object_chunks is None:
-> 1016     arr = _load_from_filelike(
   1017         data, delimiter=delimiter, comment=comment, quote=quote,
   1018         imaginary_unit=imaginary_unit,
   1019         usecols=usecols, skiplines=skiplines, max_rows=max_rows,
   1020         converters=converters, dtype=dtype,
   1021         encoding=encoding, filelike=filelike,
   1022         byte_converters=byte_converters)
   1024 else:
   1025     # This branch reads the file into chunks of object arrays and then
   1026     # casts them to the desired actual dtype.  This ensures correct
   1027     # string-length and datetime-unit discovery (like `arr.astype()`).
   1028     # Due to chunking, certain error reports are less clear, currently.
   1029     if filelike:

ValueError: invalid column index 3 at row 1 with 1 columns
```

116. logical_and()

In [597…
```
from numpy import *
logical_and(array([0,0,1,1]), array([0,1,0,1]))
```

Out[597]:
```
array([False, False, False,  True])
```

In [598…
```
logical_and(array([False,False,True,True]), array([False,True,False,True]))
```

Out[598]:
```
array([False, False, False,  True])
```

117. logical_not()

In [599…
```
logical_not(array([0,1]))
```

Out[599]:
```
array([ True, False])
```

In [600…
```
logical_not(array([False, True]))
```

Out[600]:
```
array([ True, False])
```

118. logical_or()

In [601…
```
logical_or(array([0,0,1,1]), array([0,1,0,1]))
logical_or(array([False,False,True,True]), array([False,True,False,True]))
```

Out[601]:
```
array([False,  True,  True,  True])
```

119. logical_xor()

In [602…
```
logical_xor(array([0,0,1,1]), array([0,1,0,1]))
logical_xor(array([False,False,True,True]), array([False,True,False,True]))
```

Out[602]:
```
array([False,  True,  True, False])
```

120. logspace()

In [603…
```
from numpy import *
logspace(-2, 3, num = 6) # 6 evenly spaced pts on a logarithmic scale, from 10^{-2
```

Out[603]:
```
array([1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+02, 1.e+03])
```

In [604…
```
logspace(-2, 3, num = 10) # 10 evenly spaced pts on a logarithmic scale, from 10^{
```

Out[604]:
```
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])
```

In [605…
```
logspace(-2, 3, num = 6, endpoint=False) # 6 evenly spaced pts on a logarithmic sc
```

Out[605]:
```
array([1.00000000e-02, 6.81292069e-02, 4.64158883e-01, 3.16227766e+00,
       2.15443469e+01, 1.46779927e+02])
```

In [606…
```
exp(linspace(log(0.01), log(1000), num=6, endpoint=False)) # for comparison
```

Out[606]:
```
array([1.00000000e-02, 6.81292069e-02, 4.64158883e-01, 3.16227766e+00,
       2.15443469e+01, 1.46779927e+02])
```

121. 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 = p0 *f0(t)* + *p1* f1(t) + ... + pN-1 * fN-1(t) + noise

where the pi are the parameters you want to obtain through fitting and the fi(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:

In [607…
```
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) = p0 *sin(2.*pit0.6) + p1 *cos(2.*pit0.6), where p0 and p1 are the unknown fit parameters. Here we go:

In [608…
```
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
```

```
C:\Users\nithi\AppData\Local\Temp\ipykernel_20556\876336982.py:6: FutureWarning: `
rcond` parameter will change to the default of machine precision times ``max(M, N)
`` where M and N are the input matrix dimensions.
To use the future default and silence this warning we advise to pass `rcond=None`,
to keep using the old, explicitly pass `rcond=-1`.
  (p, residuals, rank, s) = lstsq(A,y)
```

Out[608]:
```
array([1.97577955, 2.79729667])
```

In [609…
```
residuals # sum of the residuals: sum((p[0] * A[:,0] + p[1] * A[:,1] - y)**2)
```

Out[609]:
```
array([211.00248166])
```

In [610…
```
rank # rank of the array A
```

Out[610]:
```
2
```

In [611…
```
s # singular values of A
```

Out[611]:
```
array([10., 10.])
```

122. mat()

In [612…
```
from numpy import *
mat('1 3 4; 5 6 9') # matrices are always 2-dimensional
```

Out[612]:
```
matrix([[1, 3, 4],
        [5, 6, 9]])
```

In [613…
```
a = array([[1,2],[3,4]])
m = mat(a) # convert 2-d array to matrix
m
```

Out[613]:
```
matrix([[1, 2],
        [3, 4]])
```

In [614…
```
a[0] # result is 1-dimensional
```

Out[614]:
```
array([1, 2])
```

In [615…
```
m[0] # result is 2-dimensional
```

Out[615]:
```
matrix([[1, 2]])
```

In [616…
```
a.ravel() # result is 1-dimensional
```

Out[616]:
```
array([1, 2, 3, 4])
```

In [617…
```
m.ravel() # result is 2-dimensional
```

Out[617]:
```
matrix([[1, 2, 3, 4]])
```

In [618…
```
a*a # element-by-element multiplication
```

Out[618]:
```
array([[ 1,  4],
       [ 9, 16]])
```

In [619…
```
m*m # (algebraic) matrix multiplication
```

Out[619]:
```
matrix([[ 7, 10],
        [15, 22]])
```

In [620…
```
a**3 # element-wise power
```

Out[620]:
```
array([[ 1,  8],
       [27, 64]], dtype=int32)
```

In [621…
```
m**3 # matrix multiplication m*m*m
```

Out[621]:
```
matrix([[ 37,  54],
        [ 81, 118]])
```

In [622…
```
m.T # transpose of the matrix
```

Out[622]:
```
matrix([[1, 3],
        [2, 4]])
```

In [623…
```
m.H # conjugate transpose (differs from .T for complex matrices)
```

Out[623]:
```
matrix([[1, 3],
        [2, 4]])
```

In [624…
```
m.I # inverse matrix
```

Out[624]:
```
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
```

123. matrix()

In [625…
```
matrix('1 2 3;4 5 6')
```

Out[625]:
```
matrix([[1, 2, 3],
        [4, 5, 6]])
```

124. max()

In [626…
```
a = array([10,20,30])
```

In [627…
```
a.max()
```

Out[627]:
```
30
```

In [628…
```
a = array([[10,20,30],[40,50,60]])
```

In [629…
```
a.max()
```

Out[629]:
```
60
```

In [630…
```
a.max(axis=0)
```

Out[630]:
```
array([40, 50, 60])
```

In [631…
```
a.max(axis=1)
```

Out[631]:
```
array([30, 60])
```

In [632…
```
max(a)
```

Out[632]:
```
60
```

125. maximum()

In [633…
```
a = array([1,2,7])
b = array([4,5,6])
```

In [634…
```
maximum(a,b)
```

Out[634]:
```
array([4, 5, 7])
```

In [635…
```
max(a.tolist(),b.tolist()) # standard Python function does not give the same!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[635], line 1
----> 1 max(a.tolist(),b.tolist())

File ~\anaconda3\conda\Lib\site-packages\numpy\core\fromnumeric.py:2810, in max(a,
axis, out, keepdims, initial, where)
   2692 @array_function_dispatch(_max_dispatcher)
   2693 @set_module('numpy')
   2694 def max(a, axis=None, out=None, keepdims=np._NoValue, initial=np._NoValue,
   2695          where=np._NoValue):
   2696     """
   2697     Return the maximum of an array or maximum along an axis.
   2698
   (...)
   2808     5
   2809     """
-> 2810     return _wrapreduction(a, np.maximum, 'max', axis, None, out,
   2811                           keepdims=keepdims, initial=initial, where=where)

File ~\anaconda3\conda\Lib\site-packages\numpy\core\fromnumeric.py:88, in _wrapred
uction(obj, ufunc, method, axis, dtype, out, **kwargs)
     85         else:
     86             return reduction(axis=axis, out=out, **passkwargs)
---> 88 return ufunc.reduce(obj, axis, dtype, out, **passkwargs)

TypeError: 'list' object cannot be interpreted as an integer
```

126. mean()

In [636…
```
a = array([1,7,9])
```

In [637…
```
a.mean()
```

Out[637]:
```
5.666666666666667
```

In [638…
```
a = array([[1,2,3],[6,9,2]])
```

In [639…
```
a.mean()
```

Out[639]:
```
3.8333333333333335
```

In [640…
```
a.mean(axis=0)
```

Out[640]:
```
array([3.5, 5.5, 2.5])
```

In [641…
```
a.mean(axis=1)
```

Out[641]:
```
array([2.        , 5.66666667])
```

127. median()

In [642…
```
a = array([1,8,2,6,0,3])
```

In [643…
```
median(a)
```

Out[643]:
```
2.5
```

In [644…
```
a = array([1,8,2,6,3])
```

In [645…
```
median(a)
```

Out[645]:
```
3.0
```

128. mgrid()

In [646…
```
m = mgrid[1:4,6:9]
```

In [647…
```
m
```

Out[647]:
```
array([[[1, 1, 1],
        [2, 2, 2],
        [3, 3, 3]],

       [[6, 7, 8],
        [6, 7, 8],
        [6, 7, 8]]])
```

In [648…
```
m[0,2,0]
```

Out[648]:
```
3
```

129. min()

In [649…
```
a =array([12,3,4,11,10])
```

In [650…
```
min(a)
```

Out[650]:
```
3
```

In [651…
```
a.min()
```

Out[651]:
```
3
```

130. minimum()

In [652…
```
a = array([1,12,31])
b = array([10,20,3])
```

In [653…
```
minimum(a,b)
```

Out[653]:
```
array([ 1, 12,  3])
```

131. iscomplex()

In [654…
```
import numpy as np
```

In [655…
```
a = np.array([1,2,3.j])
```

In [656…
```
np.iscomplex(a)
```

Out[656]:
```
array([False, False,  True])
```

132. multiply()

In [657…
```
from numpy import *
multiply(array([1,2,3]), array([4,5,6]))
```

Out[657]:
```
array([ 4, 10, 18])
```

133. nan

In [658…
```
sqrt(array([-1.0]))
```

```
C:\Users\nithi\AppData\Local\Temp\ipykernel_20556\494075515.py:1: RuntimeWarning:
invalid value encountered in sqrt
  sqrt(array([-1.0]))
```

Out[658]:
```
array([nan])
```

In [659…
```
x = array([2, nan, 1])
```

In [660…
```
isnan(x)
```

Out[660]:
```
array([False,  True, False])
```

In [661…
```
isfinite(x)
```

Out[661]:
```
array([ True, False,  True])
```

In [662…
```
nansum(x)
```

Out[662]:
```
3.0
```

In [663…
```
nanmax(x)
```

Out[663]:
```
2.0
```

In [664…
```
nanmin(x)
```

Out[664]:
```
1.0
```

In [665…
```
nanargmin(x)
```

Out[665]:
```
2
```

In [666…
```
nanargmax(x)
```

Out[666]:
```
0
```

In [667…
```
nan_to_num(x)
```

Out[667]:
```
array([2., 0., 1.])
```

134. ndenumerate()

In [668…
```
a = arange(9).reshape(3,3) + 10
```

In [669…
```
a
```

Out[669]:
```
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
```

In [670…
```
b = ndenumerate(a)
```

In [671…
```
for position,value in b:
    print(position, value)
```

```
(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
```

In [672…
```
b = ndenumerate(a)
for position,value in b: print(position,value) # position is the N-dimensional ind
```

```
(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
```

135. ndim

In [673…
```
a = arange(12).reshape(4,3)
```

In [674…
```
a
```

Out[674]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [675…
```
a.ndim  # a has 2 axes
```

Out[675]:
```
2
```

In [676…
```
a.shape = (2,2,3)
```

In [677…
```
a
```

Out[677]:
```
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
```

In [678…
```
a.ndim
```

Out[678]:
```
3
```

In [679…
```
len(a.shape)
```

Out[679]:
```
3
```

136. ndindex()

In [680…
```
for index in ndindex(4,3,2):
    print(index)
```

```
(0, 0, 0)
(0, 0, 1)
(0, 1, 0)
(0, 1, 1)
(0, 2, 0)
(0, 2, 1)
(1, 0, 0)
(1, 0, 1)
(1, 1, 0)
(1, 1, 1)
(1, 2, 0)
(1, 2, 1)
(2, 0, 0)
(2, 0, 1)
(2, 1, 0)
(2, 1, 1)
(2, 2, 0)
(2, 2, 1)
(3, 0, 0)
(3, 0, 1)
(3, 1, 0)
(3, 1, 1)
(3, 2, 0)
(3, 2, 1)
```

137. newaxis

In [681…
```
x = arange(3)
```

In [682…
```
x
```

Out[682]:
```
array([0, 1, 2])
```

In [683…
```
x[:,newaxis]
```

Out[683]:
```
array([[0],
       [1],
       [2]])
```

In [684…
```
x[:,newaxis,newaxis]
```

Out[684]:
```
array([[[0]],

       [[1]],

       [[2]]])
```

In [685…
```
x[:,newaxis]*x
```

Out[685]:
```
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4]])
```

In [686…
```
y = arange(3,6)
```

In [687…
```
y
```

Out[687]:
```
array([3, 4, 5])
```

In [688…
```
x[:,newaxis]*y
```

Out[688]:
```
array([[ 0,  0,  0],
       [ 3,  4,  5],
       [ 6,  8, 10]])
```

In [689…
```
x.shape
```

Out[689]:
```
(3,)
```

In [690…
```
x[newaxis,:].shape
```

Out[690]:
```
(1, 3)
```

In [691…
```
x[:,newaxis].shape
```

Out[691]:
```
(3, 1)
```

138. nonzero()

In [692…
```
from numpy import *
x = array([1,0,2,-1,0,0,8])
indices = x.nonzero() # find the indices of the nonzero elements
indices
```

Out[692]:
```
(array([0, 2, 3, 6], dtype=int64),)
```

In [693…
```
x[indices]
```

Out[693]:
```
array([ 1,  2, -1,  8])
```

In [694…
```
y = array([[0,1,0],[2,0,3]])
indices = y.nonzero()
indices
```

Out[694]:
```
(array([0, 1, 1], dtype=int64), array([1, 0, 2], dtype=int64))
```

In [695…
```
y[indices[0],indices[1]] # one way of doing it, explains what's in indices[0] and
```

Out[695]:
```
array([1, 2, 3])
```

In [696…
```
y[indices] # this way is shorter
```

Out[696]:
```
array([1, 2, 3])
```

In [697…
```
y = array([1,3,5,7])
indices = (y >= 5).nonzero()
y[indices]
```

Out[697]:
```
array([5, 7])
```

In [698…
```
nonzero(y) # function also exists
```

Out[698]:
```
(array([0, 1, 2, 3], dtype=int64),)
```

139. ogrid()

In [699…
```
from numpy import *
x,y = ogrid[0:3,0:3] # x and y are useful to use with broadcasting rules
x
```

Out[699]:
```
array([[0],
       [1],
       [2]])
```

In [700…
```
y
```

Out[700]:
```
array([[0, 1, 2]])
```

In [701…
```
print(x*y) # example how to use broadcasting rules
```

```
[[0 0 0]
 [0 1 2]
 [0 2 4]]
```

140. ones()

In [702…
```
ones(5)
```

Out[702]:
```
array([1., 1., 1., 1., 1.])
```

In [703…
```
ones((2,3), int)
```

Out[703]:
```
array([[1, 1, 1],
       [1, 1, 1]])
```

141. ones_like()

In [704…
```
from numpy import *
a = array([[1,2,3],[4,2,5]])
ones_like(a)  # ones initialised array with the same shape and datatype as 'a'
```

Out[704]:
```
array([[1, 1, 1],
       [1, 1, 1]])
```

142. outer()

In [705…
```
a = array([1,1,3])
b= array([10,19,18])
outer(a,b)  # outer product
```

Out[705]:
```
array([[10, 19, 18],
       [10, 19, 18],
       [30, 57, 54]])
```

143. permutation()

In [706…
```
from numpy import *
from numpy.random import permutation
permutation(3)
```

Out[706]:
```
array([2, 0, 1])
```

In [707…
```
permutation(3)
```

Out[707]:
```
array([0, 1, 2])
```

144. piecewise()

In [708…
```
f1 = lambda x: x*x
f2 = lambda x: 2*x
x= arange(-2.,3.,0.1)
```

In [709…
```
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[
print(y)
```

```
[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 1.00000000e+00 8.10000000e-01
 6.40000000e-01 4.90000000e-01 3.60000000e-01 2.50000000e-01
 1.60000000e-01 9.00000000e-02 4.00000000e-02 1.00000000e-02
 3.15544362e-30 1.00000000e-02 4.00000000e-02 9.00000000e-02
 1.60000000e-01 2.50000000e-01 3.60000000e-01 4.90000000e-01
 6.40000000e-01 8.10000000e-01 2.00000000e+00 2.20000000e+00
 2.40000000e+00 2.60000000e+00 2.80000000e+00 3.00000000e+00
 3.20000000e+00 3.40000000e+00 3.60000000e+00 3.80000000e+00
 4.00000000e+00 4.20000000e+00 4.40000000e+00 4.60000000e+00
 4.80000000e+00 5.00000000e+00 5.20000000e+00 5.40000000e+00
 5.60000000e+00 5.80000000e+00]
```

145. pinv()

In [710…
```
 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]]
```

In [711…
```
x = dot(pinvA, b)
print(x)
```

```
[ 1.91666667  0.66666667 -0.58333333]
```

In [712…
```
 # 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]
```

```
C:\Users\nithi\AppData\Local\Temp\ipykernel_20556\1646301873.py:3: FutureWarning:
`rcond` parameter will change to the default of machine precision times ``max(M,
N)`` where M and N are the input matrix dimensions.
To use the future default and silence this warning we advise to pass `rcond=None`,
to keep using the old, explicitly pass `rcond=-1`.
  x,resids,rank,s = lstsq(A,b)
```

In [713…
```
# 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)
```

Out[713]:
```
array([[-1.33333333,  1.08333333],
       [-0.33333333,  0.33333333],
       [ 0.66666667, -0.41666667]])
```

146. poisson()

In [714…
```
from numpy import *
from numpy.random import *
poisson(lam=0.5, size=(2,3)) # poisson distribution lambda=0.5
```

Out[714]:
```
array([[0, 1, 1],
       [1, 0, 0]])
```

147. poly1d()

In [715…
```
from numpy import *
p1 = poly1d([2,3],r=1) # specify polynomial by its roots
print(p1)
```

```
   2
1 x - 5 x + 6
```

In [716…
```
p2 = poly1d([2,3],r=0) # specify polynomial by its coefficients
print (p2)
```

```
2 x + 3
```

In [717…
```
print(p1+p2) # +,-,*,/ and even ** are supported
```

```
   2
1 x - 3 x + 9
```

In [718…
```
quotient,remainder = p1/p2 # division gives a tupple with the quotient and remainde
print(quotient,remainder)
```

```
0.5 x - 3.25
15.75
```

In [719…
```
p3 = p1*p2
print (p3)
```

```
   3     2
2 x - 7 x - 3 x + 18
```

In [720…
```
p3([1,2,3,4]) # evaluate the polynomial in the values [1,2,3,4]
```

Out[720]:
```
array([10.,  0.,  0., 22.])
```

In [721…
```
p3[2] # the coefficient of x**2
```

Out[721]:
```
-7.0
```

In [722…
```
p3.r # the roots of the polynomial
```

Out[722]:
```
array([-1.5,  3. ,  2. ])
```

In [723…
```
p3.c # the coefficients of the polynomial
```

Out[723]:
```
array([ 2., -7., -3., 18.])
```

In [724…
```
p3.o # the order of the polynomial
```

Out[724]:
```
3
```

In [725…
```
print (p3.deriv(m=2)) # the 2nd derivative of the polynomial
```

```
12 x - 14
```

In [726…
```
print (p3.integ(m=2,k=[1,2])) # integrate polynomial twice and use [1,2] as integr
```

```
     5          4       3     2
0.1 x - 0.5833 x - 0.5 x + 9 x + 1 x + 2
```

148. polyfit()

In [727…
```
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
```

Out[727]:
```
array([1., 2., 3.])
```

In [728…
```
polyfit(x,y,1) # fit a 1st degree polynomial (straight line), result is 8x-4
```

Out[728]:
```
array([ 8., -4.])
```

149. prod()

In [729…
```
from numpy import *
a = array([1,2,3])
a.prod() # 1 * 2 * 3 = 6
```

Out[729]:
```
6
```

In [730…
```
prod(a) # also exists
```

Out[730]:
```
6
```

In [731…
```
a = array([[1,2,3],[4,5,6]])
a.prod(dtype=float) # specify type of output
```

Out[731]:
```
720.0
```

In [732…
```
a.prod(axis=0) # for each of the 3 columns: product
```

Out[732]:
```
array([ 4, 10, 18])
```

In [733…
```
a.prod(axis=1) # for each of the two rows: product
```

Out[733]:
```
array([  6, 120])
```

150.ptp()

In [734…
```
from numpy import *
a = array([5,15,25])
a.ptp() # peak-to-peak = maximum - minimum
```

Out[734]:
```
20
```

In [735…
```
a = array([[5,15,25],[3,13,33]])
a.ptp()
```

Out[735]:
```
30
```

In [736…
```
a.ptp(axis=0) # peak-to-peak value for each of the 3 columns
```

Out[736]:
```
array([2, 2, 8])
```

In [737…
```
a.ptp(axis=1) # peak-to-peak value for each of the 2 rows
```

Out[737]:
```
array([20, 30])
```

151. put()

In [738…
```
from numpy import *
a = array([1,2,3,4,5,6])
a.put([60,20,89],[0,4,1])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[738], line 3
      1 from numpy import *
      2 a = array([1,2,3,4,5,6])
----> 3 a.put([60,20,89],[0,4,1])

IndexError: index 60 is out of bounds for axis 0 with size 6
```

In [739…
```
from numpy import *
a = array([10,20,30,40])
a.put([10], [0]) # first values, then indices
a
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[739], line 3
      1 from numpy import *
      2 a = array([10,20,30,40])
----> 3 a.put([10], [0]) # first values, then indices
      4 a

IndexError: index 10 is out of bounds for axis 0 with size 4
```

152. putmask()

In [740…
```
from numpy 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
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[740], line 4
      2 a = array([10,20,30,40])
      3 mask = array([True,False,True,True]) # size mask = size a
----> 4 a.putmask([60,70,80,90], mask) # first values, then the mask
      5 a

AttributeError: 'numpy.ndarray' object has no attribute 'putmask'
```

In [741…
```
a = array([10,20,30,40,50])
append(a,60)
```

Out[741]:
```
array([10, 20, 30, 40, 50, 60])
```

153. r_[]

In [742…
```
from numpy import *
r_[1:5] # same as arange(1,5)
```

Out[742]:
```
array([1, 2, 3, 4])
```

In [743…
```
r_[1:10:4] # same as arange(1,10,4)
```

Out[743]:
```
array([1, 5, 9])
```

In [744…
```
r_[1:10:4j] # same as linspace(1,10,4), 4 equally-spaced elements between 1 and 10
```

Out[744]:
```
array([ 1.,  4.,  7., 10.])
```

In [745…
```
r_[1:5,7,1:10:4] # sequences separated with commas are concatenated
```

Out[745]:
```
array([1, 2, 3, 4, 7, 1, 5, 9])
```

In [746…
```
r_['r', 1:3] # return a matrix. If 1-d, result is a 1xN matrix
```

Out[746]:
```
matrix([[1, 2]])
```

In [747…
```
r_['c',1:3] # return a matrix. If 1-d, result is a Nx1 matrix
```

Out[747]:
```
matrix([[1],
        [2]])
```

In [748…
```
a = array([[1,2,3],[4,5,6]])
r_[a,a] # concatenation along 1st (default) axis (row-wise, that's why it's called
```

Out[748]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

In [749…
```
r_['-1',a,a] # concatenation along last axis, same as c_[a,a]
```

Out[749]:
```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

154. rand()

In [750…
```
from numpy.random import *
rand(3,2)
```

Out[750]:
```
array([[0.24102529, 0.75738955],
       [0.30372818, 0.92608852],
       [0.1600426 , 0.88828344]])
```

155.randint()

In [751…
```
similar to random_integers()
```

```
  Cell In[751], line 1
    similar to random_integers()
            ^
SyntaxError: invalid syntax
```

156. randn()

In [752…
```
randn(2,3)
```

Out[752]:
```
array([[ 1.32614311,  0.79773172, -1.14378126],
       [-0.73062919,  1.07249008,  1.36396574]])
```

In [753…
```
randn(3,4)
```

Out[753]:
```
array([[-0.51590351, -1.05677278,  1.58526515,  0.10180744],
       [ 0.71284616, -1.31647713, -0.71169163, -0.23730187],
       [ 0.22199829,  0.98147906,  1.68762734,  1.06827572]])
```

157. random_integers()

In [754…
```
random_integers(-1,5,(2,2))
```

```
C:\Users\nithi\AppData\Local\Temp\ipykernel_20556\1584869344.py:1: DeprecationWarn
ing: This function is deprecated. Please call randint(-1, 5 + 1) instead
  random_integers(-1,5,(2,2))
```

Out[754]:
```
array([[3, 3],
       [0, 1]])
```

In [755…
```
randint(-1,5 ,(2,2))
```

Out[755]:
```
array([[ 0,  1],
       [-1,  0]])
```

158. random_sample()

In [756…
```
random_sample((3,2))
```

Out[756]:
```
array([[0.00388765, 0.38873438],
       [0.85245003, 0.23142359],
       [0.59142888, 0.81696262]])
```

159. ranf()

In [757…
```
ranf((2,4))  # similar to random_sample()
```

Out[757]:
```
array([[0.06341494, 0.90637912, 0.93043232, 0.15415405],
       [0.52767997, 0.41595012, 0.4633249 , 0.91341926]])
```

160. ravel()

In [758…
```
from numpy import *
a = array([[1,2],[3,4]])
a.ravel() # 1-d version of a
```

Out[758]:
```
array([1, 2, 3, 4])
```

In [759…
```
b = a[:,0].ravel() # a[:,0] does not occupy a single memory segment, thus b is a c
b
```

Out[759]:
```
array([1, 3])
```

In [760…
```
c = a[0,:].ravel() # a[0,:] occupies a single memory segment, thus c is a reference
c
```

Out[760]:
```
array([1, 2])
```

In [761…
```
b[0] = -1
c[1] = -2
a
```

Out[761]:
```
array([[ 1, -2],
       [ 3,  4]])
```

In [762…
```
ravel(a) # also exists
```

Out[762]:
```
array([ 1, -2,  3,  4])
```

161. real()

In [763…
```
from numpy import *
a = array([1+2j,3+4j,5+6j])
a.real
```

Out[763]:
```
array([1., 3., 5.])
```

In [764…
```
a.real = 9
a
```

Out[764]:
```
array([9.+2.j, 9.+4.j, 9.+6.j])
```

In [765…
```
a.real = array([9,8,7])
a
```

Out[765]:
```
array([9.+2.j, 8.+4.j, 7.+6.j])
```

162. recarray()

In [766…
```
from numpy import *
num=2
a = recarray(num, formats='i4,f8,f8', names='id,x,y')
a['id']= [8,4]
a['id']
```

Out[766]:
```
array([8, 4])
```

In [767…
```
a= rec.fromrecords([(35,1.4,5.6),(26,2.3,4.5)], names='id,x,y')  # fromrecords is
a['id']
```

Out[767]:
```
array([35, 26])
```

163. reduce()

In [768…
```
add.reduce(array([1,2,3,4]))
```
Out[768]:
```
10
```

In [769…
```
multiply.reduce(array([1,2,3,4]))
```
Out[769]:
```
24
```

In [770…
```
add.reduce(array([[1,2,3],[4,5,9]]), axis=0)
```
Out[770]:
```
array([ 5,  7, 12])
```

In [771…
```
add.reduce(array([[1,9,8],[10,5,2]]), axis=1)
```
Out[771]:
```
array([18, 17])
```

164. repeat()

In [772…
```
repeat(7.,4)
```
Out[772]:
```
array([7., 7., 7., 7.])
```

In [773…
```
a = array([10,20])
```

In [774…
```
a.repeat([3,2])
```
Out[774]:
```
array([10, 10, 10, 20, 20])
```

In [775…
```
repeat(a,[3,2])
```
Out[775]:
```
array([10, 10, 10, 20, 20])
```

In [776…
```
a = array([[10,30],[2,3]])
```

In [777…
```
a.repeat([2,2,1,3])
```
Out[777]:
```
array([10, 10, 30, 30,  2,  3,  3,  3])
```

In [778…
```
a.repeat([2,4], axis=0)
```
Out[778]:
```
array([[10, 30],
       [10, 30],
       [ 2,  3],
       [ 2,  3],
       [ 2,  3],
       [ 2,  3]])
```

In [779…
```
a.repeat([2,1],axis=1)
```
Out[779]:
```
array([[10, 10, 30],
       [ 2,  2,  3]])
```

165. reshape()

In [780…
```
x = arange(12)
```

In [781…
```
x.reshape(3,4)
```

Out[781]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [782…
```
x.reshape(3,2,2)
```

Out[782]:
```
array([[[ 0,  1],
        [ 2,  3]],

       [[ 4,  5],
        [ 6,  7]],

       [[ 8,  9],
        [10, 11]]])
```

In [783…
```
x.reshape(2,-2)
```

Out[783]:
```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

166. resize()

In [784…
```
a = array([1,2,3,4])
```

In [785…
```
a
```

Out[785]:
```
array([1, 2, 3, 4])
```

In [786…
```
a.resize(2,2)
```

In [787…
```
print(a)
```

```
[[1 2]
 [3 4]]
```

In [788…
```
import numpy as np
np.resize(a,(2,4))
```

Out[788]:
```
array([[1, 2, 3, 4],
       [1, 2, 3, 4]])
```

In [789…
```
np.resize(a,(3,2))
```

Out[789]:
```
array([[1, 2],
       [3, 4],
       [1, 2]])
```

167. rollaxis()

In [790…
```
a=arange(3*2*2).reshape(3,2,2)
```

In [791…
```
a
```

Out[791]:
```
array([[[ 0,  1],
        [ 2,  3]],

       [[ 4,  5],
        [ 6,  7]],

       [[ 8,  9],
        [10, 11]]])
```

In [792…
```
a.shape
```

Out[792]:
```
(3, 2, 2)
```

In [793…
```
b = rollaxis(a,1,0)
```

In [794…
```
b.shape
```

Out[794]:
```
(2, 3, 2)
```

In [795…
```
c = rollaxis(a,0,1)
```

In [796…
```
c.shape
```

Out[796]:
```
(3, 2, 2)
```

168. round()

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

In [797…
```
array([1.78,-4.2899]).round()
```

Out[797]:
```
array([ 2., -4.])
```

In [798…
```
array([1.567,-3.8900]).round(decimals=2)
```

Out[798]:
```
array([ 1.57, -3.89])
```

169. rot90()

In [799…
```
a = arange(16).reshape(4,4)
```

In [800…
```
a
```

Out[800]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [801…
```
rot90(a)
```

Out[801]:
```
array([[ 3,  7, 11, 15],
       [ 2,  6, 10, 14],
       [ 1,  5,  9, 13],
       [ 0,  4,  8, 12]])
```

170. s_[]

In [802…
```
s_[1:5]
```

Out[802]:
```
slice(1, 5, None)
```

In [803…
```
print(s_[1:5])
```

```
slice(1, 5, None)
```

171. sample()

Synonym for random_sample

172. savetxt()

In [804…
```
from numpy import *
data = array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
```

In [805…
```
x = array([1,2,3])
y = array([7,8,9])
```

In [806…
```
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 '
savetxt("myfile.txt", data, delimiter = ';') # use ';' to separate columns instead
```

173. searchsorted()

In [807…
```
a = array([1,2,3,4,4])
```

In [808…
```
a.searchsorted(2)
```

Out[808]:
```
1
```

In [809…
```
a.searchsorted(4)
```

Out[809]:
```
3
```

174. seed()

In [810…
```
from numpy import *
seed([1]) # seed the pseudo-random number generator
```

In [811…
```
rand(3)
```

Out[811]:
```
array([0.13436424, 0.84743374, 0.76377462])
```

175. select()

In [812…
```
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.)
```

Out[812]:
```
array([  5.2,  -2.1,   1.2,   0. ,   4.2,  -1.1,   3.2, 100. ])
```

In [813…
```
#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 f
    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
```

Out[813]:
```
array([  5.2,  -2.1,   1.2,   0. ,   4.2,  -1.1,   3.2, 100. ])
```

176. set_printoptions()

In [814…
```
from numpy import *
x = array([pi, 1.e-200])
x
```

Out[814]:
```
array([3.14159265e+000, 1.00000000e-200])
```

In [815…
```
set_printoptions(precision=3, suppress=True) # 3 digits behind decimal point + supp
x
```

Out[815]:
```
array([3.142, 0.   ])
```

In [816…
```
help(set_printoptions) # see help() for keywords 'threshold','edgeitems' and 'linew
```

```
Help on function set_printoptions in module numpy:

set_printoptions(precision=None, threshold=None, edgeitems=None, linewidth=None, s
uppress=None, nanstr=None, infstr=None, formatter=None, sign=None, floatmode=None,
*, legacy=None)
    Set printing options.

    These options determine the way floating point numbers, arrays and
    other NumPy objects are displayed.

    Parameters
    ----------
    precision : int or None, optional
        Number of digits of precision for floating point output (default 8).
        May be None if `floatmode` is not `fixed`, to print as many digits as
        necessary to uniquely specify the value.
    threshold : int, optional
        Total number of array elements which trigger summarization
        rather than full repr (default 1000).
        To always use the full repr without summarization, pass `sys.maxsize`.
    edgeitems : int, optional
        Number of array items in summary at beginning and end of
        each dimension (default 3).
    linewidth : int, optional
        The number of characters per line for the purpose of inserting
        line breaks (default 75).
    suppress : bool, optional
        If True, always print floating point numbers using fixed point
        notation, in which case numbers equal to zero in the current precision
        will print as zero.  If False, then scientific notation is used when
        absolute value of the smallest number is < 1e-4 or the ratio of the
        maximum absolute value to the minimum is > 1e3. The default is False.
    nanstr : str, optional
        String representation of floating point not-a-number (default nan).
    infstr : str, optional
        String representation of floating point infinity (default inf).
    sign : string, either '-', '+', or ' ', optional
        Controls printing of the sign of floating-point types. If '+', always
        print the sign of positive values. If ' ', always prints a space
        (whitespace character) in the sign position of positive values.  If
        '-', omit the sign character of positive values. (default '-')
    formatter : dict of callables, optional
        If not None, the keys should indicate the type(s) that the respective
        formatting function applies to.  Callables should return a string.
        Types that are not specified (by their corresponding keys) are handled
        by the default formatters.  Individual types for which a formatter
        can be set are:

        - 'bool'
        - 'int'
        - 'timedelta' : a `numpy.timedelta64`
        - 'datetime' : a `numpy.datetime64`
        - 'float'
        - 'longfloat' : 128-bit floats
        - 'complexfloat'
        - 'longcomplexfloat' : composed of two 128-bit floats
        - 'numpystr' : types `numpy.bytes_` and `numpy.str_`
        - 'object' : `np.object_` arrays

        Other keys that can be used to set a group of types at once are:

        - 'all' : sets all types
        - 'int_kind' : sets 'int'
        - 'float_kind' : sets 'float' and 'longfloat'
```

```
    - 'complex_kind' : sets 'complexfloat' and 'longcomplexfloat'
    - 'str_kind' : sets 'numpystr'
floatmode : str, optional
    Controls the interpretation of the `precision` option for
    floating-point types. Can take the following values
    (default maxprec_equal):

    * 'fixed': Always print exactly `precision` fractional digits,
            even if this would print more or fewer digits than
            necessary to specify the value uniquely.
    * 'unique': Print the minimum number of fractional digits necessary
            to represent each value uniquely. Different elements may
            have a different number of digits. The value of the
            `precision` option is ignored.
    * 'maxprec': Print at most `precision` fractional digits, but if
            an element can be uniquely represented with fewer digits
            only print it with that many.
    * 'maxprec_equal': Print at most `precision` fractional digits,
            but if every element in the array can be uniquely
            represented with an equal number of fewer digits, use that
            many digits for all elements.
legacy : string or `False`, optional
    If set to the string `'1.13'` enables 1.13 legacy printing mode. This
    approximates numpy 1.13 print output by including a space in the sign
    position of floats and different behavior for 0d arrays. This also
    enables 1.21 legacy printing mode (described below).

    If set to the string `'1.21'` enables 1.21 legacy printing mode. This
    approximates numpy 1.21 print output of complex structured dtypes
    by not inserting spaces after commas that separate fields and after
    colons.

    If set to `False`, disables legacy mode.

    Unrecognized strings will be ignored with a warning for forward
    compatibility.

    .. versionadded:: 1.14.0
    .. versionchanged:: 1.22.0

See Also
--------
get_printoptions, printoptions, set_string_function, array2string

Notes
-----
`formatter` is always reset with a call to `set_printoptions`.

Use `printoptions` as a context manager to set the values temporarily.

Examples
--------
Floating point precision can be set:

>>> np.set_printoptions(precision=4)
>>> np.array([1.123456789])
[1.1235]

Long arrays can be summarised:

>>> np.set_printoptions(threshold=5)
>>> np.arange(10)
array([0, 1, 2, ..., 7, 8, 9])
```

```
Small results can be suppressed:

>>> eps = np.finfo(float).eps
>>> x = np.arange(4.)
>>> x**2 - (x + eps)**2
array([-4.9304e-32, -4.4409e-16,  0.0000e+00,  0.0000e+00])
>>> np.set_printoptions(suppress=True)
>>> x**2 - (x + eps)**2
array([-0., -0.,  0.,  0.])

A custom formatter can be used to display array elements as desired:

>>> np.set_printoptions(formatter={'all':lambda x: 'int: '+str(-x)})
>>> x = np.arange(3)
>>> x
array([int: 0, int: -1, int: -2])
>>> np.set_printoptions()  # formatter gets reset
>>> x
array([0, 1, 2])

To put back the default options, you can use:

>>> np.set_printoptions(edgeitems=3, infstr='inf',
... linewidth=75, nanstr='nan', precision=8,
... suppress=False, threshold=1000, formatter=None)

Also to temporarily override options, use `printoptions` as a context manager:

>>> with np.printoptions(precision=2, suppress=True, threshold=5):
...     np.linspace(0, 10, 10)
array([ 0.  ,  1.11,  2.22, ...,  7.78,  8.89, 10.  ])
```

177. shape()

In [817…
```
from numpy import *
x = arange(12)
x.shape
```

Out[817]:
```
(12,)
```

In [818…
```
x.shape = (3,4) # array with 3 rows and 4 columns. 3x4=12. Total number of elements
x
```

Out[818]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [819…
```
x.shape = (3,2,2) # 3x2x2 array; 3x2x2 = 12. x itself _does_ change, unlike reshape
x
```

Out[819]:
```
array([[[ 0,  1],
        [ 2,  3]],

       [[ 4,  5],
        [ 6,  7]],

       [[ 8,  9],
        [10, 11]]])
```

In [820…
```
x.shape = (2,-1) # 'missing' -1 value n is calculated so that 2xn=12, so n=6
```

```
x
```

Out[820]:
```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

In [821…
```
x.shape = 12 # x.shape = (1,12) is not the same as x.shape = 12
x
```

Out[821]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

178. shuffle()

In [822…
```
from numpy import *
from numpy.random import shuffle
x = array([1,50,-1,3])
shuffle(x) # shuffle the elements of x
print(x)
```

```
[ 3 -1 50  1]
```

In [823…
```
x = ['a','b','c','z']
shuffle(x) # works with any sequence
print(x)
```

```
['z', 'c', 'b', 'a']
```

179. slice()

In [824…
```
s = slice(3,9,2) # slice objects exist outside numpy
from numpy import *
a = arange(20)
a[s]
```

Out[824]:
```
array([3, 5, 7])
```

In [825…
```
a[3:9:2] # same thing
```

Out[825]:
```
array([3, 5, 7])
```

180. solve()

In [826…
```
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
```

```
[-3.286  9.429  1.286]
```

In [827…
```
dot(a,x) # Just checking if we indeed obtain the righthand side
```

Out[827]:
```
array([6., 7., 8.])
```

181. sometrue()

In [828…
```
from numpy import *
b = array([True, False, True, True])
sometrue(b)
```

Out[828]:
```
True
```

In [829…
```
a = array([1, 5, 2, 7])
sometrue(a >= 5)
```

Out[829]:
```
True
```

182. sort()

In [830…
```
from numpy import *
a = array([2,0,8,4,1])
a.sort() # in-place sorting with quicksort (default)
a
```

Out[830]:
```
array([0, 1, 2, 4, 8])
```

In [831…
```
a.sort(kind='mergesort') # algorithm options are 'quicksort', 'mergesort' and 'heap
a = array([[8,4,1],[2,0,9]])
a.sort(axis=0)
a
```

Out[831]:
```
array([[2, 0, 1],
       [8, 4, 9]])
```

In [832…
```
a = array([[8,4,1],[2,0,9]])
a.sort(axis=1) # default axis = -1
a
```

Out[832]:
```
array([[1, 4, 8],
       [0, 2, 9]])
```

In [833…
```
sort(a) # there is a functional form
```

Out[833]:
```
array([[1, 4, 8],
       [0, 2, 9]])
```

183. split()

In [834…
```
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
```

Out[834]:
```
[array([[1, 2, 3, 4]]), array([[5, 6, 7, 8]])]
```

In [835…
```
split(a,4,axis=1) # split a in 4 parts, column-wise
```

Out[835]:
```
[array([[1],
        [5]]),
 array([[2],
        [6]]),
 array([[3],
        [7]]),
 array([[4],
        [8]])]
```

In [836…
```
split(a,2,axis=1) # impossible to split in 3 equal parts -> error (SEE: array_spli
```

Out[836]:
```
[array([[1, 2],
        [5, 6]]),
 array([[3, 4],
        [7, 8]])]
```

In [837…
```
split(a,[2,3],axis=1) # make a split before the 2nd and the 3rd column
```

Out[837]:
```
[array([[1, 2],
        [5, 6]]),
 array([[3],
        [7]]),
 array([[4],
        [8]])]
```

184. squeeze()

In [838…
```
from numpy import *
a = arange(6)
a = a.reshape(1,2,1,1,3,1)
a
```

Out[838]:
```
array([[[[[[0],
           [1],
           [2]]]],



       [[[[3],
          [4],
          [5]]]]])
```

In [839…
```
a.squeeze() # result has shape 2x3, all dimensions with length 1 are removed
```

Out[839]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [840…
```
squeeze(a) # also exists
```

Out[840]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

185. std()

In [841…
```
from numpy import *
a = array([1.,2,7])
a.std() # normalized by N (not N-1)
```

Out[841]:
```
2.6246692913372702
```

In [842…
```
a = array([[1.,2,7],[4,9,6]])
a.std()
```

Out[842]:
```
2.793842435706702
```

In [843…
```
a.std(axis=0) # standard deviation of each of the 3 columns
```

Out[843]:
```
array([1.5, 3.5, 0.5])
```

In [844…
```
a.std(axis=1) # standard deviation of each of the 2 columns
```

Out[844]:
```
array([2.625, 2.055])
```

186. standard_normal()

In [845…
```
standard_normal((2,3))
```

Out[845]:
```
array([[-1.26 , -0.523,  0.978],
       [-0.03 ,  1.516, -1.554]])
```

187. sum()

In [846…
```
from numpy import *
a = array([1,2,3])
a.sum()
```

Out[846]:
```
6
```

In [847…
```
sum(a) # also exists
a = array([[1,2,3],[4,5,6]])
a.sum()
```

Out[847]:
```
21
```

In [848…
```
a.sum(dtype=float) # specify type of output
```

Out[848]:
```
21.0
```

In [849…
```
a.sum(axis=0) # sum over rows for each of the 3 columns
```

Out[849]:
```
array([5, 7, 9])
```

In [850…
```
a.sum(axis=1) # sum over columns for each of the 2 rows
```

Out[850]:
```
array([ 6, 15])
```

188. svd()

In [851…
```
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
```

```
[[-0.62  -0.785]
 [-0.785  0.62 ]]
```

In [852…
```
print(sigma) # non-zero diagonal elements of Sigma
```

```
[9.526 0.514]
```

In [853…
```
print(V) # V is a (3x3) unitary matrix
```

```
[[-0.23  -0.525 -0.82 ]
 [ 0.883  0.241 -0.402]
 [ 0.408 -0.816  0.408]]
```

In [854…
```
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.]]
```

189. swapaxes()

In [855…
```
from numpy import *
a = arange(30)
a = a.reshape(2,3,5)
a
```

Out[855]:
```
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]]])
```

In [856…
```
b = a.swapaxes(1,2) # swap the 2nd and the 3rd axis
b
```

Out[856]:
```
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]]])
```

In [857…
```
b.shape
```

Out[857]:
```
(2, 5, 3)
```

In [858…
```
b[0,0,0] = -1 # be aware that b is a reference, not a copy
print (a[0,0,0])
```

```
-1
```

190. T

In [859…
```
from numpy import *
x = array([[1.,2.],[3.,4.]])
x
```

Out[859]:
```
array([[1., 2.],
       [3., 4.]])
```

In [860…
```
x.T # shortcut for transpose()
```

Out[860]:
```
array([[1., 3.],
       [2., 4.]])
```

191. take()

In [861…
```
from numpy import *
a = array([10,20,30,40])
```

In [862…
```
a
```

Out[862]:
```
array([10, 20, 30, 40])
```

In [863…
```
a.take([0,0,3])
```

Out[863]:
```
array([10, 10, 40])
```

In [864…
```
a[[0,0,3]]
```

Out[864]:
```
array([10, 10, 40])
```

In [865…
```
a.take([[0,1],[0,1]])
```

Out[865]:
```
array([[10, 20],
       [10, 20]])
```

192. tensordot()

In [866…
```
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
```

Out[866]:
```
(5, 2)
```

In [867…
```
 # 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]
```

193. tile()

In [868…
```
a = array([10,20])
tile(a, (3,2))
```

Out[868]:
```
array([[10, 20, 10, 20],
       [10, 20, 10, 20],
       [10, 20, 10, 20]])
```

In [869…
```
tile(9,(3,2))
```

Out[869]:
```
array([[9, 9],
       [9, 9],
       [9, 9]])
```

In [870…
```
tile(23.,(3,2))
```

Out[870]:
```
array([[23., 23.],
       [23., 23.],
       [23., 23.]])
```

In [871…
```
tile([1,2,3],(2,3))
```

Out[871]:
```
array([[1, 2, 3, 1, 2, 3, 1, 2, 3],
       [1, 2, 3, 1, 2, 3, 1, 2, 3]])
```

194. tofile()

In [872…
```
x = arange(10.)
```

In [873…
```
y = x**2
```

In [874…
```
x
```

Out[874]:
```
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

In [875…
```
y
```

Out[875]:
```
array([ 0.,  1.,  4.,  9., 16., 25., 36., 49., 64., 81.])
```

In [876…
```
y.tofile("myfile.dat") # binary format
y.tofile("myfile.txt", sep=' ', format = "%e") # ascii format, one row, exp notati
y.tofile("myfile.txt", sep='\n', format = "%e") # ascii format, one column, exponer
```

195. tolist()

In [877…
```
a = array([[10,30],[3,7]])
a.tolist()
```

Out[877]:
```
[[10, 30], [3, 7]]
```

196. trace()

In [878…
```
a = arange(12).reshape(3,4)
```

In [879…
```
a
```

Out[879]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [880…
```
a.diagonal()
```

Out[880]:
```
array([ 0,  5, 10])
```

In [881…
```
a.trace()
```

Out[881]:
```
15
```

In [882…
```
a.diagonal(offset=1)
```

Out[882]:
```
array([ 1,  6, 11])
```

In [883…
```
a.trace(offset=1)
```

Out[883]:
```
18
```

197. transpose()

In [884…
```
a = array([[1,2,3],[4,5,6]])
```

In [885…
```
a
```

Out[885]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [886…
```
print(a.shape)
```

```
(2, 3)
```

In [887…
```
b = a.transpose()
```

In [888…
```
b
```

Out[888]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

In [889…
```
b.shape
```

Out[889]:
```
(3, 2)
```

In [890…
```
a = arange(30)
a = a.reshape(2,3,5)
a
```

Out[890]:
```
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]]])
```

In [891…
```
b = a.transpose()
b
```

Out[891]:
```
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]]])
```

In [892…
```
b.shape
```

Out[892]:
```
(5, 3, 2)
```

In [893…
```
b = a.transpose(1,0,2) # First axis 1, then axis 0, then axis 2
b
```

Out[893]:
```
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]]])
```

In [894…
```
b.shape
```

Out[894]:
```
(3, 2, 5)
```

In [895…
```
b = transpose(a, (1,0,2)) # A separate transpose() function also exists
```

In [896…
```
b
```

Out[896]:
```
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]]])
```

198. tri()

In [897…
```
from numpy import *
tri(3,4,k=0,dtype=float) # 3x4 matrix of Floats, triangular, the k=0-th diagonal ar
```

Out[897]:
```
array([[1., 0., 0., 0.],
       [1., 1., 0., 0.],
       [1., 1., 1., 0.]])
```

In [898…
```
tri(3,4,k=1,dtype=int)
```

Out[898]:
```
array([[1, 1, 0, 0],
       [1, 1, 1, 0],
       [1, 1, 1, 1]])
```

199. tril()

In [899…
```
from numpy import *
a = arange(10,100,10).reshape(3,3)
a
```

Out[899]:
```
array([[10, 20, 30],
       [40, 50, 60],
       [70, 80, 90]])
```

In [900…
```
tril(a,k=0)
```

Out[900]:
```
array([[10,  0,  0],
       [40, 50,  0],
       [70, 80, 90]])
```

In [901…
```
tril(a,k=1)
```

Out[901]:
```
array([[10, 20,  0],
       [40, 50, 60],
       [70, 80, 90]])
```

200. trim_zeros()

In [902…
```
from numpy import *
x = array([0, 0, 0, 1, 2, 3, 0, 0])
trim_zeros(x,'f') # remove zeros at the front
```

Out[902]:
```
array([1, 2, 3, 0, 0])
```

In [903…
```
trim_zeros(x,'b') # remove zeros at the back
```

Out[903]:
```
array([0, 0, 0, 1, 2, 3])
```

In [904…
```
trim_zeros(x,'bf') # remove zeros at the back and the front
```

Out[904]:
```
array([1, 2, 3])
```

201. triu()

In [905…
```
from numpy import *
a = arange(10,100,10).reshape(3,3)
a
```

Out[905]:
```
array([[10, 20, 30],
       [40, 50, 60],
       [70, 80, 90]])
```

In [906…
```
triu(a,k=0)
```

Out[906]:
```
array([[10, 20, 30],
       [ 0, 50, 60],
       [ 0,  0, 90]])
```

In [907…
```
triu(a,k=1)
```

Out[907]:
```
array([[ 0, 20, 30],
       [ 0,  0, 60],
       [ 0,  0,  0]])
```

202. typeDict()

In [908…
```
from numpy import *
typeDict['short']
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[908], line 2
      1 from numpy import *
----> 2 typeDict['short']

NameError: name 'typeDict' is not defined
```

In [909…
```
typeDict['uint16']
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[909], line 1
----> 1 typeDict['uint16']

NameError: name 'typeDict' is not defined
```

In [910…
```
>>> typeDict['void']

>>> typeDict['S']
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[910], line 1
----> 1 typeDict['void']
      3 typeDict['S']

NameError: name 'typeDict' is not defined
```

203. uniform()

In [911…
```
from numpy import *
from numpy.random import *
uniform(low=0,high=10,size=(2,3)) # uniform numbers in range [0,10)
```

Out[911]:
```
array([[7.672, 6.958, 2.663],
       [8.018, 5.912, 1.022]])
```

204. unique()

In [912…
```
from numpy import *
x = array([2,3,2,1,0,3,4,0])
unique(x) # remove double values
```

Out[912]:
```
array([0, 1, 2, 3, 4])
```

205. unique1d()

In [913…
```
import numpy as np
unique1d([1,2,3,4,2,2,3])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[913], line 2
      1 import numpy as np
----> 2 unique1d([1,2,3,4,2,2,3])

NameError: name 'unique1d' is not defined
```

206. vander()

In [914…
```
from numpy import *
x = array([1,2,3,5])
N=3
vander(x,N) # Vandermonde matrix of the vector x
```

Out[914]:
```
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

In [915…
```
column_stack([x**(N-1-i) for i in range(N)]) # to understand what a Vandermonde ma
```

Out[915]:
```
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

207. var()

In [916…
```
from numpy import *
a = array([1,2,7])
a.var() # normalised with N (not N-1)
```

Out[916]:
```
6.888888888888888
```

In [917…
```
a = array([[1,2,7],[4,9,6]])
a.var()
```

Out[917]:
```
7.805555555555557
```

In [918…
```
a.var(axis=0) # the variance of each of the 3 columns
```

Out[918]:
```
array([ 2.25, 12.25,  0.25])
```

In [919…
```
a.var(axis=1) # the variance of each of the 2 rows
```

Out[919]:
```
array([6.889, 4.222])
```

208. vdot()

In [920…
```
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)
```

Out[920]:
```
(70-8j)
```

209. vectorize()

In [921…
```
from numpy import *
def myfunc(x):
    if x >= 0: return x**2
    else: return -x
```

In [922…
```
myfunc(2.) # works fine
```

Out[922]:
```
4.0
```

In [923…
```
myfunc(array([-2,2])) # doesn't work, try it...
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[923], line 1
----> 1 myfunc(array([-2,2]))

Cell In[921], line 3, in myfunc(x)
      2 def myfunc(x):
----> 3     if x >= 0: return x**2
      4     else: return -x

ValueError: The truth value of an array with more than one element is ambiguous. U
se a.any() or a.all()
```

In [924…
```
vecfunc = vectorize(myfunc, otypes=[float]) # declare the return type as float
vecfunc(array([-2,2])) # works fine!
```

Out[924]:
```
array([2., 4.])
```

210. view()

In [925…
```
from numpy import *
a = array([1., 2.])
a.view() # new array referring to the same data as 'a'
```

Out[925]:
```
array([1., 2.])
```

In [926…
```
a.view(complex) # pretend that a is made up of complex numbers
```

Out[926]:
```
array([1.+2.j])
```

In [927…
```
a.view(int) # view(type) is NOT the same as astype(type)!
```

Out[927]:
```
array([         0, 1072693248,          0, 1073741824])
```

In [928…
```
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
```

In [929…
```
a['age'] # works with 'a' but not with 'b'
```

Out[929]:
```
array([25, 30], dtype=int16)
```

In [930…
```
b.age # works with 'b' but not with 'a'
```

Out[930]:
```
array([25, 30], dtype=int16)
```

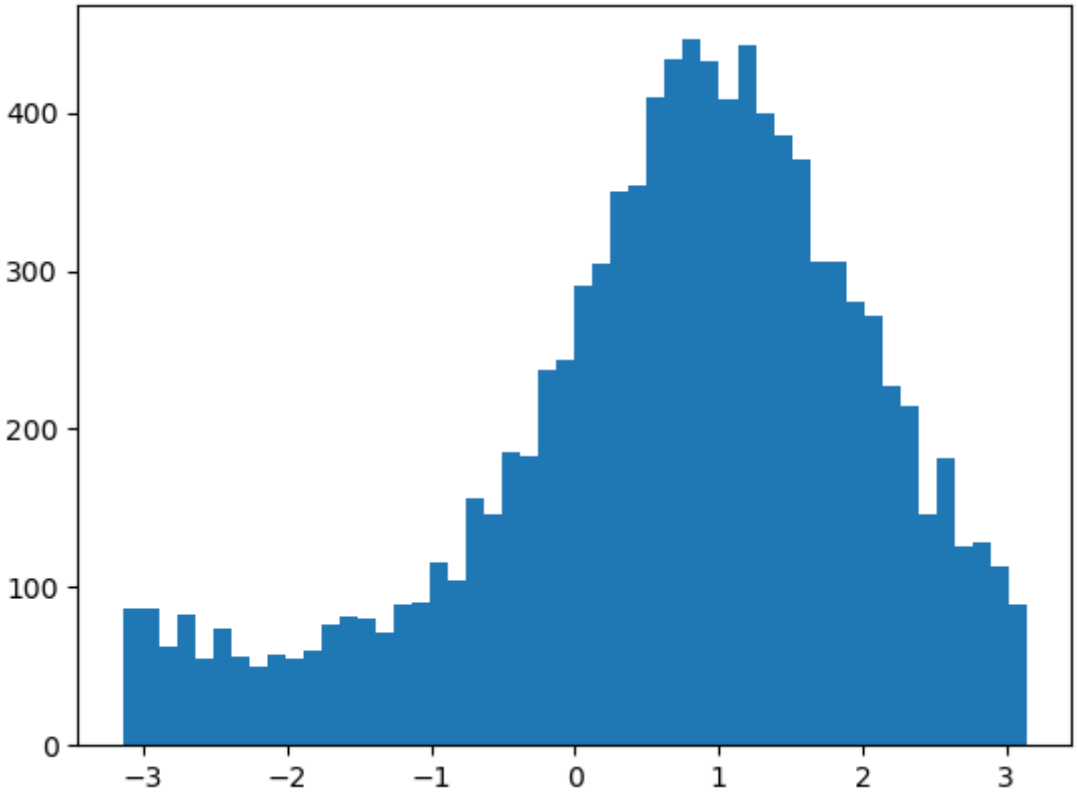
211. vonmises()

In [931…
```
from numpy import *
from numpy.random import *
vonmises(mu=1,kappa=1,size=(2,3)) # Von Mises distribution mean=1.0, kappa=1
```

Out[931]:
```
array([[ 1.517,  1.014,  1.975],
       [ 2.042, -2.61 , -2.637]])
```

In [932…
```
from pylab import * # histogram plot example
hist(vonmises(1,1,(10000)), 50)
```

Out[932]:
```
(array([ 87.,  87.,  62.,  83.,  55.,  74.,  56.,  50.,  57.,  55.,  60.,
         76.,  82.,  80.,  71.,  89.,  90., 116., 104., 156., 146., 185.,
        183., 237., 243., 290., 305., 350., 354., 409., 434., 446., 433.,
        408., 442., 400., 386., 370., 306., 306., 281., 272., 227., 214.,
        146., 181., 126., 128., 113.,  89.]),
 array([-3.141, -3.015, -2.889, -2.764, -2.638, -2.513, -2.387, -2.261,
        -2.136, -2.01 , -1.885, -1.759, -1.633, -1.508, -1.382, -1.257,
        -1.131, -1.005, -0.88 , -0.754, -0.629, -0.503, -0.377, -0.252,
        -0.126, -0.001,  0.125,  0.251,  0.376,  0.502,  0.627,  0.753,
         0.878,  1.004,  1.13 ,  1.255,  1.381,  1.506,  1.632,  1.758,
         1.883,  2.009,  2.134,  2.26 ,  2.386,  2.511,  2.637,  2.762,
         2.888,  3.014,  3.139]),
 <BarContainer object of 50 artists>)
```



212. vsplit()

In [933…
```
from numpy import *
a = array([[1,2],[3,4],[5,6],[7,8]])
vsplit(a,2) # split, row-wise, in 2 equal parts
```

Out[933]:
```
[array([[1, 2],
        [3, 4]]),
 array([[5, 6],
        [7, 8]])]
```

In [934…
```
vsplit(a,[1,2]) # split, row-wise, before row 1 and before row 2
```

Out[934]:
```
[array([[1, 2]]),
 array([[3, 4]]),
 array([[5, 6],
        [7, 8]])]
```

213. vstack()

In [935…
```
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 differen
```

Out[935]:
```
array([[1, 2],
       [3, 4],
       [5, 6],
       [1, 2]])
```

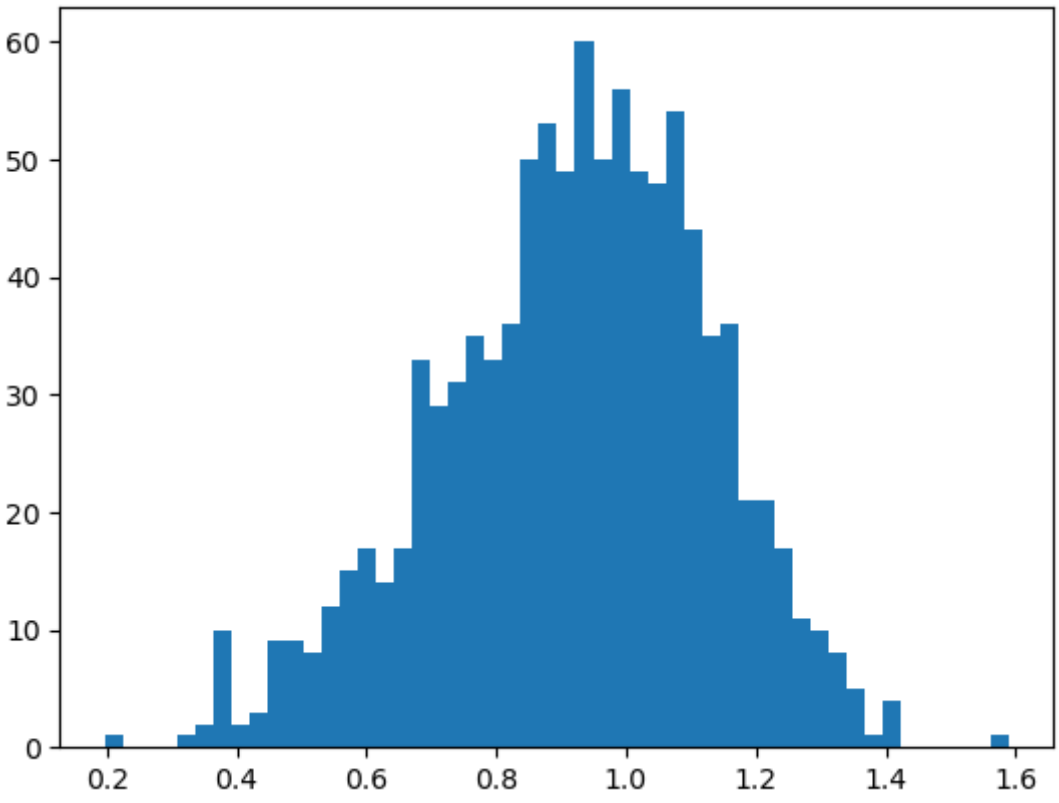
214. weibull()

In [936…
```
from numpy import *
from numpy.random import *
weibull(a=1,size=(2,3)) # I think a is the shape parameter
```

Out[936]:
```
array([[0.485, 0.65 , 0.266],
       [2.636, 0.612, 0.619]])
```

In [937…
```
from pylab import * # histogram plot example
hist(weibull(5, (1000)), 50)
```

Out[937]:
```
(array([ 1.,  0.,  0.,  0.,  1.,  2., 10.,  2.,  3.,  9.,  9.,  8., 12.,
        15., 17., 14., 17., 33., 29., 31., 35., 33., 36., 50., 53., 49.,
        60., 50., 56., 49., 48., 54., 44., 35., 36., 21., 21., 17., 11.,
        10.,  8.,  5.,  1.,  4.,  0.,  0.,  0.,  0.,  0.,  1.]),
 array([0.198, 0.225, 0.253, 0.281, 0.309, 0.337, 0.365, 0.392, 0.42 ,
        0.448, 0.476, 0.504, 0.532, 0.559, 0.587, 0.615, 0.643, 0.671,
        0.699, 0.726, 0.754, 0.782, 0.81 , 0.838, 0.866, 0.893, 0.921,
        0.949, 0.977, 1.005, 1.033, 1.06 , 1.088, 1.116, 1.144, 1.172,
        1.2  , 1.228, 1.255, 1.283, 1.311, 1.339, 1.367, 1.395, 1.422,
        1.45 , 1.478, 1.506, 1.534, 1.562, 1.589]),
 <BarContainer object of 50 artists>)
```



215. where()

In [938…
```
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)
```

Out[938]:
```
array([10, 20,  6,  8])
```

In [939…
```
where(a <= 6, b, -1)
```

Out[939]:
```
array([10, 20, -1, -1])
```

In [940…
```
indices = where(a <= 6) # returns a tuple; the array contains indices.
indices
```

Out[940]:
```
(array([0, 1], dtype=int64),)
```

In [941…
```
b[indices]
```

Out[941]:
```
array([10, 20])
```

In [942…
```
b[a <= 6] # an alternative syntax
```

Out[942]:
```
array([10, 20])
```

In [943…
```
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
```

Out[943]:
```
(array([0, 0, 1, 1, 1], dtype=int64), array([0, 1, 0, 1, 2], dtype=int64))
```

216. zeros()

In [944…
```
zeros(4)
```

Out[944]:
```
array([0., 0., 0., 0.])
```

In [945…
```
zeros((2,3),int)
```

Out[945]:
```
array([[0, 0, 0],
       [0, 0, 0]])
```

217. zeros_like()

In [946…
```
a = array([[1,2,3],[4,5,6]])
zeros_like(a) # with zeros initialised array with the same shape and datatype as 'a
```

Out[946]:
```
array([[0, 0, 0],
       [0, 0, 0]])
```

END