In [323]:
```
import numpy
numpy.__version__
```

Out[323]:
```
'1.16.2'
```

In [3]:
```
import numpy as np
np.array([10,40,50,60])
```

Out[3]:
```
array([10, 40, 50, 60])
```

In [4]:
```
a=np.array([10,40,50,60])
```

In [5]:
```
print(type(a))
print(a.ndim)    #No of dimensions
print(a.dtype)   #datatype of an array
print(a.shape)   #shape of the array as a tuple
print(a.size)    #no of elements
print(a.itemsize)  #each elements size in bytes
```

```
<class 'numpy.ndarray'>
1
int32
(4,)
4
4
```

In [6]:
```
a=np.array([1,2,3,4],np.int8)  #creating array with 8 bits
a.dtype
```

Out[6]:
```
dtype('int8')
```

In [7]:
```
a.itemsize
```

Out[7]:
```
1
```

In [8]:
```
a=np.array([1,2,3,4])  #creating array with default system config 32 bits
a.dtype
```

Out[8]:
```
dtype('int32')
```

In [9]:
```
2**8-1
```

Out[9]:
```
255
```

In [10]:
```
a=np.array([1,2,3,4,512],np.int8)  #creating array with 8 bits,here it wont acce
print(a)
```

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

In [11]:
```
a=np.array([2,3,4,6.7,9.2])
print(a.dtype)
print(a)
```

```
float64
[2.  3.  4.  6.7 9.2]
```

In [12]:
```
a=np.array([2,3,4,6.7,9.2,'numpy'])
a.dtype
```

Out[12]:
```
dtype('<U32')
```

In [13]:
```
dir(numpy.dtype)
```

Out[13]:
```
['__bool__',
 '__class__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmul__',
 '__setattr__',
 '__setstate__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'alignment',
 'base',
 'byteorder',
 'char',
 'descr',
 'fields',
 'flags',
 'hasobject',
 'isalignedstruct',
 'isbuiltin',
 'isnative',
 'itemsize',
 'kind',
 'metadata',
 'name',
 'names',
 'ndim',
 'newbyteorder',
 'num',
 'shape',
 'str',
 'subdtype',
 'type']
```

In [14]:
```
np.array((34,67))
```

Out[14]:
```
array([34, 67])
```

In [15]:
```
help(np.array)
```

```
Help on built-in function array in module numpy:

array(...)
    array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)

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

In [16]:
```
np.array(range(9))
```

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

In [17]:
```
l=[1,23,4]
l1=np.array(l)
print(l)
print(l1)
```

```
[1, 23, 4]
[ 1 23  4]
```

In [27]:
```
print(np.arange(11,21))
print(np.arange(5))
print(np.arange(4,1))
print(np.arange(1,10))   #Last parameter is excluded
```

```
[11 12 13 14 15 16 17 18 19 20]
[0 1 2 3 4]
[]
[1 2 3 4 5 6 7 8 9]
```

In [325]:
```
print(np.linspace(1,10,4))  #last parameter is no of elements,included last para
```

```
[ 1.  4.  7. 10.]
```

Array Initilization

In [20]:
```
np.ones((3,2,1),np.int)  #last parameter is shape of the array
```

Out[20]:
```
array([[[1],
        [1]],

       [[1],
        [1]],

       [[1],
        [1]]])
```

In [37]:
```
np.zeros((3,2,1))
#help(np.zeros)
```

Out[37]:
```
array([[[0.],
        [0.]],

       [[0.],
        [0.]],

       [[0.],
        [0.]]])
```

In [39]:
```
print(np.zeros((1,2)))  #2d array
print(np.zeros(1))      #1d array
```

```
[[0. 0.]]
[0.]
```

In [30]:
```
help(np.full)
```

```
Help on function full in module numpy:

full(shape, fill_value, dtype=None, order='C')
    Return a new array of given shape and type, filled with `fill_value`.

    Parameters
    ----------
    shape : int or sequence of ints
        Shape of the new array, e.g., ``(2, 3)`` or ``2``.
    fill_value : scalar
        Fill value.
    dtype : data-type, optional
        The desired data-type for the array  The default, `None`, means
         `np.array(fill_value).dtype`.
    order : {'C', 'F'}, optional
        Whether to store multidimensional data in C- or Fortran-contiguous
        (row- or column-wise) order in memory.

    Returns
    -------
    out : ndarray
        Array of `fill_value` with the given shape, dtype, and order.

    See Also
    --------
    full_like : Return a new array with shape of input filled with value.
    empty : Return a new uninitialized array.
    ones : Return a new array setting values to one.
    zeros : Return a new array setting values to zero.

    Examples
    --------
    >>> np.full((2, 2), np.inf)
    array([[ inf,  inf],
           [ inf,  inf]])
    >>> np.full((2, 2), 10)
    array([[10, 10],
           [10, 10]])
```

In [33]:
```
np.full((2,3),6)   #create an array with given size and type , filled with fill v
```

Out[33]:
```
array([[6, 6, 6],
       [6, 6, 6]])
```

In [35]:
```
help(np.eye)
```

```
Help on function eye in module numpy:

eye(N, M=None, k=0, dtype=<class 'float'>, order='C')
    Return a 2-D array with ones on the diagonal and zeros elsewhere.

    Parameters
    ----------
    N : int
      Number of rows in the output.
    M : int, optional
      Number of columns in the output. If None, defaults to `N`.
    k : int, optional
      Index of the diagonal: 0 (the default) refers to the main diagonal,
      a positive value refers to an upper diagonal, and a negative value
      to a lower diagonal.
    dtype : data-type, optional
      Data-type of the returned array.
    order : {'C', 'F'}, optional
        Whether the output should be stored in row-major (C-style) or
        column-major (Fortran-style) order in memory.

        .. versionadded:: 1.14.0

    Returns
    -------
    I : ndarray of shape (N,M)
      An array where all elements are equal to zero, except for the `k`-th
      diagonal, whose values are equal to one.

    See Also
    --------
    identity : (almost) equivalent function
    diag : diagonal 2-D array from a 1-D array specified by the user.

    Examples
    --------
    >>> np.eye(2, dtype=int)
    array([[1, 0],
           [0, 1]])
    >>> np.eye(3, k=1)
    array([[ 0.,  1.,  0.],
           [ 0.,  0.,  1.],
           [ 0.,  0.,  0.]])
```

In [40]:
```
np.eye(4)
```

Out[40]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

In [60]:
```
np.diag([3,5,7,9])
```

Out[60]:
```
array([[3, 0, 0, 0],
       [0, 5, 0, 0],
       [0, 0, 7, 0],
       [0, 0, 0, 9]])
```

Random

In [51]:
```
print(np.random.randint(10))  #random value in given range
print(np.random.randint(10,100,10)) #start,stop,shape
print(np.random.randint(10,100,(2,3)))
```

```
2
[94 80 33 54 67 63 21 51 93 15]
[[25 41 77]
 [32 59 60]]
```

In [55]:
```
print(np.random.rand(2,3))
print('------------------')
print(np.random.random((2,3)))   #giving i/p as tuple or list
```

```
[[0.02613352 0.90434673 0.88412416]
 [0.25652299 0.19763121 0.49701849]]
------------------
[[0.72842006 0.34014782 0.93901598]
 [0.71340217 0.03374922 0.78813069]]
```

In [58]:
```
print(np.random.normal())
```

```
1.1064594369189977
```

In [59]:
```
help(np.random.normal)
```

```
Help on built-in function normal:

normal(...) method of mtrand.RandomState instance
    normal(loc=0.0, scale=1.0, size=None)

    Draw random samples from a normal (Gaussian) distribution.

    The probability density function of the normal distribution, first
    derived by De Moivre and 200 years later by both Gauss and Laplace
    independently [2]_, is often called the bell curve because of
    its characteristic shape (see the example below).

    The normal distributions occurs often in nature.  For example, it
    describes the commonly occurring distribution of samples influenced
    by a large number of tiny, random disturbances, each with its own
    unique distribution [2]_.

    Parameters
    ----------
    loc : float or array_like of floats
        Mean ("centre") of the distribution.
    scale : float or array_like of floats
        Standard deviation (spread or "width") of the distribution.
    size : int or tuple of ints, optional
        Output shape.  If the given shape is, e.g., ``(m, n, k)``, then
        ``m * n * k`` samples are drawn.  If size is ``None`` (default),
        a single value is returned if ``loc`` and ``scale`` are both scalars.
        Otherwise, ``np.broadcast(loc, scale).size`` samples are drawn.

    Returns
    -------
    out : ndarray or scalar
        Drawn samples from the parameterized normal distribution.

    See Also
    --------
    scipy.stats.norm : probability density function, distribution or
        cumulative density function, etc.

    Notes
    -----
    The probability density for the Gaussian distribution is

    .. math:: p(x) = \frac{1}{\sqrt{ 2 \pi \sigma^2 }}
                     e^{ - \frac{ (x - \mu)^2 } {2 \sigma^2} },

    where :math:`\mu` is the mean and :math:`\sigma` the standard
    deviation. The square of the standard deviation, :math:`\sigma^2`,
    is called the variance.

    The function has its peak at the mean, and its "spread" increases with
    the standard deviation (the function reaches 0.607 times its maximum at
    :math:`x + \sigma` and :math:`x - \sigma` [2]_).  This implies that
    `numpy.random.normal` is more likely to return samples lying close to
    the mean, rather than those far away.
```

```
    References
    ----------
    .. [1] Wikipedia, "Normal distribution",
           https://en.wikipedia.org/wiki/Normal_distribution (https://en.wikipe
dia.org/wiki/Normal_distribution)
    .. [2] P. R. Peebles Jr., "Central Limit Theorem" in "Probability,
           Random Variables and Random Signal Principles", 4th ed., 2001,
           pp. 51, 51, 125.

    Examples
    --------
    Draw samples from the distribution:

    >>> mu, sigma = 0, 0.1 # mean and standard deviation
    >>> s = np.random.normal(mu, sigma, 1000)

    Verify the mean and the variance:

    >>> abs(mu - np.mean(s)) < 0.01
    True

    >>> abs(sigma - np.std(s, ddof=1)) < 0.01
    True

    Display the histogram of the samples, along with
    the probability density function:

    >>> import matplotlib.pyplot as plt
    >>> count, bins, ignored = plt.hist(s, 30, density=True)
    >>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
    ...                np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
    ...          linewidth=2, color='r')
    >>> plt.show()
```

In [61]:
```
np.random.randn()
```

Out[61]:
```
1.7467488564682845
```

In [62]:
```
help(np.random.randn)
```

```
Help on built-in function randn:

randn(...) method of mtrand.RandomState instance
    randn(d0, d1, ..., dn)

    Return a sample (or samples) from the "standard normal" distribution.

    If positive, int_like or int-convertible arguments are provided,
    `randn` generates an array of shape ``(d0, d1, ..., dn)``, filled
    with random floats sampled from a univariate "normal" (Gaussian)
    distribution of mean 0 and variance 1 (if any of the :math:`d_i` are
    floats, they are first converted to integers by truncation). A single
    float randomly sampled from the distribution is returned if no
    argument is provided.

    This is a convenience function.  If you want an interface that takes a
    tuple as the first argument, use `numpy.random.standard_normal` instead.

    Parameters
    ----------
    d0, d1, ..., dn : int, optional
        The dimensions of the returned array, should be all positive.
        If no argument is given a single Python float is returned.

    Returns
    -------
    Z : ndarray or float
        A ``(d0, d1, ..., dn)``-shaped array of floating-point samples from
        the standard normal distribution, or a single such float if
        no parameters were supplied.

    See Also
    --------
    standard_normal : Similar, but takes a tuple as its argument.

    Notes
    -----
    For random samples from :math:`N(\mu, \sigma^2)`, use:

    ``sigma * np.random.randn(...) + mu``

    Examples
    --------
    >>> np.random.randn()
    2.1923875335537315 #random

    Two-by-four array of samples from N(3, 6.25):

    >>> 2.5 * np.random.randn(2, 4) + 3
    array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],  #random
           [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]]) #random
```

In [63]:
```
np.random.randn(2,3) #mean=0,std=1
```

Out[63]:
```
array([[-0.8829798 , -0.82544445,  0.77617775],
       [ 0.35407667,  0.8249929 ,  1.19648087]])
```

Indexing slicing and subsetting

In [66]:
```
a=np.arange(25)
print(a[:])
print(a[1:20:2])
print(a[20:])
```

```
[ 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]
[ 1  3  5  7  9 11 13 15 17 19]
[20 21 22 23 24]
```

In [68]:
```
print(a[[0,3,5]])
```

```
[0 3 5]
```

In [70]:
```
a2=np.arange(25).reshape(5,5)   #2d array
print(a2)
```

```
[[ 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]]
```

In [75]:
```
a2[1:]
```

Out[75]:
```
array([[ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

In [78]:
```
a2[1:4:2]   #rnd and 4th rows  (row with index 1, and 3)
```

Out[78]:
```
array([[ 5,  6,  7,  8,  9],
       [15, 16, 17, 18, 19]])
```

In [79]:
```
a2[[1,4,2]]   #2nd , 5th, 3rd row (*****)( index in list )
```

Out[79]:
```
array([[ 5,  6,  7,  8,  9],
       [20, 21, 22, 23, 24],
       [10, 11, 12, 13, 14]])
```

In [80]:
```
a2[0,0]
```

Out[80]:
```
0
```

In [81]:
```
a2[0][0]
```

Out[81]:
```
0
```

##

In [82]:
```
a2[2:]
```

Out[82]:
```
array([[10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

In [83]:
```
a2[2:,3:]
```

Out[83]:
```
array([[13, 14],
       [18, 19],
       [23, 24]])
```

In [84]:
```
a2[1:5:2]
```

Out[84]:
```
array([[ 5,  6,  7,  8,  9],
       [15, 16, 17, 18, 19]])
```

In [85]:
```
a2[1:5:2,1:5:2]
```

Out[85]:
```
array([[ 6,  8],
       [16, 18]])
```

In [86]:
```
a2[[1,2],[3,4]]   #[1,2],[3,4]===>(1,3)(2,4)
```

Out[86]:
```
array([ 8, 14])
```

In [88]:
```
a2[1:3,3:5]    #slicing index 1,2 rows and column index 3,4
```

Out[88]:
```
array([[ 8,  9],
       [13, 14]])
```

In [105]:
```
a=np.array([10,20,30])
for i in a:
    print(i)        #printing each element
```

```
10
20
30
```

In [107]:
```
a2=np.arange(8).reshape(4,2)
print(a2)
for i in a2:
    print(i)     #printing each list 1-d array
```

```
[[0 1]
 [2 3]
 [4 5]
 [6 7]]
[0 1]
[2 3]
[4 5]
[6 7]
```

In [110]:
```
a3=np.arange(8).reshape(2,2,2)
print(a3)
```

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

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

In [112]:
```
for i in a3:
    print(i)    #giving 2d array as each element,list of list
```

```
[[0 1]
 [2 3]]
[[4 5]
 [6 7]]
```

Stacking

- hstack
- vstack

In [113]:
```
help(np.hstack)
```

```
Help on function hstack in module numpy:

hstack(tup)
    Stack arrays in sequence horizontally (column wise).

    This is equivalent to concatenation along the second axis, except for 1-D
    arrays where it concatenates along the first axis. Rebuilds arrays divided
    by `hsplit`.

    This function makes most sense for arrays with up to 3 dimensions. For
    instance, for pixel-data with a height (first axis), width (second axis),
    and r/g/b channels (third axis). The functions `concatenate`, `stack` and
    `block` provide more general stacking and concatenation operations.

    Parameters
    ----------
    tup : sequence of ndarrays
        The arrays must have the same shape along all but the second axis,
        except 1-D arrays which can be any length.

    Returns
    -------
    stacked : ndarray
        The array formed by stacking the given arrays.

    See Also
    --------
    stack : Join a sequence of arrays along a new axis.
    vstack : Stack arrays in sequence vertically (row wise).
    dstack : Stack arrays in sequence depth wise (along third axis).
    concatenate : Join a sequence of arrays along an existing axis.
    hsplit : Split array along second axis.
    block : Assemble arrays from blocks.

    Examples
    --------
    >>> a = np.array((1,2,3))
    >>> b = np.array((2,3,4))
    >>> np.hstack((a,b))
    array([1, 2, 3, 2, 3, 4])
    >>> a = np.array([[1],[2],[3]])
    >>> b = np.array([[2],[3],[4]])
    >>> np.hstack((a,b))
    array([[1, 2],
           [2, 3],
           [3, 4]])
```

In [115]:
```
a = np.array((1,2,3))
b = np.array((2,3,4))
np.hstack((a,b))
```

Out[115]:
```
array([1, 2, 3, 2, 3, 4])
```

In [118]:
```
a = np.array([[1],[2],[3]])
b = np.array([[2],[3],[4]])
np.hstack((a,b))
```

Out[118]:
```
array([[1, 2],
       [2, 3],
       [3, 4]])
```

In [123]:
```
b1=np.random.randint(1,10,(2,2))
b2=np.random.randint(1,10,(2))
print(b1)
print(b2)
np.vstack((b1,b2))  #will add rows
```

```
[[3 1]
 [5 7]]
[3 7]
```

Out[123]:
```
array([[3, 1],
       [5, 7],
       [3, 7]])
```

In [124]:
```
b3=np.random.randint(1,10,(2,1))
print(b1)
print(b3)
np.hstack((b1,b3))   #will add columns
```

```
[[3 1]
 [5 7]]
[[4]
 [3]]
```

Out[124]:
```
array([[3, 1, 4],
       [5, 7, 3]])
```

In [ ]:
```
```

fancy indexing or boolean indexing

In [313]:
```
a=np.random.randint(1,100,12)
a>50
```

Out[313]:
```
array([False, False,  True,  True,  True, False, False,  True, False,
       False,  True,  True])
```

In [314]:
```
(a>50).sum()
```

Out[314]:
```
6
```

In [316]:
```
a[a>50]   #boolean exp as index
```

Out[316]:
```
array([69, 63, 72, 68, 93, 65])
```

In [317]:
```
(a>50) & (a<80)
```

Out[317]:
```
array([False, False,  True,  True,  True, False, False,  True, False,
       False, False,  True])
```

In [322]:
```
a[(a>50)&(a<80)]
```

Out[322]:
```
array([], dtype=int32)
```

In [321]:
```
a[(a>50)&(a<80)]=6   #changing original array elements
a
```

Out[321]:
```
array([14,  7,  6,  6,  6, 20, 21,  6, 22, 45, 93,  6])
```

Array Manipulation

In [125]:
```
np.arange(32).reshape(4,8)
```

Out[125]:
```
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]])
```

In [126]:
```
np.arange(32).reshape(1,32)  #1 row, 32 columns
```

Out[126]:
```
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]])
```

In [127]:
```
np.arange(32).reshape(32,1)  #32 rows, 1 column
```

Out[127]:
```
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]])
```

In [128]:
```
np.arange(32).reshape(4,2,4)
```

Out[128]:
```
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]]])
```

In [129]:
```
np.arange(32).reshape(2,4,4)  #2 , 4by4 matrix
```

Out[129]:
```
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]]])
```

In [130]:
```
np.arange(32).reshape(8,-1)
```

Out[130]:
```
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]])
```

In [131]:
```
np.arange(32).reshape(-1,4)
```

Out[131]:
```
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]])
```

In [137]:
```
print(np.random.randint(1,100,(4,4)))
print(np.random.randint(1,100,(4,4)).reshape(16)) #converting 2d into 1d array
```

```
[[84 82 79  9]
 [10 19  1 45]
 [87 17 54 98]
 [91 59 80  7]]
[94 37  2 15 37 10 74 66 79 66 24 37 25 12  5 15]
```

In [140]:
```
#converting 3d array into 1d array

print(np.random.randint(1,100,(4,2,2)))
print(np.random.randint(1,100,(4,2,2)).flatten())
```

```
[[[64 35]
  [58 39]]

 [[55 79]
  [78 31]]

 [[81 76]
  [10 16]]

 [[70 88]
  [78 69]]]
[ 8  7 15 80 93 33 82  6 21 11 17 56 40 10 89 46]
```

In [150]:
```
a=np.array([5,2,3])   #1-d array
print(a)
print(np.expand_dims(a,axis=0)) #2d array  # expanding row
print(np.expand_dims(a,axis=1))   #expanding column
```

```
[5 2 3]
[[5 2 3]]
[[5]
 [2]
 [3]]
```

Mathematical Operations

Add two lists elementswise

In [146]:
```
l1=[1,2,3]
l2=[4,5,6]
list(map(lambda x,y:x+y,l1,l2))
```

Out[146]: [5, 7, 9]

In [147]:
```
l1+l2
```

Out[147]: [1, 2, 3, 4, 5, 6]

In [148]:
```
l1*l2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-148-bd77ec8884ac> in <module>
----> 1 l1*l2

TypeError: can't multiply sequence by non-int of type 'list'
```

In [149]:
```
a1=np.array(l1)
a2=np.array(l2)
a1+a2
```

Out[149]:
```
array([5, 7, 9])
```

In [151]:
```
print(a1)
print(a2)
print(a1+a2)
print(a1*a2)
print(a1%a2)
print(a1/a2)
print(a1//2)
```

```
[1 2 3]
[4 5 6]
[5 7 9]
[ 4 10 18]
[1 2 3]
[0.25 0.4  0.5 ]
[0 1 1]
```

In [153]:
```
print(a1+1)  #adding 1 to each element of a1
```

```
[2 3 4]
```

Array broadcasting rules

- When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward. Two dimensions are compatible when
- they are equal, or one of them is 1

In [183]:
```
a3=np.full((3,4),1)   #smaller one
a4=np.full((2,3,4),2)  #larger one
print(a3)
print(a4)
a5=a3-a4   # (2,3,4)
```

```
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
[[[2 2 2 2]
  [2 2 2 2]
  [2 2 2 2]]

 [[2 2 2 2]
  [2 2 2 2]
  [2 2 2 2]]]
```

In [184]:
```
a5  # (2,3,4)
```

Out[184]:
```
array([[[-1, -1, -1, -1],
        [-1, -1, -1, -1],
        [-1, -1, -1, -1]],

       [[-1, -1, -1, -1],
        [-1, -1, -1, -1],
        [-1, -1, -1, -1]]])
```

In [185]:
```
a3=np.arange(12).reshape(3,4)   #smaller
a4=np.full((2,1,4),2)    #larger
a3+a4   #(2,3,4)
```

Out[185]:
```
array([[[ 2,  3,  4,  5],
        [ 6,  7,  8,  9],
        [10, 11, 12, 13]],

       [[ 2,  3,  4,  5],
        [ 6,  7,  8,  9],
        [10, 11, 12, 13]]])
```

In [174]:
```
a3
```

Out[174]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [175]:
```
a4
```

Out[175]:
```
array([[[2, 2, 2, 2]],

       [[2, 2, 2, 2]]])
```

In [176]:
```
a3=np.arange(12).reshape(3,4)
a4=np.array([10,20,30,40,50,60,70,80]).reshape((2,1,4))
a3+a4
```

Out[176]:
```
array([[[10, 21, 32, 43],
        [14, 25, 36, 47],
        [18, 29, 40, 51]],

       [[50, 61, 72, 83],
        [54, 65, 76, 87],
        [58, 69, 80, 91]]])
```

In [177]:
```
a3
```

Out[177]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [178]:
```
a4
```

Out[178]:
```
array([[[10, 20, 30, 40]],

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

In [179]:
```
help(np.mean)
```

```
Help on function mean in module numpy:

mean(a, axis=None, dtype=None, out=None, keepdims=<no value>)
    Compute the arithmetic mean along the specified axis.

    Returns the average of the array elements.  The average is taken over
    the flattened array by default, otherwise over the specified axis.
    `float64` intermediate and return values are used for integer inputs.

    Parameters
    ----------
    a : array_like
        Array containing numbers whose mean is desired. If `a` is not an
        array, a conversion is attempted.
    axis : None or int or tuple of ints, optional
        Axis or axes along which the means are computed. The default is to
        compute the mean of the flattened array.

        .. versionadded:: 1.7.0

        If this is a tuple of ints, a mean is performed over multiple axes,
        instead of a single axis or all the axes as before.
    dtype : data-type, optional
        Type to use in computing the mean.  For integer inputs, the default
        is `float64`; for floating point inputs, it is the same as the
        input dtype.
    out : ndarray, optional
        Alternate output array in which to place the result.  The default
        is ``None``; if provided, it must have the same shape as the
        expected output, but the type will be cast if necessary.
        See `doc.ufuncs` for details.

    keepdims : bool, optional
        If this is set to True, the axes which are reduced are left
        in the result as dimensions with size one. With this option,
        the result will broadcast correctly against the input array.

        If the default value is passed, then `keepdims` will not be
        passed through to the `mean` method of sub-classes of
        `ndarray`, however any non-default value will be.  If the
        sub-class' method does not implement `keepdims` any
        exceptions will be raised.

    Returns
    -------
    m : ndarray, see dtype parameter above
        If `out=None`, returns a new array containing the mean values,
        otherwise a reference to the output array is returned.

    See Also
    --------
    average : Weighted average
    std, var, nanmean, nanstd, nanvar

    Notes
```

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

Note that for floating-point input, the mean is computed using the
same precision the input has.  Depending on the input data, this can
cause the results to be inaccurate, especially for `float32` (see
example below).  Specifying a higher-precision accumulator using the
`dtype` keyword can alleviate this issue.

By default, `float16` results are computed using `float32` intermediates
for extra precision.

Examples
--------
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([ 2.,  3.])
>>> np.mean(a, axis=1)
array([ 1.5,  3.5])

In single precision, `mean` can be inaccurate:

>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.54999924

Computing the mean in float64 is more accurate:

>>> np.mean(a, dtype=np.float64)
0.55000000074505806
```

In [190]:
```
a=np.array([40,30,20,10,50])
print(np.mean(a))
print(a.mean())
print(np.median(a))
print(a.max())
print(a.min())
```

```
30.0
30.0
30.0
50
10
```

In [308]:
```
ar1=np.arange(6).reshape(2,3)
ar2=np.arange(6).reshape(2,3)
#ar2=np.array([10,20,30,40,50,60]).reshape(2,3)
ar1
ar2
```

Out[308]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [194]:
```
print(a)
print(np.quantile(a,0.5))  #Medain q2
print(np.quantile(a,0.25))  #Median of first half q1
print(np.quantile(a,0.75))  #Median of second half q3
```

```
[40 30 20 10 50]
30.0
20.0
40.0
```

Logarithemic and exponential

In [209]:
```
np.log(5)
```

Out[209]:
```
1.6094379124341003
```

In [210]:
```
import math
math.log(5)
```

Out[210]:
```
1.6094379124341003
```

In [211]:
```
np.log2(4)
```

Out[211]:
```
2.0
```

In [212]:
```
np.log([2,3,4,6])
```

Out[212]:
```
array([0.69314718, 1.09861229, 1.38629436, 1.79175947])
```

In [213]:
```
np.log2([2,3,4,6])
```

Out[213]:
```
array([1.       , 1.5849625, 2.       , 2.5849625])
```

In [200]:
```
np.log(1)
```

Out[200]:
```
0.0
```

In [201]:
```
np.log(0)
```

```
C:\Users\APSSDC\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: RuntimeWar
ning: divide by zero encountered in log
  """Entry point for launching an IPython kernel.
```

Out[201]:
```
-inf
```

In [202]:
```
np.log(10)
```

Out[202]:
```
2.302585092994046
```

In [240]:
```
#np.log1p(x)=np.log(1+x)

print(np.log1p(0))
print(np.log(1))
```

```
0.0
0.0
```

In [204]:
```
np.log1p([2,3,4,6])
```

Out[204]:
```
array([1.09861229, 1.38629436, 1.60943791, 1.94591015])
```

In [205]:
```
np.log(np.exp(1))
```

Out[205]:
```
1.0
```

In [229]:
```
np.exp2(5)  #  2 power 5
```

Out[229]:
```
32.0
```

In [214]:
```
np.log(2)
```

Out[214]:
```
0.6931471805599453
```

In [207]:
```
np.expm1(2)
```

Out[207]:
```
6.38905609893065
```

In [223]:
```
np.exp(5)-1  #(e power 5 -1)  (e=2.7)
```

Out[223]:
```
147.4131591025766
```

In [224]:
```
np.exp([2,3,4,5,6])
```

Out[224]:
```
array([  7.3890561 ,  20.08553692,  54.59815003, 148.4131591 ,
       403.42879349])
```

In [226]:
```
np.exp2([2,3,4,78])
```

Out[226]:
```
array([4.00000000e+00, 8.00000000e+00, 1.60000000e+01, 3.02231455e+23])
```

Vectorize functions

In [227]:
```
def greater(a,b):
    if a>b:
        return a
    else:
        return b
```

In [230]:
```
greater([2,3,98],[54,1,2])
```

Out[230]:
```
[54, 1, 2]
```

In [231]:
```
[2,3,98]>[54,1,2]
```

Out[231]:
```
False
```

In [236]:
```
#element wise comparison

vgreater=np.vectorize(greater)
vgreater([2,3,98],[54,1,2])
```

Out[236]:
```
array([54,  3, 98])
```

In [246]:
```
def exp3(n):
    return 3**n

vexp3=np.vectorize(exp3)
vexp3([1,2,3])
```

Out[246]:
```
array([ 3,  9, 27])
```

In [248]:
```
a=np.array([[3,4],[1,2]])
b=np.eye(2)
```

In [249]:
```
a.T
```

Out[249]:
```
array([[3, 1],
       [4, 2]])
```

In [267]:
```
print(a)
print(b)
```

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

In [268]:
```
a*b   #normal multiplication
```

Out[268]:
```
array([[3., 0.],
       [0., 2.]])
```

In [266]:
```
np.dot(a,b)  #matrix multiplication
```

Out[266]:
```
array([[3., 4.],
       [1., 2.]])
```

Linear algebra

In [259]:
```
np.linalg.det(a)   #determinant of the matrix (ad-bc)
```

Out[259]:
```
2.0000000000000004
```

In [265]:
```
print(a)
```

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

In [260]:
```
np.linalg.det(b)
```

Out[260]:
```
1.0
```

In [262]:
```
ia=np.linalg.inv(a)
```

In [263]:
```
np.dot(a,ia)
```

Out[263]:
```
array([[1.00000000e+00, 0.00000000e+00],
       [1.11022302e-16, 1.00000000e+00]])
```

In [264]:
```
np.trace(a)
```

Out[264]:
```
5
```

In [283]:
```
%matplotlib inline

import matplotlib.pyplot as plt
img=plt.imread('python.jpg')
type(img)    #image is taking as ndarray
```

Out[283]:
```
numpy.ndarray
```

In [284]:
```
img.shape
```

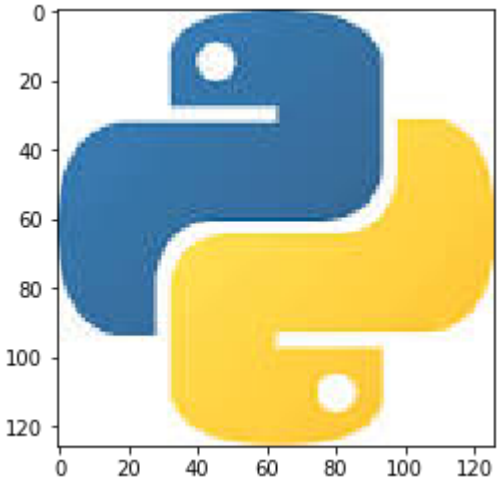
Out[284]:
```
(126, 126, 3)
```

In [285]:
```
img
```

Out[285]:
```
array([[[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       ...,

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[255, 255, 255],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]]], dtype=uint8)
```

In [286]:
```
plt.imshow(img)
```

Out[286]:
```
<matplotlib.image.AxesImage at 0x1d406a9eb70>
```

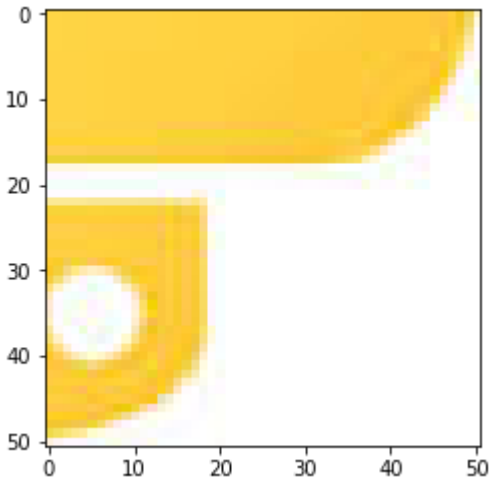


In [287]:
```
img[1,1,:]
```

Out[287]:
```
array([255, 255, 255], dtype=uint8)
```

In [295]:
```
plt.imshow(img[75:150,75:150,])
```

Out[295]:
```
<matplotlib.image.AxesImage at 0x1d407039908>
```



In [290]:
```
img[75,75,:]
```

Out[290]:
```
array([255, 213,  69], dtype=uint8)
```

In [291]:
```
img.flatten()
```

Out[291]:
```
array([255, 255, 255, ..., 255, 255, 255], dtype=uint8)
```

In [299]:
```
from sklearn.datasets import load_iris
data=load_iris()
```

In [297]:
```
data.keys()
```

Out[297]:
```
dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filenam
e'])
```

In [305]:
```
type(data['data'])
```

Out[305]:
```
numpy.ndarray
```

In [298]:
```
data['data']
```

Out[298]:
```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4],
       [4.6, 3.4, 1.4, 0.3],
       [5. , 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.1],
       [5.4, 3.7, 1.5, 0.2],
       [4.8, 3.4, 1.6, 0.2],
       [4.8, 3. , 1.4, 0.1],
       [4.3, 3. , 1.1, 0.1],
       [5.8, 4. , 1.2, 0.2],
       [5.7, 4.4, 1.5, 0.4],
       [5.4, 3.9, 1.3, 0.4],
       [5.1, 3.5, 1.4, 0.3],
       [5.7, 3.8, 1.7, 0.3],
```

In [307]:
```
data['data'].shape
```

Out[307]:
```
(150, 4)
```

In [301]:
```
print(data['DESCR'])
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988

The famous Iris database, first used by Sir R.A. Fisher. The dataset is taken
from Fisher's paper. Note that it's the same as in R, but not as in the UCI
Machine Learning Repository, which has two wrong data points.

This is perhaps the best known database to be found in the
pattern recognition literature.  Fisher's paper is a classic in the field and
is referenced frequently to this day.  (See Duda & Hart, for example.)  The
data set contains 3 classes of 50 instances each, where each class refers to a
type of iris plant.  One class is linearly separable from the other 2; the
latter are NOT linearly separable from each other.

.. topic:: References

   - Fisher, R.A. "The use of multiple measurements in taxonomic problems"
     Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to
     Mathematical Statistics" (John Wiley, NY, 1950).
   - Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analysis.
     (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.
   - Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System
```

Structure and Classification Rule for Recognition in Partially Exposed Environments". IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-2, No. 1, 67-71.
- Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule". IEEE Transactions on Information Theory, May 1972, 431-433.
- See also: 1988 MLC Proceedings, 54-64. Cheeseman et al"s AUTOCLASS II conceptual clustering system finds 3 classes in the data.
- Many, many more ...

In [326]:
```
arr = np.arange(9).reshape(3,3)
print(arr)
np.roll(arr,shift=1)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

Out[326]:
```
array([[8, 0, 1],
       [2, 3, 4],
       [5, 6, 7]])
```

In [ ]: