```
import numpy as np

a = np.array([100,101,102,103])
a
```

```
array([100, 101, 102, 103])
```

```
dir(a)
```

```
['T',
 '__abs__',
 '__add__',
 '__and__',
 '__array__',
 '__array_finalize__',
 '__array_function__',
 '__array_interface__',
 '__array_prepare__',
 '__array_priority__',
 '__array_struct__',
 '__array_ufunc__',
 '__array_wrap__',
 '__bool__',
 '__class__',
 '__class_getitem__',
 '__complex__',
 '__contains__',
 '__copy__',
 '__deepcopy__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__divmod__',
 '__dlpack__',
 '__dlpack_device__',
 '__doc__',
 '__eq__',
 '__float__',
 '__floordiv__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__iand__',
 '__ifloordiv__',
 '__ilshift__',
 '__imatmul__',
 '__imod__',
 '__imul__',
 '__index__',
 '__init__',
 '__init_subclass__',
 '__int__',
 '__invert__',
 '__ior__',
 '__ipow__',
 '__irshift__',
 '__isub__',
 '__iter__',
 '__itruediv__',
 '__ixor__',
 '__le__',
 '__len__',
 '__lshift__',
```

```
# create a two-dimensional array
#shape
#ndim
#size
b = np.array([[1,2],[3]])
b.shape
```

```
C:\Users\Sampath\AppData\Local\Temp\ipykernel_16572\2150675660.py:5: VisibleDeprecationWarning: Creating an ndarray from ragged nes
  b = np.array([[1,2],[3]])
(2,)
```

```
# Indexing multidimensional
b[0,1] # same as b[(0,1)]
```

```
2.0
```

```
#write a np.array of shape (2,3,4)
c = np.array([[[1,21,3,44],[0,-1,0.,4],[1,-1.,2.,10]],[[1,8,3,4],[9,-1,0.,4],[20,-1.,2.,0]]])
c.ndim
```

```
3
```

```
c[0,0,1]
```

```
21.0
```

```
b[1,1] = 10 # also for assignment
b
```

```
array([[ 1.,  2.,  1.],
       [ 3., 10.,  0.],
       [ 5.,  6.,  1.]])
```

```
np.array([0,4,-4], dtype=complex)
```

```
array([ 0.+0.j,  4.+0.j, -4.+0.j])
```

```
np.empty((2,3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

```
x = np.zeros((4,2,3)) # default dtype is 'float'
x.shape
```

```
(4, 2, 3)
```

```
y = np.zeros((4,3), dtype=int)
y
```

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

```
a
```

```
array([100, 101, 102, 103])
```

```
np.ones_like(y) # Similar to a with 1
```

```
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

```
np.zeros_like(y, dtype=float)
```

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

```
np.arange?
```

```
Docstring:
arange([start,] stop[, step,], dtype=None, *, like=None)

Return evenly spaced values within a given interval.

``arange`` can be called with a varying number of positional arguments:

* ``arange(stop)``: Values are generated within the half-open interval
  ``[0, stop)`` (in other words, the interval including `start` but
  excluding `stop`).
* ``arange(start, stop)``: Values are generated within the half-open
  interval ``[start, stop)``.
* ``arange(start, stop, step)`` Values are generated within the half-open
```

```
  interval ``[start, stop)``, with spacing between values given by
  ``step``.

For integer arguments the function is roughly equivalent to the Python
built-in :py:class:`range`, but returns an ndarray rather than a ``range``
instance.

When using a non-integer step, such as 0.1, it is often better to use
`numpy.linspace`.

See the Warning sections below for more information.

Parameters
----------
start : integer or real, optional
    Start of interval.  The interval includes this value.  The default
    start value is 0.
stop : integer or real
    End of interval.  The interval does not include this value, except
    in some cases where `step` is not an integer and floating point
    round-off affects the length of `out`.
step : integer or real, optional
    Spacing between values.  For any output `out`, this is the distance
    between two adjacent values, ``out[i+1] - out[i]``.  The default
    step size is 1.  If `step` is specified as a position argument,
    `start` must also be given.
dtype : dtype, optional
    The type of the output array.  If `dtype` is not given, infer the data
    type from the other input arguments.
like : array_like, optional
    Reference object to allow the creation of arrays which are not
    NumPy arrays. If an array-like passed in as ``like`` supports
    the ``__array_function__`` protocol, the result will be defined
    by it. In this case, it ensures the creation of an array object
    compatible with that passed in via this argument.

    .. versionadded:: 1.20.0

Returns
-------
arange : ndarray
    Array of evenly spaced values.

    For floating point arguments, the length of the result is
```

```
list(range(7))
```

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

```
# Initializing an array from a sequence
# np.arange()  - the NumPy equivalent of range, except that it can generate floating-point sequences.
# np.linspace() - more useful way of creating an sequence with given length

list(range(7))
```

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

```
np.arange(1.5,4,0.5) # Generate a sequence starting with 1.5 with width 0.5 and excluding 4
```

```
array([1.5, 2. , 2.5, 3. , 3.5])
```

```
np.linspace?
```

```
Signature:
np.linspace(
    start,
    stop,
    num=50,
    endpoint=True,
    retstep=False,
    dtype=None,
    axis=0,
)
Docstring:
Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the
interval [`start`, `stop`].

The endpoint of the interval can optionally be excluded.

.. versionchanged:: 1.16.0
    Non-scalar `start` and `stop` are now supported.

.. versionchanged:: 1.20.0
    Values are rounded towards ``-inf`` instead of ``0`` when an
```

```
    integer ``dtype`` is specified. The old behavior can
    still be obtained with ``np.linspace(start, stop, num).astype(int)``

Parameters
----------
start : array_like
    The starting value of the sequence.
stop : array_like
    The end value of the sequence, unless `endpoint` is set to False.
    In that case, the sequence consists of all but the last of ``num + 1``
    evenly spaced samples, so that `stop` is excluded.  Note that the step
    size changes when `endpoint` is False.
num : int, optional
    Number of samples to generate. Default is 50. Must be non-negative.
endpoint : bool, optional
    If True, `stop` is the last sample. Otherwise, it is not included.
    Default is True.
retstep : bool, optional
    If True, return (`samples`, `step`), where `step` is the spacing
    between samples.
dtype : dtype, optional
    The type of the output array.  If `dtype` is not given, the data type
    is inferred from `start` and `stop`. The inferred dtype will never be
    an integer; `float` is chosen even if the arguments would produce an
    array of integers.

    .. versionadded:: 1.9.0

axis : int, optional
    The axis in the result to store the samples.  Relevant only if start
    or stop are array-like.  By default (0), the samples will be along a
    new axis inserted at the beginning. Use -1 to get an axis at the end.

    .. versionadded:: 1.16.0
```

```
np.linspace(1,10,20).reshape(4,5)
```

```
array([[ 1.        ,  1.47368421,  1.94736842,  2.42105263,  2.89473684],
       [ 3.36842105,  3.84210526,  4.31578947,  4.78947368,  5.26315789],
       [ 5.73684211,  6.21052632,  6.68421053,  7.15789474,  7.63157895],
       [ 8.10526316,  8.57894737,  9.05263158,  9.52631579, 10.        ]])
```

```
np.linspace(10,20,5) # generate  evenly spaced array of the five numbers between 1 and 20 inclusive:
```

```
array([10. , 12.5, 15. , 17.5, 20. ])
```

```
np.linspace?
```

```
Signature:
np.linspace(
    start,
    stop,
    num=50,
    endpoint=True,
    retstep=False,
    dtype=None,
    axis=0,
)
Docstring:
Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the
interval [`start`, `stop`].

The endpoint of the interval can optionally be excluded.

.. versionchanged:: 1.16.0
    Non-scalar `start` and `stop` are now supported.

.. versionchanged:: 1.20.0
    Values are rounded towards ``-inf`` instead of ``0`` when an
    integer ``dtype`` is specified. The old behavior can
    still be obtained with ``np.linspace(start, stop, num).astype(int)``

Parameters
----------
start : array_like
    The starting value of the sequence.
stop : array_like
    The end value of the sequence, unless `endpoint` is set to False.
    In that case, the sequence consists of all but the last of ``num + 1``
    evenly spaced samples, so that `stop` is excluded.  Note that the step
    size changes when `endpoint` is False.
num : int, optional
    Number of samples to generate. Default is 50. Must be non-negative.
```

```
    endpoint : bool, optional
        If True, `stop` is the last sample. Otherwise, it is not included.
        Default is True.
    retstep : bool, optional
        If True, return (`samples`, `step`), where `step` is the spacing
        between samples.
    dtype : dtype, optional
        The type of the output array.  If `dtype` is not given, the data type
        is inferred from `start` and `stop`. The inferred dtype will never be
        an integer; `float` is chosen even if the arguments would produce an
        array of integers.

        .. versionadded:: 1.9.0

    axis : int, optional
        The axis in the result to store the samples.  Relevant only if start
        or stop are array-like.  By default (0), the samples will be along a
        new axis inserted at the beginning. Use -1 to get an axis at the end.

        .. versionadded:: 1.16.0
```

```
x = np.linspace(10,20,10, retstep=True)
x
```

```
(array([10.        , 11.11111111, 12.22222222, 13.33333333, 14.44444444,
        15.55555556, 16.66666667, 17.77777778, 18.88888889, 20.        ]),
 1.1111111111111112)
```

```
x = np.linspace(0,200,100,retstep=True)
#dx # dx = (end-start)/(num-1) = 2*pi/99 = 0.0634665...
x
```

```
(array([  0.        ,   2.02020202,   4.04040404,   6.06060606,
          8.08080808,  10.1010101 ,  12.12121212,  14.14141414,
         16.16161616,  18.18181818,  20.2020202 ,  22.22222222,
         24.24242424,  26.26262626,  28.28282828,  30.3030303 ,
         32.32323232,  34.34343434,  36.36363636,  38.38383838,
         40.4040404 ,  42.42424242,  44.44444444,  46.46464646,
         48.48484848,  50.50505051,  52.52525253,  54.54545455,
         56.56565657,  58.58585859,  60.60606061,  62.62626263,
         64.64646465,  66.66666667,  68.68686869,  70.70707071,
         72.72727273,  74.74747475,  76.76767677,  78.78787879,
         80.80808081,  82.82828283,  84.84848485,  86.86868687,
         88.88888889,  90.90909091,  92.92929293,  94.94949495,
         96.96969697,  98.98989899, 101.01010101, 103.03030303,
        105.05050505, 107.07070707, 109.09090909, 111.11111111,
        113.13131313, 115.15151515, 117.17171717, 119.19191919,
        121.21212121, 123.23232323, 125.25252525, 127.27272727,
        129.29292929, 131.31313131, 133.33333333, 135.35353535,
        137.37373737, 139.39393939, 141.41414141, 143.43434343,
        145.45454545, 147.47474747, 149.49494949, 151.51515152,
        153.53535354, 155.55555556, 157.57575758, 159.5959596 ,
        161.61616162, 163.63636364, 165.65656566, 167.67676768,
        169.6969697 , 171.71717172, 173.73737374, 175.75757576,
        177.77777778, 179.7979798 , 181.81818182, 183.83838384,
        185.85858586, 187.87878788, 189.8989899 , 191.91919192,
        193.93939394, 195.95959596, 197.97979798, 200.        ]),
 2.0202020202020203)
```

```
x = np.linspace(0,5, 10, endpoint=False) #endpoint to False omits the final point in the sequence, as for np.arange
x
```

```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
```

```
# Initializing an Array from a Function
def f(i,j):
    return 2*i*j

np.fromfunction(f,(4,3))
```

```
array([[ 0.,  0.,  0.],
       [ 0.,  2.,  4.],
       [ 0.,  4.,  8.],
       [ 0.,  6., 12.]])
```

```
np.fromfunction(lambda i,j: 2*i*j,(4,3))
```

```
array([[ 0.,  0.,  0.],
       [ 0.,  2.,  4.],
       [ 0.,  4.,  8.],
       [ 0.,  6., 12.]])
```

```
# create a “comb” of values in an array of length N for which every nth element is one but with zeros everywhere else
N, n = 20, 5
def f(i):
    return (i% n == 0) * 1

comb = np.fromfunction(f,(N,), dtype=int)
comb
```

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

```
#ndarray Attributes for Introspection
a = np.array(((1,0,1),(0,1,0)))
a.shape # returns the axis dimensions
        # 2 rows, 3 columns
```

```
(2, 3)
```

```
a.ndim # rank (number of dimensions)
```

```
2
```

```
a = np.array([[[1,2],[2,3]],[[2,1],[1,2]],[[2,1],[1,2]]])
a.shape # depthxlengthxwidth
```

```
(3, 2, 2)
```

```
a.size # total number of elements
a.dtype
a.data
```

```
<memory at 0x000001F2BF383D60>
```

NumPy's Basic Data Types (commonly used)

- `shape` - The array dimensions: the size of the array along each of its axes, returned as a tuple of integers
- `ndim` - Number of axes (dimensions); note that `ndim == len(shape)`
- `size` - The total number of elements in the array, equal to the product of the elements of shape
- `dtype` -The array’s data type
- `data` -The “buffer” in memory containing the actual elements of the array
- `itemsize` -The size in bytes of each element

```
# Universal Functions
x = np.linspace(1,5,5)
x**3
```

```
array([  1.,   8.,  27.,  64., 125.])
```

```
x-1
```

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

```
np.sqrt(x-1)
```

```
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ])
```

```
y = np.exp(-np.linspace(0,2,5))
np.sin(x-y)
```

```
array([ 0.        ,  0.98431873,  0.48771645, -0.59340065, -0.98842844])
```

```
# matrix multiplication in numpy
a = np.array(((1,2),(2,1)))
b = a
a*b # elementwise multiplication
```

```
array([[1, 4],
       [4, 1]])
```

```
#a@b # matrix multiplication
b = a.T
```

```
np.matmul(a,b)
```

```
array([[14, 14],
       [14, 26]])
```

```
a.dot(b) # also try np.dot(a,b)
np.dot(a,b)
```

```
array([[5, 4],
       [4, 5]])
```

```
np.dot?
```

```
Docstring:
dot(a, b, out=None)

Dot product of two arrays. Specifically,

- If both `a` and `b` are 1-D arrays, it is inner product of vectors
  (without complex conjugation).

- If both `a` and `b` are 2-D arrays, it is matrix multiplication,
  but using :func:`matmul` or ``a @ b`` is preferred.

- If either `a` or `b` is 0-D (scalar), it is equivalent to :func:`multiply`
  and using ``numpy.multiply(a, b)`` or ``a * b`` is preferred.

- If `a` is an N-D array and `b` is a 1-D array, it is a sum product over
  the last axis of `a` and `b`.

- If `a` is an N-D array and `b` is an M-D array (where ``M>=2``), it is a
  sum product over the last axis of `a` and the second-to-last axis of `b`::

    dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])

Parameters
----------
a : array_like
    First argument.
b : array_like
    Second argument.
out : ndarray, optional
    Output argument. This must have the exact kind that would be returned
    if it was not used. In particular, it must have the right type, must be
    C-contiguous, and its dtype must be the dtype that would be returned
    for `dot(a,b)`. This is a performance feature. Therefore, if these
    conditions are not met, an exception is raised, instead of attempting
    to be flexible.

Returns
-------
output : ndarray
    Returns the dot product of `a` and `b`.  If `a` and `b` are both
    scalars or both 1-D arrays then a scalar is returned; otherwise
    an array is returned.
    If `out` is given, then it is returned.

Raises
------
ValueError
    If the last dimension of `a` is not the same size as
    the second-to-last dimension of `b`.

See Also
--------
vdot : Complex-conjugating dot product.
tensordot : Sum products over arbitrary axes.
einsum : Einstein summation convention.
matmul : '@' operator as method with out parameter.
linalg.multi_dot : Chained dot product.
```

```
# Comparision and logic opera
a = np.array([[1,2,3],[2,3,4]])
b = np.array([1,2])
np.dot(b,a)
```

```
array([ 5,  8, 11])
```

```
a>100
```

```
array([False, False, False, False,  True,  True])
```

```
(a<10) | (a>100) # | for or
```

```
array([ True,  True, False, False,  True,  True])
```

NumPy's Special Values, `nan` and `inf`

NumPy defines two special values to represent the outcome of calculations, which are not mathematically defined or not finite.

- The value `np.nan` ("Not a Number," NaN) represents the outcome of a calculation that is not a well-defined mathematical operation (e.g. 0/0);
- `np.inf` represents infinity.

Creating a magic square

Its a square matrix of order N in which the entries in each row, column and main diagonal sum to the same number equal to $N(N^2+1)/2$.

Steps:

- Start in the middle of the top row, and let n=1
- Insert n into the current grid position.
- If $n = N^2$the grid is complete so stop. Otherwise, increment n.
- Move diagonally up and right, wrapping to the first column or last row if the move leads outside the grid. If this cell is already filled, move vertically down one space instead.
- Return to step 2.

```
# Creating an NxN magic square. N must be odd
N = 5
msquare = np.zeros((N,N), dtype = int)

n = 1
i, j = 0, N//2

while n <= N**2:
    msquare[i,j] = n
    n += 1
    newi, newj = (i-1)%N, (j+1)%N
    if msquare[newi, newj]:
        i += 1
    else:
        i, j = newi, newj
msquare
```

```
array([[17, 24,  1,  8, 15],
       [23,  5,  7, 14, 16],
       [ 4,  6, 13, 20, 22],
       [10, 12, 19, 21,  3],
       [11, 18, 25,  2,  9]])
```

flatten and ravel

- Suppose you wish to "flatten" a multidimensional array onto a single axis.
- NumPy provides two methods to do this: `flatten` and `ravel`.
  - Both flatten the array into its internal (row-major) ordering, as described earlier.
  - `flatten` returns an independent copy of the elements and is generally slower than `ravel`, which tries to return a view to the flattened array.

```
a = np.array([[[1,2,3],[4,5,6],[7,8,9]],[[1,2,3],[4,5,6],[7,8,9]]])
b = a.flatten() # Create a independent, flatten copy of a
a
```

```
array([[[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]],

       [[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]]])
```

```
b[3] = 0
b
```

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

```
a
```

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

```
c = a.ravel()
c
```

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

```
c[3] = 0
c
```

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

```
a
```

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

`resize` and `reshape`

- An array may be resized to a compatible shape with `resize` method, which takes the new dimensions as its arguments.
- It's invese method to `flatten`.
- The reshape method returns a view on the array with its elements reshaped as required.
- The original array is not modified, but the objects share the same underlying data.

```
import numpy as np
a = np.linspace(1,8,8)
a
```

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

```
a.resize(2,2,2) # reshapes a in place, doesn't return anything
a
```

```
array([[[1., 2.],
        [3., 4.]],

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

```
b = a.reshape(4,3)
b
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-13-c1de3da34992> in <module>
----> 1 b = a.reshape(4,3)
      2 b

ValueError: cannot reshape array of size 8 into shape (4,3)
```

SEARCH STACK OVERFLOW

```
b[0,0] = -99
b
```

```
array([[-99.,   2.],
       [  3.,   4.]])
```

```
a
```

```
array([-99.,   2.,   3.,   4.])
```

Transposing an Array

- The method `transpose` returns a view of an array with the axes transposed.
- For a two-dimensional array, this is the usual matrix transpose.
- Transposing a one-dimensional array returns the array unchanged.

```
a = np.linspace(1,16,16)
a = a.reshape(2,2,4)
```

```
print(a)
print(a.transpose()) # Think about it???
print(a.transpose().shape)
```

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

 [[ 9. 10. 11. 12.]
  [13. 14. 15. 16.]]]
[[[ 1.  9.]
  [ 5. 13.]]

 [[ 2. 10.]
  [ 6. 14.]]

 [[ 3. 11.]
  [ 7. 15.]]

 [[ 4. 12.]
  [ 8. 16.]]]
(4, 2, 2)
```

```
a.T
```

```
array([[[ 1.,  9.],
        [ 3., 11.],
        [ 5., 13.],
        [ 7., 15.]],

       [[ 2., 10.],
        [ 4., 12.],
        [ 6., 14.],
        [ 8., 16.]]])
```

```
b = np.array([100,101,102,103])
b.transpose()
```

```
array([100, 101, 102, 103])
```

Merging and Splitting Arrays

- A clutch of NumPy methods merge and split arrays in different ways.
- `np.vstack` - stack arrays vertically (in sequential rows);
- `np.hstack` - stack arrays horizontally (in sequential columns), and
- `np.dstack` - stack arrays depthwise(along a third axis);
- The inverse operations, `np.vsplit`, `np.hsplit` and `np.dsplit`, split a single array into multiple arrays by rows, columns or depth.
- If this argument is a single integer, the array is split into that number of equal-sized arrays along the appropriate axis

```
a = np.array([0,0,0,0])
b = np.array([1,1,1,1])
c = np.array([2,2,2,2])
d = np.dstack((a,b,c)) # stack arrays vertically
# depthxrowsXcoln
print(d.ndim)
print(d1.ndim)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-39-a4f1877a89d0> in <module>
      2 b = np.array([1,1,1,1])
      3 c = np.array([2,2,2,2])
----> 4 d = np.dstack((a,b,c),(a,b,c)) # stack arrays vertically
      5 # depthxrowsXcoln
      6 print(d.ndim)

<__array_function__ internals> in dstack(*args, **kwargs)

TypeError: _dstack_dispatcher() takes 1 positional argument but 2 were given
```

SEARCH STACK OVERFLOW

```
a = np.array([0,0,0,0])
b = np.array([1,1,1,1])
c = np.array([2,2,2,2])
d1 = np.vstack((a,b,c))
d1.shape
```

```
(3, 4)
```

```
np.hstack((a,b,c))
```

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

```
np.dstack((a,b,c))
```

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

```
a = np.arange(5)
a
```

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

```
b = np.hsplit(a,2)
b
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-45-cb6d4ed6c810> in <module>
----> 1 b = np.hsplit(a,2)
      2 b

<__array_function__ internals> in hsplit(*args, **kwargs)

~\AppData\Local\Programs\Python\Python39\lib\site-packages\numpy\lib\shape_base.py
in hsplit(ary, indices_or_sections)
    940         return split(ary, indices_or_sections, 1)
    941     else:
--> 942         return split(ary, indices_or_sections, 0)
    943
    944

<__array_function__ internals> in split(*args, **kwargs)

~\AppData\Local\Programs\Python\Python39\lib\site-packages\numpy\lib\shape_base.py
in split(ary, indices_or_sections, axis)
    870         N = ary.shape[axis]
    871         if N % sections:
--> 872             raise ValueError(
    873                 'array split does not result in an equal division') from
None
    874     return array_split(ary, indices_or_sections, axis)

ValueError: array split does not result in an equal division
```

Indexing and Slicing an Array

-

```
a = np.linspace(10,16,6)
a
```

```
array([10. , 11.2, 12.4, 13.6, 14.8, 16. ])
```

```
a[1:5:2] # a stride of 2
```

```
array([11.2, 13.6])
```

```
a[2::2] # a stride of -2
```

```
array([12.4, 14.8])
```

```
a[2::-2] # a stride of -2
```

```
array([12.4, 10. ])
```

- Multidimensional arrays have an index for each axis.
- If you want to select every item along a particular axis, replace its index with a single colon:
- The special ellipsis notation (`...`)is useful for high-rank arrays: in an index, it represents as many colons as are necessary to represent the remaining axes.

```
a = np.linspace(1,12,12).reshape(4,3)
a
```

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

```
a[3,2]
```

```
12.0
```

```
a[:0] # everything in the third row
```

```
array([], shape=(0, 3), dtype=float64)
```

```
a[:,1] # the second column
```

```
array([ 2.,  5.,  8., 11.])
```

```
a = np.linspace(1,12,12).reshape(4,3)
a
```

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

```
a[1:,1:2] # middle rows, second column onwards
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-91-8878ee542c49> in <module>
----> 1 a[1:,1,2] # middle rows, second column onwards

IndexError: too many indices for array: array is 2-dimensional, but 3 were indexed
```

SEARCH STACK OVERFLOW

```
a[3,1, ...]
```

```
array(11.)
```

```
a[:,1] = 0 # set all elements in the second column to zero
a
```

```
array([[ 1.,  0.,  3.],
       [ 4.,  0.,  6.],
       [ 7.,  0.,  9.],
       [10.,  0., 12.]])
```

Advanced Indexing

- NumPy arrays can be indexed by sequences that aren't simple tuples of integers, including other lists, arrays of integers and tuple of tuples.

```
a = np.linspace(1,10,10)
a
```

```
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

```
ia = [1,4,5]
a[ia]
```

```
array([0.1, 0.4, 0.5])
```

```
ia = np.array(((1,3),(5,7),(2,3)))
a[ia] # an array to be formed from the specified indexes
```

```
array([[2., 4.],
       [6., 8.],
       [3., 4.]])
```

```
a = np.linspace(1,12,12).reshape(4,3)
a
```

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

```
ia = np.array(((1,0),(2,1)))
ja = np.array( ((0,1),(1,2)))
a[ia,ja]
```

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

Adding an Axis

- To add an axis (i.e, dimesion) to an array, insert `np.newaxis` in the desired position.

```
a = np.linspace(1,4,4).reshape(2,2)
a # a 2x2 array (rk=2)
```

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

```
a.shape
```

```
(2, 2)
```

```
b = a[:, np.newaxis]
```

```
b
```

```
array([[[1., 2.]],

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

```
b.shape
```

```
(2, 1, 2)
```

Meshes

- `mesh` is useful to evaluate a multidimensional function on a grid of points.
- The function `np.meshgrid` is passes a series of N one-dimensional arrays representing co-ordinates along each dimension and returns a set of N–dimensional arrays comprising a mesh of cordinates at which the function can be evaluated.

```
x = np.linspace(0,5,6)
y = np.linspace(0,3,4)
X, Y = np.meshgrid(x,y)
X
```

```
array([[0., 1., 2., 3., 4., 5.],
       [0., 1., 2., 3., 4., 5.],
       [0., 1., 2., 3., 4., 5.],
       [0., 1., 2., 3., 4., 5.]])
```

```
Y
```

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

- The x array is repeated as rows down X and the y array as columns across Y.
- A function of two co-ordinates can be evaluated on the grid as simply $f(X, Y)$
- Optinal argument `sparse=True`, will return sparse grid to conserve memory.

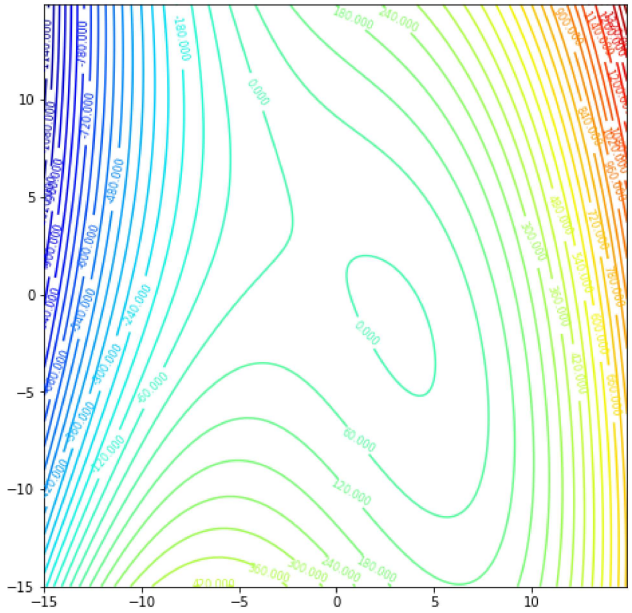
```
X, Y = np.meshgrid(x,y, sparse=True)
X,Y
```

```
(array([[0., 1., 2., 3., 4., 5.]]),
 array([[0.],
        [1.],
        [2.],
        [3.]]))
```

```
import matplotlib.pyplot as plt

f = lambda x,y : 0.3*x**3+y**2+2*x*y-6*x-3*y+4
x = np.arange(-15,15,0.1)
y = np.arange(-15,15,0.1)

fig= plt.figure(figsize = (16,8))
X, Y = np.meshgrid(x,y)
Z  = f(X,Y)
N=50
cp = plt.contour(X, Y, Z, N, cmap ='jet') # you can use ax.contourf()
plt.clabel(cp, inline=1, fontsize=8)
plt.axis('scaled')
plt.show()
```




```
import matplotlib.pyplot as plt

f = lambda x,y : x**2 - 2*x*y + 4*y**2
x = np.arange(-5,5,0.1)
y = np.arange(-5,5,0.1)

fig= plt.figure(figsize = (8,16))
X, Y = np.meshgrid(x,y)
Z  = f(X,Y)
levels = np.arange(-1.0,50,4)
cp = plt.contour(X, Y, Z, cmap ='jet',linestyles = 'dashed',levels=levels) # you can use ax.contourf()
plt.clabel(cp, inline=1, fontsize=8)
plt.axis('scaled')
plt.show()
```

Broadcasting

- Recall that for arrays of the same size, binary operations are performed on an element-by-element basis.

```
a = np.array((0,1,2))
b = np.array([5,5,5])
a+b
```

```
array([5, 6, 7])
```

Broadcasting allows these types of binary operations to be performed on arrays of different sizes-for example, we can just as easily add a scalar (think of it as a zero-dimensional array) to an array:

```
a+5
```

```
array([5, 6, 7])
```

We can similarly extend this to arrays of higher dimension. Observe the result when we add a one-dimensional array to a two-dimensional array:

```
M = np.ones((3,3))
M
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

```
M+a
```

```
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

Here the one-dimensional array `a` is stretched, or broadcast across the second dimension in order to match the shape of `M`.

```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]
print(a)
print(b)
```

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

```
a+b
```

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

Just as before we stretched or broadcasted one value to match the shape of the other, here we've stretched both `a` and `b` to match a common shape, and the result is a two-dimensional array! The geometry of these are visualized in the following figure: Drag Racing

Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

- [Rule 1:] If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.
- [Rule 2:] If the shape of the two arrays doesn't match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
- [Rule 3:] If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

```
# Example 1
M = np.ones((2,3))
a = np.arange(3)
```

```
print(M.shape)
print(a.shape)
```

```
(2, 3)
(3,)
```

```
M
```

```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

```
a
```

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

```
M+a
```

```
array([[1., 2., 3.],
       [1., 2., 3.]])
```

```
# Example 2
a = np.arange(3).reshape((3,1))
b = np.arange(3)
```

```
print(a)
print(a.shape)
```

```
[[0]
 [1]
 [2]]
(3, 1)
```

```
print(b)
print(b.shape)
```

```
[0 1 2]
(3,)
```

```
a+b
```

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

```
# Example 3

M = np.ones((3,2))
a = np.arange(3).reshape((3,1))
```

```
M
```

```
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

```
a = np.arange(3)
a
M+a
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-132-b84d995db4c1> in <module>
      1 a = np.arange(3)
      2 a
----> 3 M+a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

SEARCH STACK OVERFLOW

```
b = a[:, np.newaxis]
b.shape
```

```
(3, 1)
```

```
M+b
```

```
array([[1., 1.],
       [2., 2.],
       [3., 3.]])
```

Maximum and Minimum Values

- NumPy arrays have the methods `min` and `max`, which return the minimum and maximum values in th array.
- By default, a single value for the flattened array is returned.
- To find maximum/ minimum along a given axis, use the following axis argument.
- The methods `argmax` and `argmin` given the index of max and min values of the array after flatten.
- `a.max(axis=0)` and `a.argmin(axis=0)` giving the maximum and index of the maximum values of each column in array `a`.
- The same for `axis=1`: maximum values along each row.

```
a = np.array([[3,0,1,-1],[2,-1,-2,4],[1,7,0,4]])
a
```

```
array([[ 3,  0,  1, -1],
       [ 2, -1, -2,  4],
       [ 1,  7,  0,  4]])
```

```
a.flatten()[9]
```

```
7
```

```
print(np.argmin(a))
print(np.argmax(a))
```

```
6
9
```

```
a.max()
```

```
7
```

```
print(a.max(axis=0)) # maxima in each column
print(a.min(axis=1)) # minima in each row
```

```
[3 7 1 4]
[-1 -2  0]
```

```
a.argmin()
```

```
6
```

```
a.ravel()[a.argmin()] == a.min()
```

```
True
```

```
print(a.argmax(axis=0)) # row indexes of maxima in each column
```

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

```
print(a.argmax(axis=1))
```

```
[0 3 1]
```

Example: Consider the following function on the interval $[0, L]$:

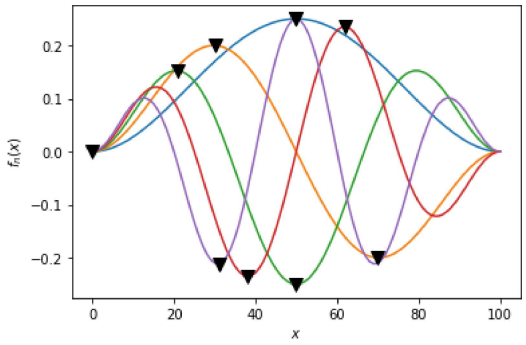
$f_n(x) = x(L-x)\sin(\frac{2\pi x}{\lambda_n}),\ n = 1, 2, 3, \ldots$

```
N = 100
L = 1

def f(i,n):
    x = i * L / N
    lam = 2 * L / (n+1)
    return x * (L-x) * np.sin(2*np.pi*x/lam)
```

```
import matplotlib.pyplot as plt
a = np.fromfunction(f,(N+1,5))
min_i = a.argmin(axis=0)
```

```
max_i = a.argmax(axis=0)
plt.plot(a)
plt.plot(min_i,a[min_i,np.arange(5)], 'v', c='k', markersize=10)
plt.plot(max_i,a[max_i,np.arange(5)], 'v', c='k',markersize=10)
plt.xlabel(r'$x$')
plt.ylabel(r'$f_n(x)$')
plt.show()
```



Sorting an Array

- NumPy arrays can be sorted in several different ways with the `sort` method.
- By default, this method sorts multidimensional arrays along their last axis.
- To sort along some other axis, set the `axis` argument.
- `np.argsort` returns the indexes
- `np.searchsorted` takes a *sorted* array, `a` and one or more values, v, and returns the indexes in `a` at which the values should be entered to maintain its order.

```
a = np.array([5,-1,2,4,0,4])
a.sort()
print(a)
```

```
[-1  0  2  4  4  5]
```

```
b = np.array([[0,3,-2],[7,1,3],[4,0,-1]])
print(b)
```

```
[[ 0  3 -2]
 [ 7  1  3]
 [ 4  0 -1]]
```

```
b.sort() # sort the numbers along each row
print(b)
```

```
[[-2  0  3]
 [ 1  3  7]
 [-1  0  4]]
```

```
b = np.array([[0,3,-2],[7,1,3],[4,0,-1]])
print(b)
```

```
[[ 0  3 -2]
 [ 7  1  3]
 [ 4  0 -1]]
```

```
b.sort(axis=0) #for each column, order the numbers by row”
print(b)
```

```
[[ 0  0 -2]
 [ 4  1 -1]
 [ 7  3  3]]
```

```
b = np.array([[0,3,-2],[7,1,3],[4,0,-1]])
b.sort(axis=0) #for each column, order the numbers by row
b
```

```
array([[ 0,  0, -2],
       [ 4,  1, -1],
       [ 7,  3,  3]])
```

```
a= np.array([3,0,-1,1])
np.argsort(a)
```

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

```
a[np.argsort(a)]
```

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

```
a = np.array([1,2,3,4])
np.searchsorted(a,3.5)
```

```
3
```

```
np.searchsorted(a,(3.5,0,1.1))
```

```
array([3, 0, 1], dtype=int64)
```

Statistics

We discuss several satatistical methods in `numpy`.

- maxima and minima
- Percentiles
- Average
- Variance
- Correlations
- Statistical Distributions

Maxima and Minima

- `np.max` and `np.min` for the maxima and minima.
- `np.nanmin` and `np.nanmax` to avoid giving `np.nan`
- There are variants to avoid `np.nan`: `np.nanargmax` and `np.nanargmin`
- To compare two arrays, element by element, we use `np.fmin` / `np.fmax`

```
a = np.sqrt(np.linspace(-2,2,4))
print(a)
```

```
[       nan        nan 0.81649658 1.41421356]
<ipython-input-82-8f16baae91f2>:1: RuntimeWarning: invalid value encountered in sqrt
  a = np.sqrt(np.linspace(-2,2,4))
```

```
np.min(a), np.max(a)
```

```
(nan, nan)
```

```
np.nanmax(a),np.nanmin(a)
```

```
(1.4142135623730951, 0.8164965809277259)
```

```
np.argmax(a), np.argmin(a)
```

```
(0, 0)
```

```
np.nanargmin(a),np.nanargmax(a)
```

```
(2, 3)
```

```
b = np.array([1, -5, 6, 2])
c = np.array([0,np.nan,-1,3])
```

```
np.fmax(b,c)
```

```
array([ 0., -5., -1.,  2.])
```

```
np.fmin(b,c)
```

```
array([ 0., -5., -1.,  2.])
```

Percentiles

- `np.percentile` method returnd a specified percentile, q of the data along an axis (or along a flatted version of the array if no axis is given).
- The minimum of an array is the value at $q = 0$(0th percentile), the maximum is the value at $q = 100$(100th percentile)
- The median is the value at $q = 50$(50th percentile)

```
a = np.array([[0,0.6,1.2],[1.8,2.4,3.0]])
#np.percentile(a,q=75)
a
```

```
array([[0. , 0.6, 1.2],
       [1.8, 2.4, 3. ]])
```

```
np.percentile(a,0)
```

```
0.0
```

```
np.percentile(a,100)
```

```
3.0
```

```
np.percentile(a,75)
```

```
2.25
```

```
np.percentile(a,50,axis=0)
```

```
array([0.9, 1.5, 2.1])
```

```
np.percentile(a,50,axis=1)
```

```
array([0.6, 2.4])
```

Sample Statistics

- `np.mean` - calculates the arithmetic mean of the values along a specified axis of an array.
- `np.average` - calculates the weighted average.
- `np.median` - calculates the arithmetic median of the values along a specified axis of an array.
- `np.std` - calculates the standard deviation of the values along a specified axis of an array.
- `np.var` - calculates the variance
- `np.cov` - calculates the covariance matrix
- `np.corrcoef` - calculates the correlation matrix

```
x =np.array([1.,4.,9,16])
np.mean(x)
```

```
7.5
```

```
np.median(x)
```

```
6.5
```

```
np.average(x, weights=[0.3, 0,1,0])
```

```
7.153846153846154
```

```
x = np.array([[1,8,27],[-0.5,1,0.]])
av,sw = np.average(x,weights=[0,1.,0.1], axis=1,returned=True)
```

```
print(av)
print(sw)
```

```
[9.72727273 0.90909091]
[1.1 1.1]
```

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

```
1.118033988749895
```

```
np.std(x,ddof=1)
```

```
1.2909944487358056
```

```
np.sqrt(np.var(x))
```

```
1.118033988749895
```

```
np.var(x,ddof=1)
```

```
1.6666666666666667
```

```
x = np.array([ [0.1,0.3,0.4,0.8,0.9],
               [3.1,3.8,0.4,1.8,5.9],
               [10,8.3,3.4,2.8,0.9],
               [10.2,18.3,3.34,3.48,1.9]
])
print(x)
```

```
[[ 0.1   0.3   0.4   0.8   0.9 ]
 [ 3.1   3.8   0.4   1.8   5.9 ]
 [10.    8.3   3.4   2.8   0.9 ]
 [10.2  18.3   3.34  3.48  1.9 ]]
```

```
x = np.cov(x)
```

```
(array([1.11746018e+09, 2.05707027e+02, 3.30796511e-09, 1.51513515e-02]), array([[-0.03501364, -0.08449775,  0.99551761, -0.0240599
       [-0.01252619, -0.62083032, -0.03420962,  0.78309803],
       [ 0.45242944,  0.68999579,  0.08796629,  0.55809974],
       [ 0.89102454, -0.36240262, -0.00602717, -0.27331901]]))
```

```
print(np.var(x,axis=1, ddof=1))
```

```
[2.96045817e+14 3.78899727e+13 4.94294915e+16 1.91718353e+17]
```

```
print(np.corrcoef(x))
```

```
[[ 1.          0.30521027 -0.91314708]
 [ 0.30521027  1.         -0.06056094]
 [-0.91314708 -0.06056094  1.        ]]
```

```
x = np.array([1,2,3,4,5])
y = np.array([0.08,0.32,0.41,0.48,0.86])
print(np.cov(x,y))
```

```
[[2.5    0.43  ]
 [0.43   0.0806]]
```

```
print(np.corrcoef(x,y))
```

```
[[1.         0.95792373]
 [0.95792373 1.        ]]
```

Special Distributions

- `np.random` - generate random numbers from any of several distributions
- `.random_sample` - create an array of given shape of random numbers from the uniform distribution [0,1).
- `np.random.randint` - take upto three arguments and return integer: *low*, *high* and *size*
- `np.random.normal` - select a random samples from the normal distribution having mean = `loc` and standard deviation = `scale`, which default to 0 and 1.

```
np.random.seed(3)
np.random.randint(1,10,10) # random integers in [1,10]
```

```
array([9, 4, 9, 9, 1, 6, 4, 6, 8, 7])
```

```
np.random.randint(1,10,size=5)
```

```
array([3, 2, 4, 6, 9])
```

```
np.random.seed(42)
np.random.randint(1,10,10)
```

```
array([7, 4, 8, 5, 7, 3, 7, 8, 5, 4])
```

```
# Generate random floating-point numbers
np.random.random_sample((2,3))
```

```
array([[0.28352508, 0.69313792, 0.44045372],
       [0.15686774, 0.54464902, 0.78031476]])
```

```
np.random.rand(2,3)
```

```
array([[0.30636353, 0.22195788, 0.38797126],
       [0.93638365, 0.97599542, 0.67238368]])
```

```
np.random.randint(4) # random integer from [0,4]
```

```
2
```

```
np.random.randint(4,size=10)
```

```
array([1, 3, 3, 3, 3, 2, 1, 1, 2, 1])
```

```
np.random.randint(4,size=(3,2))
```

```
array([[2, 3],
       [2, 3],
       [3, 0]])
```

```
np.random.randint(1,4,(3,5))
```

```
array([[3, 1, 3, 3, 1],
       [1, 3, 2, 1, 2],
       [2, 2, 1, 2, 1]])
```

```
np.random.normal()
```

```
-0.76883635031923
```

```
np.random.normal(scale=5,size=3) #mean=0
```

```
array([-1.15015361,  3.72528133,  9.88055392])
```

```
x=np.random.normal(100.,8.,size=100)
print(np.mean(x))
print(np.std(x,ddof=1))
```

```
99.63981245017402
7.969975013305313
```

```
np.random.gamma?
```

```
Docstring:
gamma(shape, scale=1.0, size=None)

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters,
`shape` (sometimes designated "k") and `scale` (sometimes designated
"theta"), where both parameters are > 0.

.. note::
    New code should use the ``gamma`` method of a ``default_rng()``
    instance instead; please see the :ref:`random-quick-start`.

Parameters
----------
shape : float or array_like of floats
    The shape of the gamma distribution. Must be non-negative.
scale : float or array_like of floats, optional
    The scale of the gamma distribution. Must be non-negative.
    Default is equal to 1.
size : int or tuple of ints, optional
    Output shape.  If the given shape is, e.g., ``(m, n, k)``, then
    ``m * n * k`` samples are drawn.  If size is ``None`` (default),
    a single value is returned if ``shape`` and ``scale`` are both scalars.
    Otherwise, ``np.broadcast(shape, scale).size`` samples are drawn.
```

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

        See Also
        --------
        scipy.stats.gamma : probability density function, distribution or
            cumulative density function, etc.
        Generator.gamma: which should be used for new code.

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

        .. math:: p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},

        where :math:`k` is the shape and :math:`\theta` the scale,
        and :math:`\Gamma` is the Gamma function.

        The Gamma distribution is often used to model the times to failure of
        electronic components, and arises naturally in processes for which the
        waiting times between Poisson distributed events are relevant.

        References
        ----------
        .. [1] Weisstein, Eric W. "Gamma Distribution." From MathWorld--A
               Wolfram Web Resource.
               http://mathworld.wolfram.com/GammaDistribution.html
        .. [2] Wikipedia, "Gamma distribution",
               https://en.wikipedia.org/wiki/Gamma_distribution
```

```
import numpy as np
```

```
A = np.array([[2,-1,0,0],[0,7,2,-4],[1,2,3,5],[1,2,5,6]])
b = np.array([0,0,10,15])
```

```
np.linalg.solve(A,b)
```

```
array([-0.06756757, -0.13513514,  2.09459459,  0.81081081])
```