Vectors, Matrices, Multidimensionals Arrays

NumPy Website

Importing the Modules

In [59]:
```
import numpy as np
```

The NumPy Array Object

In [ ]:
```
np.ndarray?
```

```
Init signature: np.ndarray(self, /, *args, **kwargs)
Docstring:
ndarray(shape, dtype=float, buffer=None, offset=0,
        strides=None, order=None)

An array object represents a multidimensional, homogeneous array
of fixed-size items.  An associated data-type object describes the
format of each element in the array (its byte-order, how many bytes it
occupies in memory, whether it is an integer, a floating point number,
or something else, etc.)

Arrays should be constructed using `array`, `zeros` or `empty` (refer
to the See Also section below).  The parameters given here refer to
a low-level method (`ndarray(...)`) for instantiating an array.

For more information, refer to the `numpy` module and examine the
methods and attributes of an array.

Parameters
----------
(for the __new__ method; see Notes below)

shape : tuple of ints
    Shape of created array.
dtype : data-type, optional
    Any object that can be interpreted as a numpy data type.
buffer : object exposing buffer interface, optional
    Used to fill the array with data.
offset : int, optional
    Offset of array data in buffer.
strides : tuple of ints, optional
    Strides of data in memory.
order : {'C', 'F'}, optional
    Row-major (C-style) or column-major (Fortran-style) order.

Attributes
----------
T : ndarray
    Transpose of the array.
data : buffer
    The array's elements, in memory.
```

```
dtype : dtype object
    Describes the format of the elements in the array.
flags : dict
    Dictionary containing information related to memory use, e.g.,
    'C_CONTIGUOUS', 'OWNDATA', 'WRITEABLE', etc.
flat : numpy.flatiter object
    Flattened version of the array as an iterator.  The iterator
    allows assignments, e.g., ``x.flat = 3`` (See `ndarray.flat` for
    assignment examples; TODO).
imag : ndarray
    Imaginary part of the array.
real : ndarray
    Real part of the array.
size : int
    Number of elements in the array.
itemsize : int
    The memory use of each array element in bytes.
nbytes : int
    The total number of bytes required to store the array data,
    i.e., ``itemsize * size``.
ndim : int
    The array's number of dimensions.
shape : tuple of ints
    Shape of the array.
strides : tuple of ints
    The step-size required to move from one element to the next in
    memory. For example, a contiguous ``(3, 4)`` array of type
    ``int16`` in C-order has strides ``(8, 2)``.  This implies that
    to move from element to element in memory requires jumps of 2 bytes.
    To move from row-to-row, one needs to jump 8 bytes at a time
    (``2 * 4``).
ctypes : ctypes object
    Class containing properties of the array needed for interaction
    with ctypes.
base : ndarray
    If the array is a view into another array, that array is its `base`
    (unless that array is also a view).  The `base` array is where the
    array data is actually stored.

See Also
--------
array : Construct an array.
zeros : Create an array, each element of which is zero.
empty : Create an array, but leave its allocated memory unchanged (i.e.,
        it contains "garbage").
dtype : Create a data-type.
numpy.typing.NDArray : A :term:`generic <generic type>` version
                       of ndarray.

Notes
-----
There are two modes of creating an array using ``__new__``:

1. If `buffer` is None, then only `shape`, `dtype`, and `order`
   are used.
2. If `buffer` is an object exposing the buffer interface, then
   all keywords are interpreted.

No ``__init__`` method is needed because the array is fully initialized
after the ``__new__`` method.
```

```
Examples
--------
These examples illustrate the low-level `ndarray` constructor.  Refer
to the `See Also` section above for easier ways of constructing an
ndarray.

First mode, `buffer` is None:

>>> np.ndarray(shape=(2,2), dtype=float, order='F')
array([[0.0e+000, 0.0e+000], # random
       [     nan, 2.5e-323]])

Second mode:

>>> np.ndarray((2,), buffer=np.array([1,2,3]),
...            offset=np.int_().itemsize,
...            dtype=int) # offset = 1*itemsize, i.e. skip first element
array([2, 3])
File:           c:\miniconda3\lib\site-packages\numpy\__init__.py
Type:           type
Subclasses:     chararray, recarray, memmap, matrix, MaskedArray
```

In [ ]:
```
data = np.array([[1, 2], [3, 4], [5, 6]]) # Matrix data
```

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

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

In [ ]:
```
data
```

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

In [ ]:
```
data.ndim s# data dimension --> or use np.ndim(data)
```

Out[ ]:
```
2
```

In [ ]:
```
data.shape # shape of data
```

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

In [ ]:
```
data.size # elements number
```

Out[ ]:
```
6
```

In [ ]:
```
data.dtype # each element's data type
```

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

In [ ]:
```
data.nbytes # whole data's size in bytes
```
Out[ ]:
```
24
```

Data Type

In [ ]:
```
# int -> int 8, 16, 32, 64
# uint -> 8, 16, 32, 64
# bool
# float -> 16, 32, 64, 128
# complex -> 64, 128, 256
```

In [ ]:
```
np.array([1, 2, 3], dtype = int)
```
Out[ ]:
```
array([1, 2, 3])
```

In [ ]:
```
np.array([1, 2, 3], dtype = float)
```
Out[ ]:
```
array([1., 2., 3.])
```

In [ ]:
```
np.array([1, 2, 3], dtype = complex)
```
Out[ ]:
```
array([1.+0.j, 2.+0.j, 3.+0.j])
```

In [ ]:
```
data = np.array([1.0, 2.0, 3.0])
print(data.dtype)
```
```
float64
```

In [ ]:
```
data = np.array([1, 2, 3], dtype = float)
```

In [ ]:
```
data
```
Out[ ]:
```
array([1., 2., 3.])
```

In [ ]:
```
data.dtype
```
Out[ ]:
```
dtype('float64')
```

In [ ]:
```
data = np.array(data, dtype = int) # Type casting
```

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

In [ ]:
```
data.dtype
```
Out[ ]:
```
dtype('int32')
```
In [ ]:
```
data = np.array([1, 2, 3], dtype = float)
```
In [ ]:
```
data
```
Out[ ]:
```
array([1., 2., 3.])
```
In [ ]:
```
data.astype(int) # another Type casting
```
Out[ ]:
```
array([1, 2, 3])
```
In [ ]:
```
data
```
Out[ ]:
```
array([1., 2., 3.])
```
In [ ]:
```
d1 = np.array([1, 2, 3], dtype = float)
d2 = np.array([1, 2, 3], dtype = complex)
```
In [ ]:
```
d1 + d2 # Automatic Type casting
```
Out[ ]:
```
array([2.+0.j, 4.+0.j, 6.+0.j])
```
In [ ]:
```
_.dtype # last cell's data type
```
Out[ ]:
```
dtype('complex128')
```
In [ ]:
```
np.sqrt(np.array([-1, 0, 1]))
```
```
C:\Users\vision\AppData\Local\Temp/ipykernel_16044/208196152.py:1: RuntimeWarning: inval
id value encountered in sqrt
  np.sqrt(np.array([-1, 0, 1]))
```
Out[ ]:
```
array([nan,  0.,  1.])
```
In [ ]:
```
np.sqrt(np.array([-1, 0, 1]), dtype = complex)
```
Out[ ]:
```
array([0.+1.j, 0.+0.j, 1.+0.j])
```
In [ ]:
```
data = np.array([1, 2, 3], dtype = complex)
```
In [ ]:
```
data
```

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

In [ ]:
```
data.real
```

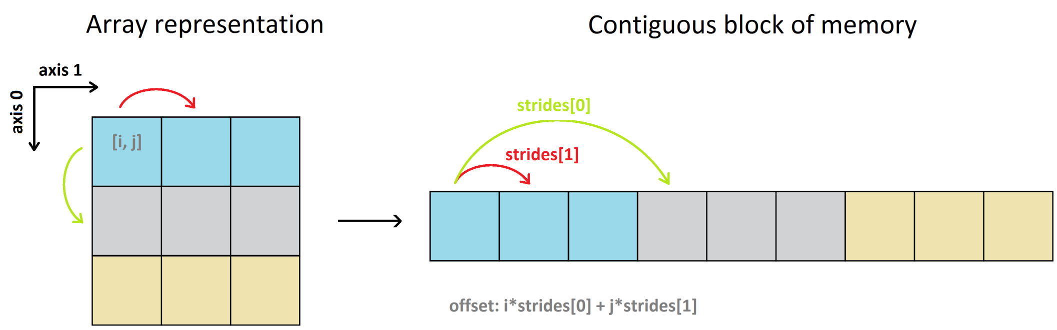
Out[ ]: `array([1., 2., 3.])`

In [ ]:
```
data.imag
```

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

Order of Array Data in Memory

row-major format : keyword argument `order= 'C'` | column-major format : keyword argument `order= 'F'` default format is row-major

> NumPy array attribute : `ndarray.strides`
>
> Example: C-order array A with shape `(2, 3)` | data type : `int32` | total memory buffer for the array : `2 × 3 × 4 = 24`
>
> strides attribute of this array : `(4 × 3, 4 × 1) = (12, 4) | F` order , the strides : `(4, 8)`
>
> Stride
>
> 
>
>
> Ref

In [ ]:
```
a = np.arange(1, 10).reshape(3, 3)
```

In [ ]:
```
a
```

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

In [ ]:
```
a.itemsize # each elemnt's byte size
```

Out[ ]: `4`

In [ ]:

```
a.size
```

Out[ ]:
```
9
```

In [ ]:
```
a.nbytes
```

Out[ ]:
```
36
```

In [ ]:
```
a.strides
```

Out[ ]:
```
(12, 4)
```

In [ ]:
```
b = np.array([[1, 4, 7],
              [2, 5, 8],
              [3, 6, 9]]).T
```

In [ ]:
```
b
```

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

In [ ]:
```
b.strides
```

Out[ ]:
```
(4, 12)
```

In [ ]:
```
a == b
```

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

Creating Arrays

> ## Arrays Created from Lists and Other Array-Like Objects

In [60]:
```
data = np.array([1, 2, 3, 4])
```

In [ ]:
```
data.ndim
```

Out[ ]:
```
1
```

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

Out[ ]: (4,)

In [ ]:
```
data.size
```

Out[ ]: 4

In [ ]:
```
data = np.array((1, 2, 3, 4))
```

In [ ]:
```
data
```

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

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

Out[ ]: (4,)

In [ ]:
```
data = np.array({1, 2, 3, 4})
```

In [ ]:
```
data
```

Out[ ]:
```
array({1, 2, 3, 4}, dtype=object)
```

In [ ]:
```
data = np.array([[1, 2],
                 [3, 4]])
```

In [ ]:
```
data
```

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

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

Out[ ]: (2, 2)

Arrays Filled with Constant Values

In [61]:
```
np.zeros(5)
```

Out[61]:
```
array([0., 0., 0., 0., 0.])
```

In [ ]:
```
np.zeros((3, 4))
```

In [ ]:
```
np.ones(10)
```

In [66]:
```
np.ones((5, 5))
```

Out[66]:
```
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., 1.]])
```

In [65]:
```
data = np.ones(4)
```

In [64]:
```
data
```

Out[64]:
```
array([1, 1, 1, 1])
```

In [63]:
```
data.dtype
```

Out[63]:
```
dtype('int32')
```

In [67]:
```
data = np.ones(4, dtype = int)
```

In [68]:
```
data
```

Out[68]:
```
array([1, 1, 1, 1])
```

In [69]:
```
data.dtype
```

Out[69]:
```
dtype('int32')
```

In [70]:
```
x1 = 8.3 * np.ones(15)
```

In [71]:
```
x1
```

Out[71]:
```
array([8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3,
       8.3, 8.3])
```

In [72]:
```
x2 = np.full(15, 8.3)
```

In [73]:
```
x2
```

Out[73]:
```
array([8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3, 8.3,
       8.3, 8.3])
```

In [74]:
```
x3 = np.full((4, 3), 8.3)
```

In [75]:
```
x3
```

Out[75]:
```
array([[8.3, 8.3, 8.3],
       [8.3, 8.3, 8.3],
       [8.3, 8.3, 8.3],
       [8.3, 8.3, 8.3]])
```

In [76]:
```
x4 = np.empty(5)
```

In [77]:
```
x4
```

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

In [78]:
```
x4.fill(6.9)
```

In [79]:
```
x4
```

Out[79]:
```
array([6.9, 6.9, 6.9, 6.9, 6.9])
```

In [80]:
```
x5 = np.full(10, 5.8)
```

In [81]:
```
x5
```

Out[81]:
```
array([5.8, 5.8, 5.8, 5.8, 5.8, 5.8, 5.8, 5.8, 5.8, 5.8])
```

Arrays Filled with Incremental Sequences

In [ ]:
```
# np.arange(start = 0, end (up to but not including), increment = 1)
```

In [82]:
```
np.arange(0, 10 , 1)
```

Out[82]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [83]:
```
np.arange(2)
```

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

In [84]:
```
np.arange(0)
```

Out[84]:
```
array([], dtype=int32)
```

In [85]:
```
np.arange(1, 11, 3)
```

Out[85]:
```
array([ 1,  4,  7, 10])
```

In [ ]:
```
# np.linspace(start, end ( up to but including), total number of elements in the array)
```

In [87]:
```
np.linspace(0, 10 , 11, dtype = int)
```

Out[87]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [86]:
```
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

Returns
-------
samples : ndarray
    There are `num` equally spaced samples in the closed interval
    ``[start, stop]`` or the half-open interval ``[start, stop)``
    (depending on whether `endpoint` is True or False).
step : float, optional
    Only returned if `retstep` is True

    Size of spacing between samples.


See Also
--------
arange : Similar to `linspace`, but uses a step size (instead of the
         number of samples).
geomspace : Similar to `linspace`, but with numbers spaced evenly on a log
            scale (a geometric progression).
logspace : Similar to `geomspace`, but with the end points specified as
           logarithms.

Examples
--------
>>> np.linspace(2.0, 3.0, num=5)
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
array([2. ,  2.2,  2.4,  2.6,  2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
(array([2.  ,  2.25,  2.5 ,  2.75,  3.  ]), 0.25)

Graphical illustration:

>>> import matplotlib.pyplot as plt
>>> N = 8
>>> y = np.zeros(N)
>>> x1 = np.linspace(0, 10, N, endpoint=True)
>>> x2 = np.linspace(0, 10, N, endpoint=False)
>>> plt.plot(x1, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(x2, y + 0.5, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-0.5, 1])
(-0.5, 1)
>>> plt.show()
File:      c:\miniconda3\lib\site-packages\numpy\core\function_base.py
Type:      function
```

In [88]:

```
np.linspace(0, 10)
```

Out[88]:
```
array([ 0.        ,  0.20408163,  0.40816327,  0.6122449 ,  0.81632653,
        1.02040816,  1.2244898 ,  1.42857143,  1.63265306,  1.83673469,
        2.04081633,  2.24489796,  2.44897959,  2.65306122,  2.85714286,
        3.06122449,  3.26530612,  3.46938776,  3.67346939,  3.87755102,
        4.08163265,  4.28571429,  4.48979592,  4.69387755,  4.89795918,
        5.10204082,  5.30612245,  5.51020408,  5.71428571,  5.91836735,
        6.12244898,  6.32653061,  6.53061224,  6.73469388,  6.93877551,
        7.14285714,  7.34693878,  7.55102041,  7.75510204,  7.95918367,
        8.16326531,  8.36734694,  8.57142857,  8.7755102 ,  8.97959184,
        9.18367347,  9.3877551 ,  9.59183673,  9.79591837, 10.        ])
```

In [89]:
```
np.linspace(0, 10 , 11, endpoint = False)
```

Out[89]:
```
array([0.        , 0.90909091, 1.81818182, 2.72727273, 3.63636364,
       4.54545455, 5.45454545, 6.36363636, 7.27272727, 8.18181818,
       9.09090909])
```

In [90]:
```
np.arange?
```

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

Return evenly spaced values within a given interval.

Values are generated within the half-open interval ``[start, stop)``
(in other words, the interval including `start` but excluding `stop`).
For integer arguments the function is equivalent to the Python built-in
`range` function, but returns an ndarray rather than a list.

When using a non-integer step, such as 0.1, the results will often not
be consistent.  It is better to use `numpy.linspace` for these cases.

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
    The type of the output array.  If `dtype` is not given, infer the data
    type from the other input arguments.
like : array_like
    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
    ``ceil((stop - start)/step)``.  Because of floating point overflow,
    this rule may result in the last element of `out` being greater
    than `stop`.

See Also
--------
numpy.linspace : Evenly spaced numbers with careful handling of endpoints.
numpy.ogrid: Arrays of evenly spaced numbers in N-dimensions.
numpy.mgrid: Grid-shaped arrays of evenly spaced numbers in N-dimensions.

Examples
--------
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0.,  1.,  2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 5])
Type:      builtin_function_or_method
```

Arrays Filled with Logarithmic Sequences

In [91]:

```python
np.logspace(0, 3, 20) # 20 data points between 10**0 =1 to 10**3 = 1000
```

Out[91]:

```
array([   1.        ,    1.43844989,    2.06913808,    2.97635144,
          4.2813324 ,    6.15848211,    8.8586679 ,   12.74274986,
         18.32980711,   26.36650899,   37.92690191,   54.55594781,
         78.47599704,  112.88378917,  162.37767392,  233.57214691,
        335.98182863,  483.29302386,  695.19279618, 1000.        ])
```

In [92]:

```python
np.logspace?
```

```
Signature:
np.logspace(
    start,
    stop,
    num=50,
    endpoint=True,
    base=10.0,
    dtype=None,
    axis=0,
)
Docstring:
Return numbers spaced evenly on a log scale.

In linear space, the sequence starts at ``base ** start``
(`base` to the power of `start`) and ends with ``base ** stop``
(see `endpoint` below).

.. versionchanged:: 1.16.0
```

```
    Non-scalar `start` and `stop` are now supported.

Parameters
----------
start : array_like
    ``base ** start`` is the starting value of the sequence.
stop : array_like
    ``base ** stop`` is the final value of the sequence, unless `endpoint`
    is False.  In that case, ``num + 1`` values are spaced over the
    interval in log-space, of which all but the last (a sequence of
    length `num`) are returned.
num : integer, optional
    Number of samples to generate.  Default is 50.
endpoint : boolean, optional
    If true, `stop` is the last sample. Otherwise, it is not included.
    Default is True.
base : array_like, optional
    The base of the log space. The step size between the elements in
    ``ln(samples) / ln(base)`` (or ``log_base(samples)``) is uniform.
    Default is 10.0.
dtype : dtype
    The type of the output array.  If `dtype` is not given, the data type
    is inferred from `start` and `stop`. The inferred type will never be
    an integer; `float` is chosen even if the arguments would produce an
    array of integers.
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


Returns
-------
samples : ndarray
    `num` samples, equally spaced on a log scale.

See Also
--------
arange : Similar to linspace, with the step size specified instead of the
         number of samples. Note that, when used with a float endpoint, the
         endpoint may or may not be included.
linspace : Similar to logspace, but with the samples uniformly distributed
           in linear space, instead of log space.
geomspace : Similar to logspace, but with endpoints specified directly.

Notes
-----
Logspace is equivalent to the code

>>> y = np.linspace(start, stop, num=num, endpoint=endpoint)
... # doctest: +SKIP
>>> power(base, y).astype(dtype)
... # doctest: +SKIP

Examples
--------
>>> np.logspace(2.0, 3.0, num=4)
array([ 100.        ,  215.443469  ,  464.15888336, 1000.        ])
```

```
>>> np.logspace(2.0, 3.0, num=4, endpoint=False)
array([100.        ,  177.827941  ,  316.22776602,  562.34132519])
>>> np.logspace(2.0, 3.0, num=4, base=2.0)
array([4.        ,  5.0396842 ,  6.34960421,  8.        ])

Graphical illustration:

>>> import matplotlib.pyplot as plt
>>> N = 10
>>> x1 = np.logspace(0.1, 1, N, endpoint=True)
>>> x2 = np.logspace(0.1, 1, N, endpoint=False)
>>> y = np.zeros(N)
>>> plt.plot(x1, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(x2, y + 0.5, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-0.5, 1])
(-0.5, 1)
>>> plt.show()
File:      c:\miniconda3\lib\site-packages\numpy\core\function_base.py
Type:      function
```

In [93]:

```
np.logspace(0, 3, 20, base = 3)
```

Out[93]:

```
array([ 1.        ,  1.18941917,  1.41471797,  1.68269268,  2.00142693,
        2.38053557,  2.83145465,  3.36778645,  4.00570977,  4.764468  ,
        5.66694959,  6.7403785 ,  8.01713542,  9.53573458, 11.34198554,
       13.49037507, 16.04571076, 19.08507602, 22.70015534, 27.        ])
```

Meshgrid Arrays

In [94]:

```
x = np.array([-1, 0 , 1])
y = np.array([-2, 0 , 2])
```

In [95]:

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

In [96]:

```
X
```

Out[96]:

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

In [97]:

```
Y
```

Out[97]:

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

In [98]:

```
x = np.array([-1.5, -1, -0.5, 0, 0.5, 1, 1.5])
```

In [99]:

```
y = np.array([-2, -1, 0, 1, 2])
```

In [100…
```
X, Y = np.meshgrid(x, y)
```

In [101…
```
X
```

Out[101…
```
array([[-1.5, -1. , -0.5,  0. ,  0.5,  1. ,  1.5],
       [-1.5, -1. , -0.5,  0. ,  0.5,  1. ,  1.5],
       [-1.5, -1. , -0.5,  0. ,  0.5,  1. ,  1.5],
       [-1.5, -1. , -0.5,  0. ,  0.5,  1. ,  1.5],
       [-1.5, -1. , -0.5,  0. ,  0.5,  1. ,  1.5]])
```

In [102…
```
Y
```

Out[102…
```
array([[-2, -2, -2, -2, -2, -2, -2],
       [-1, -1, -1, -1, -1, -1, -1],
       [ 0,  0,  0,  0,  0,  0,  0],
       [ 1,  1,  1,  1,  1,  1,  1],
       [ 2,  2,  2,  2,  2,  2,  2]])
```

In [103…
```
z = (X + Y) ** 2
```

In [104…
```
z
```

Out[104…
```
array([[12.25,  9.  ,  6.25,  4.  ,  2.25,  1.  ,  0.25],
       [ 6.25,  4.  ,  2.25,  1.  ,  0.25,  0.  ,  0.25],
       [ 2.25,  1.  ,  0.25,  0.  ,  0.25,  1.  ,  2.25],
       [ 0.25,  0.  ,  0.25,  1.  ,  2.25,  4.  ,  6.25],
       [ 0.25,  1.  ,  2.25,  4.  ,  6.25,  9.  , 12.25]])
```

In [ ]:
```
# np.mgrid, np.ogrid
```

Creating Uninitialized Arrays

In [107…
```
np.empty(10, dtype = float)
```

Out[107…
```
array([0.90909091, 1.81818182, 2.72727273, 3.63636364, 4.54545455,
       5.45454545, 6.36363636, 7.27272727, 8.18181818, 9.09090909])
```

Creating Arrays with Properties of Other Arrays

In [ ]:
```
# np.ones_like, np.zeros_like, np.full_like, np.empty_like
```

In [108…
```
def f(x):
    y = np.ones_like(x)
    return y
```

In [109…
```
x = np.array([1, 5, 8, 9, 12], dtype = complex)
```

In [111…

```
x
```

Out[111…
```
array([ 1.+0.j,  5.+0.j,  8.+0.j,  9.+0.j, 12.+0.j])
```

In [110…
```
f(x)
```

Out[110…
```
array([1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j])
```

Creating Matrix Arrays

In [ ]:
```
# np.array([[], []])
```

In [112…
```
np.identity(5)
```

Out[112…
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

In [116…
```
np.eye(5)
```

Out[116…
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

In [113…
```
np.eye(5, k = 0)
```

Out[113…
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

In [114…
```
np.eye(5, k = 3)
```

Out[114…
```
array([[0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

In [115…
```
np.eye(5, k = -1)
```

Out[115…
```
array([[0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

In [123…

```
np.diag?
```

```
Signature: np.diag(v, k=0)
Docstring:
Extract a diagonal or construct a diagonal array.

See the more detailed documentation for ``numpy.diagonal`` if you use this
function to extract a diagonal and wish to write to the resulting array;
whether it returns a copy or a view depends on what version of numpy you
are using.

Parameters
----------
v : array_like
    If `v` is a 2-D array, return a copy of its `k`-th diagonal.
    If `v` is a 1-D array, return a 2-D array with `v` on the `k`-th
    diagonal.
k : int, optional
    Diagonal in question. The default is 0. Use `k>0` for diagonals
    above the main diagonal, and `k<0` for diagonals below the main
    diagonal.

Returns
-------
out : ndarray
    The extracted diagonal or constructed diagonal array.

See Also
--------
diagonal : Return specified diagonals.
diagflat : Create a 2-D array with the flattened input as a diagonal.
trace : Sum along diagonals.
triu : Upper triangle of an array.
tril : Lower triangle of an array.

Examples
--------
>>> x = np.arange(9).reshape((3,3))
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

>>> np.diag(x)
array([0, 4, 8])
>>> np.diag(x, k=1)
array([1, 5])
>>> np.diag(x, k=-1)
array([3, 7])

>>> np.diag(np.diag(x))
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])
File:      c:\miniconda3\lib\site-packages\numpy\lib\twodim_base.py
Type:      function
```

In [122…

```
np.diag(np.arange(0, 27, 5))
```

Out[122…
```
array([[ 0,  0,  0,  0,  0,  0],
       [ 0,  5,  0,  0,  0,  0],
       [ 0,  0, 10,  0,  0,  0],
       [ 0,  0,  0, 15,  0,  0],
       [ 0,  0,  0,  0, 20,  0],
       [ 0,  0,  0,  0,  0, 25]])
```

Indexing and Slicing

In [ ]:
```
# [m = 0: n(up to but not including) = end: p(step) = 1]
```

One-Dimensional Arrays

In [124…
```
a = np.arange(0, 11)
```

In [125…
```
a
```

Out[125…
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [126…
```
a[0]
```

Out[126…
```
0
```

In [127…
```
a[-1]
```

Out[127…
```
10
```

In [128…
```
a[5]
```

Out[128…
```
5
```

In [129…
```
a[-5]
```

Out[129…
```
6
```

In [130…
```
a[2:-2]
```

Out[130…
```
array([2, 3, 4, 5, 6, 7, 8])
```

In [131…
```
a[2:-2:2]
```

Out[131…
```
array([2, 4, 6, 8])
```

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

In [132…

```
a[-6:]
```

Out[132…

```
array([ 5,  6,  7,  8,  9, 10])
```

In [133…

```
a[::-2]
```

Out[133…

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

In [134…

```
a[:]
```

Out[134…

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

In [137…

```
a[0:]
```

Out[137…

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

In [138…

```
a[0:-1]
```

Out[138…

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

In [135…

```
a[1:8:-1]
```

Out[135…

```
array([], dtype=int32)
```

In [139…

```
a[8:1:-1]
```

Out[139…

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

Multidimensional Arrays

In [140…

```
f = lambda m, n: n + 10 * m
```

In [141…

```
A = np.fromfunction(f, (7, 7), dtype = int)
```

In [142…

```
A
```

Out[142…

```
array([[ 0,  1,  2,  3,  4,  5,  6],
       [10, 11, 12, 13, 14, 15, 16],
       [20, 21, 22, 23, 24, 25, 26],
       [30, 31, 32, 33, 34, 35, 36],
       [40, 41, 42, 43, 44, 45, 46],
       [50, 51, 52, 53, 54, 55, 56],
       [60, 61, 62, 63, 64, 65, 66]])
```

In [143…
```
A[2, 3]
```
Out[143…
```
23
```
In [144…
```
A[:, 3]
```
Out[144…
```
array([ 3, 13, 23, 33, 43, 53, 63])
```
In [145…
```
A[:2, :4]
```
Out[145…
```
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13]])
```
In [146…
```
A[2:, 4:]
```
Out[146…
```
array([[24, 25, 26],
       [34, 35, 36],
       [44, 45, 46],
       [54, 55, 56],
       [64, 65, 66]])
```
In [147…
```
A[::3, ::2]
```
Out[147…
```
array([[ 0,  2,  4,  6],
       [30, 32, 34, 36],
       [60, 62, 64, 66]])
```

Views

In [148…
```
f = lambda m ,n : n + 10 * m
a = np.fromfunction(f, (7, 7), dtype = int); a
```
Out[148…
```
array([[ 0,  1,  2,  3,  4,  5,  6],
       [10, 11, 12, 13, 14, 15, 16],
       [20, 21, 22, 23, 24, 25, 26],
       [30, 31, 32, 33, 34, 35, 36],
       [40, 41, 42, 43, 44, 45, 46],
       [50, 51, 52, 53, 54, 55, 56],
       [60, 61, 62, 63, 64, 65, 66]])
```
In [154…
```
b = a[::3, ::2]; b
```
Out[154…
```
array([[100, 100, 100, 100],
       [100, 100, 100, 100],
       [100, 100, 100, 100]])
```
In [156…
```
b[:, :] = 100; b
```
Out[156…
```
array([[100, 100, 100, 100],
       [100, 100, 100, 100],
       [100, 100, 100, 100]])
```
In [157…

```
a
```

Out[157…

```
array([[100,   1, 100,   3, 100,   5, 100],
       [ 10,  11,  12,  13,  14,  15,  16],
       [ 20,  21,  22,  23,  24,  25,  26],
       [100,  31, 100,  33, 100,  35, 100],
       [ 40,  41,  42,  43,  44,  45,  46],
       [ 50,  51,  52,  53,  54,  55,  56],
       [100,  61, 100,  63, 100,  65, 100]])
```

In [158…

```
b[:2, 1:]
```

Out[158…

```
array([[100, 100, 100],
       [100, 100, 100]])
```

In [159…

```
c = b[:2, 1:].copy(); c
```

Out[159…

```
array([[100, 100, 100],
       [100, 100, 100]])
```

In [160…

```
c[:, :] = -500
```

In [161…

```
c
```

Out[161…

```
array([[-500, -500, -500],
       [-500, -500, -500]])
```

In [162…

```
b
```

Out[162…

```
array([[100, 100, 100, 100],
       [100, 100, 100, 100],
       [100, 100, 100, 100]])
```

Fancy Indexing and Boolean-Valued Indexing

In [165…

```
A = np.linspace(0, 1, 11); A
```

Out[165…

```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

In [167…

```
A[np.array([0, 4, 6])]
```

Out[167…

```
array([0. , 0.4, 0.6])
```

In [173…

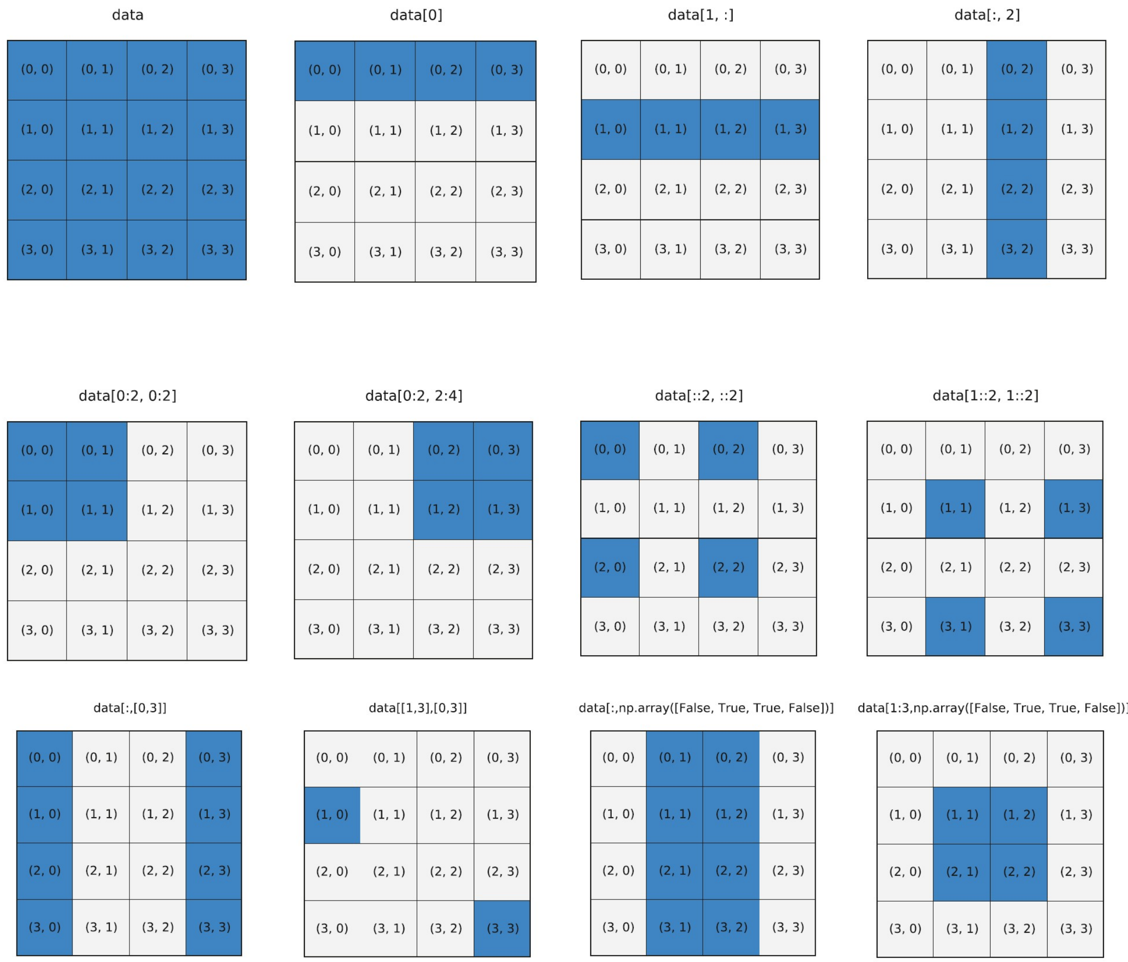
```
A[0, 4, 6]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_15992/3363190361.py in <module>
----> 1 A[0, 4, 6]

IndexError: too many indices for array: array is 1-dimensional, but 3 were indexed
```

In [168…
```
A[[0, 4, 6]]
```
Out[168…
```
array([0. , 0.4, 0.6])
```
In [169…
```
A > 0.5
```
Out[169…
```
array([False, False, False, False, False, False,  True,  True,  True,
        True,  True])
```
In [170…
```
A[A > 0.5]
```
Out[170…
```
array([0.6, 0.7, 0.8, 0.9, 1. ])
```
In [171…
```
A = np.arange(10)
indexes = [1, 3, 5]
B = A[indexes]
print(f'A: {A}')
print(f'B: {B}')
```
```
A: [0 1 2 3 4 5 6 7 8 9]
B: [1 3 5]
```
In [172…
```
B[0] = -200
print(f'A: {A}')
print(f'B: {B}')
```
```
A: [0 1 2 3 4 5 6 7 8 9]
B: [-200    3    5]
```
In [175…
```
A[indexes] = -200; A
```
Out[175…
```
array([   0, -200,    2, -200,    4, -200,    6,    7,    8,    9])
```
In [186…
```
A = np.arange(10); A
```
Out[186…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```
In [187…
```
B = A[A > 5]; B
```
Out[187…
```
array([6, 7, 8, 9])
```
In [188…
```
B[0] = -500; B
```
Out[188…
```
array([-500,    7,    8,    9])
```
In [189…
```
A
```
Out[189…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [190…
```
A[A > 5] = -500; A
```

Out[190…
```
array([   0,    1,    2,    3,    4,    5, -500, -500, -500, -500])
```



Reshaping and Resizing

In [191…
```
data = np.array([[1, 2], [3, 4]])
```

In [193…
```
np.reshape(data, (1, 4))
```

Out[193…
```
array([[1, 2, 3, 4]])
```

In [197…
```
a = data.reshape((1, 4)); a
```

Out[197…
```
array([[1, 2, 3, 4]])
```

In [198…
```
a.shape
```

Out[198…
```
(1, 4)
```

In [199…
```
b = data.reshape(4); b
```
Out[199…
```
array([1, 2, 3, 4])
```
In [200…
```
b.shape
```
Out[200…
```
(4,)
```
In [ ]:
```
# np.ravel() -> change view , np.flatten() -> give a copy
```
In [202…
```
data = np.array([[1, 2], [3, 4]]); data
```
Out[202…
```
array([[1, 2],
       [3, 4]])
```
In [203…
```
data.flatten()
```
Out[203…
```
array([1, 2, 3, 4])
```
In [204…
```
data.flatten().shape
```
Out[204…
```
(4,)
```
In [205…
```
np.shape(data.flatten())
```
Out[205…
```
(4,)
```
In [208…
```
data = np.arange(5); data
```
Out[208…
```
array([0, 1, 2, 3, 4])
```
In [209…
```
data.shape
```
Out[209…
```
(5,)
```
In [210…
```
col = data[:, np.newaxis]; col
```
Out[210…
```
array([[0],
       [1],
       [2],
       [3],
       [4]])
```
In [211…
```
col.shape
```
Out[211…
```
(5, 1)
```

In [212…

```
row = data[np.newaxis, :]; row
```

Out[212…

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

In [3]:

```
data = np.arange(5); data
```

Out[3]:

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

In [4]:

```
np.expand_dims(data, axis = 1) # -> data[:, np.newaxis]
```

Out[4]:

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

In [5]:

```
np.expand_dims(data, axis = 0) # -> data[np.newaxis, :]
```

Out[5]:

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

In [7]:

```
data = np.arange(5); data
```

Out[7]:

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

In [8]:

```
np.vstack((data, data, data, data))
```

Out[8]:

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

In [9]:

```
data
```

Out[9]:

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

In [10]:

```
np.hstack((data,data,data,data))
```

Out[10]:

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

In [11]:

```
data = data[:, np.newaxis]
```

In [12]:

```
data
```

Out[12]:

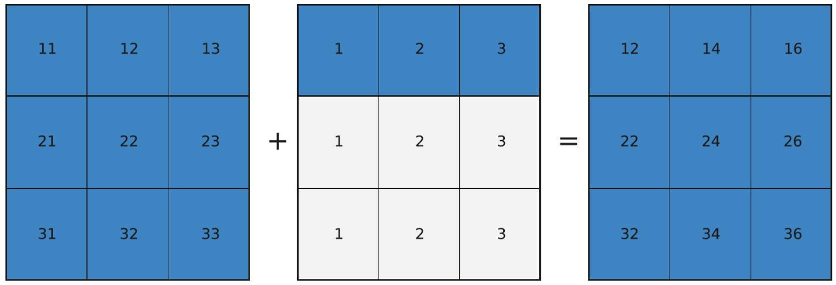
```
array([[0],
       [1],
       [2],
       [3],
       [4]])
```

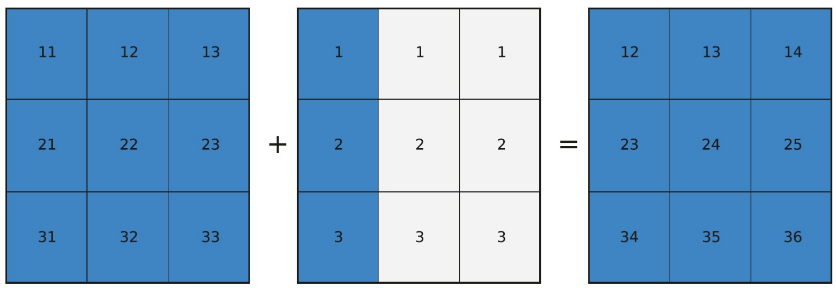
In [13]:
```
np.hstack((data, data, data, data))
```
Out[13]:
```
array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2],
       [3, 3, 3, 3],
       [4, 4, 4, 4]])
```
In [14]:
```
np.concatenate((data, data, data, data), axis = 1)
```
Out[14]:
```
array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2],
       [3, 3, 3, 3],
       [4, 4, 4, 4]])
```
In [19]:
```
data = np.arange(5); data
```
Out[19]:
```
array([0, 1, 2, 3, 4])
```
In [21]:
```
np.concatenate((data, data, data, data), axis = 0)
```
Out[21]:
```
array([0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
```
In [22]:
```
np.concatenate((data, data, data, data), axis = 1)
```
```
---------------------------------------------------------------------------
AxisError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_6048/1188247215.py in <module>
----> 1 np.concatenate((data, data, data, data), axis = 1)

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

AxisError: axis 1 is out of bounds for array of dimension 1
```
In [23]:
```
data = data[np.newaxis, :]; data
```
Out[23]:
```
array([[0, 1, 2, 3, 4]])
```
In [25]:
```
np.concatenate((data, data, data, data), axis = 0)
```
Out[25]:
```
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
```
In [26]:
```
np.concatenate((data, data, data, data), axis = 1)
```
Out[26]:
```
array([[0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4]])
```

Vectorized Expressions

Broadcasting Law in NumPy

luzumi be yek size budan araye ha nist, amma size ha sazgar bashand jam ettefagh mioftad. mehvar ha ya bayad bar ruye yek mabnaye moshtarak tul e yeksan dashte bashand va ya tule yek kodum az anha 1 bashad.





Arithmetic Operations

Operators for Elementwise Arithmetic Operation on NumPy Arrays

Operator	Operation
+, +=	Addition
-, -=	Subtraction
*, *=	Multiplication
/, /=	Division
//, //=	Integer division
**, **=	Exponentiation

In [27]:
```
x = np.array([[1, 2], [3, 4]]); x
```

Out[27]:
```
array([[1, 2],
       [3, 4]])
```

In [28]:
```
y = np.array([[5, 6], [7, 8]]); y
```

Out[28]:
```
array([[5, 6],
```

```
       [7, 8]])
```

In [29]:
```
x + y
```
Out[29]:
```
array([[ 6,  8],
       [10, 12]])
```

In [30]:
```
y - x
```
Out[30]:
```
array([[4, 4],
       [4, 4]])
```

In [31]:
```
x * y
```
Out[31]:
```
array([[ 5, 12],
       [21, 32]])
```

In [32]:
```
x * 2
```
Out[32]:
```
array([[2, 4],
       [6, 8]])
```

In [33]:
```
2 ** x
```
Out[33]:
```
array([[ 2,  4],
       [ 8, 16]], dtype=int32)
```

In [34]:
```
y / 2
```
Out[34]:
```
array([[2.5, 3. ],
       [3.5, 4. ]])
```

In [36]:
```
(y/ 2).dtype
```
Out[36]:
```
dtype('float64')
```

In [6]:
```
x = np.array([1, 2, 3, 4]).reshape((2,2)); x
```
Out[6]:
```
array([[1, 2],
       [3, 4]])
```

In [40]:
```
z = np.array([1, 2, 3, 4])
```

In [41]:
```
x / z
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_6048/2868904118.py in <module>
----> 1 x / z

ValueError: operands could not be broadcast together with shapes (2,2) (4,)
```

In [4]:
```
z = np.array([[2, 4]])
```

In [43]:
```
z.shape
```

Out[43]:
```
(1, 2)
```

In [44]:
```
x / z
```

Out[44]:
```
array([[0.5, 0.5],
       [1.5, 1. ]])
```

In [46]:
```
zz = np.concatenate((z ,z), axis = 0); zz
```

Out[46]:
```
array([[2, 4],
       [2, 4]])
```

In [48]:
```
x / zz
```

Out[48]:
```
array([[0.5, 0.5],
       [1.5, 1. ]])
```

In [8]:
```
z = np.array([[2], [4]]); z
```

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

In [50]:
```
z.shape
```

Out[50]:
```
(2, 1)
```

In [51]:
```
x / z
```

Out[51]:
```
array([[0.5 , 1.  ],
       [0.75, 1.  ]])
```

In [52]:
```
z / x
```

Out[52]:
```
array([[2.        , 1.        ],
       [1.33333333, 1.        ]])
```

In [9]:
```
zz = np.concatenate((z, z), axis = 1); zz
```

Out[9]:
```
array([[2, 2],
       [4, 4]])
```

In [10]:
```
x / zz
```

Out[10]:
```
array([[0.5 , 1.  ],
       [0.75, 1.  ]])
```

In [11]:

```
# x = x + y |  x += y (In place operator)
```

Elementwise Functions

NumPy Functions for Elementwise Elementary Mathematical Functions

NumPy Function	Description
np.cos, np.sin, np.tan	Trigonometric functions.
np.arccos, np.arcsin, np.arctan	Inverse trigonometric functions.
np.cosh, np.sinh, np.tanh	Hyperbolic trigonometric functions.
np.arccosh, np.arcsinh, np.arctanh	Inverse hyperbolic trigonometric functions.
np.sqrt	Square root.
np.exp	Exponential.
np.log, np.log2, np.log10	Logarithms of base e, 2, and 10, respectively.

In [13]:
```
x = np.linspace(-1, 1, 11); x
```

Out[13]:
```
array([-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
```

In [15]:
```
y = np.sin(np.pi * x); y
```

Out[15]:
```
array([-1.22464680e-16, -5.87785252e-01, -9.51056516e-01, -9.51056516e-01,
       -5.87785252e-01,  0.00000000e+00,  5.87785252e-01,  9.51056516e-01,
        9.51056516e-01,  5.87785252e-01,  1.22464680e-16])
```

In [16]:
```
np.round(y, decimals = 5)
```

Out[16]:
```
array([-0.     , -0.58779, -0.95106, -0.95106, -0.58779,  0.     ,
        0.58779,  0.95106,  0.95106,  0.58779,  0.     ])
```

NumPy Functions for Elementwise Mathematical Operations

NumPy Function	Description
np.add, np.subtract, np.multiply, np.divide	Addition, subtraction, multiplication, and division of two NumPy arrays.
np.power	Raises first input argument to the power of the second input argument (applied elementwise).
np.remainder	The remainder of division.
np.reciprocal	The reciprocal (inverse) of each element.
np.real, np.imag, np.conj	The real part, imaginary part, and the complex conjugate of the elements in the input arrays.
np.sign, np.abs	The sign and the absolute value.
np.floor, np.ceil, np.rint	Convert to integer values.
np.round	Rounds to a given number of decimals.

In [17]:
```
np.add(np.sin(x)**2 , np.cos(x) ** 2)
```
Out[17]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```
In [18]:
```
np.sin(x)**2 + np.cos(x) ** 2
```
Out[18]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```
In [19]:
```
def heaviside(x):
    return 1 if x>0 else 0
```
In [20]:
```
heaviside(5)
```
Out[20]:
```
1
```
In [21]:
```
heaviside(-5)
```
Out[21]:
```
0
```
In [22]:
```
x = np.linspace(-5 , 5 , 11); x
```
Out[22]:
```
array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.,  5.])
```
In [23]:
```
heaviside(x)
```
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_9040/1443328523.py in <module>
----> 1 heaviside(x)

~\AppData\Local\Temp/ipykernel_9040/2135315997.py in heaviside(x)
      1 def heaviside(x):
----> 2     return 1 if x>0 else 0

ValueError: The truth value of an array with more than one element is ambiguous. Use a.a
ny() or a.all()
```
In [24]:
```
heaviside = np.vectorize(heaviside)
```
In [25]:
```
heaviside(x)
```
Out[25]:
```
array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1])
```
In [26]:
```
heaviside(-3)
```
Out[26]:
```
array(0)
```

In [27]:
```
def heaviside(x):
    return 1.0 * (x > 0)
```

Aggregate Functions

NumPy Functions for Calculating Aggregates of NumPy Arrays

NumPy Function	Description
np.mean	The average of all values in the array.
np.std	Standard deviation.
np.var	Variance.
np.sum	Sum of all elements.
np.prod	Product of all elements.
np.cumsum	Cumulative sum of all elements.
np.cumprod	Cumulative product of all elements.
np.min, np.max	The minimum/maximum value in an array.
np.argmin, np.argmax	The index of the minimum/maximum value in an array.
np.all	Returns True if all elements in the argument array are nonzero.
np.any	Returns True if any of the elements in the argument array is nonzero.

In [28]:
```
# np.mean(data) | data.mean()
```

In [31]:
```
data = np.random.normal(size = (15, 15)); data
```

Out[31]:
```
array([[ 9.65902062e-02, -9.07721677e-01,  1.16723803e+00,
         3.67566263e-01, -5.75895256e-01, -1.56814517e-01,
        -5.41257181e-01,  2.22830952e+00,  1.02806566e+00,
         6.57531730e-01, -1.25418649e+00, -2.64031712e-01,
        -8.90088450e-01, -1.51032774e-02, -3.40704562e-01],
       [-6.20874908e-01, -6.59477208e-01, -3.71718250e-01,
        -2.31770143e-01, -8.84834179e-01, -7.56117024e-01,
         1.15035541e+00,  1.82426459e+00, -6.79074145e-01,
        -4.24984183e-01,  6.20873701e-02, -2.46049071e+00,
        -4.74779083e-01, -3.73983079e-01,  1.09242337e+00],
       [ 8.33546128e-01,  1.78947322e+00,  8.11402812e-01,
         1.69684672e-02,  3.74188521e-02, -5.42847927e-01,
         1.13197657e+00, -8.18438158e-01, -6.01928615e-01,
        -1.25159753e+00, -5.93448032e-01,  1.16860355e+00,
        -1.02980399e+00,  6.36123215e-01,  8.30023034e-01],
       [ 1.71240265e-01, -5.74340350e-01,  4.75304824e-01,
         7.13236787e-01,  1.17382373e+00, -9.18379811e-01,
        -1.12849000e-01, -1.21924139e+00, -8.63430237e-01,
         6.11082639e-01, -2.17986776e+00, -1.88198907e+00,
        -2.46002337e-01,  2.58312900e+00, -5.21903467e-01],
       [ 4.44057850e-01, -5.91581273e-01, -7.28954318e-02,
        -1.67553208e+00, -5.92632519e-01, -3.09638207e-01,
         2.32435422e-02,  8.17934054e-01,  6.80482873e-01,
         2.69029230e+00, -8.62156856e-01,  1.09390940e-01,
         4.50016068e-01, -6.32972533e-01,  2.47814456e-02],
       [ 1.53391392e+00,  2.88796068e-01, -1.83171205e-01,
         4.78465930e-01, -1.62516774e+00, -6.51653452e-01,
```

```
        -4.51785167e-01,  9.61677474e-01, -1.55062010e-03,
         5.44447815e-01, -9.21585557e-01,  8.43838703e-01,
         1.48459821e+00, -6.99801384e-01, -9.20405604e-01],
       [ 5.76121927e-01, -3.53390875e-01,  8.30625254e-01,
         2.89804058e-01,  1.96014994e+00, -2.91897348e-01,
        -1.56103120e-01, -6.20505724e-01,  7.88314121e-01,
         1.10673050e+00,  2.37003986e-02, -6.50565807e-01,
         1.04233577e+00, -1.30039671e-01, -1.05516961e+00],
       [-7.72371807e-01,  9.87239142e-01,  1.25884169e+00,
         2.27751632e-01,  1.64752238e+00,  2.41279909e+00,
        -1.52295552e+00,  1.28114015e-01, -3.15248483e-01,
         8.62723551e-01,  5.02291285e-01,  1.24406693e-01,
         1.23745557e+00, -2.26893207e-01, -1.21927956e+00],
       [ 1.27846642e+00,  9.23821924e-01,  1.33263242e+00,
         3.49152568e-01, -1.32613389e+00, -2.07179876e-01,
        -4.58400879e-01,  8.29701588e-02,  7.81653770e-01,
        -1.85244361e-01,  6.31011182e-01, -4.75063017e-01,
         1.16881259e+00,  1.81775373e-01, -2.73820130e+00],
       [-9.98469145e-01,  1.25492489e-01, -3.89747751e-02,
        -2.90608616e-01,  9.76366270e-01,  2.57680111e-01,
        -7.97439177e-01,  1.41321642e+00, -2.40754458e-01,
        -1.44511031e+00,  1.54415991e+00,  3.24465057e-01,
         4.36140649e-01,  2.55483651e-01,  1.62508093e+00],
       [ 2.53824798e+00, -7.35248735e-01,  2.32185466e+00,
         1.29568722e+00,  5.12326415e-01,  9.63211069e-01,
        -5.28567897e-01, -2.66216285e-01, -4.86743918e-01,
         8.99952872e-01,  1.65877754e+00,  1.04025399e+00,
        -7.28895323e-01, -9.73239387e-01,  1.33936789e+00],
       [-5.74743970e-02,  3.45086088e-01, -1.71756225e+00,
         1.92792109e+00,  4.49398617e-02, -7.48841304e-01,
         5.44385345e-01,  9.43639154e-01,  1.70134584e+00,
        -7.72962594e-01, -2.04242475e+00,  2.55831337e-01,
        -2.97280397e-01,  2.25083612e-01,  1.88345191e+00],
       [-4.82322998e-02, -4.89783742e-01, -1.50605196e+00,
        -3.32715844e-01,  7.53883477e-02,  8.96628165e-01,
         1.11997524e-02, -7.08127448e-01,  2.27980415e-01,
        -2.37269479e-01,  5.05060351e-01,  8.59336705e-01,
        -1.11873260e+00,  1.89788903e+00, -2.70875176e-01],
       [ 6.65079648e-01, -2.92062944e-01,  4.28167726e-02,
        -1.14517109e+00, -1.17562335e+00, -1.57985621e-01,
         9.70839554e-01,  9.71456058e-01,  1.41239805e+00,
        -8.42636955e-01, -1.47816193e+00,  1.28203347e-01,
        -4.46151366e-01, -1.09236386e+00,  1.81221889e+00],
       [-1.01105743e+00, -1.60538202e+00,  2.30666959e-01,
         1.23156846e+00,  1.59446405e+00,  3.17872228e-01,
        -2.55353261e+00, -1.08274855e-02,  1.36427260e-01,
         8.96182406e-01, -1.43112300e+00,  9.71547737e-01,
         3.54745106e-01,  7.69945816e-01,  1.22146037e-01]])
```

In [32]:

```
np.mean(data)
```

Out[32]:

```
0.09339834249582513
```

In [33]:

```
data.mean()
```

Out[33]:

```
0.09339834249582513
```

In [34]:
```
data = np.random.normal(size = (5, 10, 15))
```

In [35]:
```
data.sum(axis = 0).shape
```

Out[35]:
```
(10, 15)
```

In [36]:
```
data.sum(axis = (0, 2)).shape
```

Out[36]:
```
(10,)
```

In [37]:
```
data.sum()
```

Out[37]:
```
-16.285187474805525
```

In [40]:
```
data = np.arange(1, 10).reshape((3, 3)); data
```

Out[40]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [41]:
```
data.sum()
```

Out[41]:
```
45
```

In [42]:
```
data.sum(axis = 0)
```

Out[42]:
```
array([12, 15, 18])
```

In [44]:
```
data.sum(axis = 1)
```

Out[44]:
```
array([ 6, 15, 24])
```

In [43]:
```
np.sum(data, axis = 0)
```

Out[43]:
```
array([12, 15, 18])
```

Boolean Arrays and Conditional Expressions

NumPy Functions for Conditional and Logical Expressions

Function	Description
np.where	Chooses values from two arrays depending on the value of a condition array.
np.choose	Chooses values from a list of arrays depending on the values of a given index array.
np.select	Chooses values from a list of arrays depending on a list of conditions.
np.nonzero	Returns an array with indices of nonzero elements.
np.logical_and	Performs an elementwise AND operation.
np.logical_or, np.logical_xor	Elementwise OR/XOR operations.
np.logical_not	Elementwise NOT operation (inverting).

In [ ]:
```
# <, >, <=, >=, ==, =!
```

In [45]:
```
a = np.array([1, 2, 3, 4])
```

In [46]:
```
b = np.array([4, 3, 2, 1])
```

In [47]:
```
a < b
```

Out[47]:
```
array([ True,  True, False, False])
```

In [49]:
```
(a < b).dtype
```

Out[49]:
```
dtype('bool')
```

In [50]:
```
np.all(a < b)
```

Out[50]:
```
False
```

In [52]:
```
np.any(a < b)
```

Out[52]:
```
True
```

In [53]:
```
if np.all(a < b):
    print('All elements in a is lower than all elements in b.')
```

```
elif np.any(a < b):
    print('some elements in a is lower than some elements in b.')
else:
    print('all elements in a is bigger than all elements in b.')
```

```
some elements in a is lower than some elements in b.
```

In [54]:
```
x = np.array([-2, -1 , 0, 1, 2])
```

In [55]:
```
x > 0
```

Out[55]:
```
array([False, False, False,  True,  True])
```

In [56]:
```
1.0 * (x > 0)
```

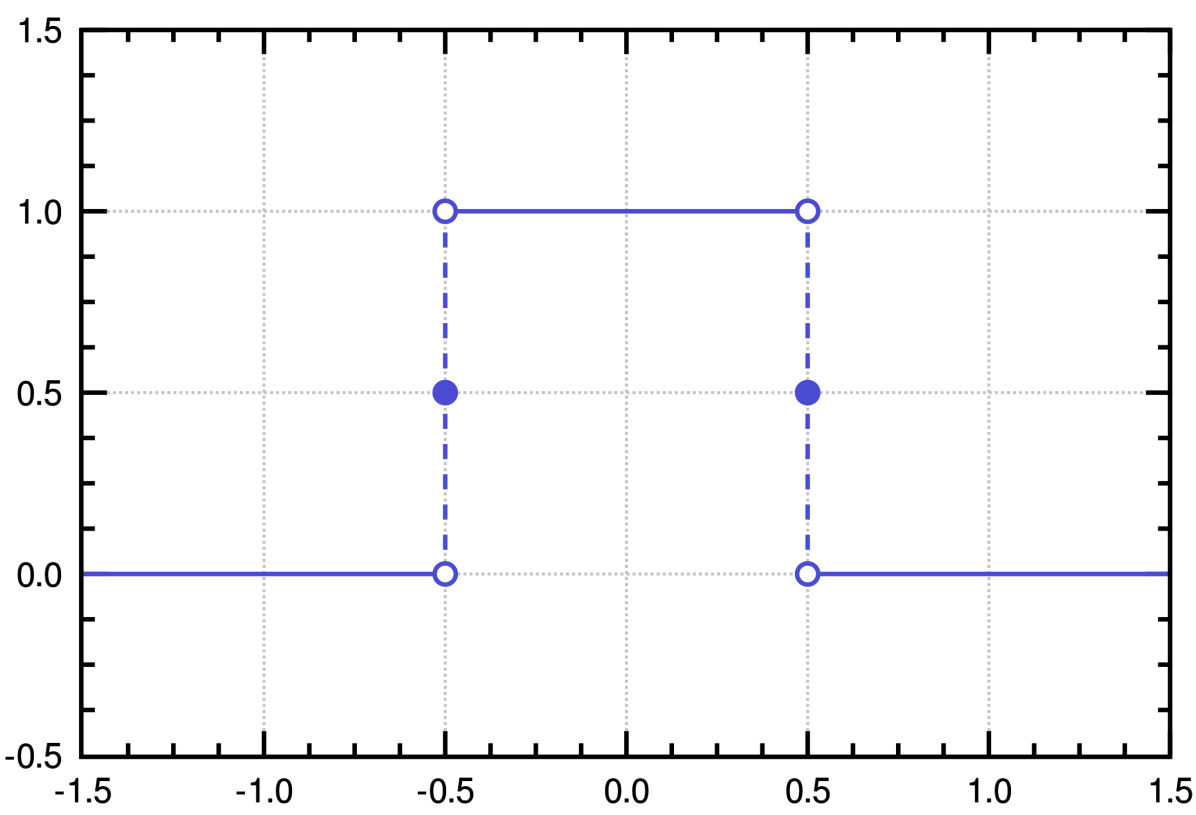
Out[56]:
```
array([0., 0., 0., 1., 1.])
```

In [57]:
```
x * (x > 0)
```

Out[57]:
```
array([0, 0, 0, 1, 2])
```

In [62]:
```
from IPython.display import Image
Image(data = 'PulseFunc.png')
```

Out[62]:



In [67]:
```
# Desing piece wise functions like pulse function
```

```python
def pulse(x, position, height, width):
    return height * (x >= position) * (x <= (position + width))
```

In [66]:

```python
x = np.linspace(-5, 5, 11); x
```

Out[66]:

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

In [68]:

```python
pulse(x, position = -3, height = 5, width = 6)
```

Out[68]:

```
array([0, 0, 5, 5, 5, 5, 5, 5, 5, 0, 0])
```

In [73]:

```python
def pulse(x, position, height, width):
    return height * np.logical_and(x >= position, x <= (position + width))
```

In [74]:

```python
pulse(x, position = -3, height = 5, width = 6)
```

Out[74]:

```
array([0, 0, 5, 5, 5, 5, 5, 5, 5, 0, 0])
```

In [78]:

```python
x = np.linspace(-4, 4, 9);x
```

Out[78]:

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

In [77]:

```python
np.where(x < 0 , x**2, x**3)
```

Out[77]:

```
array([16.,  9.,  4.,  1.,  0.,  1.,  8., 27., 64.])
```

In [79]:

```python
np.select([x < -1, x < 2, x >= 2], [x**2, x**3, x**4])
```

Out[79]:

```
array([ 16.,   9.,   4.,  -1.,   0.,   1.,  16.,  81., 256.])
```

In [80]:

```python
np.choose([0, 0 ,0, 1, 1, 1, 2, 2, 2], [x**2, x**3, x**4])
```

Out[80]:

```
array([ 16.,   9.,   4.,  -1.,   0.,   1.,  16.,  81., 256.])
```

In [81]:

```python
x
```

Out[81]:

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

In [82]:

```python
np.nonzero(abs(x) > 2)
```

Out[82]:

```
(array([0, 1, 7, 8], dtype=int64),)
```

In [83]:

```python
x[np.nonzero(abs(x) > 2)]
```

Out[83]:
```
array([-4., -3.,  3.,  4.])
```

In [84]:
```
x[abs(x) > 2]
```

Out[84]:
```
array([-4., -3.,  3.,  4.])
```

Set Operations

NumPy Functions for Operating on Sets

Function	Description
np.unique	Creates a new array with unique elements, where each value only appears once.
np.in1d	Tests for the existence of an array of elements in another array.
np.intersect1d	Returns an array with elements that are contained in two given arrays.
np.setdiff1d	Returns an array with elements that are contained in one, but not the other, of two given arrays.
np.union1d	Returns an array with elements that are contained in either, or both, of two given arrays.

In [3]:
```
a = np.unique([1, 2, 2, 3, 3, 3]); a
```

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

In [5]:
```
b = np.unique([2, 3, 3, 4, 4, 4, 5, 6, 3, 2, 4]); b
```

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

In [6]:
```
np.in1d(a, b)
```

Out[6]:
```
array([False,  True,  True])
```

In [7]:
```
a[np.in1d(a, b)]
```

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

In [8]:
```
np.in1d(b, a)
```

Out[8]:
```
array([ True,  True, False, False, False])
```

In [9]:
```
1 in a
```

Out[9]:
```
True
```

In [10]:
```
1 in b
```

Out[10]: False

In [11]:
```
np.all(np.in1d(a , b))  # subset checking
```

Out[11]: False

In [12]:
```
np.union1d(a, b)
```

Out[12]: array([1, 2, 3, 4, 5, 6])

In [13]:
```
np.union1d(b, a)
```

Out[13]: array([1, 2, 3, 4, 5, 6])

In [14]:
```
np.intersect1d(a, b)
```

Out[14]: array([2, 3])

In [15]:
```
np.intersect1d(b, a)
```

Out[15]: array([2, 3])

In [18]:
```
np.setdiff1d(a, b)
```

Out[18]: array([1])

In [19]:
```
np.setdiff1d(b, a)
```

Out[19]: array([4, 5, 6])

Operations on Arrays

NumPy Functions for Array Operations

Function	Description
np.transpose, np.ndarray.transpose, np.ndarray.T	The transpose (reverse axes) of an array.
np.fliplr/np.flipud	Reverse the elements in each row/column.
np.rot90	Rotates the elements along the first two axes by 90 degrees.
np.sort, np.ndarray.sort	Sort the elements of an array along a given specified axis (which default to the last axis of the array). The np.ndarray method sort performs the sorting in place, modifying the input array.

In [4]:
```
data = np.arange(9).reshape((3, 3)); data
```

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

In [5]:
```
np.transpose(data)
```

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

In [10]:
```
data.transpose()
```

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

In [8]:
```
data.T
```

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

In [13]:
```
data
```

Out[13]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

In [11]:
```
np.fliplr(data)
```

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

In [14]:
```
np.flipud(data)
```

Out[14]:
```
array([[6, 7, 8],
       [3, 4, 5],
       [0, 1, 2]])
```

In [15]:
```
np.rot90(data)
```

Out[15]:
```
array([[2, 5, 8],
       [1, 4, 7],
       [0, 3, 6]])
```

Matrix and Vector Operations

> > ## NumPy Functions for Matrix Operations

NumPy Function	Description
np.dot	Matrix multiplication (dot product) between two given arrays representing vectors, arrays, or tensors.
np.inner	Scalar multiplication (inner product) between two arrays representing vectors.
np.cross	The cross product between two arrays that represent vectors.
np.tensordot	Dot product along specified axes of multidimensional arrays.
np.outer	Outer product (tensor product of vectors) between two arrays representing vectors.
np.kron	Kronecker product (tensor product of matrices) between arrays representing matrices and higher-dimensional arrays.
np.einsum	Evaluates Einstein's summation convention for multidimensional arrays.

In [5]:
```
A = np.arange(1, 7).reshape((2, 3)); A
```
Out[5]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```
In [6]:
```
B = np.arange(1, 7).reshape((3, 2)); B
```
Out[6]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```
In [7]:
```
# N * M and M * P -> N * P
```
In [8]:
```
np.dot(A, B)
```
Out[8]:
```
array([[22, 28],
       [49, 64]])
```
In [9]:
```
np.dot(B , A)
```
Out[9]:
```
array([[ 9, 12, 15],
       [19, 26, 33],
       [29, 40, 51]])
```
In [10]:
```
A @ B
```
Out[10]:
```
array([[22, 28],
       [49, 64]])
```
In [12]:
```
B @ A
```
Out[12]:
```
array([[ 9, 12, 15],
       [19, 26, 33],
       [29, 40, 51]])
```
In [14]:
```
A = np.arange(9).reshape((3 , 3)); A
```
Out[14]:
```
array([[0, 1, 2],
       [3, 4, 5],
```

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

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

Out[15]:
```
array([0, 1, 2])
```

In [17]:
```
A.shape
```

Out[17]:
```
(3, 3)
```

In [18]:
```
x.shape
```

Out[18]:
```
(3,)
```

In [16]:
```
np.dot(A, x)
```

Out[16]:
```
array([ 5, 14, 23])
```

In [19]:
```
y = np.arange(3).reshape((1, 3)); y
```

Out[19]:
```
array([[0, 1, 2]])
```

In [20]:
```
y.shape
```

Out[20]:
```
(1, 3)
```

In [21]:
```
np.dot(A, y)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_14404/3453443763.py in <module>
----> 1 np.dot(A, y)

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

ValueError: shapes (3,3) and (1,3) not aligned: 3 (dim 1) != 1 (dim 0)
```

In [26]:
```
print(y)
print('\n',A)
```

```
[[0 1 2]]

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

In [22]:
```
np.dot(y, A)
```

Out[22]:
```
array([[15, 18, 21]])
```

In [27]:
```
A.dot(x)
```

Out[27]:
```
array([ 5, 14, 23])
```

In [28]:
```
# A_prime = BAB^-1
```

In [32]:
```
A = np.random.rand(3, 3); A
```

Out[32]:
```
array([[0.42444768, 0.08570787, 0.23705496],
       [0.50194924, 0.8918437 , 0.85327381],
       [0.78860255, 0.74432695, 0.87303542]])
```

In [33]:
```
B = np.random.rand(3, 3); B
```

Out[33]:
```
array([[0.46871292, 0.42345985, 0.5602959 ],
       [0.16776235, 0.06976333, 0.65776935],
       [0.62478447, 0.51727267, 0.26160816]])
```

In [35]:
```
A_prime = np.dot(B ,np.dot(A, np.linalg.inv(B))); A_prime
```

Out[35]:
```
array([[ 2.54959175, -0.55123481, -0.39885741],
       [ 1.24304873, -0.06916957,  0.08629156],
       [ 2.10241501, -0.43167853, -0.29109538]])
```

In [37]:
```
A_prime = B.dot(A.dot(np.linalg.inv(B))); A_prime
```

Out[37]:
```
array([[ 2.54959175, -0.55123481, -0.39885741],
       [ 1.24304873, -0.06916957,  0.08629156],
       [ 2.10241501, -0.43167853, -0.29109538]])
```

In [40]:
```
A = np.matrix(A); A
```

Out[40]:
```
matrix([[0.42444768, 0.08570787, 0.23705496],
        [0.50194924, 0.8918437 , 0.85327381],
        [0.78860255, 0.74432695, 0.87303542]])
```

In [41]:
```
B = np.matrix(B); B
```

Out[41]:
```
matrix([[0.46871292, 0.42345985, 0.5602959 ],
        [0.16776235, 0.06976333, 0.65776935],
        [0.62478447, 0.51727267, 0.26160816]])
```

In [43]:
```
A_prime = B * A * B.I; A_prime
```

Out[43]:
```
matrix([[ 2.54959175, -0.55123481, -0.39885741],
        [ 1.24304873, -0.06916957,  0.08629156],
        [ 2.10241501, -0.43167853, -0.29109538]])
```

In [44]:
```
A = np.random.rand(3, 3); A
```

```
array([[0.39468434, 0.4131507 , 0.11883634],
```

Out[44]:
```
       [0.1883368 , 0.00684695, 0.94127215],
       [0.01192458, 0.18653736, 0.29896111]])
```

In [45]:
```
B = np.random.rand(3, 3); B
```

Out[45]:
```
array([[0.91005693, 0.7889583 , 0.23023307],
       [0.11238977, 0.9456988 , 0.42102974],
       [0.45365817, 0.39384245, 0.38422564]])
```

In [46]:
```
type(A)
```

Out[46]:
```
numpy.ndarray
```

In [47]:
```
A = np.asmatrix(A)
```

In [48]:
```
B = np.asmatrix(B)
```

In [49]:
```
type(A)
```

Out[49]:
```
numpy.matrix
```

In [51]:
```
A_prime = B * A * B.I; A_prime
```

Out[51]:
```
matrix([[-0.91548479, -0.02343996,  2.96764898],
        [-1.59612519, -0.07995528,  3.72315548],
        [-0.56756356,  0.04404366,  1.69593247]])
```

In [53]:
```
A_prime = np.asarray(A_prime); A_prime
```

Out[53]:
```
array([[-0.91548479, -0.02343996,  2.96764898],
       [-1.59612519, -0.07995528,  3.72315548],
       [-0.56756356,  0.04404366,  1.69593247]])
```

In [54]:
```
x
```

Out[54]:
```
array([0, 1, 2])
```

In [55]:
```
np.inner(x, x) # np.inner just accepts vectors
```

Out[55]:
```
5
```

In [56]:
```
np.sum(x * x)
```

Out[56]:
```
5
```

In [57]:
```
np.dot(x, x) # np.dot could accept vbectors with dimensions like 1*N or N*1
```

Out[57]: 5

In [58]:
```
y = x[:, np.newaxis]; y
```

Out[58]:
```
array([[0],
       [1],
       [2]])
```

In [59]:
```
np.dot(y.T, y)
```

Out[59]:
```
array([[5]])
```

Outer product

> ## Given two vectors
>
> $$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$
>
> ## their outer product u ⊗ v is defined as the m × n matrix A obtained by multiplying each element of u by each element of v:
>
> $$\mathbf{u} \otimes \mathbf{v} = \mathbf{A} = \begin{bmatrix} u_1v_1 & u_1v_2 & \dots & u_1v_n \\ u_2v_1 & u_2v_2 & \dots & u_2v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_mv_1 & u_mv_2 & \dots & u_mv_n \end{bmatrix}$$
>
> ## index notation :
>
> $$(\mathbf{u} \otimes \mathbf{v})_{ij} = u_iv_j$$
>
> ## The outer product u ⊗ v is equivalent to a matrix multiplication uv^T, provided that u is represented as a m × 1 column vector and v as a n × 1 column vector (which makes v^T a row vector). For instance, if m = 4 and n = 3, then

$$\mathbf{u} \otimes \mathbf{v} = \mathbf{u}\mathbf{v}^\mathsf{T} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} = \begin{bmatrix} u_1v_1 & u_1v_2 & u_1v_3 \\ u_2v_1 & u_2v_2 & u_2v_3 \\ u_3v_1 & u_3v_2 & u_3v_3 \\ u_4v_1 & u_4v_2 & u_4v_3 \end{bmatrix}.$$

> Wikipedia

In [60]:
```
x = np.array([1, 2, 3])
```

In [61]:
```
np.outer(x, x)
```

Out[61]:
```
array([[1, 2, 3],
       [2, 4, 6],
       [3, 6, 9]])
```

Kronecker product

> ## If A is an m × n matrix and B is a p × q matrix, then the Kronecker product A ⊗ B is the pm × qn block matrix:
>
> $$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix},$$
>
> ## more explicitly:
>
> $$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{11}b_{1q} & \cdots & \cdots & a_{1n}b_{11} & a_{1n}b_{12} & \cdots & a_{1n}b_{1q} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{11}b_{2q} & \cdots & \cdots & a_{1n}b_{21} & a_{1n}b_{22} & \cdots & a_{1n}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{p1} & a_{11}b_{p2} & \cdots & a_{11}b_{pq} & \cdots & \cdots & a_{1n}b_{p1} & a_{1n}b_{p2} & \cdots & a_{1n}b_{pq} \\ \vdots & \vdots & & \vdots & \ddots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & & \ddots & \vdots & \vdots & & \vdots \\ a_{m1}b_{11} & a_{m1}b_{12} & \cdots & a_{m1}b_{1q} & \cdots & \cdots & a_{mn}b_{11} & a_{mn}b_{12} & \cdots & a_{mn}b_{1q} \\ a_{m1}b_{21} & a_{m1}b_{22} & \cdots & a_{m1}b_{2q} & \cdots & \cdots & a_{mn}b_{21} & a_{mn}b_{22} & \cdots & a_{mn}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & a_{m1}b_{p2} & \cdots & a_{m1}b_{pq} & \cdots & \cdots & a_{mn}b_{p1} & a_{mn}b_{p2} & \cdots & a_{mn}b_{pq} \end{bmatrix}.$$
>
> Wikipedia

In [62]:
```
np.kron(x, x)
```

Out[62]:
```
array([1, 2, 3, 2, 4, 6, 3, 6, 9])
```

In [66]:
```
np.kron(x[:, np.newaxis], x[np.newaxis, :]) # to act like outer product our inputs shpu
```

Out[66]:
```
array([[1, 2, 3],
       [2, 4, 6],
       [3, 6, 9]])
```

In [65]:
```
np.outer(x, x)
```

Out[65]:
```
array([[1, 2, 3],
       [2, 4, 6],
       [3, 6, 9]])
```

In [67]:
```
np.kron(np.ones((2, 2)), np.identity(2))
```

Out[67]:
```
array([[1., 0., 1., 0.],
       [0., 1., 0., 1.],
       [1., 0., 1., 0.],
       [0., 1., 0., 1.]])
```

In [70]:
```
np.kron(np.identity(2), np.ones((2, 2)))
```

Out[70]:
```
array([[1., 1., 0., 0.],
       [1., 1., 0., 0.],
       [0., 0., 1., 1.],
       [0., 0., 1., 1.]])
```

> ## Einstein notation
>
> > ### scalar product between two vectors x and y: $x_n y_n$
> >
> > ### matrix multiplication of A and B : $A_{mk} B_{kn}$

In [71]:
```
x = np.arange(1,5)
```

In [72]:
```
y = np.array([5, 6, 7, 8])
```

In [73]:
```
np.einsum('n, n', x, y)
```

Out[73]:
```
70
```

In [74]:
```
np.inner(x, y)
```

Out[74]:
```
70
```

In [75]:
```
A = np.arange(9).reshape((3, 3))
```

In [76]:
```
B = A.T
```

In [77]:
```
np.einsum('mk, kn', A, B)
```

Out[77]:
```
array([[  5,  14,  23],
       [ 14,  50,  86],
       [ 23,  86, 149]])
```

In [78]:
```
np.alltrue(np.einsum('mk, kn', A, B) == np.dot(A, B))
```

Out[78]:
```
True
```

In [1]:
```
!jupyter nbconvert --to html Numpy.ipynb
```

In [ ]: