ch02-vectors-matrices-ndarrays

November 14, 2023

0.0.1 Ch 02: Vectors, matrices and multidimensional arrays

NumPy manual (latest version, ReadTheDocs)

[1]:
```
import numpy as np
import seaborn as sn
import pandas as pd
```

0.0.2 NumPy arrays

- **NOT THE SAME AS PYTHON LISTS**.
- All array elements have same data type; arrays are fixed size.
- (Need to edit the array? create a new one.)
- **Attributes**:
  - *shape*: tuple; contains # of elements for each axis of the array
  - *size*: total # of elements
  - *ndim*: number of dimensions (axes)
  - *nbytes*: number of bytes used for storage
  - *dtype*: datatype

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

[2]:
```
numpy.ndarray
```

[3]:
```
data.ndim, data.shape, data.size, data.dtype, data.nbytes
```

[3]:
```
(2, (3, 2), 6, dtype('int64'), 48)
```

[4]:
```
data
```

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

0.0.3 Integers (8b,16,32b,64)

(python 3.11: dtype=np.int now dtype=int)

[5]:
```python
data = np.array([1, 2, 3], dtype=int); print(data.dtype, data)
data = np.array([1, 2, 3], dtype=np.int32); print(data.dtype, data)
data = np.array([1, 2, 3], dtype=np.int16); print(data.dtype, data)
data = np.array([1, 2, 3], dtype=np.int8); print(data.dtype, data)
```

```
int64 [1 2 3]
int32 [1 2 3]
int16 [1 2 3]
int8 [1 2 3]
```

0.0.4 Unsigned Integers (8b,16,32b,64b)

[6]:
```python
data = np.array([1, 2, 3], dtype=np.uint); print(data.dtype, data)
data = np.array([1, 2, 3], dtype=np.uint32); print(data.dtype, data)
data = np.array([1, 2, 3], dtype=np.uint16); print(data.dtype, data)
data = np.array([1, 2, 3], dtype=np.uint8); print(data.dtype, data)
```

```
uint64 [1 2 3]
uint32 [1 2 3]
uint16 [1 2 3]
uint8 [1 2 3]
```

0.0.5 Booleans

[7]:
```python
data = np.array([True,False,1,0], dtype=bool); print(data.dtype, data)
```

```
bool [ True False  True False]
```

0.0.6 Floating Point (16b,32b,64b,128b)

- NumPy 1.20: numpy.float deprecated; use 'float' by itself.

[8]:
```python
data = np.array([1., 2., 3.], dtype=float); print(data.dtype, data)
data = np.array([1., 2., 3.], dtype=np.float128); print(data.dtype, data)
data = np.array([1., 2., 3.], dtype=np.float32); print(data.dtype, data)
data = np.array([1., 2., 3.], dtype=np.float16); print(data.dtype, data)
```

```
float64 [1. 2. 3.]
float128 [1. 2. 3.]
float32 [1. 2. 3.]
float16 [1. 2. 3.]
```

0.0.7 Complex Data (64b,128b,256b)

- NumPy 1.20: np.complex deprecated. use 'float' by itself.

[9]:
```python
data = np.array([1., 2., 3.], dtype=complex);       print(data.dtype, data)
data = np.array([1., 2., 3.], dtype=np.complex64);  print(data.dtype, data)
data = np.array([1., 2., 3.], dtype=np.complex256); print(data.dtype, data)
```

```
complex128 [1.+0.j 2.+0.j 3.+0.j]
complex64 [1.+0.j 2.+0.j 3.+0.j]
complex256 [1.+0.j 2.+0.j 3.+0.j]
```

0.0.8 Real and imaginary parts

- All numpy arrays (**not just complex vals**) have real & imaginary attributes.

[10]:
```
data = np.array([1, 2, 3], dtype=complex)
print(data,"\n",data.real,"\n",data.imag)
```

```
[1.+0.j 2.+0.j 3.+0.j]
 [1. 2. 3.]
 [0. 0. 0.]
```

0.0.9 Typecasting

- Once created, dtype cannot be changed. Create a copy by **typecasting** (*astype*).

[11]:
```
data.astype(int) # previously: astype(np.int)
```

```
/tmp/ipykernel_10580/295343611.py:1: ComplexWarning: Casting complex values to
real discards the imaginary part
  data.astype(int) # previously: astype(np.int)
```

[11]:
```
array([1, 2, 3])
```

0.0.10 Promoting

- Data types can get "promoted" to support math ops:

[12]:
```
d1 = np.array([1, 2, 3], dtype=float)
d2 = np.array([1, 2, 3], dtype=complex)
(d1+d2).dtype
```

[12]:
```
dtype('complex128')
```

- Some cases may require creation of arrays set to appropriate data types. The default datatype is 'float'.

[13]:
```
# NumPy sqrt returns different datatypes depending on argument:
print(np.sqrt(np.array([-1, 0, 1]               )))
print(np.sqrt(np.array([-1, 0, 1], dtype=complex)))
```

```
[nan  0.  1.]
[0.+1.j 0.+0.j 1.+0.j]
```

```
/tmp/ipykernel_10580/2548364883.py:2: RuntimeWarning: invalid value encountered
in sqrt
  print(np.sqrt(np.array([-1, 0, 1]               )))
```

0.0.11 Array Data Order in Memory

- Multidimensional arrays are stored as contiguous data in memory. There is a freedom of choice in how to arrange the array elements in this memory segment.
- **Row-major** and **column-major** ordering are special cases of strategies for mapping an element's index using `ndarray.strides`.
- Operations that require changing `strides` return "views" that refer to the same data as the original array. For efficiency, NumPy strives to create views rather than copies when applying operations on arrays.
- Two options:
  - **Row-major** (row-wise storage; C std, Numpy default. Use `order='C'`)
  - **Column-major** (column-wise storage; Fortran std. Use `order='F'`)

0.0.12 Creating Arrays

Function name	Type of array
`np.array`	Creates an array for which the elements are given by an array-like object, which, for example, can be a (nested) Python list, a tuple, an iterable sequence, or another `ndarray` instance.
`np.zeros`	Creates an array – with the specified dimensions and data type – that is filled with zeros.
`np.ones`	Creates an array – with the specified dimensions and data type – that is filled with ones.
`np.diag`	Creates a diagonal array with specified values along the diagonal, and zeros elsewhere.
`np.arange`	Creates an array with evenly spaced values between specified start, end, and increment values.
`np.linspace`	Creates an array with evenly spaced values between specified start and end values, using a specified number of elements.
`np.logspace`	Creates an array with values that are logarithmically spaced between the given start and end values.
`np.meshgrid`	Generate coordinate matrices (and higher-dimensional coordinate arrays) from one-dimensional coordinate vectors.
`np.fromfunction`	Create an array and fill it with values specified by a given function, which is evaluated for each combination of indices for the given array size.
`np.fromfile`	Create an array with the data from a binary (or text) file. NumPy also provides a corresponding function `np.tofile` with which NumPy arrays can be stored to disk, and later read back using `np.fromfile`.
`np.genfromtxt`, `np.loadtxt`	Creates an array from data read from a text file. For example, a comma-separated value (CSV) file. The function `np.genfromtxt` also supports data files with missing values.
`np.random.rand`	Generates an array with random numbers that are uniformly distributed between 0 and 1. Other types of distributions are also available in the `np.random` module.

0.0.13 Arrays created from lists and other array-like objects

[14]:
```
data = np.array([1, 2, 3, 4]) # 1D array
data.ndim, data.shape
```

[14]: (1, (4,))

[15]:
```
data = np.array([[1, 2], [3, 4]]) # 2D array
data.ndim, data.shape
```

[15]: (2, (2, 2))

0.0.14 Arrays filled with constants:

- zeros(), ones(), full(), fill(), empty()

[16]:
```
np.zeros((2, 3))
```

[16]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

[17]:
```
data = np.ones(4); data
```

[17]: array([1., 1., 1., 1.])

[18]:
```
5.4*data
```

[18]: array([5.4, 5.4, 5.4, 5.4])

[19]:
```
np.full(10, 5.4)
```

[19]: array([5.4, 5.4, 5.4, 5.4, 5.4, 5.4, 5.4, 5.4, 5.4, 5.4])

[20]:
```
x1 = np.empty(5); x1
```

[20]:
```
array([3.7632544e-316, 0.0000000e+000, 3.6977106e-316, 3.7673960e-316,
       2.3715151e-322])
```

[21]:
```
x1.fill(3.0); x1
```

[21]: array([3., 3., 3., 3., 3.])

0.0.15 Arrays filled with increments

- arange(start,stop,increment)
- linspace(start,stop,#points)

[22]:
```
np.arange(0.0, 10, 1)
```

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

[23]:
```
print(np.linspace(0, 10, 20))
```

```
[ 0.          0.52631579  1.05263158  1.57894737  2.10526316  2.63157895
  3.15789474  3.68421053  4.21052632  4.73684211  5.26315789  5.78947368
  6.31578947  6.84210526  7.36842105  7.89473684  8.42105263  8.94736842
  9.47368421 10.        ]
```

0.0.16 Arrays filled with logarithmic sequences

- starting value, ending value, base (optional)

```
[24]: # 4 data points between 10**0=1 to 10**2=100
      np.logspace(0, 2, 4)
```

```
[24]: array([  1.        ,   4.64158883,  21.5443469 , 100.        ])
```

0.0.17 Mesh-grid arrays

- Given two 1D coordinate arrays, generate 2D coordinate array.
- Often used when plotting function over two variables (ex: contour plots).

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

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

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

```
[26]: Y
```

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

```
[27]: (X+Y)**2
```

```
[27]: array([[9, 4, 1],
             [1, 0, 1],
             [1, 4, 9]])
```

- np.mgrid & np.ogrid generate coordinate arrays with slightly different syntaxes.

```
[28]: np.mgrid[0:3,0:5]
```

```
[28]: array([[[0, 0, 0, 0, 0],
              [1, 1, 1, 1, 1],
              [2, 2, 2, 2, 2]],

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

[29]:
```
np.ogrid[0:3,0:5]
```

[29]:
```
[array([[0],
        [1],
        [2]]),
 array([[0, 1, 2, 3, 4]])]
```

0.0.18 Creating arrays with properties of other arrays

- Typical use case: a function that takes arrays of unspecified type & size as arguments & requires working arrays of the same type & size.
- like(), ones_like(), zeros_like(), full_like(), empty_like().

[30]:
```
np.ones_like([1,2,3,4])
```

[30]:
```
array([1, 1, 1, 1])
```

[31]:
```
np.zeros_like([1,2,3,4])
```

[31]:
```
array([0, 0, 0, 0])
```

[32]:
```
np.full_like([1,2,3,4],5)
```

[32]:
```
array([5, 5, 5, 5])
```

[33]:
```
np.empty_like([1,2,3,4])
```

[33]:
```
array([   18093,        0, 75570320, 74591632])
```

0.0.19 Creating matrix arrays

- **np.identity()**: creates square matrix with ones on diagonal, zero elsewhere.
- **np.eye()**: ones on diagonal, optionally offset
- **diag()**: arbitrary 1D array on the diagonal of a matrix

[34]:
```
np.identity(5)
```

[34]:
```
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.]])
```

[35]:
```
np.eye(4, k=1)
```

[35]:
```
array([[0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.],
```

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

[36]:
```
np.eye(4, k=-1)
```

[36]:
```
array([[0., 0., 0., 0.],
       [1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.]])
```

[37]:
```
np.diag(np.arange(0, 20, 5))
```

[37]:
```
array([[ 0,  0,  0,  0],
       [ 0,  5,  0,  0],
       [ 0,  0, 10,  0],
       [ 0,  0,  0, 15]])
```

0.1 Index and slicing

- Elements and subarrays of NumPy arrays are accessed using the standard square bracket notation that is also used with Python lists.

0.1.1 One-dimensional arrays

- Positive integers index elements from the beginning of the array (index starts at 0). Negative integers index elements from the end of the array.

Expression	Description
`a[m]`	Select element at index m, where m is an integer (start counting form 0).
`a[-m]`	Select the mth element from the end of the list, where m is an integer. The last element in the list is addressed as -1, the second-to-last element as -2, and so on.
`a[m:n]`	Select elements with index starting at m and ending at $n-1$ (m and n are integers).
`a[:]` or `a[0:-1]`	Select all elements in the given axis.
`a[:n]`	Select elements starting with index 0 and going up to index $n-1$ (integer).
`a[m:]` or `a[m:-1]`	Select elements starting with index m (integer) and going up to the last element in the array.
`a[m:n:p]`	Select elements with index m through n (exclusive), with increment p.
`a[::-1]`	Select all the elements, in reverse order.

[38]:
```
a = np.arange(0, 11); a
```

[38]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

[39]:
```
a[0], a[-1], a[4] # first, last, 5th elements
```

[39]:
```
(0, 10, 4)
```

[40]:
```
a[1:-1] # range (2nd..2nd to last)
```

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

[41]:
```
a[1:-1:2] # range (1st..last, by 2)
```

[41]: array([1, 3, 5, 7, 9])

[42]:
```
a[:5], a[-5:] # first five elements, last five elements
```

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

0.1.2 Reversed order

[43]:
```
a[::-1]
```

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

[44]:
```
a[::-3]
```

[44]: array([10, 7, 4, 1])

0.1.3 Multidimensional arrays

[45]:
```
f = lambda m,n: n+10*m
A = np.fromfunction(f, (6, 6), dtype=int); A
```

[45]:
```
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
```

[46]:
```
A[:,0], A[0,:] # 1st col, 1st row
```

[46]: (array([0, 10, 20, 30, 40, 50]), array([0, 1, 2, 3, 4, 5]))

[47]:
```
A[:3,:3], A[3:,:3] # upper left 3x3, lower left 3x3
```

[47]:
```
(array([[ 0,  1,  2],
        [10, 11, 12],
        [20, 21, 22]]),
 array([[30, 31, 32],
        [40, 41, 42],
        [50, 51, 52]]))
```

[48]:
```
A[::2, ::2] # every 2nd element
```

[48]:
```
array([[ 0,  2,  4],
       [20, 22, 24],
       [40, 42, 44]])
```

[49]:
```
A[1::2, 1::3]  # every (2nd,3rd) element starting from 1,1
```

[49]:
```
array([[11, 14],
       [31, 34],
       [51, 54]])
```

0.1.4 Views

- Subarray extractions using slice ops are alternative *views* of same underlying data. (They refer to same data, but using different "strides".)
- np.copy()
- np.array(,copy=True)

[50]:
```
B = A[1:5, 1:5]; B
```

[50]:
```
array([[11, 12, 13, 14],
       [21, 22, 23, 24],
       [31, 32, 33, 34],
       [41, 42, 43, 44]])
```

[51]:
```
# modifying B (created from A) also modifies A.
B[:,:] = 0; A
```

[51]:
```
array([[ 0,  1,  2,  3,  4,  5],
       [10,  0,  0,  0,  0, 15],
       [20,  0,  0,  0,  0, 25],
       [30,  0,  0,  0,  0, 35],
       [40,  0,  0,  0,  0, 45],
       [50, 51, 52, 53, 54, 55]])
```

[52]:
```
# explicitly copy B to C (B not affected.)
C = B.copy(); C
```

[52]:
```
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

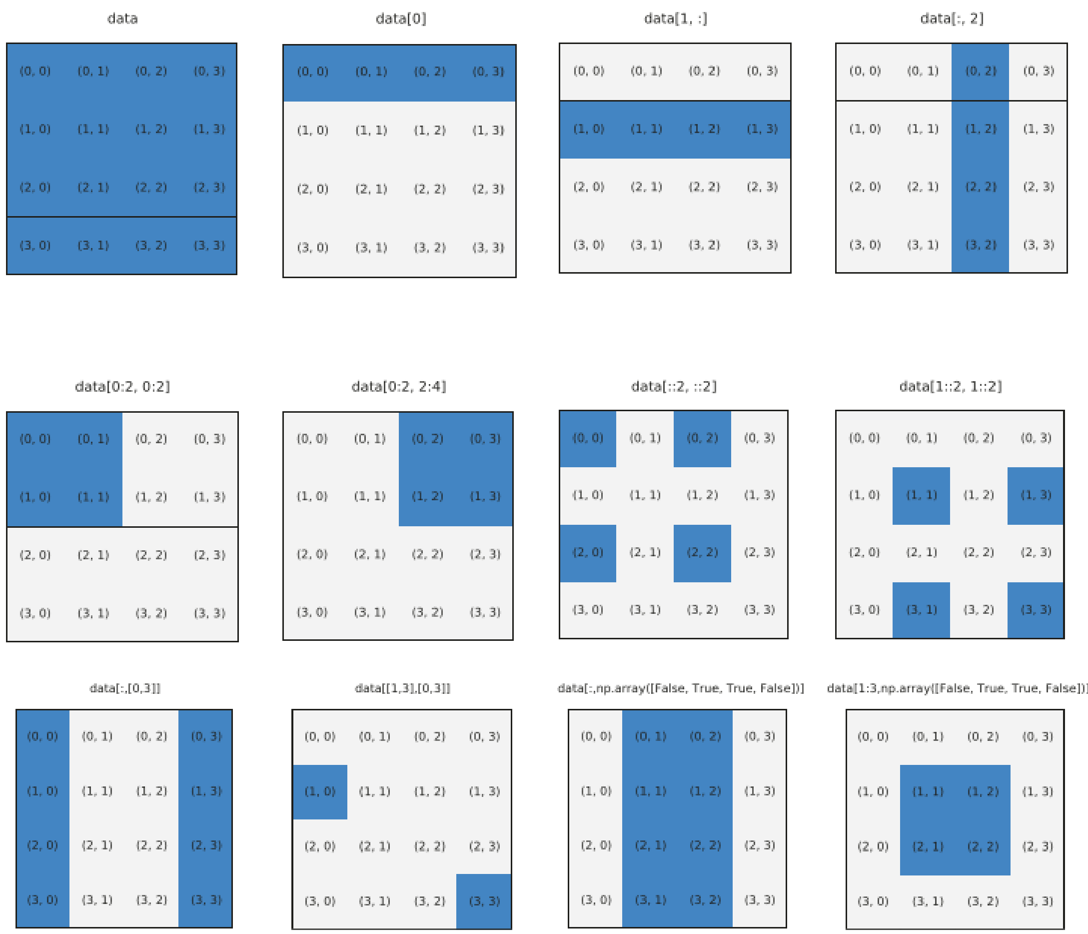
[53]:
```
C[:,:] = 1; C,B # C is a *copy* of the view B.
```

[53]:
```
(array([[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]),
```

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

0.1.5 Fancy indexing

- Arrays can be indexed using another array, a list, or sequence of integers.



[54]:
```python
A = np.linspace(0, 1, 11); A
```

[54]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

[55]:
```python
A[np.array([0, 2, 4])]
```

[55]:
```
array([0. , 0.2, 0.4])
```

[56]:
```python
A[[0, 2, 4]]
```

[56]:
```
array([0. , 0.2, 0.4])
```

0.1.6 Boolean-based indexing: great for filtering!

[57]:
```
A>0.8, A[A>0.8]
```

[57]:
```
(array([False, False, False, False, False, False, False, False, False,
         True,  True]),
 array([0.9, 1. ]))
```

- Arrays from fancy|boolean indexing are *new, independent structures* - not just views of existing data.

[58]:
```
A = np.arange(10,20); A
```

[58]:
```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

[59]:
```
indices = [2, 4, 6]; B = A[indices]; B
```

[59]:
```
array([12, 14, 16])
```

[60]:
```
B[0] = -1; B,A # this does not affect A
```

[60]:
```
(array([-1, 14, 16]), array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19]))
```

[61]:
```
A[indices] = -1; A
```

[61]:
```
array([10, 11, -1, 13, -1, 15, -1, 17, 18, 19])
```

[62]:
```
A = np.arange(10); A
```

[62]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[63]:
```
B = A[A > 5]; B
```

[63]:
```
array([6, 7, 8, 9])
```

[64]:
```
B[0] = -1; B,A # this does not affect A
```

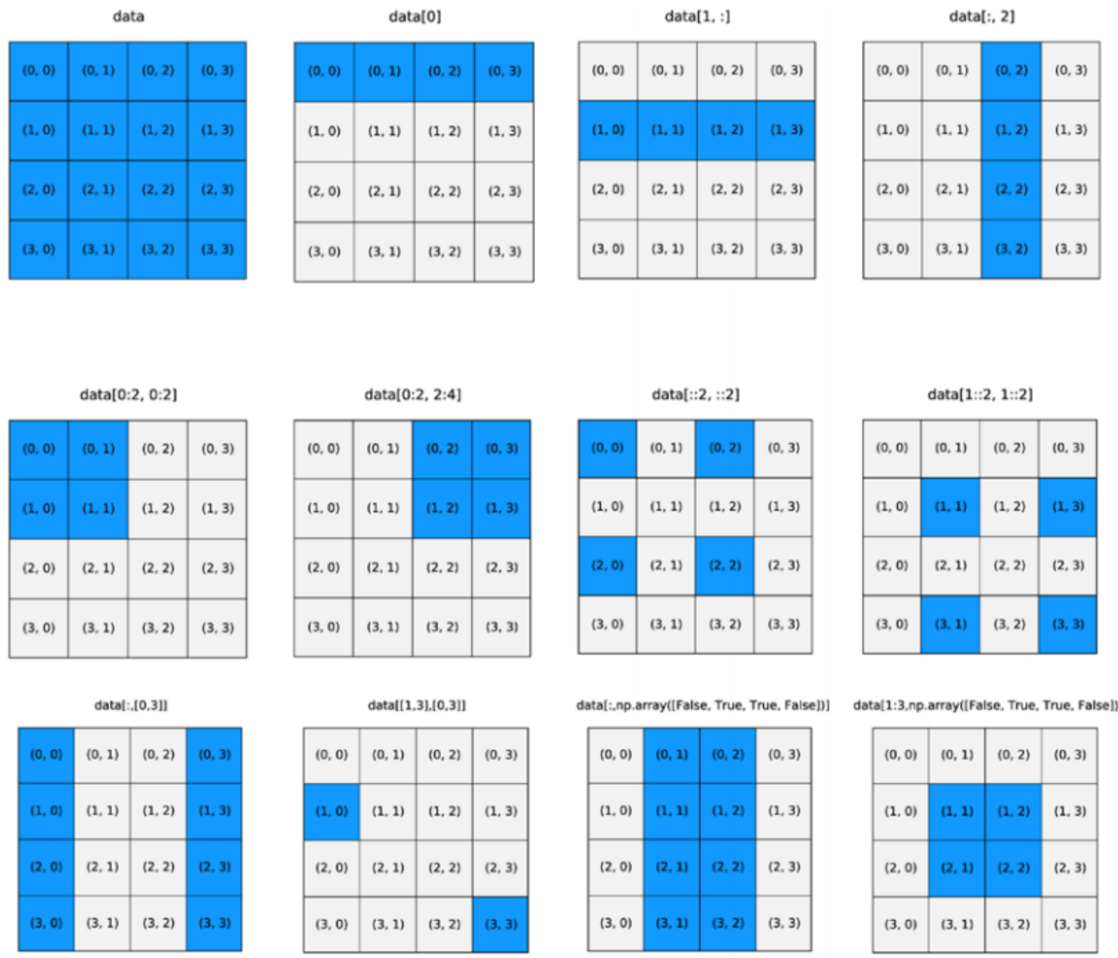
[64]:
```
(array([-1,  7,  8,  9]), array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]))
```

[65]:
```
A[A > 5] = -1; A
```

[65]:
```
array([ 0,  1,  2,  3,  4,  5, -1, -1, -1, -1])
```

0.1.7 Reshaping and resizing ops

Function / method	Description
np.reshape, np.ndarray.reshape	Reshape an N-dimensional array. The total number of elements must remain the same.
np.ndarray.flatten	Create a copy of an N-dimensional array and reinterpret it as a one-dimensional array (that is, all dimensions are collapsed into one).
np.ravel, np.ndarray.ravel	Create a view (if possible, otherwise a copy) of an N-dimensional array in which it is interpreted as a one-dimensional array.
np.squeeze	Remove axes with length 1.
np.expand_dims, np.newaxis	Adds a new axis (dimension) of length 1 to an array, where np.newaxis is used with array indexing.
np.transpose, np.ndarray.transpose, np.ndarray.T	Transpose the array. The transpose operation corresponds to reversing (or more generally, permuting) the axes of the array.
np.hstack	Stack a list of arrays horizontally (along axis 1): For example, given a list of column vectors, append the columns to form a matrix.
np.vstack	Stack a list of arrays vertically (along axis 0): For example, given a list of row vectors, append the rows to form a matrix.
np.dstack	Stack arrays depth-wise (along axis 2).
np.concatenate	Create a new array by appending arrays after each other, along a given axis.
np.resize	Resize an array. Creates a new copy of the original array, with the requested size. If necessary, the orignal array will repeated to fill up the new array.
np.append	Append an element to an array. Creates a new copy of the array.
np.insert	Insert a new element at a given position. Creates a new copy of the array.
np.delete	Delete an element at a given position. Creates a new copy of the array.



* Reshaping doesn't modify underlying data, only changes *stride* attribute

[66]:
```
data = np.array([[1, 2], [3, 4]])
np.reshape(data, (1, 4))
```

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

[67]:
```
data.reshape(4)
```

[67]:
```
array([1, 2, 3, 4])
```

0.1.8 ravel(), flatten()

- np.ravel() = special case of reshape. It collapses all array dimensions & returns a flattened 1D array with length = total number of original array elements.
- flatten() does the same thing, but returns a copy instead of a view.

[68]:
```
data, data.flatten(), data.flatten().shape
```

[68]:
```
(array([[1, 2],
        [3, 4]]),
```

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

[69]:
```python
data, data.ravel(), data.ravel().shape
```

[69]:
```
(array([[1, 2],
        [3, 4]]),
 array([1, 2, 3, 4]),
 (4,))
```

0.1.9 newaxis()

- np.newaxis() = add axis to existing array.

[70]:
```python
data = np.arange(0, 5); data
```

[70]:
```
array([0, 1, 2, 3, 4])
```

[71]:
```python
col = data[:, np.newaxis]; col
```

[71]:
```
array([[0],
       [1],
       [2],
       [3],
       [4]])
```

[72]:
```python
row = data[np.newaxis, :]; row
```

[72]:
```
array([[0, 1, 2, 3, 4]])
```

0.1.10 hstack(), vstack(), concatenate()

- np.hstack(): horizontal stacking
- np.vstack(): vertical stacking rows into a matrix
- np.concatenate(): similar to stack, but accepts an *axis* keyword

[73]:
```python
data = np.arange(5); data
```

[73]:
```
array([0, 1, 2, 3, 4])
```

[74]:
```python
# stack vertically along axis 0
np.vstack((data, data, data))
```

[74]:
```
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
```

[75]:
```python
# stack horizontally along axis 0np.hstack((data, data, data))
```

[76]:
```python
# to make hstack treat input arrays as columns:
data = data[:, np.newaxis]; data
```

[76]:
```
array([[0],
       [1],
       [2],
       [3],
       [4]])
```

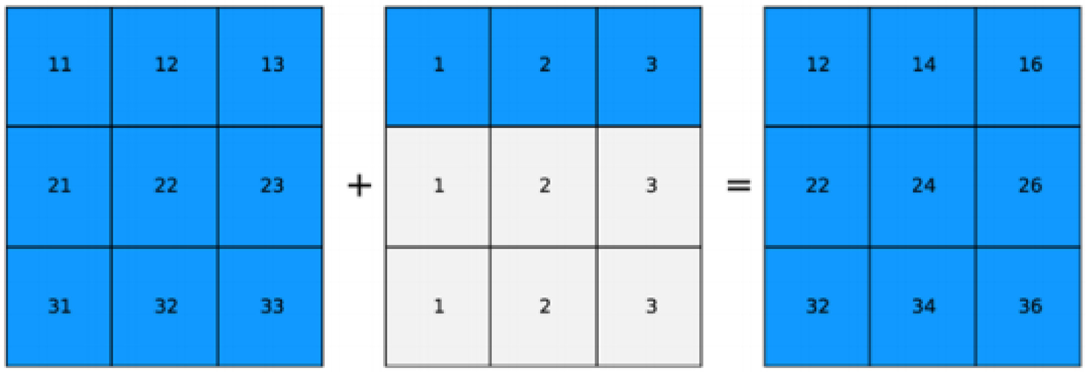
[77]:
```python
np.hstack((data, data, data))
```

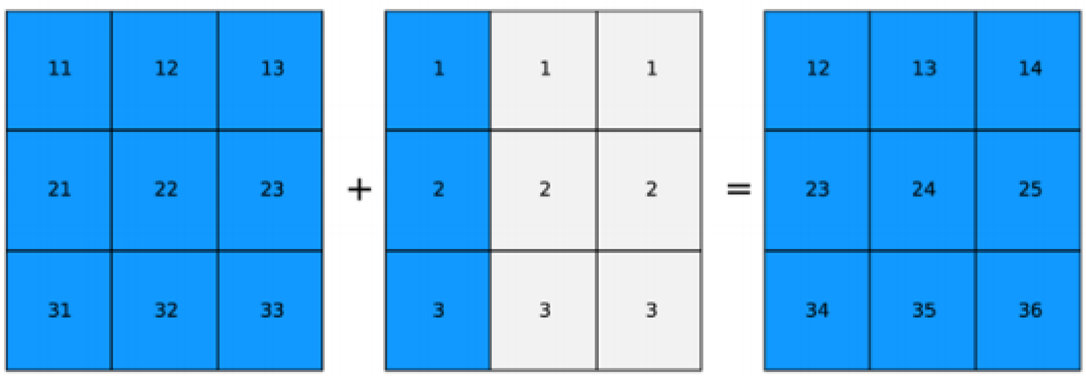
[77]:
```
array([[0, 0, 0],
       [1, 1, 1],
       [2, 2, 2],
       [3, 3, 3],
       [4, 4, 4]])
```

- Number of elements in NumPy arrays can't be changed once created. **append**, **insert**, **delete** all use a fresh copy of an array.
- **Not a best practice** due to the overhead of creating & copying the arrays. Start with preallocated arrays whenever possible to avoid resizing.

0.1.11 Vectorized expressions & Broadcasting

- Designed to avoid need for "*for*" loops. **Broadcasting** = a scalar being distributed and an operation being applied to each element in an array.





0.1.12 Arithmetic operations

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

[78]:
```
x = np.array([[1, 2], [3, 4]])
y = np.array([[5, 6], [7, 8]])
```

[79]:
```
x+y, x-y
```

[79]:
```
(array([[ 6,  8],
        [10, 12]]),
 array([[-4, -4],
        [-4, -4]]))
```

[80]:
```
x*y, y/x
```

[80]:
```
(array([[ 5, 12],
        [21, 32]]),
 array([[5.        , 3.        ],
        [2.33333333, 2.        ]]))
```

[81]:
```
x*2, 2**x
```

[81]:
```
(array([[2, 4],
        [6, 8]]),
 array([[ 2,  4],
        [ 8, 16]]))
```

[82]:
```
y/2, (y/2).dtype
```

[82]:
```
(array([[2.5, 3. ],
        [3.5, 4. ]]),
 dtype('float64'))
```

- If a math operation is performed on incompatible (size or shape) arrays, a **ValueError** is raised.

[83]:
```
x = np.array([1, 2, 3, 4]).reshape(2,2); x
```

[83]:
```
array([[1, 2],
       [3, 4]])
```

[84]:
```
z = np.array([1, 2, 3, 4]); z
```

[84]:
```
array([1, 2, 3, 4])
```

[85]:
```
try:
    x / z # incompatible size/shape
except ValueError:
    print("Nope. Can't do that.")
```

```
Nope. Can't do that.
```

- Broadcasting to a correct shape:

[86]:
```
z = np.array([[2, 4]]); z.shape
```

[86]: (1, 2)

[87]:
```
x/z
```

[87]:
```
array([[0.5, 0.5],
       [1.5, 1. ]])
```

[88]:
```
zz = np.concatenate([z, z], axis=0); zz
```

[88]:
```
array([[2, 4],
       [2, 4]])
```

[89]:
```
x/zz
```

[89]:
```
array([[0.5, 0.5],
       [1.5, 1. ]])
```

[90]:
```
z = np.array([[2], [4]]); z.shape
```

[90]: (2, 1)

[91]:
```
x/z
```

[91]:
```
array([[0.5 , 1.  ],
       [0.75, 1.  ]])
```

[92]:
```
zz = np.concatenate([z, z], axis=1); zz
```

[92]:
```
array([[2, 2],
       [4, 4]])
```

[93]:
```
x/zz
```

[93]:
```
array([[0.5 , 1.  ],
       [0.75, 1.  ]])
```

[94]:
```
x = np.array([[1, 3], [2, 4]])
x = x+y; x
```

[94]:
```
array([[ 6,  9],
       [ 9, 12]])
```

[95]:
```
x = np.array([[1, 3], [2, 4]])
x += y; x
```

[95]:
```
array([[ 6,  9],
       [ 9, 12]])
```

0.1.13 Trigonometry, square root, exponential, logarithmic 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.

[96]:
```
x = np.linspace(-1, 1, 8); print(x)
```

```
[-1.         -0.71428571 -0.42857143 -0.14285714  0.14285714  0.42857143
  0.71428571  1.        ]
```

[97]:
```
y = np.sin(np.pi*x); print(y) # sine function
```

```
[-1.22464680e-16 -7.81831482e-01 -9.74927912e-01 -4.33883739e-01
  4.33883739e-01  9.74927912e-01  7.81831482e-01  1.22464680e-16]
```

[98]:
```
print(np.round(y, decimals=4)) # round FP numbers to 4 decimals
```

```
[-0.     -0.7818 -0.9749 -0.4339  0.4339  0.9749  0.7818  0.    ]
```

[99]:
```
np.add(np.sin(x)**2, np.cos(x)**2) # sin^2+cos^2
```

[99]:
```
array([1., 1., 1., 1., 1., 1., 1., 1.])
```

0.1.14 Element-wise Math Functions

NumPy function	Description
`np.add, np.subtract, np.multiply, np.divide`	Addition, subtraction, multiplication and division of two NumPy arrays.
`np.power`	Raise 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`	Round to a given number of decimals.

0.1.15 vectorize()

- Sometimes we need to define new functions that use NumPy arrays element-by-element. **vectorize()** may help; it transforms a (usually scalar) function.

[100]:
```
def heaviside(x):
    return 1 if x > 0 else 0

heaviside(-1), heaviside(1.5)
```

[100]:
```
(0, 1)
```

[101]:
```
# won't work for Numpy arrays:
try:
    heaviside(np.linspace(-5, 5, 11))
except ValueError:
    print("Nope. Can't do that.")
```

```
Nope. Can't do that.
```

[102]:
```
# works, but relatively slow.
# better to use boolean-valued arrays (to be discussed later)
# use as quick-n-dirty check
heaviside = np.vectorize(heaviside)
heaviside(x)
```

[102]:
```
array([0, 0, 0, 0, 1, 1, 1, 1])
```

0.1.16 Aggregate functions

- Accepts array inputs, returns scalar outputs.
- Uses entire array by default - can specify an axis using `axis`.

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`	Return True if all elements in the argument array are nonzero.
`np.any`	Return True if any of the elements in the argument array is nonzero.

[103]:
```python
data = np.random.normal(size=(8,8)); data.round(2)
```

[103]:
```
array([[ 1.26, -0.09, -0.45,  0.04, -0.28, -0.01, -0.04, -0.44],
       [-1.32,  1.42, -1.72, -0.67, -1.16,  0.24,  0.76,  0.25],
       [-1.36,  1.27,  0.64,  1.67, -1.31,  0.29, -1.13,  0.36],
       [-0.61,  0.68, -0.04,  0.78,  0.3 ,  1.72,  0.51, -0.66],
       [-0.11, -1.64,  0.94, -0.23, -0.02,  0.28, -1.56, -0.02],
       [ 1.07,  0.7 ,  0.83, -0.41, -1.21, -1.72, -0.42, -1.02],
       [-1.19, -2.75,  0.51,  0.1 ,  0.22,  0.57,  0.2 , -0.66],
       [ 0.21, -0.03,  0.23,  1.14, -0.19, -1.02, -2.07, -1.75]])
```

[104]:
```python
np.mean(data), data.mean(), np.std(data), data.std()
```

[104]:
```
(-0.15802839037046623,
 -0.15802839037046623,
 0.9595463483350469,
 0.9595463483350469)
```

[105]:
```python
data = np.random.normal(size=(5, 10, 15))
```

[106]:
```python
# axis keyword controls which array axis gets aggregated
print(data.sum(axis=0    ).shape)
print(data.sum(axis=(0,2)).shape)
print(data.sum())
```

```
(10, 15)
(10,)
-28.48621544399279
```

0.1.17 Array aggregation:

1) over all elements
2) over first axis

3) over 2nd axis of a 3x3 array

[107]:
```python
data = np.arange(1,10).reshape(3,3); data
```

[107]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

[108]:
```python
print(data.sum(),data.sum(axis=0), data.sum(axis=1))
```

```
45 [12 15 18] [ 6 15 24]
```

0.1.18 Boolean arrays and vectorized conditional expressions

- Enables you to avoid using if statements. Winning!

[109]:
```python
a = np.array([1, 2, 3, 4])
b = np.array([4, 3, 2, 1]); a<b
```

[109]:
```
array([ True,  True, False, False])
```

0.1.19 Aggregate booleans

[110]:
```python
# aggregate booleans
np.all(a<b), np.any(a<b)
```

[110]:
```
(False, True)
```

[111]:
```python
if np.all(a < b):
    print("All a's < b's")
elif np.any(a < b):
    print("Some a's < b's")
else:
```

```
    print("All b's < a's")
```

```
Some a's < b's
```

0.1.20 Vectorized booleans

[112]:
```
x = np.array([-2, -1, 0, 1, 2]); x>0
```

[112]:
```
array([False, False, False,  True,  True])
```

[113]:
```
1*(x>0)
```

[113]:
```
array([0, 0, 0, 1, 1])
```

[114]:
```
x*(x>0)
```

[114]:
```
array([0, 0, 0, 1, 2])
```

0.1.21 Conditional / Logical ops

- Example use case: defining piecewise functions.

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

[115]:
```
x = np.linspace(-5, 5, 11); print(x)
```

```
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
```

[116]:
```
# expression is a multiplication of two Boolean arrays,
# so multiplication acts as an elementwise AND operator.
def pulse(x, position, height, width):
    return height * (x >= position) * (x <= (position+width))
```

[117]:
```
pulse(x, position=-2, height=1, width=5)
```

[117]:
```
array([0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0])
```

[118]:
```
pulse(x, position=1, height=1, width=5)
```

[118]: array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1])

[119]:
```
# another implementation using logical_and:
def pulse(x, position, height, width):
    return height * np.logical_and(x >= position, x <= (position + width))
```

[120]:
```
x = np.linspace(-4, 4, 9); x
```

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

0.1.22 where(), select(), choose(), nonzero()

[121]:
```
# 1st arg = boolean; 2nd,3rd args = true,false results
print(np.where(x<0, x*10, x/10))
```

```
[-40.  -30.  -20.  -10.    0.    0.1   0.2   0.3   0.4]
```

[122]:
```
# select value from list of conditions.
print(np.select(
    [x < -1, x < 2, x >= 2],
    ['bad',  'meh', 'good']))
```

```
['bad' 'bad' 'bad' 'meh' 'meh' 'meh' 'good' 'good' 'good']
```

[123]:
```
# choose value from list of arrays.
print(np.choose([0, 0, 0, 1, 1, 1, 2, 2, 2],
                [x**2, x**3, x**4]))
```

```
[ 16.   9.   4.  -1.   0.   1.  16.  81. 256.]
```

[124]:
```
# returns tuple of indices
# same result as direct indexing (abs(x)>2, but uses fancy ndxng.)
print(  np.nonzero(abs(x)>2))
print(x[np.nonzero(abs(x)>2)])
print(           x[abs(x)>2])
```

```
(array([0, 1, 7, 8]),)
[-4. -3.  3.  4.]
[-4. -3.  3.  4.]
```

0.1.23 Set operations

- Manages **unordered collections** of unique objects.

Function	Description
np.unique	Create a new array with unique elements, where each
np.in1d	Test for the existence of an array of elements in anothe
np.intersect1d	Return an array with elements that are contained in tw
np.setdiff1d	Return an array with elements that are contained in one
np.union1d	Return an array with elements that are contained in ei

[125]:
```
a = np.unique([1,2,3,3])
b = np.unique([2,3,4,4,5,6,5])
```

[126]:
```
print(np.in1d(a,b)) # test for existence of a in b (1D)
```

```
[False  True  True]
```

[127]:
```
1 in a, 1 in b # testing for single element presence
```

[127]:
```
(True, False)
```

[128]:
```
print(np.all(np.in1d(a,b))) # a = subset of b?
```

```
False
```

[129]:
```
print(np.union1d(    a,b)) # presence in either or both arrays
print(np.intersect1d(a,b)) # both arrays
```

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

[130]:
```
print(np.setdiff1d(a, b)) # presence in a, but not in b
print(np.setdiff1d(b, a)) # presence in b, but not in a
```

```
[1]
[4 5 6]
```

0.1.24 Array operations

Operations that act upon arrays **as a single entity**, and return transformed arrays of the same size.

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	Rotate the elements along the first two axes by 90 degrees.
np.sort, np.ndarray.sort	Sort the element 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.

[131]:
```
data = np.arange(9).reshape(3, 3); print(data)
```

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

[132]:
```
print(np.transpose(data))
print(data.T) # transpose also exists as special method "T"
```

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

[133]:
```
print(np.fliplr(data)) # flip left-to-right
print(np.flipud(data)) # flip up-down
```

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

[134]:
```
np.flipud(data) # flip up-to-down
```

[134]:
```
array([[6, 7, 8],
       [3, 4, 5],
       [0, 1, 2]])
```

0.1.25 Matrix and vector 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.

[135]:
```
A = np.arange(1,7).reshape(2,3); print(A)
```

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

[136]:
```
B = np.arange(1,7).reshape(3,2); print(B)
```

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

[137]:
```
print(np.dot(A,B),"\n\n",np.dot(B,A))
```

```
[[22 28]
 [49 64]]

 [[ 9 12 15]
 [19 26 33]
 [29 40 51]]
```

[138]:
```
A = np.arange(9).reshape(3, 3); print(A); print()
x = np.arange(3);               print(x)
```

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

[0 1 2]
```

[139]:
```
# dot also works for matrix-vector multiplication
print(np.dot(A, x)); print()
print(A.dot(x))
```

```
[ 5 14 23]

[ 5 14 23]
```

0.1.26 Matrix math: alternative data structure

- Matrix multiplication expressions can quickly get VERY cumbersome. Below is an example of a **similarity transform**. $A' = BAB^{-1}$:

[140]:
```
A = np.random.rand(3,3); print(A.round(3))
B = np.random.rand(3,3); print(B.round(3))
```

```
[[0.492 0.411 0.398]
 [0.841 0.7   0.542]
 [0.912 0.724 0.014]]
[[0.281 0.551 0.456]
 [0.78  0.497 0.505]
 [0.96  0.285 0.808]]
```

[141]:
```
Ap = np.dot(B,
            np.dot(A,
                   np.linalg.inv(B))); print(Ap.round(3))
```

```
[[-0.278  2.486 -0.878]
 [-0.198  2.834 -0.931]
 [-0.496  3.698 -1.35 ]]
```

[142]:
```
Ap = B.dot(A.dot(np.linalg.inv(B))); print(Ap.round(3))
```

```
[[-0.278  2.486 -0.878]
 [-0.198  2.834 -0.931]
 [-0.496  3.698 -1.35 ]]
```

- NumPy **matrix** data structure = an easier-to-read alternative.

[143]:
```
A = np.matrix(A); print(A.round(3))
B = np.matrix(B); print(B.round(3))
```

```
[[0.492 0.411 0.398]
 [0.841 0.7   0.542]
 [0.912 0.724 0.014]]
[[0.281 0.551 0.456]
 [0.78  0.497 0.505]
 [0.96  0.285 0.808]]
```

[144]:
```
Ap = B*A*B.I; print(Ap.round(3)) # I = inverse matrix
```

```
[[-0.278  2.486 -0.878]
 [-0.198  2.834 -0.931]
 [-0.496  3.698 -1.35 ]]
```

- Unfortunately **matrix** has some disadvantages & is discouraged. Expressions like A * B are context dependent, which causes readability issues.
- Consider **casting arrays to matrices** before computation, then casting the result back to ndarray instead.

[145]:
```
A = np.asmatrix(A); print(A.round(3))
B = np.asmatrix(B); print(B.round(3))
```

```
[[0.492 0.411 0.398]
 [0.841 0.7   0.542]
 [0.912 0.724 0.014]]
[[0.281 0.551 0.456]
 [0.78  0.497 0.505]
 [0.96  0.285 0.808]]
```

[146]:
```
Ap = B*A*B.I; Ap = np.asarray(Ap); print(Ap.round(3))
```

```
[[-0.278  2.486 -0.878]
 [-0.198  2.834 -0.931]
 [-0.496  3.698 -1.35 ]]
```

0.1.27 inner(), dot(), outer()

- np.inner expects two inputs with the same dimension.
- np.dot can take input vectors of shape *1xN* & *Nx1* respectively.
- np.outer maps two vectors to a matrix.

[147]:
```
print(np.inner(x,x)) # inner product between 2 arrays
print(np.dot(  x,x))
```

```
5
5
```

[148]:
```python
y = x[:, np.newaxis]; print(y)
```

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

[149]:
```python
print(np.dot(y.T, y))
```

```
[[5]]
```

[150]:
```python
x = np.array([1,2,3]); print(x)
print(np.outer(x,x))
print(np.kron( x,x))
```

```
[1 2 3]
[[1 2 3]
 [2 4 6]
 [3 6 9]]
[1 2 3 2 4 6 3 6 9]
```

0.1.28 kron()

- np.kron: often used to compute tensor products of arbitrary dimensions (both inputs must have same #axes).
- To obtain a result similar to `np.outer(x,x)`, input array x should be extended to shape (N,1) & (1,N) for `kron`'s 1st & 2nd arguments.

[151]:
```python
print(np.kron(x[:,np.newaxis], x[np.newaxis,:]))
```

```
[[1 2 3]
 [2 4 6]
 [3 6 9]]
```

[152]:
```python
# computing tensor product of two 2x2 matrices
print(np.kron(np.ones((2,2)),
              np.identity(2)))
```

```
[[1. 0. 1. 0.]
 [0. 1. 0. 1.]
 [1. 0. 1. 0.]
 [0. 1. 0. 1.]]
```

[153]:
```python
np.kron(np.identity(2), np.ones((2,2)))
```

[153]:
```
array([[1., 1., 0., 0.],
       [1., 1., 0., 0.],
       [0., 0., 1., 1.],
       [0., 0., 1., 1.]])
```

0.1.29 einsum()

- Expressing common array ops using **Einstein's summation convention** (np.einsum). (a summation is assumed over each index that occurs multiple times in an expression.)
- First argument is **an index expression** (a string with comma-separated indices, followed by arbitrary number of arrays.
- For example: $x_n y_n$ can represented using "n,n".

[154]:
```
x = np.array([1, 2, 3, 4])
y = np.array([5, 6, 7, 8])
```

[155]:
```
print(np.einsum("n,n",x,y))
print(np.inner(       x,y))
```

```
70
70
```

- Matrix multiplication $A_{mk}B_{kn}$ using "mk,kn":

[156]:
```
A = np.arange(9).reshape(3, 3); print(A)
```

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

[157]:
```
B = A.T; print(B)
```

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

[158]:
```
print(np.einsum("mk,kn",A,B))
```

```
[[  5  14  23]
 [ 14  50  86]
 [ 23  86 149]]
```

[159]:
```
# verifying...
print(np.alltrue(np.einsum("mk,kn",A,B) == np.dot(A,B)))
```

```
True
```