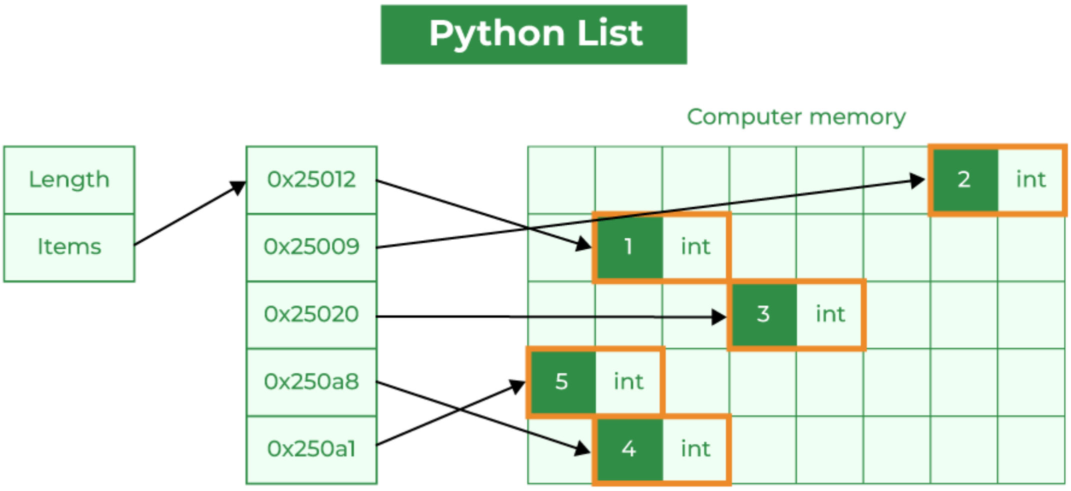
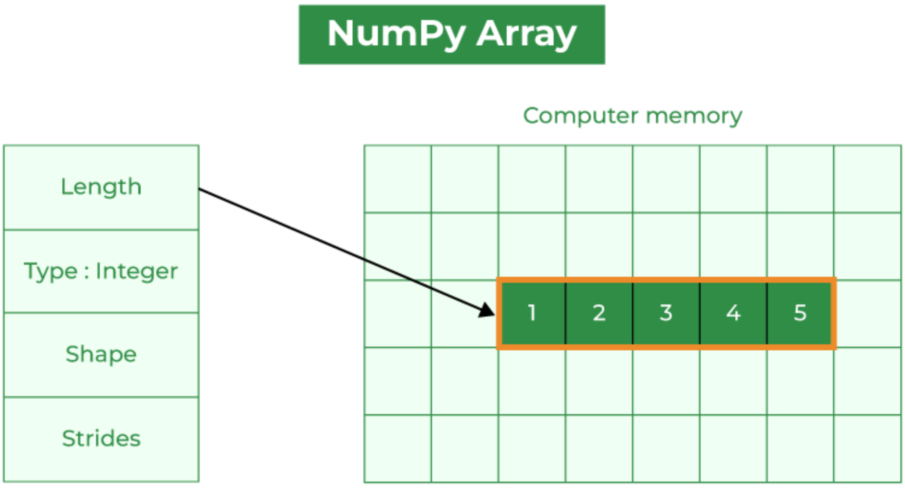
Summary

Ndarrays and the difference between ndarrays and lists

A ndarray is an array of n dimensions that has data of the same type and size. A ndarray is stored in memory like arrays common in many programming languages are stored in memory, in contiguous memory addresses. Lists, on the other hand, are arrays of pointers that point to random memory addresses of different types.





Advantages of using numpy ndarrays over regular lists

- SIMD vector processing unit makes operations on arrays faster.
- More cache utilization, since the data in arrays is contiguous the cache won't need to load data from the main memory as often.
- No type checking, arrays are always made up of the same data type.

Usages of Numpy

- Machine learning
- Mathematics
- Plotting
- Backend (Pandas, Digital Photography)

Ndarrays attributes:

Shape: the array's shape is a tuple containing the dimensions of the array. An array of 2 rows and 3 column will have a shape of (2,3).

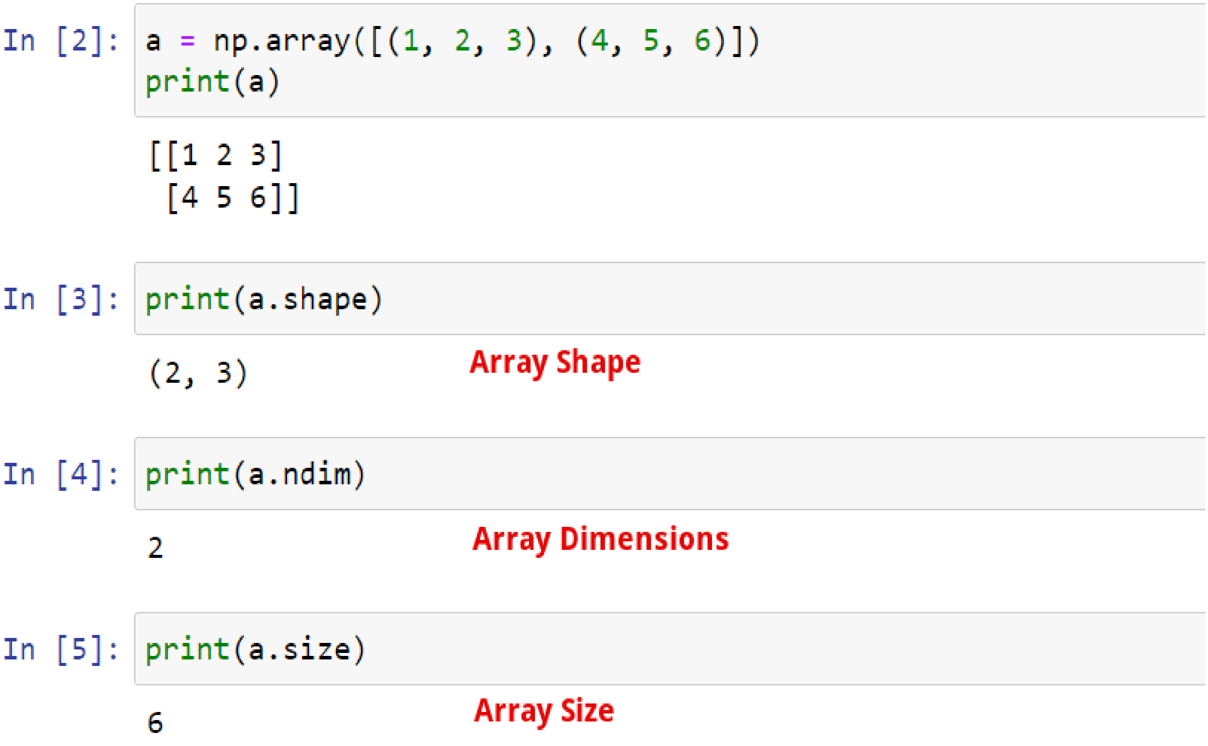
Dimensions: how many dimensions are in the array

dtype: the data type of the array.

Size: the size of the array

Itemsize: the size of the datatype of the array.

nbytes: how many bytes the array is made of , which is size* itemsize.



```
In [2]: a = np.array([(1, 2, 3), (4, 5, 6)])
        print(a)

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

In [3]: print(a.shape)

        (2, 3)

In [4]: print(a.ndim)

        2

In [5]: print(a.size)

        6
```

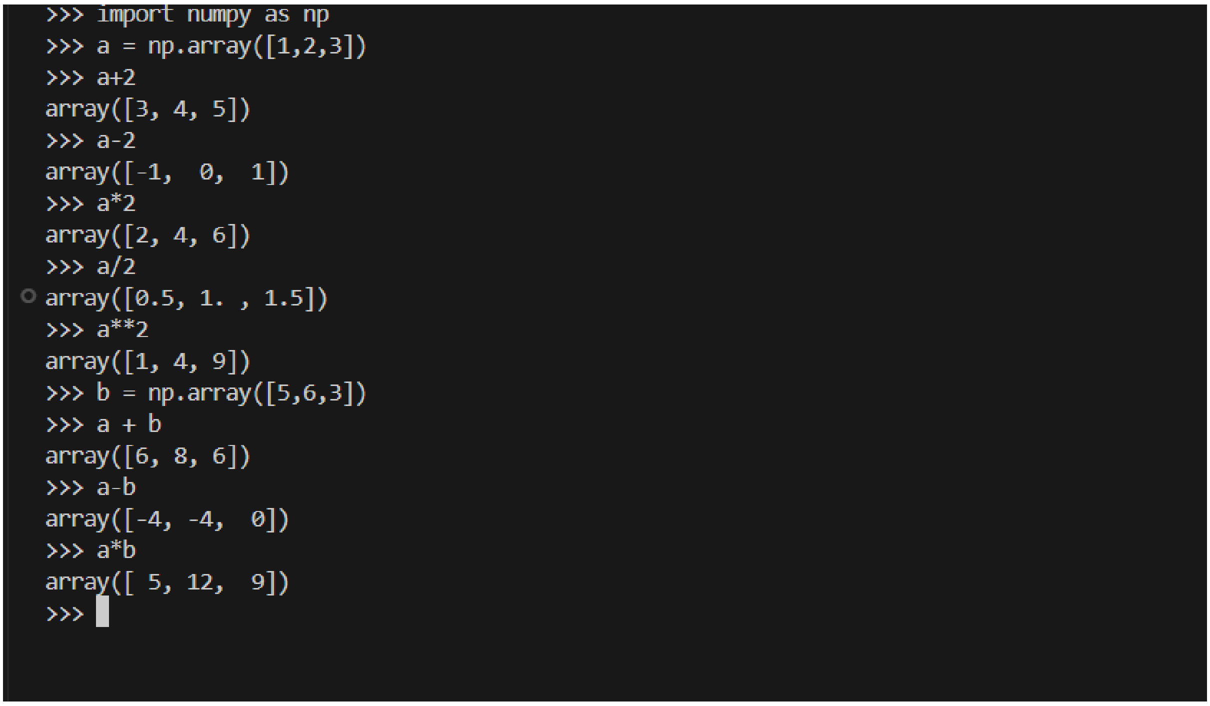
Array Shape

Array Dimensions

Array Size

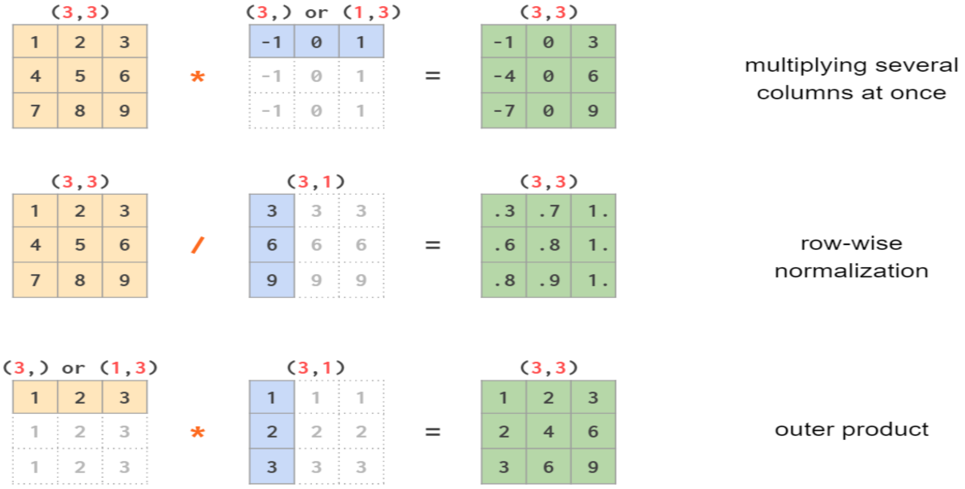
Numpy Capabalities

Arithmetic on arrays can be done much easier. You can add elements of arrays as long as they have the same shape or the one shape could be *broadcasted* to the other.

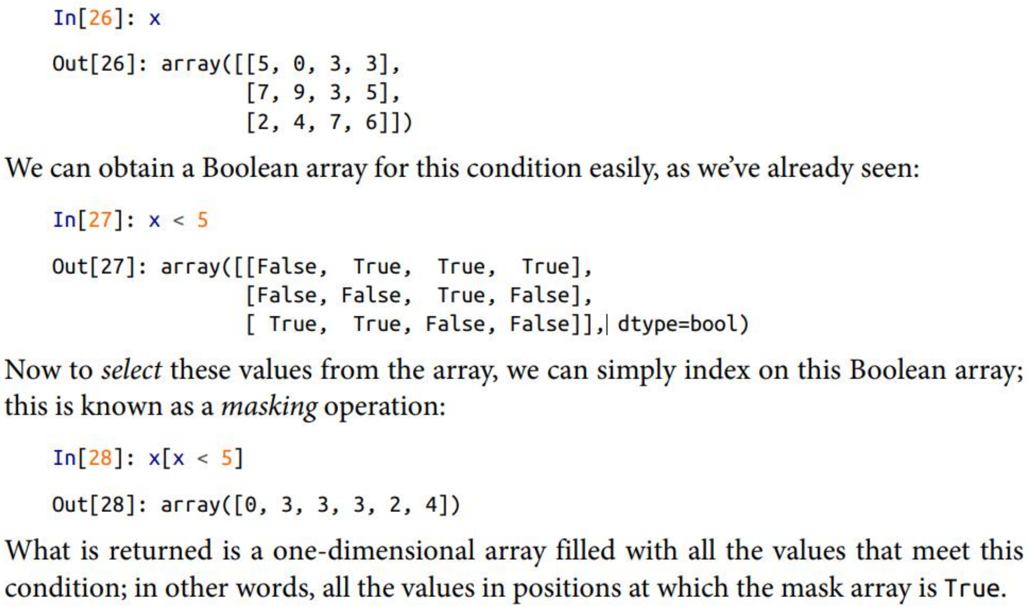


```
>>> import numpy as np
>>> a = np.array([1,2,3])
>>> a+2
array([3, 4, 5])
>>> a-2
array([-1,  0,  1])
>>> a*2
array([2, 4, 6])
>>> a/2
array([0.5, 1. , 1.5])
>>> a**2
array([1, 4, 9])
>>> b = np.array([5,6,3])
>>> a + b
array([6, 8, 6])
>>> a-b
array([-4, -4,  0])
>>> a*b
array([ 5, 12,  9])
>>>
```

Broadcasting is an operation of matching the dimensions of differently shaped arrays in order to be able to perform further operations on those arrays.



Boolean indexing is indexing through a Boolean array. a[a>2] returns all values in array a where a[i] is greater than 2.



```
In[26]: x

Out[26]: array([[5, 0, 3, 3],
                [7, 9, 3, 5],
                [2, 4, 7, 6]])
```

We can obtain a Boolean array for this condition easily, as we've already seen:

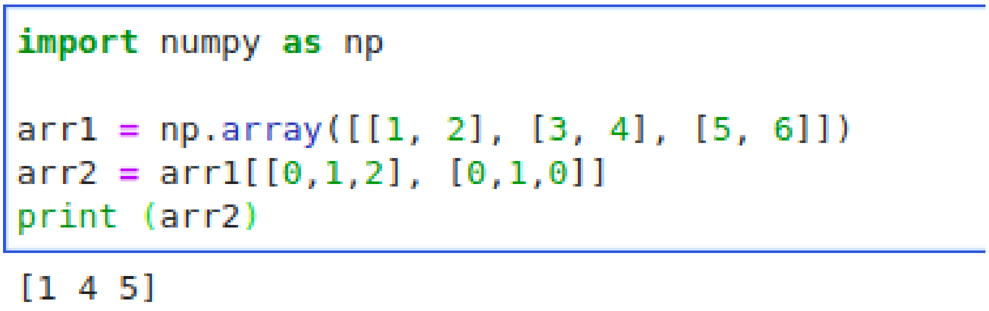
```
In[27]: x < 5

Out[27]: array([[False,  True,  True,  True],
                [False, False,  True, False],
                [ True,  True, False, False]], dtype=bool)
```

Now to *select* these values from the array, we can simply index on this Boolean array; this is known as a *masking* operation:

```
In[28]: x[x < 5]

Out[28]: array([0, 3, 3, 3, 2, 4])
```

What is returned is a one-dimensional array filled with all the values that meet this condition; in other words, all the values in positions at which the mask array is `True`.

Advanced indexing is indexing in an array through the use of lists. For example if a is an array of shape (3,3) then a[[0,1],[1,2]] will return a[0][1] and a[1][2] respectively.



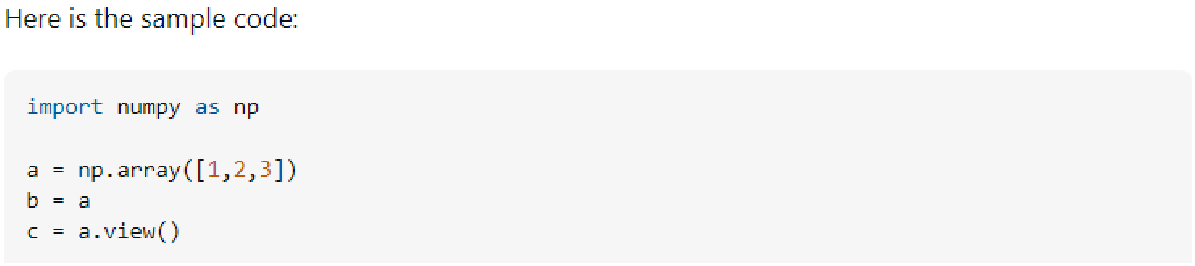
```
import numpy as np

arr1 = np.array([[1, 2], [3, 4], [5, 6]])
arr2 = arr1[[0,1,2], [0,1,0]]
print (arr2)
```

```
[1 4 5]
```

You can even index into arrays using slices, like in lists. For example, a[: ,k] will return all elements in the kth column.

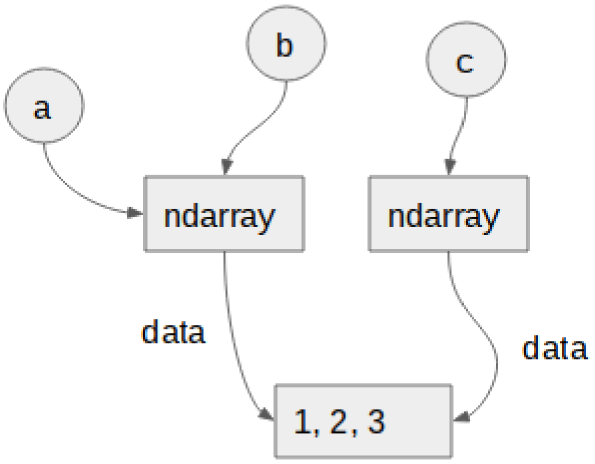
Array views and copies

A view is an object that shares the same data buffer as the original array. A copy, on the other hand, is a copy of the original array and **does not** share the same data buffer. Modifying a view's data will cause a change in the original array's data and vice-versa. Modifying a copy's data will not cause any change in the original array. The difference between assigning the original array to another variable and creating a view is as follows:



Here is the sample code:

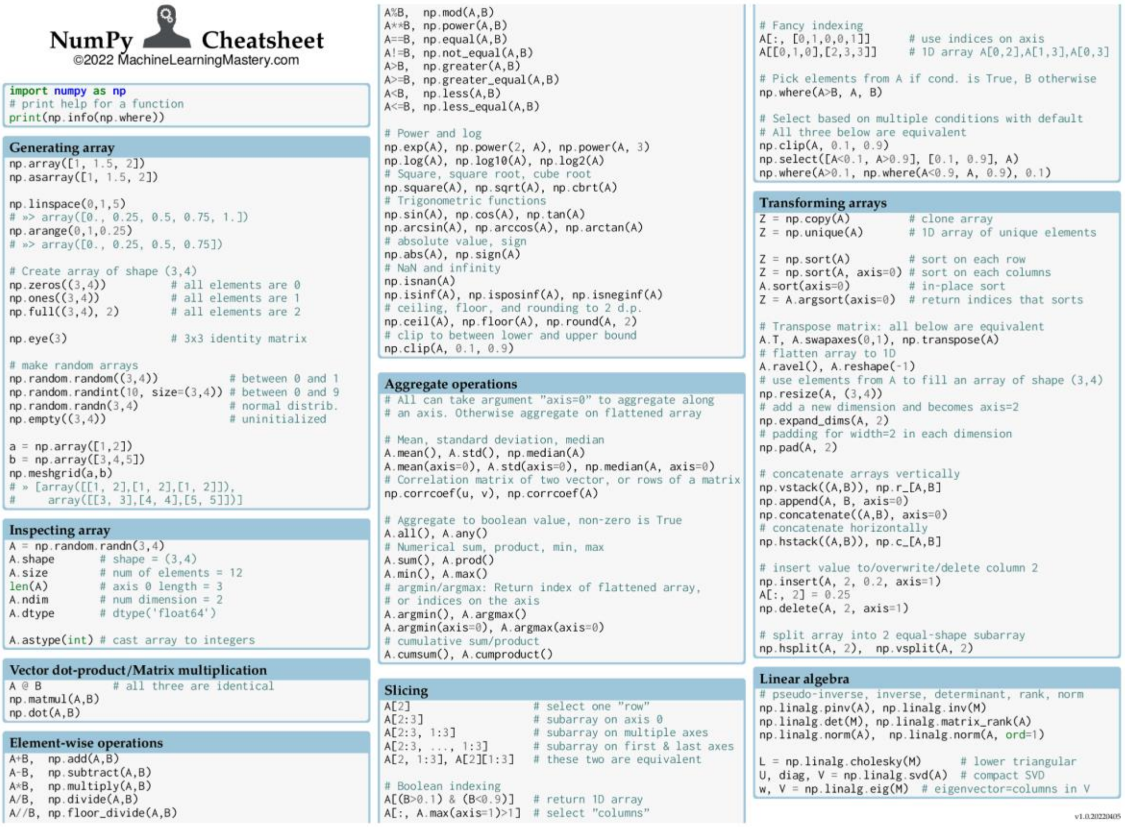
```
import numpy as np

a = np.array([1,2,3])
b = a
c = a.view()
```



Variable and variable b share the same reference/pointer while variable c is a different object that points to the same data buffer. Also, views can have a different shape from the original array. Array slicing creates views of the original array.

Array functions

The numpy module has a lot of functions that support statistics, linear algebra, array comparison, and much more.



NumPy Cheatsheet

```
import numpy as np
# print help for a function
print(np.info(np.where))
```

Generating array

```
np.array([1, 1.5, 2])
np.asarray([1, 1.5, 2])

np.linspace(0,1,5)
# »> array([0., 0.25, 0.5, 0.75, 1.])
np.arange(0,1,0.25)
# »> array([0., 0.25, 0.5, 0.75])

# Create array of shape (3,4)
np.zeros((3,4))          # all elements are 0
np.ones((3,4))           # all elements are 1
np.full((3,4), 2)        # all elements are 2

np.eye(3)                # 3x3 identity matrix

# make random arrays
np.random.random((3,4))           # between 0 and 1
np.random.randint(10, size=(3,4)) # between 0 and 9
np.random.randn(3,4)              # normal distrib.
np.empty((3,4))                   # uninitialized

a = np.array([1,2])
b = np.array([3,4,5])
np.meshgrid(a,b)
# » [array([[1, 2],[1, 2],[1, 2]]),
#     array([[3, 3],[4, 4],[5, 5]])]
```

Inspecting array

```
A = np.random.randn(3,4)
A.shape       # shape = (3,4)
A.size        # num of elements = 12
len(A)        # axis 0 length = 3
A.ndim        # num dimension = 2
A.dtype       # dtype('float64')

A.astype(int) # cast array to integers
```

Vector dot-product/Matrix multiplication

```
A @ B             # all three are identical
np.matmul(A,B)
np.dot(A,B)
```

Element-wise operations

```
A+B,  np.add(A,B)
A-B,  np.subtract(A,B)
A*B,  np.multiply(A,B)
A/B,  np.divide(A,B)
A//B, np.floor_divide(A,B)
A%B,  np.mod(A,B)
A**B, np.power(A,B)
A==B, np.equal(A,B)
A!=B, np.not_equal(A,B)
A>B,  np.greater(A,B)
A>=B, np.greater_equal(A,B)
A<B,  np.less(A,B)
A<=B, np.less_equal(A,B)

# Power and log
np.exp(A), np.power(2, A), np.power(A, 3)
np.log(A), np.log10(A), np.log2(A)
# Square, square root, cube root
np.square(A), np.sqrt(A), np.cbrt(A)
# Trigonometric functions
np.sin(A), np.cos(A), np.tan(A)
np.arcsin(A), np.arccos(A), np.arctan(A)
# absolute value, sign
np.abs(A), np.sign(A)
# NaN and infinity
np.isnan(A)
np.isinf(A), np.isposinf(A), np.isneginf(A)
# ceiling, floor, and rounding to 2 d.p.
np.ceil(A), np.floor(A), np.round(A, 2)
# clip to between lower and upper bound
np.clip(A, 0.1, 0.9)
```

Aggregate operations

```
# All can take argument "axis=0" to aggregate along
# an axis. Otherwise aggregate on flattened array

# Mean, standard deviation, median
A.mean(), A.std(), np.median(A)
A.mean(axis=0), A.std(axis=0), np.median(A, axis=0)
# Correlation matrix of two vector, or rows of a matrix
np.corrcoef(u, v), np.corrcoef(A)

# Aggregate to boolean value, non-zero is True
A.all(), A.any()
# Numerical sum, product, min, max
A.sum(), A.prod()
A.min(), A.max()
# argmin/argmax: Return index of flattened array,
# or indices on the axis
A.argmin(), A.argmax()
A.argmin(axis=0), A.argmax(axis=0)
# cumulative sum/product
A.cumsum(), A.cumproduct()
```

Slicing

```
A[2]                    # select one "row"
A[2:3]                  # subarray on axis 0
A[2:3, 1:3]             # subarray on multiple axes
A[2:3, ..., 1:3]        # subarray on first & last axes
A[2, 1:3], A[2][1:3]    # these two are equivalent

# Boolean indexing
A[(B>0.1) & (B<0.9)]    # return 1D array
A[:, A.max(axis=1)>1]   # select "columns"

# Fancy indexing
A[:, [0,1,0,0,1]]       # use indices on axis
A[[0,1,0],[2,3,3]]      # 1D array A[0,2],A[1,3],A[0,3]

# Pick elements from A if cond. is True, B otherwise
np.where(A>B, A, B)

# Select based on multiple conditions with default
# All three below are equivalent
np.clip(A, 0.1, 0.9)
np.select([A<0.1, A>0.9], [0.1, 0.9], A)
np.where(A>0.1, np.where(A<0.9, A, 0.9), 0.1)
```

Transforming arrays

```
Z = np.copy(A)          # clone array
Z = np.unique(A)        # 1D array of unique elements

Z = np.sort(A)          # sort on each row
Z = np.sort(A, axis=0) # sort on each columns
A.sort(axis=0)          # in-place sort
Z = A.argsort(axis=0)  # return indices that sorts

# Transpose matrix: all below are equivalent
A.T, A.swapaxes(0,1), np.transpose(A)
# flatten array to 1D
A.ravel(), A.reshape(-1)
# use elements from A to fill an array of shape (3,4)
np.resize(A, (3,4))
# add a new dimension and becomes axis=2
np.expand_dims(A, 2)
# padding for width=2 in each dimension
np.pad(A, 2)

# concatenate arrays vertically
np.vstack((A,B)), np.r_[A,B]
np.append(A, B, axis=0)
np.concatenate((A,B), axis=0)
# concatenate horizontally
np.hstack((A,B)), np.c_[A,B]

# insert value to/overwrite/delete column 2
np.insert(A, 2, 0.2, axis=1)
A[:, 2] = 0.25
np.delete(A, 2, axis=1)

# split array into 2 equal-shape subarray
np.hsplit(A, 2),  np.vsplit(A, 2)
```

Linear algebra

```
# pseudo-inverse, inverse, determinant, rank, norm
np.linalg.pinv(A), np.linalg.inv(M)
np.linalg.det(M), np.linalg.matrix_rank(A)
np.linalg.norm(A),  np.linalg.norm(A, ord=1)

L = np.linalg.cholesky(M)      # lower triangular
U, diag, V = np.linalg.svd(A)  # compact SVD
w, V = np.linalg.eig(M)  # eigenvector=columns in V
```

v1.0.20220405