## Aggregations: Min, Max, and Everything in Between

### Minimum and Maximum

Python has built-in min and max functions, used to find the minimum value and maximum value of any given array.

For min, max, sum, and several other NumPy aggregates, a shorter syntax is to use methods of the array object itself.

- np.min() – finds the minimum (smallest) value in the array
- np.max() – finds the maximum (largest) value in the array

Example

x=[1,2,3,4]

np.min(x)

1

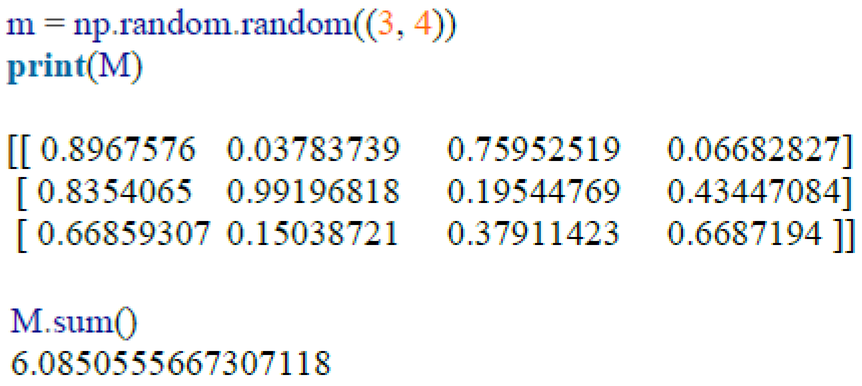
np.max(x)

4

### Multidimensional aggregates

One common type of aggregation operation is an aggregate along a row or column. By default, each NumPy aggregation function will return the aggregate over the entire array. ie. If we use the np.sum() it will calculates the sum of all elements of the array.

Example



```
m = np.random.random((3, 4))
print(M)

[[ 0.8967576   0.03783739   0.75952519   0.06682827]
 [ 0.8354065   0.99196818   0.19544769   0.43447084]
 [ 0.66859307  0.15038721   0.37911423   0.6687194 ]]

M.sum()
6.0850555667307118
```

Aggregation functions take an additional argument specifying the axis along which the aggregate is computed. The axis normally takes either 0 or 1. if the axis = 0 then it runs along with columns, if axis =1 it runs along with rows.

Example
We can find the minimum value within each column by specifying axis=0

```
M.min(axis=0)
array([ 0.66859307, 0.03783739, 0.19544769, 0.06682827])
```

Similarly, we can find the maximum value within each row

```
M.max(axis=1)
array([ 0.8967576 , 0.99196818, 0.6687194 ])
```

**Other aggregation functions**

NumPy provides many other aggregations functions most aggregates have a NaN-safe counterpart that computes the result while ignoring missing values, which are marked by the special IEEE floating-point NaN value.

| Function Name | NaN-safe Version | Description |
|---|---|---|
| np.sum | np.nansum | Compute sum of elements |
| np.prod | np.nanprod | Compute product of elements |
| np.mean | np.nanmean | Compute median of elements |
| np.std | np.nanstd | Compute standard deviation |
| np.var | np.nanvar | Compute variance |
| np.min | np.nanmin | Find minimum value |
| np.max | np.nanmax | Find maximum value |
| np.argmin | np.nanargmin | Find index of minimum value |
| np.argmax | np.nanargmax | Find index of maximum value |
| np.median | np.nanmedian | Compute median of elements |
| np.percentile | np.nanpercentile | Compute rank-based statistics of elements |
| np.any | N/A | Evaluate whether any elements are true |
| np.all | N/A | Evaluate whether all elements are true |

**Computation on Arrays: Broadcasting**

Broadcasting is simply a set of rules for applying binary ufuncs (addition, subtraction, multiplication, etc.) on arrays of different sizes.

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

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

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

Broadcasting allows these types of binary operations to be performed on arrays of different sizes.

```
a + 5

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

We can think of this as an operation that stretches or duplicates the value 5 into the array [5, 5, 5], and adds the results. The advantage of NumPy's broadcasting is that this duplication of values does not actually take place. We can similarly extend this to arrays of higher dimension. Observe the result when we add a one-dimensional array to a two-dimensional array.

Example

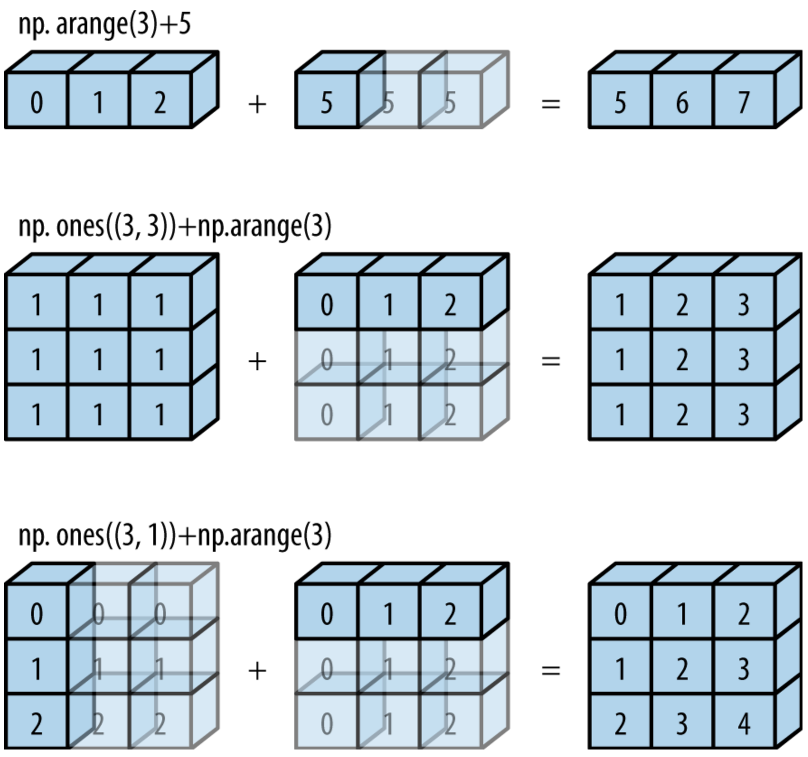
```
M = np.ones((3, 3))
M

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

M + a

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

Here the one-dimensional array a is stretched, or broadcast, across the second dimension in order to match the shape of M. Just as before we stretched or broadcasted one value to match the shape of the other, here we've stretched both a and b to match a common shape, and the result is a two dimensional array.



## Rules of Broadcasting

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

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