

NumPy

Efficient Arrays and Numerical Computing for Python

Numerical Python

Provides efficient storage and operations on dense data buffers, i.e., arrays.

- ▶ `ndarray` is the fundamental object
- ▶ Vectorized (no loop) operations on arrays
- ▶ Broadcasting
- ▶ File IO and memory-mapped files

```
In [1]: import numpy as np
```

NumPy Array Element Types

Arrays have elements of homogeneous data type

```
In [2]: nums = [1, 2, 3.14]

In [3]: nums
Out[3]: [1, 2, 3.14]

In [4]: np.array(nums)
Out[4]: array([ 1. , 2. , 3.14])

In [5]: type(Out[4][0])
Out[5]: numpy.float64
```

- ▶ Notice that the values were converted to floats.

You can specify an explicit element type with the `dtype` keyword argument:

```
In [6]: np.array(nums, dtype='int')
Out[6]: array([1, 2, 3])
```

Basic Array Creation

Pass list to `np.array()` (nested lists create multi-dimensional arrays)

```
In [9]: np.array([[1,2,3],[4,5,6]])  
Out[9]:  
array([[1, 2, 3],  
       [4, 5, 6]])
```

Create a one-dimensional array of zeros, `dtype` defaults to `float`:

```
In [10]: np.zeros(4)  
Out[10]: array([ 0., 0., 0., 0.])
```

Create a multi-dimensional array of 1s with element type `int`. Note that first argument is a tuple of array dimensions.

```
In [11]: np.ones((2, 3), dtype=int)  
Out[11]:  
array([[1, 1, 1],  
       [1, 1, 1]])
```

Create a 2-d array of the same element values:

```
In [12]: np.full((2, 3), 2.72)  
Out[12]:  
array([[ 2.72,  2.72,  2.72],  
       [ 2.72,  2.72,  2.72]])
```

`np.arange` similar to Python's built-in `range(start, end, stride)`:

```
In [13]: np.arange(0, 10, 2)  
Out[13]: array([0, 2, 4, 6, 8])
```

Creating Arrays of Random Numbers

Create a 2×3 array of values uniformly distributed between 0 and 1:

```
In [28]: np.random.random((2, 3))
Out[28]:
array([[ 0.93923457,  0.41299137,  0.07451052],
       [ 0.32800936,  0.44435825,  0.4520937 ]])
```

Create an 2×3 array of numbers normally distributed with mean 71.36 and standard deviation of 14.79:

```
In [26]: np.random.normal(71.36, 14.79, (2, 3))
Out[26]:
array([[ 71.24362489,  61.05019638,  72.25408014],
       [ 63.03759916,  70.64992342,  75.94207076]])
```

Create a 2×3 array of int values in the interval [1, 11):

```
In [29]: np.random.randint(1, 11, (2, 3))
Out[29]:
array([[9, 8, 6],
       [9, 5, 9]])
```

3-d identity matrix:

```
In [31]: np.identity(3)
Out[31]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

NumPy Array Attributes

Given:

```
In [33]: a = np.array([[1,2,3], [4,5,6]])
```

```
In [34]: a
```

```
Out[34]:  
array([[1, 2, 3],  
       [4, 5, 6]])
```

`ndim` is the number of dimensions:

```
In [37]: a.ndim
```

```
Out[37]: 2
```

`shape` is a tuple giving the number of elements in each dimension:

```
In [35]: a.shape
```

```
Out[35]: (2, 3)
```

`dtype` is the type of the elements

```
In [36]: a.dtype
```

```
Out[36]: dtype('int64')
```

1-D Array Indexing and Slicing

1-d arrays similar to Python lists:

```
In [41]: a1 = np.arange(10)
```

```
In [44]: a1[1]
```

```
Out[44]: 1
```

```
In [45]: a1[-1]
```

```
Out[45]: 9
```

```
In [46]: a1[2:5]
```

```
Out[46]: array([2, 3, 4])
```

Assignment of single value to a (sub)range *broadcasts* the value to the (sub)range:

```
In [47]: a1[2:5] = 11
```

```
In [48]: a1
```

```
Out[48]: array([ 0, 1, 11, 11, 11, 5, 6, 7, 8, 9])
```

Notice that the original array is modified.

2-D Array Indexing and Slicing

Given:

```
In [49]: a3 = np.array([[1,2,3],[4,5,6],[7,8,9]])
```

```
In [50]: a3
```

```
Out[50]:
```

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

Single scalar value:

```
In [51]: a3[1,1]
```

```
Out[51]: 5
```

Subarray:

```
In [52]: a3[1:, 1:]
```

```
Out[52]:
```

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

Single column:

```
In [53]: a3[:, 2]
```

```
Out[53]: array([3, 6, 9])
```

Single row:

```
In [54]: a3[2, :]
```

```
Out[54]: array([7, 8, 9])
```


Array Reshaping

2-d arrays

```
In [62]: a3 = np.arange(1, 13)
```

```
In [63]: a3
```

```
Out[63]: array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
```

```
In [64]: a3.reshape(3, 4)
```

```
Out[64]:
```

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

```
In [65]: a3.reshape(4, 3)
```

```
Out[65]:
```

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

Universal Functions: Vectorized Operations on Arrays

Operations between like-shaped arrays are *vectorized*, that is, the loop that applies the operations to the elements of the arrays elementwise is pushed into the compiled C-code layer instead of Python. For example:

```
In [114]: np.arange(2, 20, 2) / np.arange(1, 10)
Out[114]: array([ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.])
```

When arrays don't have the same shape, the smaller array is "broadcast" across the larger array. The simplest example is when the smaller array is a scalar value:

```
In [108]: a = np.arange(9)

In [109]: a
Out[109]: array([0, 1, 2, 3, 4, 5, 6, 7, 8])

In [110]: 2 ** a
Out[110]: array([ 1,  2,  4,  8, 16, 32, 64, 128, 256])

In [111]: 2 ** a.reshape((3, 3))
Out[111]:
array([[ 1,  2,  4],
       [ 8, 16, 32],
       [64, 128, 256]])
```

In general, broadcasting can occur between any two arrays with compatible dimensions. General broadcasting between multi-dimensional arrays is beyond the scope of this course. See [the NumPy docs](#) for details.

Array Aggregations

```
In [117]: np.arange(10).sum()
```

```
Out[117]: 45
```

```
In [119]: np.array([8,6,7,5,3,0,9]).min()
```

```
Out[119]: 0
```

```
In [120]: np.array([8,6,7,5,3,0,9]).max()
```

```
Out[120]: 9
```

2-D Aggregations

```
In [131]: np.arange(9).reshape(3,3)
Out[131]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [132]: np.arange(9).reshape(3,3).min(axis=0)
Out[132]: array([0, 1, 2])

In [133]: np.arange(9).reshape(3,3).max(axis=0)
Out[133]: array([6, 7, 8])

In [134]: np.arange(9).reshape(3,3).min(axis=1)
Out[134]: array([0, 3, 6])

In [135]: np.arange(9).reshape(3,3).max(axis=1)
Out[135]: array([2, 5, 8])
```

Boolean Operations

You can broadcast boolean expressions just like arithmetic expressions:

```
In [163]: exam1scores = np.loadtxt('exam1grades.txt')

In [164]: exam1scores
Out[164]:
array([[ 72.,  72.,  50.,  65.,  60.,  73.,  93.,  88.,  97., ...
        84.,  75.,  88.,  75.,  86.,  49.,  65.,  69.,  87.]])
```

How many people "passed"? First, you can apply a comparison operator to an array to get an array of booleans:

```
In [165]: exam1scores > 70
Out[165]:
array([ True,  True, False, False, False,  True,  True,  True,  True, ...
        True,  True,  True,  True,  True, False, False, False,  True], dtype=bool)
```

Then you can apply the `np.sum` aggregation function to count the booleans in the resulting array of booleans:

```
In [169]: np.sum(exam1scores > 70)
Out[169]: 77
```

You can also combine comparisons with logical operators. How many Bs?

```
In [173]: np.sum((exam1scores >= 80) & (exam1scores < 90))
Out[173]: 27
```

Note the syntax with single `&` – NumPy uses efficient bitwise logical operators.

Masking

First, boolean indexing: you can use a like-shaped array of bools to index into an array, which selects items from the array. The arrays of bools is called a *mask* and using it to select elements is called *masking*.

```
In [175]: xs = np.array([0,1,2,3,4,5,6,7,8,9])
```

```
In [177]: xs[[True, False, True, False, True, False, True, False, True, False]]  
Out[177]: array([0, 2, 4, 6, 8])
```

Since you can create arrays of bools easily with comparison ufuncs, you can combine boolean indexing with broadcasting to easily mask an array:

```
In [179]: xs[(xs % 2) == 0]  
Out[179]: array([0, 2, 4, 6, 8])
```

Missing Data

Missing array elements represented as `np.nan` values.

```
In [86]: xs = np.array([2, 3, 4, np.nan])
```

```
In [87]: np.mean(xs)
```

```
Out[87]: nan
```

Ways to handle missing values:

- ▶ Manually masking with `np.isnan`

```
In [90]: np.mean(xs[[not np.isnan(x) for x in xs]])
```

```
Out[90]: 3.0
```

- ▶ Masking using the [numpy.ma](#) module.

```
In [92]: np.ma.masked_invalid(xs).mean()
```

```
Out[92]: 3.0
```

- ▶ Using NaN-ignoring aggregates:

```
In [93]: np.nanmean(xs)
```

```
Out[93]: 3.0
```

Pandas gives you a few more options, but these cover many cases that come up in practice.

np.where

`np.where(cond, true_result, false_result)` is a vectorized version of Python's ternary if-else expression.

Here, we double all the even numbers:

```
In [12]: a = np.array([[1,2,3], [4,5,6], [7,8,9]])
```

```
In [14]: a
```

```
Out[14]:
```

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

```
In [15]: np.where((a % 2) == 0, a * 2, a)
```

```
Out[15]:
```

```
array([[ 1,  4,  3],  
       [ 8,  5, 12],  
       [ 7, 16,  9]])
```

Exercise: do that operation above using basic Python on a list of lists.