NumPy

Numerical Python

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

- `ndarray` is the fundamental object
- Vectorized operations on arrays
- Broadcasting
- File IO amd memory-mapped files

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

NumPy Array Element Types

Arrays have elements of homogeneous data type

```
In [2]: a = np.array([1, 2, 3.14])

In [3]: type(a)
Out[3]: numpy.ndarray

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

In [5]: type(a[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])
```

One-dimensional Arrays

Pass list to `np.array()`:

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

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

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

`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])
```

Multi-Dimensional Arrays

Passing nested lists to `np.array()` 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 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]])
```

Creating Arrays of Random Numbers

Creat a 2 x 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 ]])
```

Normally distributed with $\mu = 71.36$ and $\sigma = 14.79$:

```
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 x 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:

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]])
```

Python is slow

- Consider an array representing pixels of a “one megapixel” image:

```
In [20]: image = np.random.randint(0, 256, (1000000, 3))
```

- This is a deep underwater image which looks very green and we want to increase the “blueness” by 10% [fn:1]. So we write a function to mutiply pixel elements by a factor:

```
In [60]: def mult_elem(image, n, factor):
    ...:     for i in range(len(image)):
    ...:         image[i][n] = image[i][n] * factor
```

- This operation is *slow*:

```
In [61]: %timeit mult_elem(image, 2, 1.10)
1.85 s +/- 16.8 ms per loop (mean +/- std. dev. of 7 runs, 1 loop each)
```

- The equivalent vectorized opertation is /300 times faster/:

```
In [62]: %timeit image[:, 2] = image[:, 2] * 1.10
6.23 ms +/- .0693 ms per loop (mean +/- std. dev. of 7 runs, 100 loops each)
```

[fn:1] I’m not a graphics guy, so just indulge me here.

Vectorized Operations on Arrays

Operations between compatibly-shaped arrays or between arrays and scalars are *vectorized* – the loop applying the operations to elements of the array(s) is in the compiled C-code layer instead of Python.

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

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 [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]])
```

General braodcasting between multi-dimensional arrays is beyond the scope of this course. See the NumPy docs for details.

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])
```

The comparison operation above is a boolean universal function.

Boolean UFuncs

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 boooleans:

```
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
```

Boolean UFuncs

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.

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]])
```

We can summarize the values of each column,

```
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])
```

or summarize the values in each row:

```
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])
```

Note that *axis* here means *dimension to be collapsed*. So axis 0 means we collapse the rows into one array by aplying the aggregation function by column.

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.

Closing Thoughts

Key ideas of NumPy:

- In-memory arrays of elements with the same data type
- Static typing of arrays together with vectorized operations of universal functions provide dramatic speed up over equivalent Python code
- Ufuncs combined with with boolean masks makes it easy to partition data
- Aggregate functions make it easy to summarize data

NumPy is the foundation of the SciPy stack. Even when we don't use it directly (which we often will), it's there underneath the hood.