Numpy

June 20, 2016

1 Numpy

One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large data sets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements must be the same type. Every array has a shape, a tuple indicating the size of each dimension, and a dtype, an object describing the data type of the array

```
In [34]: import numpy as np
         from numpy import random
         data = np.array([1,2,3]); data
```

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

```
In [7]: print(data.shape); data.dtype
```

```
(3,)
```

```
Out[7]: dtype('int64')
```

The easiest way to create an array is to use the array function. This accepts any se- quence-like object (including other arrays) and produces a new NumPy array contain- ing the passed data. Nested sequences, like a list of equal-length lists, will be converted into a multidimen- sional array:

```
In [10]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
         arr2 = np.array(data2); print(arr2)
         arr2.shape
```

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

```
Out[10]: (2, 4)
```

```
In [17]: print(np.zeros((2,3)).shape); np.eye(4,2).shape
```

```
(2, 3)
```

```
Out[17]: (4, 2)
```

You can explicitly convert or cast an array from one dtype to another using ndarray's astype method:

```
In [19]: float_arr = arr2.astype(np.float64); float_arr
```

```
Out[19]: array([[ 1.,  2.,  3.,  4.],
                [ 5.,  6.,  7.,  8.]])
```

arange is an array-valued version of the built-in Python range function:

```
In [20]: np.arange(15)

Out[20]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Arrays are important because they enable you to express batch operations on data without writing any for loops. This is usually called vectorization. Any arithmetic op- erations between equal-size arrays applies the operation elementwise.

Arithmetic operations with scalars are as you would expect, propagating the value to each element.

```
In [22]: print(arr2-arr2); arr2/2

[[0 0 0 0]
 [0 0 0 0]]

Out[22]: array([[ 0.5,  1. ,  1.5,  2. ],
                [ 2.5,  3. ,  3.5,  4. ]])
```

1.1 Basic Index and slicing

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset.

```
In [23]: arr = np.arange(10)
         arr[5]

Out[23]: 5
```

In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays.

In multidimensional arrays, if you omit later indices, the returned object will be a lower- dimensional ndarray consisting of all the data along the higher dimensions (i.e., pass first dimension and np will understand as give me the other dimensions)

```
In [32]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
         arr2d[2] # third row
         arr2d[0, 2] # first row and second column
         arr2d[:, :1] # All rows and the first column

Out[32]: array([[1],
                [4],
                [7]])
```

Boolean indexing: both the boolean and the array must be of the same length. You can combine boolean indexing with integer indexs.

```
In [36]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
         data = random.randn(7, 4)
         data[names == 'Bob'] # pass me rows that match the boolean tuple name == 'Bob'

Out[36]: array([[ 1.57952361, -0.78912249, -0.90513602, -0.29086422],
                [ 0.64280752, -0.22957581,  0.96624177, -0.04311981]])
```

To select everything but 'Bob', you can either use != or negate the condition using ~:

```
In [41]: data[~(names == 'Bob')]
```

```
Out[41]: array([[-0.94562579, -0.78398283, -1.14576263,  0.38169993],
                [ 1.08518446,  1.52575289,  0.49105393, -0.37885421],
                [ 0.92645591,  1.34995855, -0.85850141,  0.34978215],
                [-1.19088969, -1.03275245,  0.2643695 , -0.64296291],
                [-0.15733807, -0.18148319,  1.19668514,  0.68572787]])

In [43]: print(data.shape)
         data.T.shape

(7, 4)

Out[43]: (4, 7)
```

A universal function, or ufunc, is a function that performs elementwise operations on data in ndarrays

```
In [45]: np.abs(data);
         np.greater(data, data)

Out[45]: array([[False, False, False, False],
                [False, False, False, False],
                [False, False, False, False],
                [False, False, False, False],
                [False, False, False, False],
                [False, False, False, False],
                [False, False, False, False]], dtype=bool)
```

1.2 Vectorization

Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops. This practice of replacing explicit loops with array expressions is commonly referred to as vectoriza- tion.

The numpy.where function is a vectorized version of the ternary expression x if condition else y. That is, it returns for the array evaluated what'd be done with a for loop.

A typical use of where in data analysis is to produce a new array of values based on another array. Suppose you had a matrix of randomly generated data and you wanted to replace all positive values with 2 and all negative values with -2. This is very easy to do with np.where:

```
In [47]: arr = random.randn(4, 4)
         print(np.where(arr > 0, 2, -2))
         np.where(arr > 0, 2, arr) # set only positive values to 2

[[ 2  2  2  2]
 [-2 -2 -2 -2]
 [ 2  2  2  2]
 [-2 -2 -2  2]]

Out[47]: array([[ 2.        ,  2.        ,  2.        ,  2.        ],
                [-0.74421119, -0.43169346, -0.21717698, -1.44943132],
                [ 2.        ,  2.        ,  2.        ,  2.        ],
                [-0.28945964, -0.79812387, -0.59861476,  2.        ]])
```

1.3 Mathematical functions

A set of mathematical functions which compute statistics about an entire array or about the data along an axis are accessible as array methods. arrays have different axes: the first running vertically downwards across rows (axis 0), and the second running horizontally across columns (axis 1).

```
In [56]: arr = np.random.randn(5, 4) # normally-distributed data
         print(arr)
         arr.mean(axis = 0) # mean for columns
         arr.mean(axis = 1) # mean for rows
```

```
[[ 0.70854543  1.33234993 -0.18716781  0.88292194]
 [ 0.01605364  1.26289966 -0.26594632  0.71641893]
 [ 1.65305611  0.17438206  1.10062451  0.98385526]
 [-0.01633144 -1.17617375  0.01698707 -1.72375708]
 [ 0.71462109  0.34357872 -0.63009946 -0.34884192]]
```

```
Out[56]: array([ 0.61518897,  0.38740733,  0.0068796 ,  0.10211943])
```

Boolean arrays also have different methods.

```
In [58]: arr = random.randn(100)
         (arr > 0).sum() # Number of positive values
```

```
Out[58]: 52
```