NumPy (Numerical Python)

February 27, 2018

0.1 NumPy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

1) a powerful N-dimensional array object
2) sophisticated (broadcasting) functions
3) tools for integrating C/C++ and Fortran code
4) useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

One of the reasons NumPy is so important for numerical computations in Python is because it is designed for efficiency on large arrays of data. There are a number of reasons for this:

NumPy internally stores data in a contiguous block of memory, independent of other built-in Python objects. NumPy's library of algorithms written in the C language can operate on this memory without any type checking or other overhead. NumPy arrays also use much less memory than built-in Python sequences.

NumPy operations perform complex computations on entire arrays without the need for Python for loops.

```
In [2]: import numpy as np
        my_arr = np.arange(1000000)
        my_list = list(range(1000000))
        %time for _ in range(10): my_arr2 = my_arr * 2

        %time for _ in range(10): my_list2 = [x * 2 for x in my_list]
```

```
Wall time: 24.6 ms
Wall time: 1.09 s
```

NumPy-based algorithms are generally 10 to 100 times faster (or more) than their pure Python counterparts and use significantly less memory.

0.2 The NumPy ndarray: A Multidimensional Array Object

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

```
In [3]: #standard NumPy convention of always using import numpy as np.
        import numpy as np
        data = np.random.randn(2, 3)
        print(data)
        print(data * 10)
        print(data + data)
```

```
[[ 1.03912885  0.48295306 -0.0292333 ]
 [ 0.46265113 -0.11299027 -1.9762025 ]]
[[ 10.39128849   4.82953061  -0.29233304]
 [  4.62651131  -1.1299027  -19.76202505]]
[[ 2.0782577   0.96590612 -0.05846661]
 [ 0.92530226 -0.22598054 -3.95240501]]
```

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 [4]: data.shape
        data.dtype
```

```
Out[4]: dtype('float64')
```

0.3 Creating ndarrays

The easiest way to create an array is to use the array function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data. For example, a list is a good candidate for conversion

```
In [5]: data1 = [6, 7.5, 8, 0, 1]
        arr1 = np.array(data1)
        print(arr1)
```

```
[ 6.   7.5  8.   0.   1. ]
```

Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

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

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

```
In [8]: arr2.ndim
        arr2.shape
```

```
Out[8]: (2, 4)
```

In addition to np.array, there are a number of other functions for creating new arrays. As examples, zeros and ones create arrays of 0s or 1s, respectively, with a given length or shape. empty creates an array without initializing its values to any particular value. To create a higher dimensional array with these methods, pass a tuple for the shape:

```
In [10]: np.zeros(10)
         np.zeros((3, 6))
         #arange is an array-valued version of the built-in Python range function
         np.arange(15)
```

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

0.4 Data Types for ndarrays

The data type or dtype is a special object containing the information (or metadata, data about data) the ndarray needs to interpret a chunk of memory as a particular type of data dtypes are a source of NumPy's flexibility for interacting with data coming from other systems. In most cases they provide a mapping directly onto an underlying disk or memory representation, which makes it easy to read and write binary streams of data to disk and also to connect to code written in a low-level language like C or Fortran. The numerical dtypes are named the same way: a type name, like float or int, followed by a number indicating the number of bits per element. A standard doubleprecision floating-point value (what's used under the hood in Python's float object) takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as float64. See Table 4-2 for a full listing of NumPy's supported data typ

```
In [3]: import numpy as np
        arr1 = np.array([1, 2, 3], dtype=np.float64)
        arr2 = np.array([1, 2, 3], dtype=np.int32)
        arr1.dtype
        arr2.dtype
```

```
Out[3]: dtype('int32')
```

0.5 Arithmetic with NumPy Arrays

Arrays are important because they enable you to express batch operations on data without writing any for loops. NumPy users call this vectorization. Any arithmetic operations between equal-size arrays applies the operation **element-wise** :

```
In [10]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])
         arr
         arr*arr
         arr-arr
         1/arr
         arr**2
         arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
         arr2>arr

Out[10]: array([[False,  True, False],
                [ True, False,  True]], dtype=bool)
```

0.6 Basic Indexing and Slicing

1) One-dimensional arrays are simple; on the surface they act similarly to Python lists
2) An important first distinction from Python's built-in lists is that array slices are views on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array. As NumPy has been designed to be able to work with very large arrays, you could imagine performance and memory problems if NumPy insisted on always copying data. If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array—for example,arr[5:8].copy().
3) The "bare" slice [:] will assign to all values in an array:
4) In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays; array2d[rowindx][colidx] or array2d[rowinx, colindx]
5) It is helpful to think of axis 0 as the "rows" of the array and axis 1 as the "columns."

```
In [25]: arr = np.arange(10)
         arr
         arr[5]
         arr[3:5]

         arr[6:9] = 100
         arr

         arr_slice = arr[5:8]
         arr_slice
         arr_slice[1] = 12345
         arr_slice
         arr

         arr_slice[:] = 2018
         arr_slice
         arr

         arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
         arr2d
         arr2d[2]
         arr2d[0][2]
         arr2d[0,2]
```

```
Out[25]: 3
```

0.7 Indexing with slices

1) 1D arry: same syntax as Python Lists
2) 2D array: it has sliced along axis 0, the first axis
3) A colon by itself means to take the entire axis
4) assigning to a slice expression assigns to the whole selection

```
In [32]: arr[1:6]
         arr2d
         arr2d[:2]
         arr2d[:2, 1:]
         arr2d[1, :2]
         arr2d[:2, 2]

         arr2d[:, :1]
         arr2d[:2, 1:] = 0
         arr2d
```

```
Out[32]: array([[1, 0, 0],
                [4, 0, 0],
                [7, 8, 9]])
```

0.8 Boolean Indexing

```
In [50]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
         data = np.random.randn(7, 4)
         data
         names == 'Bob'
         data[names == 'Bob']
         data[names == 'Bob', 2:]
         data[names == 'Bob', 3]
         names != 'Bob'
         data[~(names == 'Bob')]
         cond = names == 'Bob'
         data[~cond]
         mask = (names == 'Bob') | (names == 'Will')
         data[mask]
         data[data < 0] = 0
         data
         data[names != 'Joe'] = 7
         data
```

```
Out[50]: array([[ 7.        ,  7.        ,  7.        ,  7.        ],
                [ 0.13614019,  0.84213464,  0.58824823,  0.        ],
                [ 7.        ,  7.        ,  7.        ,  7.        ],
                [ 7.        ,  7.        ,  7.        ,  7.        ],
                [ 7.        ,  7.        ,  7.        ,  7.        ],
                [ 0.        ,  0.65777635,  0.71114127,  0.        ],
                [ 0.        ,  0.        ,  0.        ,  0.28042159]])
```

0.9 Fancy Indexing

the result of fancy indexing is **always one-dimensional**. Keep in mind that fancy indexing, unlike slicing, always copies the data into a new array

1) To select out a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order
2) Using negative indices selects rows from the end
3) Passing multiple index arrays does something slightly different; it selects a onedimensional array of elements corresponding to each tuple of indices

```
In [61]: arr = np.empty((8, 4))
         for i in range(8):
             arr[i] = i
         arr
         arr[[4, 3, 0, 6]]
         arr[[-3, -5, -7]]

         arr = np.arange(32).reshape((8, 4))

         arr
         arr[[1, 5, 7, 2], [0, 3, 1, 2]]
         arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]

Out[61]: array([[ 4,  7,  5,  6],
                [20, 23, 21, 22],
                [28, 31, 29, 30],
                [ 8, 11,  9, 10]])
```

0.10 Transposing Arrays and Swapping Axes

```
In [65]: arr = np.arange(15).reshape((3, 5))
         arr
         arr.T
         arr = np.random.randn(6, 3)
         np.dot(arr.T, arr)

Out[65]: array([[ 7.98979573,  1.48819748, -0.31197102],
                [ 1.48819748,  1.23241087, -0.66337781],
                [-0.31197102, -0.66337781,  2.18304654]])
```

0.11 Universal Functions: Fast Element-Wise Array Functions

A universal function, or ufunc, is a function that performs element-wise operations on data in ndarrays. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results

```
In [69]: arr = np.arange(10)
         arr
         np.sqrt(arr)
         np.exp(arr)
         x = np.random.randn(8)
         y = np.random.randn(8)
         np.maximum(x, y)
```

```
Out[69]: array([-0.66931476,  0.19265346, -0.3535984 ,  0.26493831,  0.89825499,
                1.06819045, -0.55193539,  0.03684383])
```

0.12 Expressing Conditional Logic as Array Operations

The numpy.where function is a vectorized version of the ternary expression x if con dition else y.

```
In [72]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
         yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
         cond = np.array([True, False, True, True, False])
         result = [(x if c else y)
             for x, y, c in zip(xarr, yarr, cond)]
         result
```

```
Out[72]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```

This has multiple problems. First, it will not be very fast for large arrays (because all the work is being done in interpreted Python code). Second, it will not work with multidimensional arrays. With np.where you can write this very concisely

```
In [74]: result = np.where(cond, xarr, yarr)
```

The second and third arguments to np.where don't need to be arrays; one or both of them can be scalars. 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 [77]: arr = np.random.randn(4, 4)
         arr
         arr > 0
         np.where(arr > 0, 2, -2)
         np.where(arr > 0, 2, arr) # set only positive values to 2
```

```
Out[77]: array([[ 2.        , -0.51691092, -0.03666309,  2.        ],
               [-2.06259724, -1.36498876,  2.        , -0.62413161],
               [ 2.        ,  2.        , -0.07064881,  2.        ],
               [-0.99580665, -0.06095904, -1.05862878, -1.39717283]])
```

0.13 Mathematical and Statistical Methods

A set of mathematical functions that compute statistics about an entire array or about the data along an axis are accessible as methods of the array class. You can use aggregations (often called reductions) like **sum, mean, and std (standard deviation)** either by calling the array instance method or using the top-level NumPy function

Functions like mean and sum take **an optional axis argument that computes the statistic over the given axis**, resulting in an array with one fewer dimension

```
In [80]: arr = np.random.randn(5, 4)
         arr
         arr.mean()
         np.mean(arr)
         arr.sum()
         arr.mean(axis=1)
         arr.sum(axis=0)

Out[80]: array([-0.71685849,  2.29175132, -1.31499241,  0.53303887])
```

0.14 Methods for Boolean Arrays

Boolean values are coerced to 1 (True) and 0 (False) in the preceding methods. Thus, sum is often used as a means of counting True values in a boolean array

```
In [81]: arr = np.random.randn(100)
         (arr > 0).sum() # Number of positive values

Out[81]: 44
```

There are two additional methods, any and all, useful especially for boolean arrays. any tests whether one or more values in an array is True, while all checks if every value is True

```
In [82]: bools = np.array([False, False, True, False])
         bools.any()
         bools.all()

Out[82]: False
```

0.15 Sorting

Like Python's built-in list type, NumPy arrays can be sorted in-place with the sort method

```
In [84]: arr = np.random.randn(6)
         arr.sort()
```

You can sort each one-dimensional section of values in a multidimensional array inplace along an axis by passing the axis number to sort

```
In [85]: arr = np.random.randn(5, 3)
```

0.16 Linear Algebra

Linear algebra, like matrix multiplication, decompositions, determinants, and other square matrix math, is an important part of any array library. Unlike some languages like MATLAB, multiplying two two-dimensional arrays with * is an element-wise product instead of a matrix dot product. Thus, there is a function dot, both an array method and a function in the numpy namespace, for matrix multiplication

```
In [86]: x = np.array([[1., 2., 3.], [4., 5., 6.]])
         y = np.array([[6., 23.], [-1, 7], [8, 9]])
         x.dot(y)
         np.dot(x, y)
         np.dot(x, np.ones(3))

         x @ np.ones(3)

Out[86]: array([  6.,  15.])
```