Numpy

NumPy, short for Numerical Python, has long been a cornerstone of numerical computing in Python. It provides the data structures, algorithms, and library glue needed for most scientific applications involving numerical data in Python. NumPy contains, among other things:

- A fast and efficient multidimensional array object ndarray
- Functions for performing element-wise computations with arrays or mathematical operations between arrays
- Tools for reading and writing array-based datasets to disk
- Linear algebra operations, Fourier transform, and random number generation
- A mature C API to enable Python extensions and native C or C++ code to access NumPy's data structures and computational facilities.

Beyond the fast array-processing capabilities that NumPy adds to Python, one of its primary uses in data analysis is as a container for data to be passed between algorithms and libraries. For numerical data, NumPy arrays are more efficient for storing and manipulating data than the other built-in Python data structures. Also, libraries written in a lower-level language, such as C or Fortran, can operate on the data stored in a NumPy array without copying data into some other memory representation. Thus, many numerical computing tools for Python either assume NumPy arrays as a primary data structure or else target seamless interoperability with NumPy.

Why Numpy?

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. To give you an idea of the performance difference, consider a NumPy array of one million integers, and the equivalent Python list:

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

In [2]: my_arr = np.arange(1000000)
        my_list = list(range(1000000))
```

Now let's multiply each sequence by 2:

```
In [3]: %time for _ in range(10): my_arr2 = my_arr * 2
        Wall time: 32 ms

In [4]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]
        Wall time: 1.71 s
```

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

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.

To give you a flavor of how NumPy enables batch computations with similar syntax to scalar values on built-in Python objects, we first import NumPy and generate a small array of random data:

```
In [5]: import numpy as np
        # Generate some random data
        data = np.random.randn(2, 3)

In [6]: data
Out[6]: array([[-1.72703539, -0.30991483, -0.36867274],
               [-0.9939939 , -0.35093629,  0.18523807]])
```

Then write mathematical operations with data:

```
In [7]: data * 10
Out[7]: array([[-17.27035392,  -3.09914827,  -3.68672745],
               [ -9.93993904,  -3.50936293,   1.85238069]])

In [8]: data + data
Out[8]: array([[-3.45407078, -0.61982965, -0.73734549],
               [-1.98798781, -0.70187259,  0.37047614]])
```

In the first example, all of the elements have been multiplied by 10. In the second, the corresponding values in each “cell” in the array have been added to each other.

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 [9]: data.shape
Out[9]: (2, 3)

In [10]: data.dtype
Out[10]: dtype('float64')
```

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 [11]: data1 = [6, 7.5, 8, 0, 1]

In [12]: arr1 = np.array(data1)

In [13]: arr1
Out[13]: array([6. , 7.5, 8. , 0. , 1. ])
```

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

```
In [14]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
         arr2 = np.array(data2)
         arr2
Out[14]: array([[1, 2, 3, 4],
                [5, 6, 7, 8]])
```

Since data2 was a list of lists, the NumPy array arr2 has two dimensions with shape inferred from the data. We can confirm this by inspecting the ndim and shape attributes:

```
In [16]: arr2.ndim
Out[16]: 2

In [17]: arr2.shape
Out[17]: (2, 4)
```

Unless explicitly specified, np.array tries to infer a good data type for the array that it creates. The data type is stored in a special dtype metadata object; for example, in the previous two examples we have:

```
In [18]: arr1.dtype
Out[18]: dtype('float64')

In [19]: arr2.dtype
Out[19]: dtype('int32')
```

Homogeneous n-dimensional vectors:

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 [20]: np.zeros(10)
Out[20]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [21]: np.zeros((3, 6))
Out[21]: array([[0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0.]])

In [22]: np.empty((2, 3, 2))
Out[22]: array([[[9.27054442e-312, 2.47032823e-322],
                 [0.00000000e+000, 0.00000000e+000],
                 [1.11260619e-306, 1.58817677e-052]],

                [[4.51522363e-090, 8.37963633e+169],
                 [1.52536973e-052, 1.96821284e+160],
                 [3.99910963e+252, 4.93432906e+257]]])

In [23]: np.full((5,5), 3, np.int)
Out[23]: array([[3, 3, 3, 3, 3],
                [3, 3, 3, 3, 3],
                [3, 3, 3, 3, 3],
                [3, 3, 3, 3, 3],
                [3, 3, 3, 3, 3]])
```

Note:It's not safe to assume that np.empty will return an array of all zeros. In some cases, it may return uninitialized "garbage" values.

Matrix functions

1. The identity matrix:
   In linear algebra, the identity matrix of size n is the n × n square matrix with ones on the main diagonal and zeros elsewhere.

```
In [23]: np.full((5,5), 3, np.int)

Out[23]: array([[3, 3, 3, 3, 3],
                [3, 3, 3, 3, 3],
                [3, 3, 3, 3, 3],
                [3, 3, 3, 3, 3],
                [3, 3, 3, 3, 3]])

In [24]: np.identity(5)

Out[24]: array([[1., 0., 0., 0., 0.],
                [0., 1., 0., 0., 0.],
                [0., 0., 1., 0., 0.],
                [0., 0., 0., 1., 0.],
                [0., 0., 0., 0., 1.]])

In [25]: np.identity(5, dtype = np.int)

Out[25]: array([[1, 0, 0, 0, 0],
                [0, 1, 0, 0, 0],
                [0, 0, 1, 0, 0],
                [0, 0, 0, 1, 0],
                [0, 0, 0, 0, 1]])

In [26]: np.identity(4, dtype = np.bool)

Out[26]: array([[ True, False, False, False],
                [False,  True, False, False],
                [False, False,  True, False],
                [False, False, False,  True]])
```

2. Function ‘eye’:
   It is used to return a 2-D array with ones on the diagonal and zeros elsewhere.

```
In [27]: np.eye(3)

Out[27]: array([[1., 0., 0.],
                [0., 1., 0.],
                [0., 0., 1.]])

In [28]: np.eye(3,3,-1)

Out[28]: array([[0., 0., 0.],
                [1., 0., 0.],
                [0., 1., 0.]])

In [29]: np.eye(3,3,1)

Out[29]: array([[0., 1., 0.],
                [0., 0., 1.],
                [0., 0., 0.]])

In [30]: np.eye(5,10, 3, dtype = np.int)

Out[30]: array([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]])
```

3. Function ‘diag’:
   The diag function takes two arguments:
   - An ndarray v.
   - An integer k (default = 0). If ‘v’ is dimension is 1 then the function constructs a matrix where its diagonal number k is formed by the elements of the vector ‘v’. If a is a matrix (dimension 2) then the function extracts the elements of the kth diagonal in a one-dimensional vector.

```
In [31]: np.diag([1,5,7])
         a = np.ones((3,3)) * 5
         np.diag(a)

Out[31]: array([5., 5., 5.])
```

4. Function ‘fromfunction’:
   Construct an array by executing a function over each coordinate.
   Let’s create a vector-based on its indices.

```
In [32]: np.fromfunction(lambda i,j : i-j, (3,3))
Out[32]: array([[ 0., -1., -2.],
                [ 1.,  0., -1.],
                [ 2.,  1.,  0.]])

In [33]: np.fromfunction(lambda i, j : i == j, (5,5))
Out[33]: array([[ True, False, False, False, False],
                [False,  True, False, False, False],
                [False, False,  True, False, False],
                [False, False, False,  True, False],
                [False, False, False, False,  True]])
```

5. Function ‘vectorize’

The purpose of numpy.vectorize is to transform functions which are not numpy-aware into functions that can operate on (and return) numpy arrays

Let’s build a function sign, allowing us to calculate the sign of a value.

```
In [34]: def sign(x):
           if x == 0:
              return 0
           elif x > 0:
              return 1
           else:
              return -1

In [35]: sign(8)
Out[35]: 1

In [36]: sign(-4.32)
Out[36]: -1

In [37]: sign(0)
Out[37]: 0
```

Array creation functions

Function	Description
array	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype; copies the input data by default
asarray	Convert input to ndarray, but do not copy if the input is already an ndarray arange Like the built-in range but returns an ndarray instead of a list
ones, ones_like	Produce an array of all 1s with the given shape and dtype; ones_like takes another array and produces a ones array of the same shape and dtype
zeros, zeros_like	Like ones and ones_like but producing arrays of 0s instead
empty, empty_like	Create new arrays by allocating new memory, but do not populate with any values like ones and zeros
full, full_like	Produce an array of the given shape and dtype with all values set to the indicated “fill value” full_like takes another array and produces a filled array of the same shape and dtype
eye, identity	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

Aggregation methods:

- **ndarray.sum:** Return the sum of the array elements over the given axis.
- **ndarray.sum:** Return the product of the array elements over the given axis.
- **ndarray.max:** Return the maximum along the given axis.
- **ndarray.min:** Return the minimum along the given axis.
- **ndarray.mean:** Returns the average of the array elements along the given axis.
- **ndarray.cumsum:** Return the cumulative sum of the elements along the given axis.
- **ndarray.cumprod:** Return the cumulative product of the elements along the given axis.
- **ndarray.var:** Returns the variance of the array elements, along the given axis.
- **ndarray.std:** Returns the standard deviation of the array elements along the given axis.

- **ndarray.argmin:** Return indices of the minimum values along the given axis.
- **ndarray.argmax:** Return indices of the maximum values along the given axis.

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:

```
In [23]: arr1 = np.array([1, 2, 3], dtype=np.float64)
         arr2 = np.array([1, 2, 3], dtype=np.int32)
         arr1.dtype

Out[23]: dtype('float64')

In [24]: arr2.dtype

Out[24]: dtype('int32')
```

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 double precision 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.

Type	Type code	Description
int8, uint8	i1, u1	Signed and unsigned 8-bit (1 byte) integer types
int16, uint16	i2, u2	Signed and unsigned 16-bit integer types
int32, uint32	i4, u4	Signed and unsigned 32-bit integer types
int64, uint64	i8, u8	Signed and unsigned 64-bit integer types
float16	f2	Half-precision floating point
float32	f4 or f	Standard single-precision floating point; compatible with C float

float64	f8 or d	Standard double-precision floating point; compatible with C double and Python float object
float128	f16 or g	Extended-precision floating point
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool	?	Boolean type storing True and False values
object	O	Python object type; a value can be any Python object
string_	S	Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use 'S10'
unicode_	U	Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string_ (e.g., 'U10')

Note: It's important to be cautious when using the numpy.string_ type, as string data in NumPy is fixed size and may truncate input without warning. pandas has more intuitive out-of-the-box behavior on non-numeric data.

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 [3]: import numpy as np

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

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

In [7]: arr * arr
Out[7]: array([[ 1.,  4.,  9.],
               [16., 25., 36.]])

In [8]: arr - arr
Out[8]: array([[0., 0., 0.],
               [0., 0., 0.]])
```

Arithmetic operations with scalars propagate the scalar argument to each element in the array:

```
In [9]: 1 / arr
Out[9]: array([[1.        , 0.5       , 0.33333333],
               [0.25      , 0.2       , 0.16666667]])

In [10]: arr ** 0.5
Out[10]: array([[1.        , 1.41421356, 1.73205081],
                [2.        , 2.23606798, 2.44948974]])
```

Comparisons between arrays of the same size yield boolean arrays:

```
In [11]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

In [12]: arr2
Out[12]: array([[ 0.,  4.,  1.],
                [ 7.,  2., 12.]])

In [13]: arr2 > arr
Out[13]: array([[False,  True, False],
                [ True, False,  True]])
```

Operations between differently sized arrays is called broadcasting

Basic Indexing and Slicing

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
In [20]: arr = np.arange(10)

In [21]: arr
Out[21]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [22]: arr[5]
Out[22]: 5

In [23]: arr[5:8]
Out[23]: array([5, 6, 7])

In [24]: arr[5:8] = 12

In [25]: arr
Out[25]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

As you can see, if you assign a scalar value to a slice, as in arr[5:8] = 12, the value is propagated (or broadcasted henceforth) to the entire selection. 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. To give an example of this, first create a slice of arr:

```
In [26]: arr_slice = arr[5:8]

In [27]: arr_slice
Out[27]: array([12, 12, 12])
```

Now, when we change values in arr_slice, the mutations are reflected in the original array arr:

```
In [28]: arr_slice[1] = 12345
         arr

Out[28]: array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
                9])
```

The "bare" slice [:] will assign to all values in an array:

```
In [29]: arr_slice[:] = 64
         arr
Out[29]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

Note: 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().

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
In [30]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
         arr2d[2]
Out[30]: array([7, 8, 9])
```

Thus, individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements. So these are equivalent:

```
In [31]: arr2d[0][2]
Out[31]: 3

In [32]: arr2d[0, 2]
Out[32]: 3
```

Indexing with slices

Like one-dimensional objects such as Python lists, ndarrays can be sliced with the familiar syntax:

```
In [33]: arr
Out[33]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])

In [34]: arr[1:6]
Out[34]: array([ 1,  2,  3,  4, 64])
```

Consider the two-dimensional array from before, arr2d. Slicing this array is a bit different:

```
In [35]: arr2d
Out[35]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

In [36]: arr2d[:2]
Out[36]: array([[1, 2, 3],
                [4, 5, 6]])
```

As you can see, it has sliced along axis 0, the first axis. A slice, therefore, selects a range of elements along an axis. It can be helpful to read the expression arr2d[:2] as “select the first two rows of arr2d.”

You can pass multiple slices just like you can pass multiple indexes:

```
In [37]: arr2d[:2, 1:]
Out[37]: array([[2, 3],
                [5, 6]])
```

When slicing like this, you always obtain array views of the same number of dimensions. By mixing integer indexes and slices, you get lower dimensional slices. For example, we can select the second row but only the first two columns like so:

```
In [38]: arr2d[1, :2]
Out[38]: array([4, 5])
```

Similarly, I can select the third column but only the first two rows like so:

```
In [39]: arr2d[:2, 2]
Out[39]: array([3, 6])
```

Boolean Indexing

Let’s consider an example where we have some data in an array and an array of names with duplicates. I’m going to use here the randn function in numpy.random to generate some random normally distributed data:

```
In [40]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
         data = np.random.randn(7, 4)
         names
Out[40]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')

In [41]: data
Out[41]: array([[-0.02518197,  0.80943424,  1.73293903,  0.66082271],
                [-2.39248784,  0.29922496,  0.79790917, -1.13724858],
                [ 1.89797025, -0.30737821,  0.43630279,  0.48267583],
                [ 0.48042156,  0.3953928 ,  0.22438742, -0.59274482],
                [-0.35212748, -1.29892375, -2.05463521,  0.64971741],
                [ 1.77274107, -0.68446375, -0.09880592, -0.22073544],
                [-0.69395412, -0.52897539,  0.40634899, -1.59207859]])
```

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, compari- sons (such as ==) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

```
In [42]: names == 'Bob'
Out[42]: array([ True, False, False,  True, False, False, False])
```

This boolean array can be passed when indexing the array:

```
In [43]: data[names == 'Bob']
Out[43]: array([[-0.02518197,  0.80943424,  1.73293903,  0.66082271],
                [ 0.48042156,  0.3953928 ,  0.22438742, -0.59274482]])
```

The boolean array must be of the same length as the array axis it's indexing. You can even mix and match boolean arrays with slices or integers.

What's the difference between numpy.arange and the range python built-in function?

- **Python range function**
  Python built-in function 'range' takes only integers as arguments.
  range(0, 1, 0.2) #Error

```
In [44]: range(5)

Out[44]: range(0, 5)

In [45]: range(0,1,0.2)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-45-eb5a148d1b7e> in <module>
----> 1 range(0,1,0.2)

TypeError: 'float' object cannot be interpreted as an integer
```

- **Function: 'linspace':**
  Return evenly spaced numbers over a specified interval.

```
In [46]: a = np.linspace(0,1) #by default divides the interval into 50 equidistant points
         b = np.linspace(0, 1, num = 10)

In [47]: a

Out[47]: array([0.        , 0.02040816, 0.04081633, 0.06122449, 0.08163265,
                0.10204082, 0.12244898, 0.14285714, 0.16326531, 0.18367347,
                0.20408163, 0.2244898 , 0.24489796, 0.26530612, 0.28571429,
                0.30612245, 0.32653061, 0.34693878, 0.36734694, 0.3877551 ,
                0.40816327, 0.42857143, 0.44897959, 0.46938776, 0.48979592,
                0.51020408, 0.53061224, 0.55102041, 0.57142857, 0.59183673,
                0.6122449 , 0.63265306, 0.65306122, 0.67346939, 0.69387755,
                0.71428571, 0.73469388, 0.75510204, 0.7755102 , 0.79591837,
                0.81632653, 0.83673469, 0.85714286, 0.87755102, 0.89795918,
                0.91836735, 0.93877551, 0.95918367, 0.97959184, 1.        ])

In [48]: a.shape

Out[48]: (50,)

In [49]: b

Out[49]: array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
                0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
```

- **Function: 'logspace':**
  Return numbers spaced evenly on a log scale (by default in base 10).

```
In [50]: np.logspace(0,1)

Out[50]: array([ 1.        ,  1.04811313,  1.09854114,  1.1513954 ,  1.20679264,
                 1.26485522,  1.32571137,  1.38949549,  1.45634848,  1.52641797,
                 1.59985872,  1.67683294,  1.75751062,  1.84206997,  1.93069773,
                 2.02358965,  2.12095089,  2.22299648,  2.32995181,  2.44205309,
                 2.55954792,  2.6826958 ,  2.8117687 ,  2.9470517 ,  3.0888436 ,
                 3.23745754,  3.39322177,  3.55648031,  3.72759372,  3.90693994,
                 4.09491506,  4.29193426,  4.49843267,  4.71486636,  4.94171336,
                 5.17947468,  5.42867544,  5.68986603,  5.96362332,  6.25055193,
                 6.55128557,  6.86648845,  7.19685673,  7.54312006,  7.90604321,
                 8.28642773,  8.68511374,  9.10298178,  9.54095476, 10.        ])
```