NumPy Basics

for Data Analysis

By Armin Khayati

NumPy Basics:
Arrays and Vectorized Computation

- NumPy, short for Numerical Python, is one of the most important foundational packages for numerical computing in Python.

- Here are some of the things you'll find in NumPy at next slide.

NumPy Basics: Arrays and Vectorized Computation

- `ndarray`, an efficient multidimensional array providing fast array-oriented arithmetic operations and flexible broadcasting capabilities.
- Mathematical functions for fast operations on entire arrays of data without having to write loops.
- Tools for reading/writing array data to disk and working with memory-mapped files.
- Linear algebra, random number generation, and Fourier transform capabilities.
- A C API for connecting NumPy with libraries written in C, C++, or FORTRAN.

NumPy Basics: Arrays and Vectorized Computation

- While NumPy by itself does not provide modeling or scientific functionality, having an understanding of NumPy arrays and array-oriented computing will help you use tools with array-oriented semantics, like pandas, much more effectively.

- For most data analysis applications, the main areas of functionality we’ll focus on are in next slide

NumPy Basics: Arrays and Vectorized Computation

- Fast vectorized array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
- Common array algorithms like sorting, unique, and set operations
- Efficient descriptive statistics and aggregating/summarizing data
- Data alignment and relational data manipulations for merging and joining together heterogeneous datasets
- Expressing conditional logic as array expressions instead of loops with `if-elif-else` branches
- Group-wise data manipulations (aggregation, transformation, function application)

NumPy Basics: Arrays and Vectorized Computation

- There are a number of reasons for why NumPy is so important :
- 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.

NumPy Basics: Arrays and Vectorized Computation

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

```
In [7]: import numpy as np

In [8]: my_arr = np.arange(1000000)

In [9]: my_list = list(range(1000000))
```

```
In [10]: %time for _ in range(10): my_arr2 = my_arr * 2
CPU times: user 20 ms, sys: 50 ms, total: 70 ms
Wall time: 72.4 ms

In [11]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]
CPU times: user 760 ms, sys: 290 ms, total: 1.05 s
Wall time: 1.05 s
```

The NumPy ndarray: A Multidimensional Array Object

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

```
In [12]: import numpy as np

# Generate some random data
In [13]: data = np.random.randn(2, 3)

In [14]: data
Out[14]:
array([[-0.2047,  0.4789, -0.5194],
       [-0.5557,  1.9658,  1.3934]])
```

```
In [15]: data * 10
Out[15]:
array([[ -2.0471,   4.7894,  -5.1944],
       [ -5.5573,  19.6578,  13.9341]])

In [16]: data + data
Out[16]:

array([[-0.4094,  0.9579, -1.0389],
       [-1.1115,  3.9316,  2.7868]])
```

The NumPy ndarray: A Multidimensional Array Object

- n 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 [17]: data.shape
Out[17]: (2, 3)

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

The NumPy ndarray: 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.

```
In [19]: data1 = [6, 7.5, 8, 0, 1]

In [20]: arr1 = np.array(data1)

In [21]: arr1
Out[21]: array([ 6. ,  7.5,  8. ,  0. ,  1. ])
```

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [23]: arr2 = np.array(data2)

In [24]: arr2
Out[24]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

The NumPy ndarray: Creating ndarrays

- We can confirm `arr2` dimensions and shape by inspecting the `ndim` and `shape` attributes
- `np.array` tries to infer a good data type for the array that it creates.

```
In [25]: arr2.ndim
Out[25]: 2

In [26]: arr2.shape
Out[26]: (2, 4)
```

```
In [27]: arr1.dtype
Out[27]: dtype('float64')

In [28]: arr2.dtype
Out[28]: dtype('int64')
```

The NumPy ndarray: Creating ndarrays

- `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.

```
In [29]: np.zeros(10)
Out[29]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [30]: np.zeros((3, 6))
Out[30]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])
```

```
In [31]: np.empty((2, 3, 2))
Out[31]:
array([[[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]],

       [[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]]])
```

The NumPy ndarray: Creating ndarrays

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

```
In [32]: np.arange(15)
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

The NumPy ndarray: 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)

The NumPy ndarray: 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 [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [35]: arr1.dtype
Out[35]: dtype('float64')
In [36]: arr2.dtype
Out[36]: dtype('int32')
```

The NumPy ndarray: Data Types for ndarrays

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')

The NumPy ndarray: Data Types for ndarrays

- You can explicitly convert or cast an array from one `dtype` to another using ndarray's `astype` method. In this example, integers were cast to floating point.

```
In [37]: arr = np.array([1, 2, 3, 4, 5])

In [38]: arr.dtype
Out[38]: dtype('int64')

In [39]: float_arr = arr.astype(np.float64)

In [40]: float_arr.dtype
Out[40]: dtype('float64')
```

The NumPy ndarray: Data Types for ndarrays

- If I cast some floating-point numbers to be of integer `dtype`, the decimal part will be truncated.

```
In [41]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [42]: arr
Out[42]: array([  3.7,  -1.2,  -2.6,   0.5,  12.9,  10.1])

In [43]: arr.astype(np.int32)
Out[43]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

The NumPy ndarray: Data Types for ndarrays

- If you have an array of strings representing numbers, you can use `astype` to convert them to numeric form.

```
In [44]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)

In [45]: numeric_strings.astype(float)
Out[45]: array([  1.25,  -9.6 ,  42.  ])
```

The NumPy ndarray: 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 [51]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [52]: arr
Out[52]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [53]: arr * arr
Out[53]:
array([[  1.,   4.,   9.],
       [ 16.,  25.,  36.]])

In [54]: arr - arr
Out[54]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

The NumPy ndarray: Arithmetic with NumPy Arrays

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

```
In [55]: 1 / arr
Out[55]:
array([[ 1.    ,  0.5   ,  0.3333],
       [ 0.25  ,  0.2   ,  0.1667]])

In [56]: arr ** 0.5
Out[56]:
array([[ 1.    ,  1.4142,  1.7321],
       [ 2.    ,  2.2361,  2.4495]])
```

The NumPy ndarray: Arithmetic with NumPy Arrays

- Comparisons between arrays of the same size yield boolean arrays.
- Operations between differently sized arrays is called broadcasting and we won't discuss it here. You can search about it at home :)

```
In [57]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

In [58]: arr2
Out[58]:
array([[  0.,   4.,   1.],
       [  7.,   2.,  12.]])
```

- .

```
In [59]: arr2 > arr
Out[59]:
array([[False,  True, False],
       [ True, False,  True]], dtype=bool)
```

The NumPy ndarray: 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 [60]: arr = np.arange(10)

In [61]: arr
Out[61]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [62]: arr[5]
Out[62]: 5

In [63]: arr[5:8]
Out[63]: array([5, 6, 7])
```

- .

```
In [64]: arr[5:8] = 12

In [65]: arr
Out[65]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

The NumPy ndarray: Basic Indexing and Slicing

- 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.

The NumPy ndarray: Basic Indexing and Slicing

- To give an example of this, I first create a slice of `arr`.
- Then I change values in `arr_slice` , the mutations are reflected in the original array `arr`.

```
In [66]: arr_slice = arr[5:8]

In [67]: arr_slice
Out[67]: array([12, 12, 12])
```

```
In [68]: arr_slice[1] = 12345

In [69]: arr
Out[69]: array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
    9])
```

The NumPy ndarray: Basic Indexing and Slicing

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

```
In [70]: arr_slice[:] = 64

In [71]: arr
Out[71]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

- 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()`

The NumPy ndarray: Basic Indexing and Slicing

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

```
In [72]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [73]: arr2d[2]
Out[73]: array([7, 8, 9])
```

- You can pass a comma-separated list of indices to select individual elements. So these are equivalent:

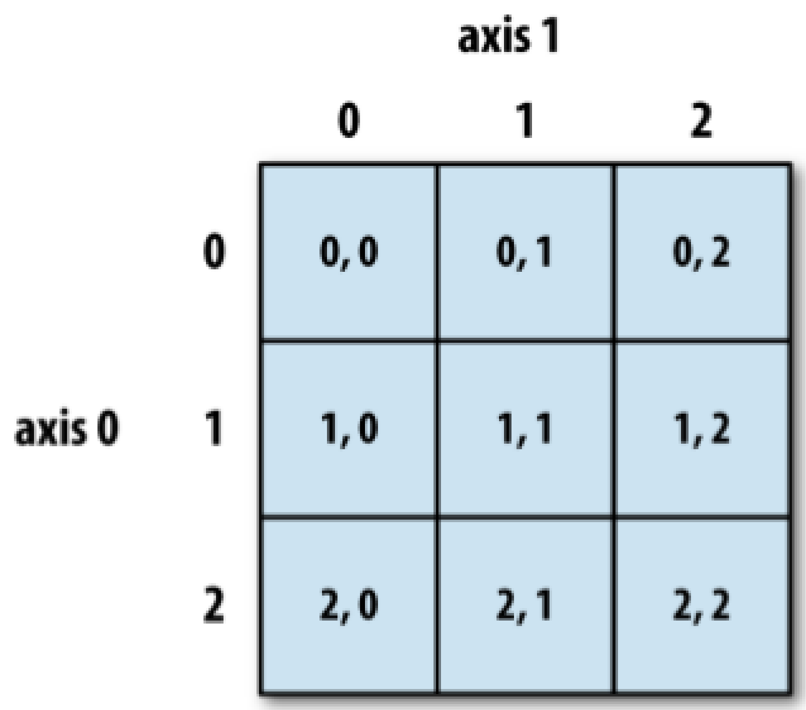
- .

```
In [74]: arr2d[0][2]
Out[74]: 3

In [75]: arr2d[0, 2]
Out[75]: 3
```

The NumPy ndarray: Basic Indexing and Slicing

- See figure bellow for an illustration of indexing on a two-dimensional array. I find it helpful to think of axis 0 as the “rows” of the array and axis 1 as the “columns.”



The NumPy ndarray: Basic Indexing and Slicing

- Example of a 3d (2 × 2 × 3) array:

```
In [76]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [77]: arr3d
Out[77]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

- `arr3d[0]` is a `2 × 3` array

```
In [78]: arr3d[0]
Out[78]:
array([[1, 2, 3],
       [4, 5, 6]])
```

The NumPy ndarray: Basic Indexing and Slicing

- Both scalar values and arrays can be assigned to `arr3d[0]`

```
In [79]: old_values = arr3d[0].copy()

In [80]: arr3d[0] = 42

In [81]: arr3d
Out[81]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])

In [82]: arr3d[0] = old_values

In [83]: arr3d
Out[83]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

The NumPy ndarray: Basic Indexing and Slicing

- Similarly, arr3d[1, 0] gives you all of the values whose indices start with (1, 0) , forming a 1-dimensional array:

```
In [84]: arr3d[1, 0]
Out[84]: array([7, 8, 9])
```

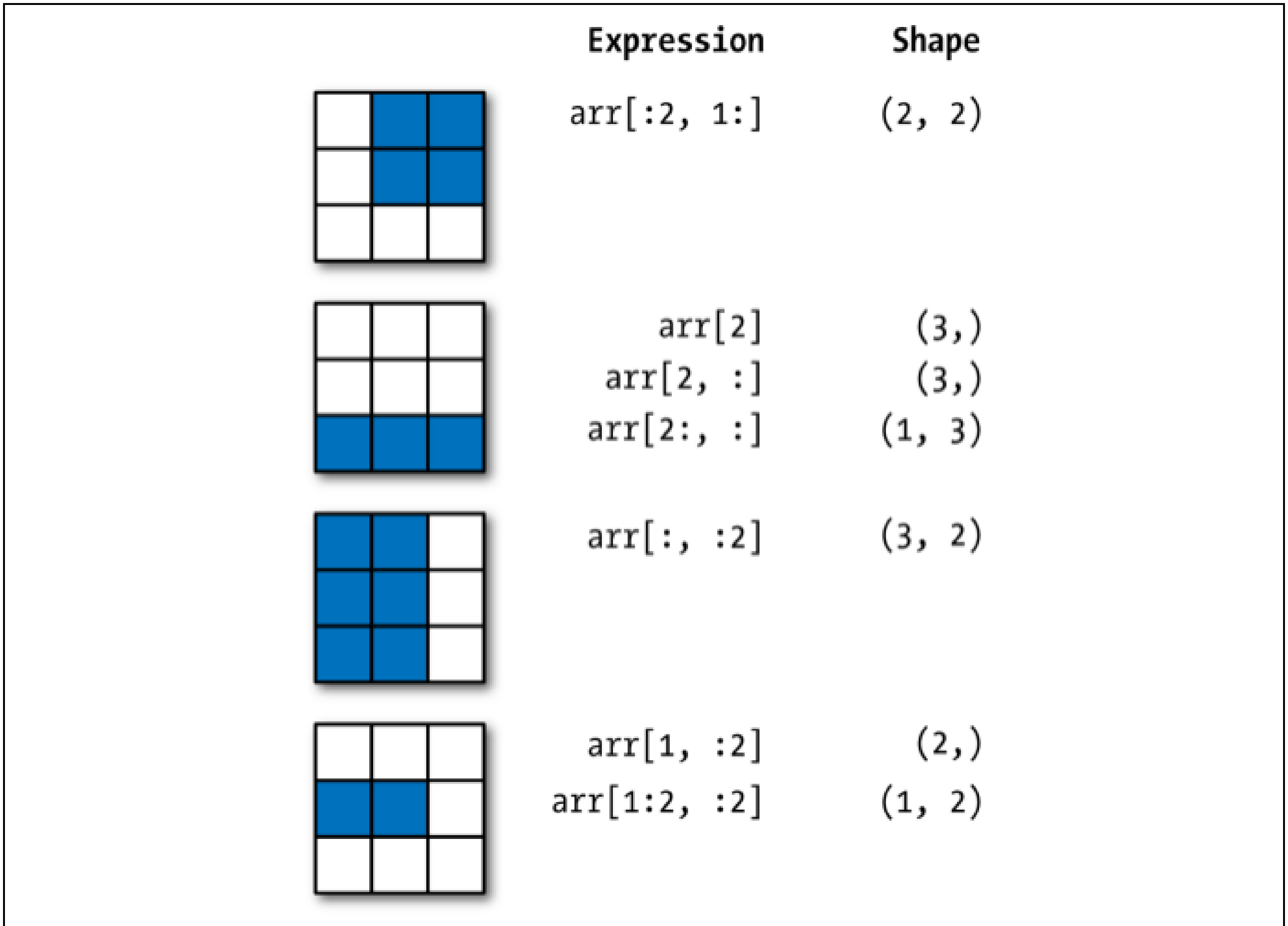
- This expression is the same as though we had indexed in two steps

```
In [85]: x = arr3d[1]

In [86]: x
Out[86]:
array([[ 7,  8,  9],
       [10, 11, 12]])

In [87]: x[0]
Out[87]: array([7, 8, 9])
```

The NumPy ndarray: Basic Indexing and Slicing



The NumPy ndarray: Boolean Indexing

- Let's consider an example where we have some data in an array and an array of names with duplicates.
- We're going to use here the `randn` function in `numpy.random` to generate some random normally distributed data.
- .

```
In [98]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [99]: data = np.random.randn(7, 4)

In [100]: names
Out[100]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
      dtype='<U4')

In [101]: data
Out[101]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

The NumPy ndarray: Boolean Indexing

- Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

```
In [102]: names == 'Bob'
Out[102]: array([ True, False, False,  True, False, False, False], dtype=bool)
```

- Suppose each `name` corresponds to a row in the `data` array and we wanted to select all the rows with corresponding name 'Bob' .This boolean array can be passed when indexing the array.

- .

```
In [103]: data[names == 'Bob']
Out[103]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])
```

The NumPy ndarray: Boolean Indexing

- In these examples, I select from the rows where `names == 'Bob'` and index the columns, too.

```
In [104]: data[names == 'Bob', 2:]
Out[104]:
array([[ 0.769 ,  1.2464],
       [-0.5397,  0.477 ]])

In [105]: data[names == 'Bob', 3]
Out[105]: array([ 1.2464,  0.477 ])
```

The NumPy ndarray: Boolean Indexing

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

```
In [107]: data[~(names == 'Bob')]
Out[107]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])

In [108]: cond = names == 'Bob'

In [109]: data[~cond]
Out[109]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

The NumPy ndarray: Boolean Indexing

- Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic operators like & (and) and | (or)
-

```
In [110]: mask = (names == 'Bob') | (names == 'Will')

In [111]: mask
Out[111]: array([ True, False,  True,  True,  True, False, False], dtype=bool)

In [112]: data[mask]
Out[112]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241]])
```

The NumPy ndarray: Boolean Indexing

- Setting values with boolean arrays works in a common-sense way. To set all of the negative values in data to 0 we need only do.
- Setting whole rows or columns using a one-dimensional boolean array is also easy.

```
In [115]: data[names != 'Joe'] = 7

In [116]: data
Out[116]:
array([[ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 1.0072,  0.    ,  0.275 ,  0.2289],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [ 0.    ,  0.    ,  0.    ,  0.    ]])
```

The NumPy ndarray: Fancy Indexing

- `Fancy indexing` is a term adopted by NumPy to describe indexing using integer arrays. Suppose we had an 8 × 4 array.
-

```
In [117]: arr = np.empty((8, 4))

In [118]: for i in range(8):
   .....:     arr[i] = i

In [119]: arr
Out[119]:
array([[ 0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.],
       [ 5.,  5.,  5.,  5.],
       [ 6.,  6.,  6.,  6.],
       [ 7.,  7.,  7.,  7.]])
```

The NumPy ndarray: Fancy Indexing

- 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.
-

```
In [120]: arr[[4, 3, 0, 6]]
Out[120]:
array([[ 4.,  4.,  4.,  4.],
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.],
       [ 6.,  6.,  6.,  6.]])
```

The NumPy ndarray: Fancy Indexing

- Passing multiple index arrays does something slightly different; it selects a one-dimensional array of elements corresponding to each tuple of indices.

```
In [122]: arr = np.arange(32).reshape((8, 4))

In [123]: arr
Out[123]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])

In [124]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[124]: array([ 4, 23, 29, 10])
```

The NumPy ndarray: Fancy Indexing

- Here the elements (1, 0), (5, 3), (7, 1) , and (2, 2) were selected. Regardless of how many dimensions the array has (here, only 2), the result of fancy indexing is always one-dimensional.

The NumPy ndarray: Transposing Arrays and Swapping Axes

- Transposing is a special form of reshaping that similarly returns a view on the underlying data without copying anything. Arrays have the `transpose` method and also the special `T` attribute.

```
In [126]: arr = np.arange(15).reshape((3, 5))

In [127]: arr
Out[127]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [128]: arr.T
Out[128]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

The NumPy ndarray: Transposing Arrays and Swapping Axes

- When doing matrix computations, you may do this very often—for example, when computing the inner matrix product using `np.dot`

```
In [130]: arr
Out[130]:
array([[-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329],
       [-2.3594, -0.1995, -1.542 ],
       [-0.9707, -1.307 ,  0.2863],
       [ 0.378 , -0.7539,  0.3313],
       [ 1.3497,  0.0699,  0.2467]])

In [131]: np.dot(arr.T, arr)
Out[131]:
array([[ 9.2291,  0.9394,  4.948 ],
       [ 0.9394,  3.7662, -1.3622],
       [ 4.948 , -1.3622,  4.3437]])
```

The NumPy ndarray: Transposing Arrays and Swapping Axes

- For higher dimensional arrays, transpose will accept a tuple of axis numbers to permute the axes (for extra mind bending)
- Here, the axes have been reordered with the second axis first, the first axis second, and the last axis unchanged.
- .

```
In [132]: arr = np.arange(16).reshape((2, 2, 4))

In [133]: arr
Out[133]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [134]: arr.transpose((1, 0, 2))
Out[134]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

The NumPy ndarray: Transposing Arrays and Swapping Axes

- Simple transposing with `.T` is a special case of swapping axes. `ndarray` has the method `swapaxes`, which takes a pair of axis numbers and switches the indicated axes to rear range the data.

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.

Universal Functions: Fast Element-Wise Array Functions

- Many `ufuncs` are simple element-wise transformations, like sqrt or exp. These are referred to as `unary ufuncs.`

```
In [137]: arr = np.arange(10)

In [138]: arr
Out[138]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [139]: np.sqrt(arr)
Out[139]:
array([ 0.    ,  1.    ,  1.4142,  1.7321,  2.    ,  2.2361,  2.4495,
        2.6458,  2.8284,  3.    ])

In [140]: np.exp(arr)
Out[140]:
array([    1.    ,     2.7183,     7.3891,    20.0855,    54.5982,
         148.4132,   403.4288,  1096.6332,  2980.958 ,  8103.0839])
```

Universal Functions: Fast Element-Wise Array Functions

- Others, such as add or maximum , take two arrays (thus, `binary ufuncs`) and return a single array as the result.

```
In [141]: x = np.random.randn(8)

In [142]: y = np.random.randn(8)

In [143]: x
Out[143]:
array([-0.0119,  1.0048,  1.3272, -0.9193, -1.5491,  0.0222,  0.7584,
       -0.6605])

In [144]: y
Out[144]:
array([ 0.8626, -0.01  ,  0.05  ,  0.6702,  0.853 , -0.9559, -0.0235,
       -2.3042])

In [145]: np.maximum(x, y)
Out[145]:
array([ 0.8626,  1.0048,  1.3272,  0.6702,  0.853 ,  0.0222,  0.7584,
       -0.6605])
```

Universal Functions: Fast Element-Wise Array Functions

- While not common, a `ufunc` can return multiple arrays. `modf` is one example, a vectorized version of the built-in Python `divmod`; it returns the fractional and integral parts of a floating-point array.

```
In [146]: arr = np.random.randn(7) * 5

In [147]: arr
Out[147]: array([-3.2623, -6.0915, -6.663 ,  5.3731,  3.6182,  3.45  ,  5.0077])

In [148]: remainder, whole_part = np.modf(arr)

In [149]: remainder
Out[149]: array([-0.2623, -0.0915, -0.663 ,  0.3731,  0.6182,  0.45  ,  0.0077])

In [150]: whole_part
Out[150]: array([-3., -6., -6.,  5.,  3.,  3.,  5.])
```

Universal Functions: Fast Element-Wise Array Functions

- `Ufuncs` accept an optional out argument that allows them to operate in-place on arrays.

```
In [151]: arr
Out[151]: array([-3.2623, -6.0915, -6.663 ,  5.3731,  3.6182,  3.45  ,  5.0077])

In [152]: np.sqrt(arr)
Out[152]: array([    nan,     nan,     nan,  2.318 ,  1.9022,  1.8574,  2.2378])

In [153]: np.sqrt(arr, arr)
Out[153]: array([    nan,     nan,     nan,  2.318 ,  1.9022,  1.8574,  2.2378])

In [154]: arr
Out[154]: array([    nan,     nan,     nan,  2.318 ,  1.9022,  1.8574,  2.2378])
```

Unary universal functions

Function	Description
abs, fabs	Compute the absolute value element-wise for integer, floating-point, or complex values
sqrt	Compute the square root of each element (equivalent to arr ** 0.5)
square	Compute the square of each element (equivalent to arr ** 2)
exp	Compute the exponent e^x of each element
log, log10, log2, log1p	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
sign	Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
ceil	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
floor	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
rint	Round elements to the nearest integer, preserving the dtype
modf	Return fractional and integral parts of array as a separate array
isnan	Return boolean array indicating whether each value is NaN (Not a Number)
isfinite, isinf	Return boolean array indicating whether each element is finite (non-inf, non-NaN) or infinite, respectively
cos, cosh, sin, sinh, tan, tanh	Regular and hyperbolic trigonometric functions
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Inverse trigonometric functions
logical_not	Compute truth value of not x element-wise (equivalent to ~arr).

Binary universal functions

Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Element-wise maximum; `fmax` ignores NaN
`minimum, fmin`	Element-wise minimum; `fmin` ignores NaN
`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument
`greater, greater_equal, less, less_equal, equal, not_equal`	Perform element-wise comparison, yielding boolean array (equivalent to infix operators `>, >=, <, <=, ==, !=`)
`logical_and, logical_or, logical_xor`	Compute element-wise truth value of logical operation (equivalent to infix operators `& |, ^`)

Array-Oriented Programming with Arrays

- 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 vectorization.

Array-Oriented Programming with Arrays

- As a simple example, suppose we wished to evaluate the function `sqrt(x^2 + y^2)` across a regular grid of values.
- The `np.meshgrid` function takes two 1D arrays and produces two 2D matrices corresponding to all pairs of `(x, y)` in the two arrays.
- .

```
In [156]: xs, ys = np.meshgrid(points, points)

In [157]: ys
Out[157]:
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

Array-Oriented Programming with Arrays

- Now, evaluating the function is a matter of writing the same expression you would write with two points:

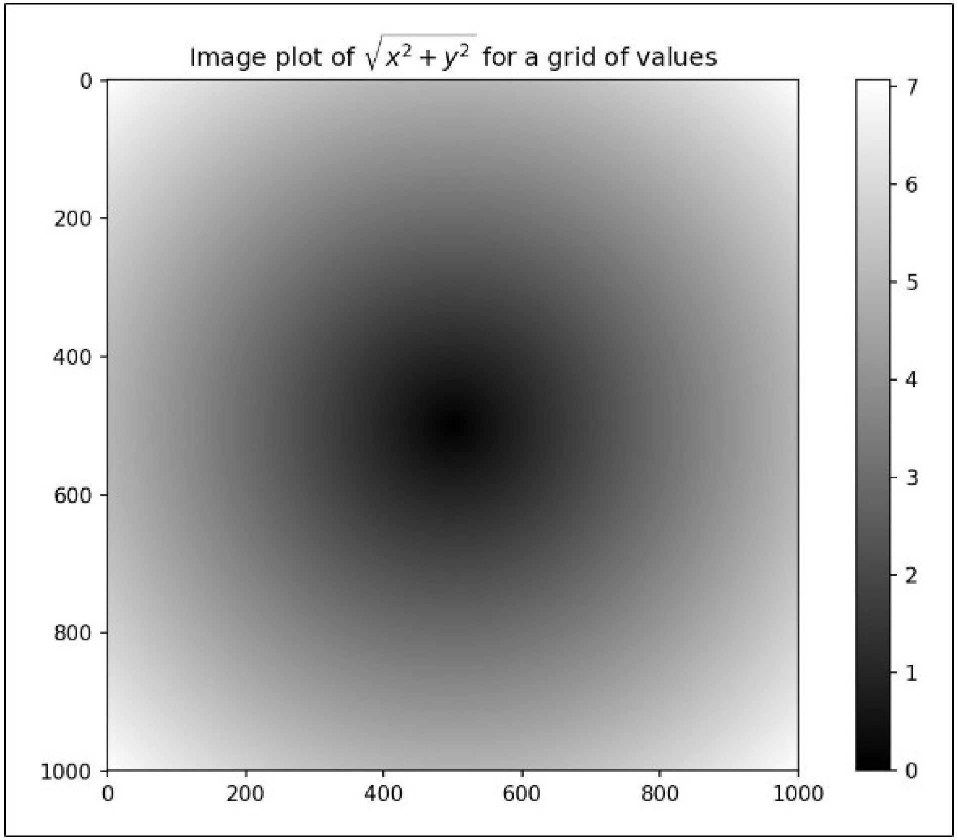
```
In [158]: z = np.sqrt(xs ** 2 + ys ** 2)

In [159]: z
Out[159]:
array([[ 7.0711,  7.064 ,  7.0569, ...,  7.0499,  7.0569,  7.064 ],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],
       ...,
       [ 7.0499,  7.0428,  7.0357, ...,  7.0286,  7.0357,  7.0428],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569]])
```

Array-Oriented Programming with Arrays

- As a preview , I use `matplotlib` to create visualizations of this two-dimensional array:

```
In [162]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")
Out[162]: <matplotlib.text.Text at 0x7f715d2de748>
```



Expressing Conditional Logic as Array Operations

- The `numpy.where` function is a vectorized version of the ternary expression `x if condition else y`. Suppose we had a boolean array and two arrays of values

```
In [165]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [166]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [167]: cond = np.array([True, False, True, True, False])
```

Expressing Conditional Logic as Array Operations

- Suppose we wanted to take a value from `xarr` whenever the corresponding value in `cond` is `True` , and otherwise take the value from `yarr` .
- A list comprehension doing this might look like:

```
In [168]: result = [(x if c else y)
   .....:           for x, y, c in zip(xarr, yarr, cond)]

In [169]: result
Out[169]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```

Expressing Conditional Logic as Array Operations

- 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 [170]: result = np.where(cond, xarr, yarr)

In [171]: result
Out[171]: array([ 1.1,  2.2,  1.3,  1.4,  2.5])
```

Expressing Conditional Logic as Array Operations

- 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.

Expressing Conditional Logic as Array Operations

- This is very easy to do with `np.where` :

```
In [172]: arr = np.random.randn(4, 4)

In [173]: arr
Out[173]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 0.2229,  0.0513, -1.1577,  0.8167],
       [ 0.4336,  1.0107,  1.8249, -0.9975],
       [ 0.8506, -0.1316,  0.9124,  0.1882]])

In [174]: arr > 0
Out[174]:
array([[False, False, False, False],
       [ True,  True, False,  True],
       [ True,  True,  True, False],
       [ True, False,  True,  True]], dtype=bool)

In [175]: np.where(arr > 0, 2, -2)
Out[175]:
array([[-2, -2, -2, -2],
       [ 2,  2, -2,  2], [ 2,  2,  2, -2],
                          [ 2, -2,  2,  2]])
```

Expressing Conditional Logic as Array Operations

- You can combine scalars and arrays when using `np.where`.
- For example, I can replace all positive values in `arr` with the constant 2 like so:

```
In [176]: np.where(arr > 0, 2, arr) # set only positive values to 2
Out[176]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 2.    ,  2.    , -1.1577,  2.    ],
       [ 2.    ,  2.    ,  2.    , -0.9975],
       [ 2.    , -0.1316,  2.    ,  2.    ]])
```

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.

Mathematical and Statistical Methods

- Here I generate some normally distributed random data and compute some aggregate statistics:

```
In [177]: arr = np.random.randn(5, 4)

In [178]: arr
Out[178]:
array([[ 2.1695, -0.1149,  2.0037,  0.0296],
       [ 0.7953,  0.1181, -0.7485,  0.585 ],
       [ 0.1527, -1.5657, -0.5625, -0.0327],
       [-0.929 , -0.4826, -0.0363,  1.0954],
       [ 0.9809, -0.5895,  1.5817, -0.5287]])

In [179]: arr.mean()
Out[179]: 0.19607051119998253

In [180]: np.mean(arr)
Out[180]: 0.19607051119998253

In [181]: arr.sum()
Out[181]: 3.9214102239996507
```

Mathematical and Statistical Methods

- 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 [182]: arr.mean(axis=1)
Out[182]: array([ 1.022 ,  0.1875, -0.502 , -0.0881,  0.3611])

In [183]: arr.sum(axis=0)
Out[183]: array([ 3.1693, -2.6345,  2.2381,  1.1486])
```

Mathematical and Statistical Methods

- Here, `arr.mean(1)` means “compute mean across the columns” where `arr.sum(0)` means “compute sum down the rows.”
- Other methods like `cumsum` and `cumprod` do not aggregate, instead producing an array of the intermediate results.

```
In [184]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

In [185]: arr.cumsum()
Out[185]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

Mathematical and Statistical Methods

- In multidimensional arrays, accumulation functions like `cumsum` return an array of the same size, but with the partial aggregates computed along the indicated axis according to each lower dimensional slice.

```
In [186]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [187]: arr
Out[187]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [188]: arr.cumsum(axis=0)
Out[188]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])
```

- .

```
In [189]: arr.cumprod(axis=1)
Out[189]:
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]])
```

Table of basic array statistical methods

Method	Description
`sum`	Sum of all the elements in the array or along an axis; zero-length arrays have sum 0
`mean`	Arithmetic mean; zero-length arrays have `NaN` mean
`std, var`	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator `n`)
`min, max`	Minimum and maximum
`argmin, argmax`	Indices of minimum and maximum elements, respectively
`cumsum`	Cumulative sum of elements starting from 0
`cumprod`	Cumulative product of elements starting from 1

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 [190]: arr = np.random.randn(100)

In [191]: (arr > 0).sum() # Number of positive values
Out[191]: 42
```

Methods for Boolean Arrays

- 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 [192]: bools = np.array([False, False, True, False])

In [193]: bools.any()
Out[193]: True

In [194]: bools.all()
Out[194]: False
```

Sorting

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

```
In [195]: arr = np.random.randn(6)

In [196]: arr
Out[196]: array([ 0.6095, -0.4938,  1.24  , -0.1357,  1.43  , -0.8469])

In [197]: arr.sort()

In [198]: arr
Out[198]: array([-0.8469, -0.4938, -0.1357,  0.6095,  1.24  ,  1.43  ])
```

Sorting

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

```
In [199]: arr = np.random.randn(5, 3)

In [200]: arr
Out[200]:
array([[ 0.6033,  1.2636, -0.2555],
       [-0.4457,  0.4684, -0.9616],
       [-1.8245,  0.6254,  1.0229],
       [ 1.1074,  0.0909, -0.3501],
       [ 0.218 , -0.8948, -1.7415]])

In [201]: arr.sort(1)

In [202]: arr
Out[202]:
array([[-0.2555,  0.6033,  1.2636],
       [-0.9616, -0.4457,  0.4684],
       [-1.8245,  0.6254,  1.0229],
       [-0.3501,  0.0909,  1.1074],
       [-1.7415, -0.8948,  0.218 ]])
```

Sorting

- The top-level method `np.sort` returns a sorted copy of an array instead of modifying the array in-place.
- A quick-and-dirty way to compute the quantiles of an array is to sort it and select the value at a particular rank:

```
In [203]: large_arr = np.random.randn(1000)

In [204]: large_arr.sort()

In [205]: large_arr[int(0.05 * len(large_arr))] # 5% quantile
Out[205]: -1.5311513550102103
```

Unique and Other Set Logic

- NumPy has some basic set operations for one-dimensional `ndarrays`.
- A commonly used one is `np.unique` , which returns the sorted unique values in an array:

```
In [206]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [207]: np.unique(names)
Out[207]:
array(['Bob', 'Joe', 'Will'],
      dtype='<U4')

In [208]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [209]: np.unique(ints)
Out[209]: array([1, 2, 3, 4])
```

Unique and Other Set Logic

- Another function, `np.in1d`, tests membership of the values in one array in another, returning a boolean array:

```
In [211]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [212]: np.in1d(values, [2, 3, 6])
Out[212]: array([ True, False, False,  True,  True, False,  True], dtype=bool)
```

Table of array set operations

Method	Description
`unique(x)`	Compute the sorted, unique elements in x
`intersect1d(x, y)`	Compute the sorted, common elements in x and y
`union1d(x, y)`	Compute the sorted union of elements
`in1d(x, y)`	Compute a boolean array indicating whether each element of x is contained in y
`setdiff1d(x, y)`	Set difference, elements in x that are not in y
`setxor1d(x, y)`	Set symmetric differences; elements that are in either of the arrays, but not both

File Input and Output with Arrays

- NumPy is able to save and load data to and from disk either in text or binary format.
- In this section I only discuss NumPy's built-in binary format, since most users will prefer pandas and other tools for loading text or tabular data.
- `np.save` and `np.load` are the two workhorse functions for efficiently saving and loading array data on disk.
- Arrays are saved by default in an uncompressed raw binary format with file extension `.npy`

```
In [215]: np.load('some_array.npy')
Out[215]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

File Input and Output with Arrays

- You save multiple arrays in an uncompressed archive using `np.savez` and passing the arrays as keyword arguments:

```
In [216]: np.savez('array_archive.npz', a=arr, b=arr)
```

- When loading an `.npz` file, you get back a dict-like object that loads the individual arrays lazily.

```
In [217]: arch = np.load('array_archive.npz')

In [218]: arch['b']
Out[218]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

File Input and Output with Arrays

- If your data compresses well, you may wish to use `numpy.savez_compressed` instead:

```
In [219]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```

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.

Linear Algebra

- Thus, there is a function `dot` , both an array method and a function in the numpy namespace, for matrix multiplication:

```
In [224]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [225]: x
Out[225]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [226]: y
Out[226]:
array([[  6.,  23.],
       [ -1.,   7.],
       [  8.,   9.]])

In [227]: x.dot(y)
Out[227]:
array([[  28.,   64.],
       [  67.,  181.]])
```

Linear Algebra

- `x.dot(y)` is equivalent to `np.dot(x, y)`:

```
In [228]: np.dot(x, y)
Out[228]:
array([[  28.,   64.],
       [  67.,  181.]])
```

- A matrix product between a two-dimensional array and a suitably sized one-dimensional array results in a one-dimensional array:

```
In [229]: np.dot(x, np.ones(3))
Out[229]: array([  6.,  15.])
```

Linear Algebra

- The `@` symbol (as of Python 3.5) also works as an infix operator that performs matrix multiplication:

```
In [230]: x @ np.ones(3)
Out[230]: array([  6.,  15.])
```

- `numpy.linalg` has a standard set of matrix decompositions and things like inverse and determinant.
- These are implemented under the hood via the same industry-standard linear algebra libraries used in other languages like MATLAB and R, such as BLAS, LAPACK, or possibly (depending on your NumPy build) the proprietary Intel MKL (Math Kernel Library)

Linear Algebra

```
In [231]: from numpy.linalg import inv, qr

In [232]: X = np.random.randn(5, 5)

In [233]: mat = X.T.dot(X)

In [234]: inv(mat)
Out[234]:
array([[  933.1189,   871.8258, -1417.6902, -1460.4005,  1782.1391],
       [  871.8258,   815.3929, -1325.9965, -1365.9242,  1666.9347],
       [-1417.6902, -1325.9965,  2158.4424,  2222.0191, -2711.6822],
       [-1460.4005, -1365.9242,  2222.0191,  2289.0575, -2793.422 ],
       [ 1782.1391,  1666.9347, -2711.6822, -2793.422 ,  3409.5128]])
```

Linear Algebra

```
In [235]: mat.dot(inv(mat))
Out[235]:
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 [236]: q, r = qr(mat)

In [237]: r
Out[237]:
array([[-1.6914,  4.38  ,  0.1757,  0.4075, -0.7838],
       [ 0.    , -2.6436,  0.1939, -3.072 , -1.0702],
       [ 0.    ,  0.    , -0.8138,  1.5414,  0.6155],
       [ 0.    ,  0.    ,  0.    , -2.6445, -2.1669],
       [ 0.    ,  0.    ,  0.    ,  0.    ,  0.0002]])
```

Table of Commonly used numpy.linalg functions

- The expression `X.T.dot(X)` computes the dot product of `X` with its transpose `X.T`.

Function	Description
`diag`	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
`dot`	Matrix multiplication
`trace`	Compute the sum of the diagonal elements
`det`	Compute the matrix determinant
`eig`	Compute the eigenvalues and eigenvectors of a square matrix
`inv`	Compute the inverse of a square matrix
`pinv`	Compute the Moore-Penrose pseudo-inverse of a matrix
`qr`	Compute the QR decomposition
`svd`	Compute the singular value decomposition (SVD)
`solve`	Solve the linear system Ax = b for x, where A is a square matrix
`lstsq`	Compute the least-squares solution to Ax = b

Pseudorandom Number Generation

- The numpy.random module supplements the built-in Python random with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions.
- For example, you can get a 4 × 4 array of samples from the standard normal distribution using normal :

```
In [238]: samples = np.random.normal(size=(4, 4))

In [239]: samples
Out[239]:
array([[ 0.5732,  0.1933,  0.4429,  1.2796],
       [ 0.575 ,  0.4339, -0.7658, -1.237 ],
       [-0.5367,  1.8545, -0.92  , -0.1082],
       [ 0.1525,  0.9435, -1.0953, -0.144 ]])
```

Pseudorandom Number Generation

- Python’s built-in random module, by contrast, only samples one value at a time.
- As you can see from this benchmark, `numpy.random` is well over an order of magnitude faster for generating very large samples:

```
In [240]: from random import normalvariate

In [241]: N = 1000000

In [242]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
1.77 s +- 126 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)

In [243]: %timeit np.random.normal(size=N)
61.7 ms +- 1.32 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Pseudorandom Number Generation

- We say that these are pseudorandom numbers because they are generated by an algorithm with deterministic behavior based on the seed of the random number generator.
- You can change NumPy's random number generation seed using `np.random.seed`:

```
In [244]: np.random.seed(1234)
```

Pseudorandom Number Generation

- The data generation functions in numpy.random use a global random seed.
- To avoid global state, you can use `numpy.random.RandomState` to create a random number generator isolated from others:

```
In [245]: rng = np.random.RandomState(1234)

In [246]: rng.randn(10)
Out[246]:
array([ 0.4714, -1.191 ,  1.4327, -0.3127, -0.7206,  0.8872,  0.8596,
       -0.6365,  0.0157, -2.2427])
```

Table of Partial list of numpy.random functions

Function	Description
seed	Seed the random number generator
permutation	Return a random permutation of a sequence, or return a permuted range
shuffle	Randomly permute a sequence in-place
rand	Draw samples from a uniform distribution
randint	Draw random integers from a given low-to-high range
randn	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
binomial	Draw samples from a binomial distribution
normal	Draw samples from a normal (Gaussian) distribution
beta	Draw samples from a beta distribution
chisquare	Draw samples from a chi-square distribution
gamma	Draw samples from a gamma distribution
uniform	Draw samples from a uniform [0, 1) distribution