

Numpy

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import statsmodels as sm
```

- 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.
-
- 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-elifelse branches
 - Expressing conditional logic as array expressions instead of loops with if-elifelse branches
 - Group-wise data manipulations (aggregation, transformation, function application)

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.

```
In [3]: my_arr = np.arange(1000000)
```

```
In [4]: my_list = list(range(1000000))
```

```
In [5]: %time for _ in range(10): my_arr2 = my_arr * 2
```

Wall time: 15.6 ms

```
In [6]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]
```

Wall time: 858 ms

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

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 [7]: data = np.random.randn(2, 3)
```

```
In [8]: data
```

```
Out[8]: array([[ -1.66213272, -1.1580821 ,  0.03110487],
               [-1.81332257,  0.76870567, -2.26895165]])
```

```
In [9]: data * 10
```

```
Out[9]: array([[ -16.62132724, -11.58082096,  0.31104871],
               [-18.13322573,  7.68705675, -22.68951646]])
```

```
In [10]: data + data
```

```
Out[10]: array([[ -3.32426545, -2.31616419,  0.06220974],
                [-3.62664515,  1.53741135, -4.53790329]])
```

An ndarray is a generic multidimensional container for **homogeneous** data.

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 [11]: data.shape
```

```
Out[11]: (2, 3)
```

```
In [12]: data.dtype
```

```
Out[12]: dtype('float64')
```

Creating ndarrays

The easiest way to create an array is to use the **array** function (accepts any sequence-like object):

```
In [13]: data1 = [6, 7.5, 8, 0, 1]
```

```
In [14]: arr1 = np.array(data1)
```

```
In [15]: arr1
```

```
Out[15]: array([6. , 7.5, 8. , 0. , 1. ])
```

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

```
In [16]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
```

```
In [17]: arr2 = np.array(data2)
```

```
In [18]: arr2
```

```
Out[18]: array([[1, 2, 3, 4],
               [5, 6, 7, 8]])
```

```
In [19]: arr2.ndim
```

```
Out[19]: 2
```

```
In [20]: arr2.shape
```

```
Out[20]: (2, 4)
```

Unless explicitly specified, `np.array` tries to infer a good data type for the array that it creates:

```
In [21]: arr1.dtype
```

```
Out[21]: dtype('float64')
```

```
In [22]: arr2.dtype
```

```
Out[22]: dtype('int32')
```

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 [23]: np.zeros(10)
```

```
Out[23]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
In [24]: np.zeros((3, 6))
```

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

```
In [25]: np.empty((2, 3, 2))
```

```
Out[25]: array([[ 9.1e-322,  0.0e+000],
               [ 0.0e+000,  0.0e+000],
               [ 0.0e+000,  0.0e+000]],
               [[ 0.0e+000,  0.0e+000],
               [ 0.0e+000,  0.0e+000],
               [ 0.0e+000,  0.0e+000]])
```

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

```
In [26]: np.arange(15)
```

```
Out[26]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Array creation functions

Function	Description
<code>array</code>	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
<code>asarray</code>	Convert input to ndarray, but do not copy if the input is already an ndarray
<code>arange</code>	Like the built-in <code>range</code> but returns an ndarray instead of a list
<code>ones</code> , <code>ones_like</code>	Produce an array of all 1s with the given shape and dtype; <code>ones_like</code> takes another array and produces a ones array of the same shape and dtype
<code>zeros</code> , <code>zeros_like</code>	Like <code>ones</code> and <code>ones_like</code> but producing arrays of 0s instead
<code>empty</code> , <code>empty_like</code>	Create new arrays by allocating new memory, but do not populate with any values like <code>ones</code> and <code>zeros</code>
<code>full</code> , <code>full_like</code>	Produce an array of the given shape and dtype with all values set to the indicated "fill value" <code>full_like</code> takes another array and produces a filled array of the same shape and dtype
<code>eye</code> , <code>identity</code>	Create a square $N \times N$ identity matrix (1s on the diagonal and 0s elsewhere)

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 [27]: arr1 = np.array([1, 2, 3], dtype=np.float64)
```

```
In [28]: arr2 = np.array([1, 2, 3], dtype=np.int32)
```

```
In [29]: arr1.dtype
```

```
Out[29]: dtype('float64')
```

```
In [30]: arr2.dtype
```

```
Out[30]: dtype('int32')
```

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.

NumPy data types

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')

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

```
In [31]: arr = np.array([1, 2, 3, 4, 5])
```

```
In [32]: arr.dtype
```

```
Out[32]: dtype('int32')
```

```
In [33]: float_arr = arr.astype(np.float64)
```

```
In [34]: float_arr.dtype
```

```
Out[34]: dtype('float64')
```

```
In [35]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
```

```
In [36]: arr
```

```
Out[36]: array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])
```

```
In [37]: arr.astype(np.int32)
```

```
Out[37]: array([ 3, -1, -2,  0, 12, 10])
```

```
In [38]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
```

```
In [39]: numeric_strings.astype(float)
```

```
Out[39]: array([ 1.25, -9.6 , 42.  ])
```

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.

You can also use another array's dtype attribute:

```
In [40]: int_array = np.arange(10)
```

```
In [41]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
```

```
In [42]: int_array.astype(calibers.dtype)
```

```
Out[42]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

There are shorthand type code strings you can also use to refer to a dtype:

```
In [43]: empty_uint32 = np.empty(8, dtype='u4')
```

Calling `astype` always creates a new array (a copy of the data), even if the new dtype is the same as the old dtype.

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

```
In [45]: arr
```

```
Out[45]: array([[1., 2., 3.],
               [4., 5., 6.]])
```

```
In [46]: arr*arr
```

```
Out[46]: array([[ 1.,  4.,  9.],
               [16., 25., 36.]])
```

```
In [47]: arr - arr
```

```
Out[47]: array([[0., 0., 0.],
               [0., 0., 0.]])
```

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

```
In [48]: 1 / arr
```

```
Out[48]: array([[1.          , 0.5          , 0.33333333],
                [0.25         , 0.2          , 0.16666667]])
```

```
In [49]: arr ** 0.5
```

```
Out[49]: array([[1.          , 1.41421356, 1.73205081],
                [2.          , 2.23606798, 2.44948974]])
```

Comparisons between arrays of the same size yield boolean arrays:

```
In [50]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
```

```
In [51]: arr2 > arr
```

```
Out[51]: array([[False,  True, False],
                [ True, False,  True]])
```

Basic Indexing and Slicing

One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
In [52]: arr = np.arange(10)
```

```
In [53]: arr
```

```
Out[53]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [54]: arr[5]
```

```
Out[54]: 5
```

```
In [55]: arr[5:8]
```

```
Out[55]: array([5, 6, 7])
```

```
In [56]: arr[5:8] = 12
```

```
In [57]: arr
```

```
Out[57]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

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.

```
In [58]: arr_slice = arr[5:8]
```

```
In [59]: arr_slice
```

```
Out[59]: array([12, 12, 12])
```

```
In [60]: arr_slice[1] = 12345
```

```
In [61]: arr
```

```
Out[61]: array([ 0,  1,  2,  3,  4, 12, 12345, 12,  8,
                9])
```

The “bare” slice `[:]` will assign to all values in an array:

```
In [62]: arr_slice[:] = 64
```

```
In [63]: arr
```

```
Out[63]: 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:

```
In [64]: arr_slice = arr[5:8].copy()
```

```
In [65]: arr_slice
```

```
Out[65]: array([64, 64, 64])
```

```
In [66]: arr_slice[1] = 12345
```

```
In [67]: arr
```

```
Out[67]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

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

```
In [68]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
In [69]: arr2d[2]
```

```
Out[69]: array([7, 8, 9])
```

You can pass a comma-separated list of indices to select individual elements:

```
In [70]: arr2d[0][2]
```

```
Out[70]: 3
```

```
In [71]: arr2d[0, 2]
```

```
Out[71]: 3
```


		axis 1		
		0	1	2
axis 0	0	0,0	0,1	0,2
	1	1,0	1,1	1,2
	2	2,0	2,1	2,2

Indexing elements in a NumPy array

In multidimensional arrays, if you omit later indices, the returned object will be an lower dimensional ndarray consisting of *all the data along the higher dimensions*:

```
In [72]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

```
In [73]: arr3d.shape
```

```
Out[73]: (2, 2, 3)
```

```
In [74]: arr3d
```

```
Out[74]: array([[[ 1,  2,  3],
                  [ 4,  5,  6]],
                [[ 7,  8,  9],
                  [10, 11, 12]]])
```

`arr3d[0]` is a 2×3 array:

```
In [75]: arr3d[0]
```

```
Out[75]: array([[1, 2, 3],
                [4, 5, 6]])
```

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

```
In [76]: old_values = arr3d[0].copy()
```

```
In [77]: arr3d[0] = 42
```

```
In [78]: arr3d
```

```
Out[78]: array([[[42, 42, 42],
                 [42, 42, 42]],

                [[ 7,  8,  9],
                 [10, 11, 12]]])
```

```
In [79]: arr3d[0] = old_values
```

```
In [80]: arr3d
```

```
Out[80]: array([[[ 1,  2,  3],
                 [ 4,  5,  6]],

                [[ 7,  8,  9],
                 [10, 11, 12]]])
```

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

```
In [81]: arr3d[1, 0]
```

```
Out[81]: array([7, 8, 9])
```

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

```
In [82]: x = arr3d[1]
```

```
In [83]: x
```

```
Out[83]: array([[ 7,  8,  9],
                 [10, 11, 12]])
```

```
In [84]: x[0]
```

```
Out[84]: array([7, 8, 9])
```

Indexing with slices

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

```
In [85]: arr
```

```
Out[85]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

```
In [86]: arr[1:6]
```

```
Out[86]: array([ 1,  2,  3,  4, 64])
```

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

```
In [90]: arr2d
```

```
Out[90]: array([[1, 2, 3],
               [4, 5, 6],
               [7, 8, 9]])
```

```
In [91]: arr2d[:2]
```

```
Out[91]: array([[1, 2, 3],
               [4, 5, 6]])
```

```
In [92]: arr2d[:2, 1:]
```

```
Out[92]: 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:

```
In [93]: arr2d[1, :2]
```

```
Out[93]: array([4, 5])
```

```
In [94]: arr2d[:2, 2]
```

```
Out[94]: array([3, 6])
```

A colon by itself means to take the entire axis, so you can slice only higher dimensional axes by doing:

```
In [95]: arr2d[:, :1]
```

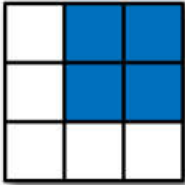
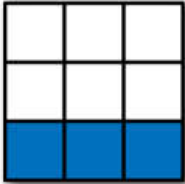
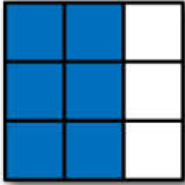
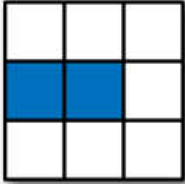
```
Out[95]: array([[1],
               [4],
               [7]])
```

Assigning to a slice expression assigns to the whole selection:

```
In [96]: arr2d[:2, 1:] = 0
```

```
In [97]: arr2d
```

```
Out[97]: array([[1, 0, 0],
               [4, 0, 0],
               [7, 8, 9]])
```

	Expression	Shape
	<code>arr[:2, 1:]</code>	<code>(2, 2)</code>
	<code>arr[2]</code>	<code>(3,)</code>
	<code>arr[2, :]</code>	<code>(3,)</code>
	<code>arr[2:, :]</code>	<code>(1, 3)</code>
	<code>arr[:, :2]</code>	<code>(3, 2)</code>
	<code>arr[1, :2]</code>	<code>(2,)</code>
	<code>arr[1:2, :2]</code>	<code>(1, 2)</code>

Two-dimensional array slicing

Boolean Indexing

```
In [120]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
```

```
In [121]: data = np.random.randn(7, 4)
```

```
In [122]: names
```

```
Out[122]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

```
In [123]: data
```

```
Out[123]: array([[ -0.73498549,  0.037139 , -1.08964642,  0.61184086],
 [ 0.96054725, -0.31456444,  1.37568801,  0.05908302],
 [ 0.3144823 , -1.29994292,  0.93648728,  0.95195609],
 [ 0.47781265,  1.63699929, -1.17075395, -0.11915952],
 [-0.40114462,  0.49274672, -1.44571477, -0.57809474],
 [ 0.04140382, -1.31601431,  1.84395403, -0.44220063],
 [-1.63392114, -0.66836314,  0.67528124, -0.0061356 ]])
```

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

```
In [124]: names == 'Bob'
Out[124]: array([ True, False, False,  True, False, False, False])

In [125]: data[names == 'Bob']
Out[125]: array([[ -0.73498549,  0.037139 , -1.08964642,  0.61184086],
                 [ 0.47781265,  1.63699929, -1.17075395, -0.11915952]])
```

The boolean array must be of the same length as the array axis it's indexing.

```
In [126]: data[names == 'Bob', 2:]
Out[126]: array([[ -1.08964642,  0.61184086],
                 [-1.17075395, -0.11915952]])

In [127]: data[names == 'Bob', 3]
Out[127]: array([ 0.61184086, -0.11915952])
```

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

```
In [128]: names != 'Bob'
Out[128]: array([False,  True,  True, False,  True,  True,  True])

In [129]: data[~(names == 'Bob')]
Out[129]: array([[ 0.96054725, -0.31456444,  1.37568801,  0.05908302],
                 [ 0.3144823 , -1.29994292,  0.93648728,  0.95195609],
                 [-0.40114462,  0.49274672, -1.44571477, -0.57809474],
                 [ 0.04140382, -1.31601431,  1.84395403, -0.44220063],
                 [-1.63392114, -0.66836314,  0.67528124, -0.0061356 ]])
```

The `~` operator can be useful when you want to invert a general condition:

```
In [130]: cond = names == 'Bob'

In [131]: data[~cond]
Out[131]: array([[ 0.96054725, -0.31456444,  1.37568801,  0.05908302],
                 [ 0.3144823 , -1.29994292,  0.93648728,  0.95195609],
                 [-0.40114462,  0.49274672, -1.44571477, -0.57809474],
                 [ 0.04140382, -1.31601431,  1.84395403, -0.44220063],
                 [-1.63392114, -0.66836314,  0.67528124, -0.0061356 ]])
```

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

```
In [132]: mask = (names == 'Bob') | (names == 'Will')
```

```
In [133]: mask
```

```
Out[133]: array([ True, False,  True,  True,  True, False, False])
```

```
In [134]: data[mask]
```

```
Out[134]: array([[ -0.73498549,  0.037139 , -1.08964642,  0.61184086],
 [ 0.3144823 , -1.29994292,  0.93648728,  0.95195609],
 [ 0.47781265,  1.63699929, -1.17075395, -0.11915952],
 [-0.40114462,  0.49274672, -1.44571477, -0.57809474]])
```

Selecting data from an array by boolean indexing always creates a copy of the data, even if the returned array is unchanged.

The Python keywords **and** and **or** do not work with boolean arrays. Use **&** (and) and **|** (or) instead.

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:

```
In [135]: data
```

```
Out[135]: array([[ -0.73498549,  0.037139 , -1.08964642,  0.61184086],
 [ 0.96054725, -0.31456444,  1.37568801,  0.05908302],
 [ 0.3144823 , -1.29994292,  0.93648728,  0.95195609],
 [ 0.47781265,  1.63699929, -1.17075395, -0.11915952],
 [-0.40114462,  0.49274672, -1.44571477, -0.57809474],
 [ 0.04140382, -1.31601431,  1.84395403, -0.44220063],
 [-1.63392114, -0.66836314,  0.67528124, -0.0061356 ]])
```

```
In [136]: data[data < 0] = 0
```

```
In [137]: data
```

```
Out[137]: array([[0.          , 0.037139 , 0.          , 0.61184086],
 [0.96054725, 0.          , 1.37568801, 0.05908302],
 [0.3144823 , 0.          , 0.93648728, 0.95195609],
 [0.47781265, 1.63699929, 0.          , 0.          ],
 [0.          , 0.49274672, 0.          , 0.          ],
 [0.04140382, 0.          , 1.84395403, 0.          ],
 [0.          , 0.          , 0.67528124, 0.          ]])
```

```
In [138]: data < 0
```

```
Out[138]: 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]])
```

Setting whole rows or columns using a one-dimensional boolean array is also easy:

```
In [139]: data[names != 'Joe'] = 7
```

```
In [140]: data
Out[140]: array([[7.          , 7.          , 7.          , 7.          ],
                 [0.96054725, 0.          , 1.37568801, 0.05908302],
                 [7.          , 7.          , 7.          , 7.          ],
                 [7.          , 7.          , 7.          , 7.          ],
                 [7.          , 7.          , 7.          , 7.          ],
                 [0.04140382, 0.          , 1.84395403, 0.          ],
                 [0.          , 0.          , 0.67528124, 0.          ]])
```

Fancy Indexing

```
In [144]: arr = np.empty((8, 4))
```

```
In [145]: for i in range(8):
          arr[i] = i
```

```
In [146]: arr
```

```
Out[146]: 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.]])
```

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 [147]: arr[[4, 3, 0, 6]]
```

```
Out[147]: array([[4., 4., 4., 4.],
                 [3., 3., 3., 3.],
                 [0., 0., 0., 0.],
                 [6., 6., 6., 6.]])
```

Using negative indices selects rows from the end:

```
In [148]: arr[[-3, -5, -7]]
```

```
Out[148]: array([[5., 5., 5., 5.],
                 [3., 3., 3., 3.],
                 [1., 1., 1., 1.]])
```

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

```
In [149]: arr = np.arange(32).reshape((8, 4))
```

```
In [150]: arr
Out[150]: 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 [151]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[151]: array([ 4, 23, 29, 10])
```

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

```
In [153]: arr
Out[153]: array([[ 0,  1,  2,  3,  4],
                 [ 5,  6,  7,  8,  9],
                 [10, 11, 12, 13, 14]])
```

```
In [154]: arr.T
Out[154]: array([[ 0,  5, 10],
                 [ 1,  6, 11],
                 [ 2,  7, 12],
                 [ 3,  8, 13],
                 [ 4,  9, 14]])
```

```
In [155]: np.dot(arr.T, arr)
Out[155]: array([[125, 140, 155, 170, 185],
                 [140, 158, 176, 194, 212],
                 [155, 176, 197, 218, 239],
                 [170, 194, 218, 242, 266],
                 [185, 212, 239, 266, 293]])
```

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 rearrange the data:

```
In [186]: arr = np.arange(16).reshape((2, 2, 4))
```

```
In [188]: arr
Out[188]: array([[[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7]],
                 [[ 8,  9, 10, 11],
                  [12, 13, 14, 15]])
```



```
In [193]: swapped = arr.swapaxes(1, 2)
```

```
In [194]: swapped.shape
```

```
Out[194]: (2, 4, 2)
```

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. Many ufuncs are simple element-wise transformations, like **sqrt** or **exp**:

```
In [195]: arr = np.arange(10)
```

```
In [196]: np.sqrt(arr)
```

```
Out[196]: array([0.          , 1.          , 1.41421356, 1.73205081, 2.          ,
                2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.          ])
```

These are referred to as *unary ufuncs*. Others, such as **add** or **maximum*, take two arrays (thus, binary ufuncs) and return a single array as the result:

```
In [197]: x = np.random.randn(8)
```

```
In [198]: y = np.random.randn(8)
```

```
In [199]: x
```

```
Out[199]: array([-0.65191932,  0.76130351, -0.5241499 , -0.23818609, -0.21331952,
                0.23886189,  0.580115   ,  2.36126907])
```

```
In [200]: y
```

```
Out[200]: array([-0.87983114, -0.27343768,  1.06693021,  1.05339703, -1.03328155,
                1.4845363 , -0.9244597 , -2.9936107 ])
```

```
In [201]: np.maximum(x, y)
```

```
Out[201]: array([-0.65191932,  0.76130351,  1.06693021,  1.05339703, -0.21331952,
                1.4845363 ,  0.580115   ,  2.36126907])
```

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

```
In [203]: arr
```

```
Out[203]: array([ 3.53572967,  3.61914284,  0.22757453, -2.55843312, -5.46982579,
                -4.40385381, -4.25154546])
```

```
In [204]: remainder, whole_part = np.modf(arr)
```

```
In [205]: remainder
```

```
Out[205]: array([ 0.53572967,  0.61914284,  0.22757453, -0.55843312, -0.46982579,  
                -0.40385381, -0.25154546])
```

```
In [206]: whole_part
```

```
Out[206]: array([ 3.,  3.,  0., -2., -5., -4., -4.])
```

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

```
In [207]: arr
```

```
Out[207]: array([ 3.53572967,  3.61914284,  0.22757453, -2.55843312, -5.46982579,  
                -4.40385381, -4.25154546])
```

```
In [209]: np.exp(arr)
```

```
Out[209]: array([3.43200480e+01, 3.73055774e+01, 1.25555101e+00, 7.74259624e-02,  
                4.21196585e-03, 1.22301164e-02, 1.42422061e-02])
```

```
In [210]: np.exp(arr, arr)
```

```
Out[210]: array([3.43200480e+01, 3.73055774e+01, 1.25555101e+00, 7.74259624e-02,  
                4.21196585e-03, 1.22301164e-02, 1.42422061e-02])
```

```
In [211]: arr
```

```
Out[211]: array([3.43200480e+01, 3.73055774e+01, 1.25555101e+00, 7.74259624e-02,  
                4.21196585e-03, 1.22301164e-02, 1.42422061e-02])
```

Unary ufuncs

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

Binary universal functions

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

Array-Oriented Programming with Arrays

The practice of replacing explicit loops with array expressions is commonly referred to as vectorization. In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents.

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 [212]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced points
```

```
In [213]: xs, ys = np.meshgrid(points, points)
```

```
In [214]: ys
```

```
Out[214]: 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]])
```

```
In [215]: xs
```

```
Out[215]: array([[ -5.   , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
                [ -5.   , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
                [ -5.   , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
                ...,
                [ -5.   , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
                [ -5.   , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
                [ -5.   , -4.99, -4.98, ...,  4.97,  4.98,  4.99]])
```

```
In [216]: xs.shape
```

```
Out[216]: (1000, 1000)
```

```
In [217]: z = np.sqrt(xs ** 2 + ys ** 2)
```

```
In [218]: z.min()
```

```
Out[218]: 1.507288760336424e-13
```

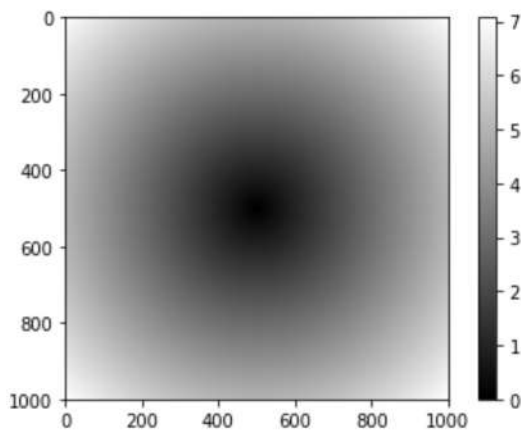
```
In [219]: z.max()
```

```
Out[219]: 7.0710678118654755
```

```
In [220]: import matplotlib.pyplot as plt
```

```
In [221]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
```

```
Out[221]: <matplotlib.colorbar.Colorbar at 0xdd07b00>
```



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 [222]: 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])
```

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

```
In [224]: result
```

```
Out[224]: [1.1, 2.2, 1.3, 1.4, 2.5]
```

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

```
In [226]: result
```

```
Out[226]: array([1.1, 2.2, 1.3, 1.4, 2.5])
```

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

```
In [228]: arr
```

```
Out[228]: array([[ -0.19965611,  0.59806866,  0.24891245, -0.53003084],
 [ 0.14185762, -0.54627093,  0.20304678,  0.60120412],
 [ 0.77438255,  1.74557866,  0.42505904,  1.93042427],
 [-0.86644984, -2.03547748, -1.05679164, -0.62683715]])
```

```
In [229]: arr > 0
```

```
Out[229]: array([[False,  True,  True, False],
 [ True, False,  True,  True],
 [ True,  True,  True,  True],
 [False, False, False, False]])
```

```
In [230]: np.where(arr > 0, 2, -2)
```

```
Out[230]: array([[ -2,  2,  2, -2],
 [ 2, -2,  2,  2],
 [ 2,  2,  2,  2],
 [-2, -2, -2, -2]])
```

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 [231]: np.where(arr > 0, 2, arr) # set only positive values to 2
```

```
Out[231]: array([[ -0.19965611,  2.,          2.,          -0.53003084],
 [ 2.,          -0.54627093,  2.,          0.60120412],
 [ 2.,          2.,          2.,          1.93042427],
 [-0.86644984, -2.03547748, -1.05679164, -0.62683715]])
```

Intuitively, *np.where* is like asking "tell me where in this array, entries satisfy a given condition":

```
In [240]: a = np.arange(5,10)
          np.where(a < 8) # Indices

Out[240]: (array([0, 1, 2], dtype=int64),)
```

It can also be used to get entries in array that satisfy the condition:

```
In [241]: a[np.where(a < 8)]

Out[241]: array([5, 6, 7])
```

When `a` is a 2d array, `np.where()` returns an array of row idx's, and an array of col idx's:

```
In [242]: arr

Out[242]: array([[ -0.19965611,  0.59806866,  0.24891245, -0.53003084],
                 [ 0.14185762, -0.54627093,  0.20304678,  0.60120412],
                 [ 0.77438255,  1.74557866,  0.42505904,  1.93042427],
                 [-0.86644984, -2.03547748, -1.05679164, -0.62683715]])

In [243]: np.where(arr > 0)

Out[243]: (array([0, 0, 1, 1, 1, 2, 2, 2, 2], dtype=int64),
          array([1, 2, 0, 2, 3, 0, 1, 2, 3], dtype=int64))
```

So that as in the 1d case, we can use `np.where()` to get entries in the 2d array that satisfy the condition

```
In [244]: arr[np.where(arr > 0)]

Out[244]: array([0.59806866, 0.24891245, 0.14185762, 0.20304678, 0.60120412,
                 0.77438255, 1.74557866, 0.42505904, 1.93042427])
```

Mathematical and Statistical Methods

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.

```
In [245]: arr = np.arange(20).reshape((5, 4))
```

```
In [246]: arr

Out[246]: array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11],
                 [12, 13, 14, 15],
                 [16, 17, 18, 19]])
```

```
In [247]: arr.mean()
```

```
Out[247]: 9.5
```

```
In [248]: np.mean(arr)
```

```
Out[248]: 9.5
```

```
In [249]: arr.sum()
```

```
Out[249]: 190
```

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 [250]: arr.mean(axis=1)
```

```
Out[250]: array([ 1.5,  5.5,  9.5, 13.5, 17.5])
```

```
In [251]: arr.sum(axis=0)
```

```
Out[251]: array([40, 45, 50, 55])
```

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

```
In [268]: arr
```

```
Out[268]: array([-0.37337865,  2.09750292, -1.62537892,  0.0823575 , -0.71395187,
                  0.45674193, -0.03811723,  0.53397143,  0.19851352, -0.23244747,
                  -0.07008176, -1.416856  , -0.67223683,  1.26952   , -0.0314119 ,
                  -0.81452874,  0.05035723, -1.80302844,  1.66249091, -0.42703449,
                  -0.46056608,  0.58130759,  1.7504585 ,  0.10065007,  0.40061188,
                  1.40469707, -0.53351237,  0.53147637, -0.18554189,  1.67932754,
                  -0.36227099,  0.39060059,  0.83191476,  0.02513846,  0.35849898,
                  -1.44102577, -0.66395072,  0.47313736, -0.25082959, -0.33571729,
                  -0.01373723,  0.11613974,  0.64584091, -0.58729123,  0.92673551,
                  -0.70449528, -0.65338644,  1.53913136, -0.53455003,  1.06322668,
                  0.3828355 ,  0.79161214,  0.84478488, -0.37271317, -0.18775885,
                  -0.08636377,  1.52006863, -0.90682767,  0.93332395,  1.34491675,
                  0.62653052,  0.98566377, -0.74774112,  0.75327842, -0.73307121,
                  -1.0308154 , -0.34791947,  0.47897183, -0.11236194,  0.12337575,
                  0.32485924, -0.69526922,  0.15546491, -1.28043551, -0.78188937,
                  -1.88071562, -0.69908542,  0.35299715,  0.16423328,  2.45668454,
                  0.89457233,  0.38697647,  1.72157921, -0.13720526,  0.12496377,
                  -1.52536148,  0.04341854, -1.26474038, -1.71439178, -0.21311658,
                  1.00966571, -0.31593656,  0.30774834,  0.41291546,  0.73974377,
                  0.45135962, -0.12576371,  0.59527236,  1.11059439,  0.35902332])
```

```
In [269]: (arr > 0).sum() # Number of positive values
```

```
Out[269]: 54
```

any tests whether one or more values in an array is True, while **all** checks if every value is True:

```
In [270]: bools = np.array([False, False, True, False])
```



```
In [271]: bools.any()
```

```
Out[271]: True
```

```
In [272]: bools.all()
```

```
Out[272]: False
```

Sorting

```
In [273]: arr = np.random.randn(6)
```

```
In [274]: arr
```

```
Out[274]: array([-0.65286122,  0.07727207,  0.47816548,  0.32290724,  0.08945407,
                -1.31068488])
```

```
In [275]: arr.sort() #inplace
```

```
In [276]: arr
```

```
Out[276]: array([-1.31068488, -0.65286122,  0.07727207,  0.08945407,  0.32290724,
                0.47816548])
```

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

```
In [278]: arr
```

```
Out[278]: array([[ 0.87488706,  0.94700978,  0.61947158],
                [-0.24314617,  0.5823176 , -1.46280925],
                [ 0.91494697,  1.19554455, -1.67824757],
                [ 0.35661813, -0.50104464,  0.46406887],
                [ 1.37961108, -0.73628495, -0.12255107]])
```

```
In [279]: arr.sort(1)
```

```
In [280]: arr
```

```
Out[280]: array([[ 0.61947158,  0.87488706,  0.94700978],
                [-1.46280925, -0.24314617,  0.5823176 ],
                [-1.67824757,  0.91494697,  1.19554455],
                [-0.50104464,  0.35661813,  0.46406887],
                [-0.73628495, -0.12255107,  1.37961108]])
```

The top-level method **np.sort** returns a sorted copy of an array instead of modifying the array in-place.

```
In [281]: np.sort(arr)
```

```
Out[281]: array([[ 0.61947158,  0.87488706,  0.94700978],
                [-1.46280925, -0.24314617,  0.5823176 ],
                [-1.67824757,  0.91494697,  1.19554455],
                [-0.50104464,  0.35661813,  0.46406887],
                [-0.73628495, -0.12255107,  1.37961108]])
```

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

```
In [283]: large_arr.sort()
```

```
In [285]: int(0.05 * len(large_arr))
```

```
Out[285]: 50
```

```
In [290]: large_arr[:int(0.05 * len(large_arr))] # 5% quantile
```

```
Out[290]: array([-2.94108493, -2.6856096 , -2.52880161, -2.41681616, -2.41310542,
                -2.41188905, -2.39774893, -2.31805149, -2.28167901, -2.26458633,
                -2.2588826 , -2.24406885, -2.23237495, -2.21751869, -2.18023115,
                -2.16792915, -2.15723696, -2.10930285, -2.09595369, -2.02361692,
                -1.97811071, -1.95008234, -1.93405426, -1.92872126, -1.89667181,
                -1.87761095, -1.85713291, -1.8209341 , -1.81616409, -1.81532986,
                -1.81387127, -1.80497234, -1.78153941, -1.77105903, -1.76753505,
                -1.75578146, -1.74258716, -1.73456216, -1.72677068, -1.72218755,
                -1.70107469, -1.69132449, -1.66952798, -1.64590533, -1.63369753,
                -1.63287004, -1.62938613, -1.60573084, -1.60341046, -1.58766134])
```

Unique and Other Set Logic

np.unique returns the sorted unique values in an array:

```
In [291]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
```

```
In [292]: np.unique(names)
```

```
Out[292]: array(['Bob', 'Joe', 'Will'], dtype='<U4')
```

Contrast **np.unique** with the pure Python alternative:

```
In [293]: sorted(set(names))
```

```
Out[293]: ['Bob', 'Joe', 'Will']
```

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

```
In [294]: values = np.array([6, 0, 0, 3, 2, 5, 6])
```

```
In [295]: np.in1d(values, [2, 3, 6])
```

```
Out[295]: array([ True, False, False,  True,  True, False,  True])
```

Array set operations

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

File Input and Output with Arrays

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 [296]: arr = np.arange(10)
In [297]: np.save('some_array', arr)
In [298]: np.load('some_array.npy')
Out[298]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

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

```
In [299]: 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 [300]: arch = np.load('array_archive.npz')
In [301]: arch['b']
Out[301]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

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

```
In [302]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```

Linear Algebra

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

```
In [304]: x
```

```
Out[304]: array([[1., 2., 3.],
                [4., 5., 6.]])
```

```
In [305]: y
```

```
Out[305]: array([[ 6., 23.],
                [-1.,  7.],
                [ 8.,  9.]])
```

```
In [306]: x.shape , y.shape
```

```
Out[306]: ((2, 3), (3, 2))
```

```
In [307]: x.dot(y)
```

```
Out[307]: array([[ 28.,  64.],
                [ 67., 181.]])
```

```
In [308]: np.dot(x, y)
```

```
Out[308]: array([[ 28.,  64.],
                [ 67., 181.]])
```

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

```
In [309]: x @ y
```

```
Out[309]: array([[ 28.,  64.],
                [ 67., 181.]])
```

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):

```
In [310]: from numpy.linalg import inv, qr
```

```
In [311]: X = np.random.randn(5, 5)
```

```
In [312]: mat = X.T.dot(X)
```

```
In [313]: inv(mat)
```

```
Out[313]: array([[ 1.02261899, -0.22069948, -1.35955365,  1.95548458,  0.31887143],
                [-0.22069948,  0.31652906,  0.19270382, -0.55455638, -0.01662178],
                [-1.35955365,  0.19270382,  3.17794727, -4.49159046, -1.08744526],
                [ 1.95548458, -0.55455638, -4.49159046,  7.31135686,  1.47795054],
                [ 0.31887143, -0.01662178, -1.08744526,  1.47795054,  0.53413015]])
```

```
In [314]: mat.dot(inv(mat))
```

```
Out[314]: array([[ 1.00000000e+00,  4.85722573e-17,  0.00000000e+00,
                  -8.88178420e-16,  0.00000000e+00],
                 [-3.60822483e-16,  1.00000000e+00, -9.99200722e-16,
                  -8.88178420e-16, -5.55111512e-17],
                 [-2.22044605e-16,  2.91433544e-16,  1.00000000e+00,
                  0.00000000e+00, -4.44089210e-16],
                 [-4.71844785e-16,  2.81025203e-16,  9.99200722e-16,
                  1.00000000e+00, -3.33066907e-16],
                 [ 0.00000000e+00,  5.55111512e-17,  0.00000000e+00,
                  -1.77635684e-15,  1.00000000e+00]])
```

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

```
In [318]: samples.mean()
```

```
Out[318]: -0.39423407058551774
```

```
In [319]: samples
```

```
Out[319]: array([[ -0.13726869, -0.05688761,  0.02510036, -1.05613295],
                 [-2.92523403,  0.22152363,  0.05539478, -0.92136896],
                 [-0.7559269 , -0.10423869, -1.31393853,  0.18113329],
                 [ 0.42645927,  0.66812156, -0.52739976, -0.08708192]])
```

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 [320]: np.random.seed(1234)
```

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

```
In [322]: rng.randn(10)
```

```
Out[322]: array([ 0.47143516, -1.19097569,  1.43270697, -0.3126519 , -0.72058873,
                  0.88716294,  0.85958841, -0.6365235 ,  0.01569637, -2.24268495])
```