

Numpy

```
In [52]: 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 [53]: my_arr = np.arange(1000000)
```

```
In [54]: my_list = list(range(1000000))
```

```
In [55]: %time for _ in range(10): my_arr2 = my_arr * 2
```

Wall time: 31.9 ms

```
In [56]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]
```

Wall time: 1.08 s

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

The NumPy ndarray: A Multidimensional Array Object

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

```
In [58]: data
```

```
Out[58]: array([[ 1.29904659, -0.35849534,  0.73192217],  
               [-0.77925505, -0.06635645, -1.44162044]])
```

```
In [59]: data * 10
```

```
Out[59]: array([[ 12.99046591, -3.58495336,  7.31922174],  
               [-7.79255053, -0.66356448, -14.41620441]])
```

```
In [60]: data + data
```

```
Out[60]: array([[ 2.59809318, -0.71699067,  1.46384435],  
               [-1.55851011, -0.1327129 , -2.88324088]])
```

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 [61]: data.shape
```

```
Out[61]: (2, 3)
```

```
In [62]: data.dtype
```

```
Out[62]: dtype('float64')
```

Creating ndarrays

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

```
In [63]: data1 = [6, 7.5, 8, 0, 1]
```

```
In [64]: arr1 = np.array(data1)
```

```
In [65]: arr1
```

```
Out[65]: array([6. , 7.5, 8. , 0. , 1. ])
```

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

```
In [66]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
```

```
In [67]: arr2 = np.array(data2)
```

```
In [68]: arr2
```

```
Out[68]: array([[1, 2, 3, 4],
                [5, 6, 7, 8]])
```

```
In [69]: arr2.ndim
```

```
Out[69]: 2
```

```
In [70]: arr2.shape
```

```
Out[70]: (2, 4)
```

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

```
In [71]: arr1.dtype
```

```
Out[71]: dtype('float64')
```

```
In [72]: arr2.dtype
```

```
Out[72]: 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 [73]: np.zeros(10)
```

```
Out[73]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
In [74]: np.zeros((3, 6))
```

```
Out[74]: array([[0., 0., 0., 0., 0., 0.],
               [0., 0., 0., 0., 0., 0.],
               [0., 0., 0., 0., 0., 0.]])
```

```
In [75]: np.empty((2, 3, 2))
```

```
Out[75]: array([[[1.23816123e-311, 2.47032823e-322],
                  [0.00000000e+000, 0.00000000e+000],
                  [6.23054632e-307, 8.60952352e-072]],

                [[5.24774425e-090, 1.93374755e+184],
                  [5.50938909e+169, 1.61873077e-051],
                  [3.99910963e+252, 9.30986379e+165]]])
```

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

```
In [76]: np.arange(15)
```

```
Out[76]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```


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

```
In [78]: arr2 = np.array([1, 2, 3], dtype=np.int32)
```

```
In [79]: arr1.dtype
```

```
Out[79]: dtype('float64')
```

```
In [80]: arr2.dtype
```

```
Out[80]: 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 [81]: arr = np.array([1, 2, 3, 4, 5])
```

```
In [82]: arr.dtype
```

```
Out[82]: dtype('int32')
```

```
In [83]: float_arr = arr.astype(np.float64)
```

```
In [84]: float_arr.dtype
```

```
Out[84]: dtype('float64')
```

```
In [85]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
```

```
In [86]: arr
```

```
Out[86]: array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])
```

```
In [87]: arr.astype(np.int32)
```

```
Out[87]: array([ 3, -1, -2,  0, 12, 10])
```

```
In [88]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
```

```
In [89]: numeric_strings.astype(float)
```

```
Out[89]: 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 [90]: int_array = np.arange(10)
```

```
In [91]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
```

```
In [92]: int_array.astype(calibers.dtype)
```

```
Out[92]: 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 [93]: 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 [94]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])
```

```
In [95]: arr
```

```
Out[95]: array([[1., 2., 3.],
               [4., 5., 6.]])
```

```
In [96]: arr*arr
```

```
Out[96]: array([[ 1.,  4.,  9.],  
               [16., 25., 36.]])
```

```
In [97]: arr - arr
```

```
Out[97]: array([[0., 0., 0.],  
               [0., 0., 0.]])
```

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

```
In [98]: 1 / arr
```

```
Out[98]: array([[1.         , 0.5         , 0.33333333],  
               [0.25        , 0.2         , 0.16666667]])
```

```
In [99]: arr ** 0.5
```

```
Out[99]: array([[1.         , 1.41421356, 1.73205081],  
               [2.         , 2.23606798, 2.44948974]])
```

Comparisons between arrays of the same size yield boolean arrays:

```
In [100]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
```

```
In [101]: arr2 > arr
```

```
Out[101]: 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 [102]: arr = np.arange(10)
```

```
In [103]: arr
```

```
Out[103]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [104]: arr[5]
```

```
Out[104]: 5
```

```
In [105]: arr[5:8]
```

```
Out[105]: array([5, 6, 7])
```

```
In [106]: arr[5:8] = 12
```

```
In [107]: arr
```

```
Out[107]: 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 [108]: arr_slice = arr[5:8]
```

```
In [109]: arr_slice
```

```
Out[109]: array([12, 12, 12])
```

```
In [110]: arr_slice[1] = 12345
```

```
In [111]: arr
```

```
Out[111]: array([ 0,  1,  2,  3,  4, 12, 12345, 12,  8,  9])
```

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

```
In [112]: arr_slice[:] = 64
```

```
In [113]: arr
```

```
Out[113]: 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 [114]: arr_slice = arr[5:8].copy()
```

```
In [115]: arr_slice
```

```
Out[115]: array([64, 64, 64])
```



```
In [116]: arr_slice[1] = 12345
```

```
In [117]: arr
```

```
Out[117]: 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 [118]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
In [119]: arr2d[2]
```

```
Out[119]: array([7, 8, 9])
```

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

```
In [120]: arr2d[0][2]
```

```
Out[120]: 3
```

```
In [121]: arr2d[0, 2]
```

```
Out[121]: 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 [122]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

```
In [123]: arr3d.shape
```

```
Out[123]: (2, 2, 3)
```

```
In [124]: arr3d
```

```
Out[124]: array([[[ 1,  2,  3],
                  [ 4,  5,  6]],
                [[ 7,  8,  9],
                 [10, 11, 12]]])
```

arr3d[0] is a 2×3 array:

```
In [125]: arr3d[0]
```

```
Out[125]: array([[1, 2, 3],
                [4, 5, 6]])
```

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

```
In [126]: old_values = arr3d[0].copy()
```

```
In [127]: arr3d[0] = 42
```

```
In [128]: arr3d
```

```
Out[128]: array([[[42, 42, 42],
                  [42, 42, 42]],
                [[ 7,  8,  9],
                 [10, 11, 12]]])
```

```
In [129]: arr3d[0] = old_values
```

```
In [130]: arr3d
```

```
Out[130]: 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:

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

```
In [131]: x = arr3d[1]
```

```
In [132]: x
```

```
Out[132]: array([[ 7,  8,  9],  
                [10, 11, 12]])
```

```
In [133]: x[0]
```

```
Out[133]: array([7, 8, 9])
```

Indexing with slices

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

```
In [134]: arr
```

```
Out[134]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

```
In [135]: arr[1:6]
```

```
Out[135]: array([ 1,  2,  3,  4, 64])
```

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

```
In [136]: arr2d
```

```
Out[136]: array([[1, 2, 3],  
                [4, 5, 6],  
                [7, 8, 9]])
```

```
In [137]: arr2d[:2]
```

```
Out[137]: array([[1, 2, 3],  
                [4, 5, 6]])
```

```
In [138]: arr2d[:2, 1:]
```

```
Out[138]: 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 [139]: arr2d[1, :2]
```

```
Out[139]: array([4, 5])
```

```
In [140]: arr2d[:2, 2]
```

```
Out[140]: array([3, 6])
```

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

```
In [141]: arr2d[:, :1]
```

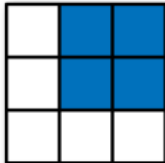
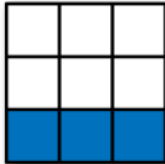
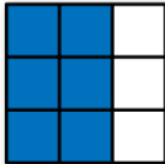
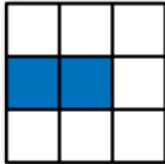
```
Out[141]: array([[1],  
                [4],  
                [7]])
```

Assigning to a slice expression assigns to the whole selection:

```
In [142]: arr2d[:2, 1:] = 0
```

```
In [143]: arr2d
```

```
Out[143]: 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 [144]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
```

```
In [145]: data = np.random.randn(7, 4)
```

```
In [146]: names
```

```
Out[146]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

```
In [147]: data
```

```
Out[147]: array([[ -1.45565612,  2.10136306,  0.08781366,  0.31314028],
 [ -1.79685007, -0.34783674,  1.13944839, -1.14777461],
 [ -0.02698795,  0.41626567, -0.83598748, -1.31247031],
 [ -0.10270405, -0.81609972, -0.07675042,  1.54351995],
 [ -0.73830382,  2.7866335 , -0.13682904,  0.024982  ],
 [ -2.24052944,  0.87798189, -0.6744322 , -0.77891158],
 [  0.076221  ,  0.70900184,  0.71623537, -0.47322533]])
```

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 [148]: names == 'Bob'
```

```
Out[148]: array([ True, False, False,  True, False, False, False])
```

```
In [149]: data[names == 'Bob']
```

```
Out[149]: array([[ -1.45565612,  2.10136306,  0.08781366,  0.31314028],
                 [-0.10270405, -0.81609972, -0.07675042,  1.54351995]])
```

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

```
In [150]: data[names == 'Bob', 2:]
```

```
Out[150]: array([[ 0.08781366,  0.31314028],
                 [-0.07675042,  1.54351995]])
```

```
In [151]: data[names == 'Bob', 3]
```

```
Out[151]: array([0.31314028, 1.54351995])
```

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

```
In [152]: names != 'Bob'
```

```
Out[152]: array([False,  True,  True, False,  True,  True,  True])
```

```
In [153]: data[~(names == 'Bob')]
```

```
Out[153]: array([[ -1.79685007, -0.34783674,  1.13944839, -1.14777461],
                 [-0.02698795,  0.41626567, -0.83598748, -1.31247031],
                 [-0.73830382,  2.7866335 , -0.13682904,  0.024982  ],
                 [-2.24052944,  0.87798189, -0.6744322 , -0.77891158],
                 [ 0.076221  ,  0.70900184,  0.71623537, -0.47322533]])
```

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

```
In [154]: cond = names == 'Bob'
```

```
In [155]: data[~cond]
```

```
Out[155]: array([[ -1.79685007, -0.34783674,  1.13944839, -1.14777461],
                 [-0.02698795,  0.41626567, -0.83598748, -1.31247031],
                 [-0.73830382,  2.7866335 , -0.13682904,  0.024982  ],
                 [-2.24052944,  0.87798189, -0.6744322 , -0.77891158],
                 [ 0.076221  ,  0.70900184,  0.71623537, -0.47322533]])
```

Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic

operators like & (and) and | (or):

```
In [156]: mask = (names == 'Bob') | (names == 'Will')
```

```
In [157]: mask
```

```
Out[157]: array([ True, False,  True,  True,  True, False, False])
```

```
In [158]: data[mask]
```

```
Out[158]: array([[ -1.45565612,  2.10136306,  0.08781366,  0.31314028],
                 [-0.02698795,  0.41626567, -0.83598748, -1.31247031],
                 [-0.10270405, -0.81609972, -0.07675042,  1.54351995],
                 [-0.73830382,  2.7866335 , -0.13682904,  0.024982  ]])
```

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 [159]: data
```

```
Out[159]: array([[ -1.45565612,  2.10136306,  0.08781366,  0.31314028],
                 [-1.79685007, -0.34783674,  1.13944839, -1.14777461],
                 [-0.02698795,  0.41626567, -0.83598748, -1.31247031],
                 [-0.10270405, -0.81609972, -0.07675042,  1.54351995],
                 [-0.73830382,  2.7866335 , -0.13682904,  0.024982  ],
                 [-2.24052944,  0.87798189, -0.6744322 , -0.77891158],
                 [ 0.076221  ,  0.70900184,  0.71623537, -0.47322533]])
```

```
In [160]: data[data < 0] = 0
```

```
In [161]: data
```

```
Out[161]: array([[0.          ,  2.10136306,  0.08781366,  0.31314028],
                 [0.          ,  0.          ,  1.13944839,  0.          ],
                 [0.          ,  0.41626567,  0.          ,  0.          ],
                 [0.          ,  0.          ,  0.          ,  1.54351995],
                 [0.          ,  2.7866335 ,  0.          ,  0.024982  ],
                 [0.          ,  0.87798189,  0.          ,  0.          ],
                 [0.076221  ,  0.70900184,  0.71623537,  0.          ]])
```

```
In [162]: data < 0
```

```
Out[162]: 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 [163]: data[names != 'Joe'] = 7
```

```
In [164]: data
```

```
Out[164]: array([[7.         , 7.         , 7.         , 7.         ],
 [0.         , 0.         , 1.13944839, 0.         ],
 [7.         , 7.         , 7.         , 7.         ],
 [7.         , 7.         , 7.         , 7.         ],
 [7.         , 7.         , 7.         , 7.         ],
 [0.         , 0.87798189, 0.         , 0.         ],
 [0.076221   , 0.70900184, 0.71623537, 0.         ]])
```

Fancy Indexing

```
In [165]: arr = np.empty((8, 4))
```

```
In [166]: for i in range(8):
          arr[i] = i
```

```
In [167]: arr
```

```
Out[167]: 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 [168]: arr[[4, 3, 0, 6]]
```

```
Out[168]: 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 [169]: arr[[-3, -5, -7]]
```

```
Out[169]: 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 [170]: arr = np.arange(32).reshape((8, 4))
```

```
In [171]: arr
```

```
Out[171]: 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 [172]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
```

```
Out[172]: array([ 4, 23, 29, 10])
```

The behavior of fancy indexing in this case is a bit different from what some users might have expected (myself included), which is the rectangular region formed by selecting a subset of the matrix's rows and columns. Here is one way to get that:

```
In [173]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
```

```
Out[173]: array([[ 4,  7,  5,  6],
                 [20, 23, 21, 22],
                 [28, 31, 29, 30],
                 [ 8, 11,  9, 10]])
```

Keep in mind that fancy indexing, unlike slicing, always copies the data into a new array.

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

```
In [175]: arr
```

```
Out[175]: array([[ 0,  1,  2,  3,  4],
                 [ 5,  6,  7,  8,  9],
                 [10, 11, 12, 13, 14]])
```

```
In [176]: arr.T
```

```
Out[176]: array([[ 0,  5, 10],
                 [ 1,  6, 11],
                 [ 2,  7, 12],
                 [ 3,  8, 13],
                 [ 4,  9, 14]])
```

```
In [177]: np.dot(arr.T, arr)
```

```
Out[177]: 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]])
```

For higher dimensional arrays, **transpose** will accept a tuple of axis numbers to permute the axes (for extra mind bending):

```
In [178]: arr = np.arange(40).reshape((2, 5, 4))
```

```
In [179]: arr
```

```
Out[179]: 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],
                  [32, 33, 34, 35],
                  [36, 37, 38, 39]]])
```

```
In [180]: arr.transpose((1, 0, 2))
```

```
Out[180]: array([[[ 0,  1,  2,  3],
                  [20, 21, 22, 23]],

                 [[ 4,  5,  6,  7],
                  [24, 25, 26, 27]],

                 [[ 8,  9, 10, 11],
                  [28, 29, 30, 31]],

                 [[12, 13, 14, 15],
                  [32, 33, 34, 35]],

                 [[16, 17, 18, 19],
                  [36, 37, 38, 39]]])
```

Here, the axes have been reordered with the second axis first, the first axis second, and the last axis unchanged.

```
In [181]: arr.transpose((1, 0, 2)).shape
```

```
Out[181]: (5, 2, 4)
```

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

```
In [183]: arr
```

```
Out[183]: array([[[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7]],

                 [[ 8,  9, 10, 11],
                  [12, 13, 14, 15]]])
```

```
In [184]: arr.swapaxes(1, 2)
```

```
Out[184]: array([[[ 0,  4],
                  [ 1,  5],
                  [ 2,  6],
                  [ 3,  7]],

                 [[ 8, 12],
                  [ 9, 13],
                  [10, 14],
                  [11, 15]]])
```

```
In [185]: arr.shape
```

```
Out[185]: (2, 2, 4)
```

swapaxes similarly returns a view on the data without making a copy

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

```
In [187]: np.sqrt(arr)
```

```
Out[187]: 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 [188]: x = np.random.randn(8)
```

```
In [189]: y = np.random.randn(8)
```

```
In [190]: x
```

```
Out[190]: array([-0.53993425, -0.09601701, -0.26128075, -0.85512342, -0.32404329,
                1.06104765, -0.48106102, -0.30156627])
```

```
In [191]: y
```

```
Out[191]: array([ 0.04115599,  1.29790112,  0.66319976,  0.30012337,  2.26238033,
                -1.16395329, -0.18757589,  1.81457026])
```

```
In [192]: np.maximum(x, y)
```

```
Out[192]: array([ 0.04115599,  1.29790112,  0.66319976,  0.30012337,  2.26238033,
                1.06104765, -0.18757589,  1.81457026])
```

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

```
In [194]: arr
```

```
Out[194]: array([ 7.24601987, -4.4896811 ,  0.90190416, 14.95250653,  1.30103197,
                -2.5256517 ,  7.42755075])
```

```
In [195]: remainder, whole_part = np.modf(arr)
```

```
In [196]: remainder
```

```
Out[196]: array([ 0.24601987, -0.4896811 ,  0.90190416,  0.95250653,  0.30103197,
                -0.5256517 ,  0.42755075])
```

```
In [197]: whole_part
```

```
Out[197]: array([ 7., -4.,  0., 14.,  1., -2.,  7.])
```

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

```
In [198]: arr
```

```
Out[198]: array([ 7.24601987, -4.4896811 ,  0.90190416, 14.95250653,  1.30103197,
                -2.5256517 ,  7.42755075])
```

```
In [199]: np.exp(arr)
```

```
Out[199]: array([1.40251155e+03, 1.12242227e-02, 2.46429106e+00, 3.11738956e+06,
                3.67308524e+00, 8.00061560e-02, 1.68168366e+03])
```

```
In [200]: np.exp(arr, arr)
```

```
Out[200]: array([1.40251155e+03, 1.12242227e-02, 2.46429106e+00, 3.11738956e+06,
                3.67308524e+00, 8.00061560e-02, 1.68168366e+03])
```

```
In [201]: arr
```

```
Out[201]: array([1.40251155e+03, 1.12242227e-02, 2.46429106e+00, 3.11738956e+06,
                3.67308524e+00, 8.00061560e-02, 1.68168366e+03])
```

 "Unary ufuncs"

 "Binary universal functions"

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

```
In [203]: xs, ys = np.meshgrid(points, points)
```

```
In [204]: ys
```

```
Out[204]: 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 [205]: xs
```

```
Out[205]: 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 [206]: xs.shape
```

```
Out[206]: (1000, 1000)
```

```
In [207]: z = np.sqrt(xs ** 2 + ys ** 2)
```

```
In [208]: z.min()
```

```
Out[208]: 1.507288760336424e-13
```

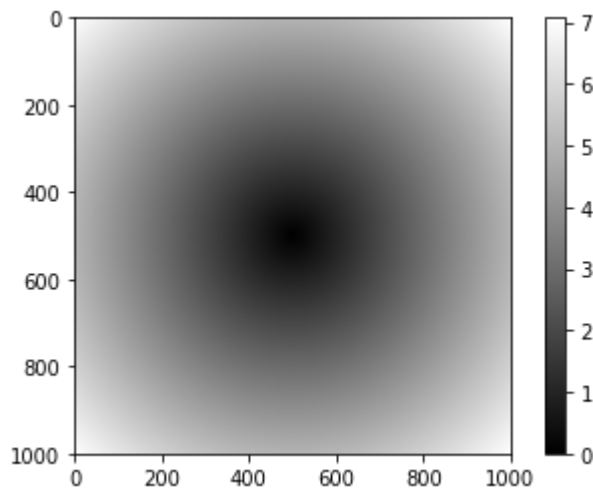
```
In [209]: z.max()
```

```
Out[209]: 7.0710678118654755
```

```
In [210]: import matplotlib.pyplot as plt
```

```
In [211]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
```

```
Out[211]: <matplotlib.colorbar.Colorbar at 0x24709b46d30>
```



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 [212]: 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 [213]: result = [(x if c else y)
                    for x, y, c in zip(xarr, yarr, cond)]
```

```
In [214]: result
```

```
Out[214]: [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 [215]: result = np.where(cond, xarr, yarr)
```

```
In [216]: result
```

```
Out[216]: 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 [217]: arr = np.random.randn(4, 4)
```

```
In [218]: arr
```

```
Out[218]: array([[ 0.80919423,  0.67016953, -0.0924767 , -0.27139904],
                 [-2.32117967,  0.27639576,  0.70977141,  2.0845821 ],
                 [-0.36316583, -0.41514498, -1.47434034,  0.13609221],
                 [-0.015347  ,  0.25335209,  0.32208006, -0.50062417]])
```

```
In [219]: arr > 0
```

```
Out[219]: array([[ True,  True, False, False],
                 [False,  True,  True,  True],
                 [False, False, False,  True],
                 [False,  True,  True, False]])
```

```
In [220]: np.where(arr > 0, 2, -2)
```

```
Out[220]: 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 [221]: np.where(arr > 0, 2, arr) # set only positive values to 2
```

```
Out[221]: array([[ 2.          ,  2.          , -0.0924767 , -0.27139904],
                 [-2.32117967,  2.          ,  2.          ,  2.          ],
                 [-0.36316583, -0.41514498, -1.47434034,  2.          ],
                 [-0.015347  ,  2.          ,  2.          , -0.50062417]])
```

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

```
In [222]: a = np.arange(5,10)
          np.where(a < 8)
```

```
Out[222]: (array([0, 1, 2], dtype=int64),)
```


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

```
In [223]: a[np.where(a < 8)]
```

```
Out[223]: 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 [224]: np.where(arr > 0)
```

```
Out[224]: (array([0, 0, 1, 1, 1, 2, 3, 3], dtype=int64),  
          array([0, 1, 1, 2, 3, 3, 1, 2], 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 [225]: arr[np.where(arr > 0)]
```

```
Out[225]: array([0.80919423, 0.67016953, 0.27639576, 0.70977141, 2.0845821 ,  
                0.13609221, 0.25335209, 0.32208006])
```

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

```
In [227]: arr
```

```
Out[227]: array([[ 0,  1,  2,  3],  
                [ 4,  5,  6,  7],  
                [ 8,  9, 10, 11],  
                [12, 13, 14, 15],  
                [16, 17, 18, 19]])
```

```
In [228]: arr.mean()
```

```
Out[228]: 9.5
```

```
In [229]: np.mean(arr)
```

```
Out[229]: 9.5
```

```
In [230]: arr.sum()
```

```
Out[230]: 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 [231]: arr.mean(axis=1)
```

```
Out[231]: array([ 1.5,  5.5,  9.5, 13.5, 17.5])
```

```
In [232]: arr.sum(axis=0)
```

```
Out[232]: array([40, 45, 50, 55])
```

Other methods like **cumsum** and **cumprod** do not aggregate, instead producing an array of the intermediate results:

```
In [233]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
In [234]: arr.cumsum()
```

```
Out[234]: array([ 0,  1,  3,  6, 10, 15, 21, 28], dtype=int32)
```

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

```
In [236]: arr
```

```
Out[236]: array([[0, 1, 2],
                 [3, 4, 5],
                 [6, 7, 8]])
```

```
In [237]: arr.cumsum(axis=0)
```

```
Out[237]: array([[ 0,  1,  2],
                 [ 3,  5,  7],
                 [ 9, 12, 15]], dtype=int32)
```

```
In [238]: arr.cumprod(axis=1)
```

```
Out[238]: array([[ 0,  0,  0],
                  [ 3, 12, 60],
                  [ 6, 42, 336]], dtype=int32)
```

```
In [239]: arr.cumsum()
```

```
Out[239]: array([ 0,  1,  3,  6, 10, 15, 21, 28, 36], dtype=int32)
```

![[alt text]](images/basic statistical.png "Basic array statistical methods")

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

```
In [241]: arr
```

```
Out[241]: array([ 3.95446099e-01,  7.05594076e-01, -1.46823155e+00,  2.08463179e-01,
                  6.42658739e-01,  7.27883844e-02,  3.00667871e-01,  1.23344922e-01,
                 -1.32800426e+00, -3.05970231e-01,  1.32095653e-01,  5.76770891e-01,
                  5.40932484e-01,  3.97237528e-01, -1.28393543e+00,  2.25105388e-01,
                  6.88497088e-01,  1.09694067e+00, -3.46368851e-02, -9.08169007e-01,
                  9.42313541e-01,  9.14495000e-01,  9.25226244e-01, -1.97845021e+00,
                  1.22086432e+00, -8.08735943e-02, -4.07479971e-01,  4.37761328e-01,
                 -6.84001067e-01,  3.97638221e-02,  2.14185283e+00,  9.19701099e-01,
                 -7.95338880e-01, -7.31785455e-01,  9.81680732e-01, -1.00489401e+00,
                 -1.40324183e+00,  1.32724313e+00, -2.96845393e-01,  3.34655527e-01,
                 -1.30761744e+00, -5.92224374e-01, -4.19205627e-01,  1.85527872e+00,
                 -3.22843698e-01, -1.49522623e+00,  3.78043818e-01, -1.77365401e+00,
                 -1.28477158e+00,  8.92759069e-01,  5.30585026e-01,  1.91040231e-01,
                 -5.50621966e-01, -1.55280945e+00,  3.72168289e-01,  4.32376748e-01,
                 -3.69601617e-01,  3.94211468e-01, -2.34629907e+00,  2.43057218e+00,
                 -8.42426481e-02, -1.22687384e+00, -4.35024142e-01, -9.69527485e-02,
                 -2.75211243e+00, -1.27264774e+00,  3.06153757e-01,  2.36695409e+00,
                 -9.21781741e-01, -1.92092550e-01, -8.21654467e-01, -4.96580237e-01,
                 -2.64440095e-01,  1.60951804e+00, -8.03001693e-01, -2.04693780e+00,
                  1.76922813e+00,  4.94708690e-01,  5.50634116e-01, -7.34664199e-01,
                  1.08319082e+00, -2.90190412e-01, -7.86507998e-01, -1.07511186e+00,
                  1.17967224e-01,  8.11810964e-01, -3.01857246e-01,  2.57256606e-01,
                  9.08429766e-01,  5.60451381e-01,  1.83757250e+00, -9.60488276e-01,
                 -1.34339903e-03, -1.07753999e+00,  5.90100813e-01,  1.14648852e+00,
                 -6.06156384e-02, -5.95107189e-01,  1.33666786e+00, -1.50832271e+00])
```

```
In [242]: (arr > 0).sum() # Number of positive values
```

```
Out[242]: 50
```

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

```
In [243]: bools = np.array([False, False, True, False])
```

```
In [244]: bools.any()
```

```
Out[244]: True
```

```
In [245]: bools.all()
```

```
Out[245]: False
```

Sorting

```
In [246]: arr = np.random.randn(6)
```

```
In [247]: arr
```

```
Out[247]: array([ 1.05180256,  1.52650292, -0.74758223,  1.27144232,  0.30594934,
                  -0.21444092])
```

```
In [248]: arr.sort() #inplace
```

```
In [249]: arr
```

```
Out[249]: array([-0.74758223, -0.21444092,  0.30594934,  1.05180256,  1.27144232,
                  1.52650292])
```

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

```
In [251]: arr
```

```
Out[251]: array([[ 0.07858387,  2.06738044, -0.21815406],
                  [ 0.86974218, -0.19014351, -0.78606876],
                  [-1.89463643, -0.97720038, -1.57718151],
                  [ 0.3007583 ,  0.52516229, -1.37144567],
                  [-0.65379091,  0.50806196,  0.36342604]])
```

```
In [252]: arr.sort(1)
```

```
In [253]: arr
```

```
Out[253]: array([[ -0.21815406,  0.07858387,  2.06738044],
                 [-0.78606876, -0.19014351,  0.86974218],
                 [-1.89463643, -1.57718151, -0.97720038],
                 [-1.37144567,  0.3007583 ,  0.52516229],
                 [-0.65379091,  0.36342604,  0.50806196]])
```

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

```
In [254]: np.sort(arr)
```

```
Out[254]: array([[ -0.21815406,  0.07858387,  2.06738044],
                 [-0.78606876, -0.19014351,  0.86974218],
                 [-1.89463643, -1.57718151, -0.97720038],
                 [-1.37144567,  0.3007583 ,  0.52516229],
                 [-0.65379091,  0.36342604,  0.50806196]])
```

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

```
In [256]: large_arr.sort()
```

```
In [257]: large_arr[int(0.05 * len(large_arr))] # 5% quantile
```

```
Out[257]: -1.6711893316258368
```

Unique and Other Set Logic

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

```
In [258]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
```

```
In [259]: np.unique(names)
```

```
Out[259]: array(['Bob', 'Joe', 'Will'], dtype='<U4')
```

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

```
In [260]: sorted(set(names))
```

```
Out[260]: ['Bob', 'Joe', 'Will']
```

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

```
In [261]: values = np.array([6, 0, 0, 3, 2, 5, 6])
```

```
In [262]: np.in1d(values, [2, 3, 6])
```

```
Out[262]: array([ True, False, False,  True,  True, False,  True])
```

![alt text](images/array set operations.png "Array set operations")

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

```
In [264]: np.save('some_array', arr)
```

```
In [265]: np.load('some_array.npy')
```

```
Out[265]: 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 [266]: 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 [267]: arch = np.load('array_archive.npz')
```

```
In [268]: arch['b']
```

```
Out[268]: 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 [269]: 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 [270]: x = np.array([[1., 2., 3.], [4., 5., 6.]])  
          y = np.array([[6., 23.], [-1, 7], [8, 9]])
```

```
In [271]: x
```

```
Out[271]: array([[1., 2., 3.],  
                [4., 5., 6.]])
```

```
In [272]: y
```

```
Out[272]: array([[ 6., 23.],  
                [-1.,  7.],  
                [ 8.,  9.]])
```

```
In [273]: x.shape , y.shape
```

```
Out[273]: ((2, 3), (3, 2))
```

```
In [274]: x.dot(y)
```

```
Out[274]: array([[ 28.,  64.],  
                [ 67., 181.]])
```

```
In [275]: np.dot(x, y)
```

```
Out[275]: array([[ 28.,  64.],  
                [ 67., 181.]])
```

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

```
In [276]: x @ y
```

```
Out[276]: 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 [277]: from numpy.linalg import inv, qr
```

```
In [278]: X = np.random.randn(5, 5)
```

```
In [279]: mat = X.T.dot(X)
```

```
In [280]: inv(mat)
```

```
Out[280]: array([[ 11.90365701, -30.09494017,  3.51795086, -5.08399306,
 26.29848104],
 [-30.09494017,  77.68130499, -8.88521991,  13.10267733,
 -67.41477796],
 [ 3.51795086, -8.88521991,  1.15840875, -1.45467158,
  7.81393431],
 [-5.08399306,  13.10267733, -1.45467158,  2.3767254 ,
 -11.36847273],
 [ 26.29848104, -67.41477796,  7.81393431, -11.36847273,
 58.80610523]])
```

```
In [281]: mat.dot(inv(mat))
```

```
Out[281]: array([[ 1.00000000e+00,  1.94963937e-14,  3.62323504e-16,
 -1.82085742e-16,  3.40466444e-14],
 [ 2.63735295e-14,  1.00000000e+00,  1.51665556e-15,
 -6.14079955e-15,  2.50156238e-15],
 [ 3.91619111e-15, -1.05905768e-14,  1.00000000e+00,
 -2.43600619e-15,  7.26431398e-15],
 [-5.56614941e-15, -8.03991754e-15, -1.23431489e-16,
  1.00000000e+00,  3.87887337e-16],
 [ 2.53597700e-14,  2.25661765e-14,  3.47664911e-15,
  1.13099554e-14,  1.00000000e+00]])
```

```
In [282]: q, r = qr(mat)
```

Commonly used numpy.linalg functions

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

```
In [284]: samples.mean()
```

```
Out[284]: 0.16164870113528143
```

```
In [285]: samples
```

```
Out[285]: array([[ 0.63518655,  0.66189529,  1.46361606, -0.1610624 ],
 [ 0.28881374,  0.09943448, -0.27397532,  0.01560321],
 [ 1.15474223,  0.10515925, -0.31596453, -0.40476898],
 [-0.53561748, -0.02026385,  0.69113503, -0.81755409]])
```

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 [286]: from random import normalvariate
```

```
In [287]: N = 1000000
```

```
In [288]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
```

```
882 ms ± 27 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [289]: %timeit np.random.normal(size=N)
```

```
34.8 ms ± 1.26 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

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 [290]: 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 [291]: rng = np.random.RandomState(1234)
```

```
In [292]: rng.randn(10)
```

```
Out[292]: array([ 0.47143516, -1.19097569,  1.43270697, -0.3126519 , -0.72058873,  
                0.88716294,  0.85958841, -0.6365235 ,  0.01569637, -2.24268495])
```

Table 4-8. 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

```
In [ ]:
```