NumPy

NumPy standard for Numerical Python.

Here are some of the highlights of NumPy:

- ndarray (or N-dimensional array object), 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.

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

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

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

In [ ]:
```
import numpy as np
my_arr2 = np.arange(1000000)
my_list = list(range(1000000))
```

Now let's multiply each sequence by 2 and compute the required time:

In [ ]:
```
%time for _ in range(10): my_arr = my_arr2 * 2 # ms => milliseconds and 1 second =
1000 milliseconds
```

- '%time' is a magic command.
  - Magic commands come in two flavors:
    - line magics, which are denoted by a single % prefix and operate on a single line of input, and
    - cell magics, which are denoted by a double %% prefix and operate on multiple lines of input within a cell.
  - 'time' provides CPU time (user + sys time) and Wall time.
    - CPU time may be available based on OS type.
    - Wall time = CPU time + I/O time + ... => total time to execute.
  - You can use %%timeit -n 100 to compute the average execution time of a cell from 100 runs.

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

- Here `_` is just a variable. Traditionally, it is used to indicate that the values (from 0 to (10-1)) of the variable is of no use.

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

NumPy enables **batch computations** with similar syntax to scalar values on built-in Python objects

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

In the first example, all of the elements have been multiplied by 10.

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

In the second, the corresponding values in each “cell” in the array have been added to each other.

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

An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements **must** be the same type.

Every array has a shape, a tuple indicating the size of each dimension, and a dtype, an object describing the data type of the array:

In [ ]:
```
data.shape
```

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

Creating ndarrays

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

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

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

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

In [ ]:
```
arr2.ndim # ndim returns number of axes
```

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

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

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

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

In addition to `np.array` , there are a number of other functions for creating new arrays. As examples, `zeros` and `ones` create arrays of 0s or 1s, respectively, with a given length or shape.

`empty` creates an array without initializing its values to any particular value. To create a higher dimensional array with these methods, pass a tuple for the shape:

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

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

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

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

In [ ]:
```
 np.empty((2, 4))
```

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

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

In [ ]:
```
np.full((2,3),6)
```

See Table 1 below for a short list of standard array creation functions. Since NumPy is focused on numerical computing, the data type, if not specified, will in many cases be float64 (floating point).

Table 1: Array creation functions.

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

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

A standard double precision floating-point value (what's used under the hood in Python's float object) takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as float64. See Table 2 for a full listing of NumPy's supported data types.

Table 2: NumPy data types.

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

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

Next, integers are cast to floating point.

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

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

If you cast some floating-point numbers to be of integer dtype, the decimal part will be truncated:

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

In [ ]:
```
arr
```

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

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

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

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.

If casting were to fail for some reason (like a string that cannot be converted to float64), a ValueError will be raised.

You can also use another array's dtype attribute:

In [ ]:
```
int_array = np.arange(10)
calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
int_array.astype(calibers.dtype)
```

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

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

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

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

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

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

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

In [ ]:
```
arr ** 0.5 # apply sqrt element-wise.
```

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

Comparisons between arrays of the same size yield boolean arrays:

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

Operations between differently sized arrays is called *broadcasting*.

Basic Indexing and Slicing

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

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

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

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

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

In [ ]:
```
arr
```

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

An important first distinction from Python's built-in lists is that array slices are **views** on the original array. This means that the data is **not copied**, and any modifications to the view will be reflected in the source array.

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

If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array—for example, `arr[5:8].copy()` .

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

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

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

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

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

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

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

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

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

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

In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions. So in the 2 × 2 × 3 array arr3d:

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

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

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

In [ ]:
```
old_values = arr3d[0].copy()
arr3d[0] = 42
arr3d
```

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

In [ ]:
```
arr3d
```

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

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

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

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

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

Note that in all of these cases where subsections of the array have been selected, the returned arrays are views.

Indexing with slices

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

In [ ]:
```
arr
```

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

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

In [ ]:
```
arr2d
```

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

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

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

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

When slicing like this, you always obtain array views of the same number of dimensions. By mixing integer indexes and slices, you get lower dimensional slices.

For example, you can select the second row but only the first two columns like so:

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

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

See Fig. 1 for an illustration.

Fig. 1:Two-dimensional array slicing

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

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

Of course, assigning to a slice expression assigns to the whole selection:

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

Boolean Indexing

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

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

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

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

This boolean array can be passed when indexing the array:

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

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

The boolean array must be of the same length as the array axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers; more on this later).

Note: Boolean selection will not fail if the boolean array is not the correct length, so care should be taken when using this feature.

In these examples, you can select from the rows where `names == 'Bob'` and index the columns, too:

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

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

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

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

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

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

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

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

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

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

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

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

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

These types of operations on two-dimensional data are convenient to do with pandas.

Fancy Indexing

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

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

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

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

Using negative indices selects rows from the end (starts counting from -1, -2, ...):

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

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

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

In [ ]:
```
arr[[1, 5, 7, 2], [0, 3, 1, 2]] # The formed indexes are (1,0), (5,3), (7,1) & (2,2).
```

The behavior of fancy indexing in this case may seem a bit unusual. Here it is the rectangular region formed by selecting a subset of the matrix's rows and columns. This is one way to get that:

In [ ]:
```
arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]  # consider this as a two-step indexing
```

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

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

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

In [ ]:
```
arr = np.random.randn(6, 3)
arr
```

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

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

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

Below, the axes have been reordered with the second axis first, the first axis second, and the last axis unchanged:

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

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

Below, the axes have been reordered with the first axis unchanges, 3rd/last axis second and the second axis last:

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

Below, the axes have been reordered with the 3rd/last axis first, the first axis second and the second axis last:

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

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 [ ]:
```
arr
```

In [ ]:
```
arr.swapaxes(1, 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 [ ]:
```
arr = np.arange(10)
arr
```

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

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

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

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

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

Here, `numpy.maximum` computed the element-wise maximum of the elements in x and y.

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

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

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

In [ ]:
```
remainder
```

In [ ]:
```
whole_part
```

In [ ]:
```
arr
```

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

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

In [ ]:
```
arr
```

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

In [ ]:
```
arr
```

See Table 3 and Table 4 for the listings of available ufuncs.

Table 3: Unary ufuncs.

Table 4: Binary universal functions.

Array-Oriented Programming with Arrays

Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops. This practice of replacing explicit loops with array expressions is commonly referred to as **vectorization**.

In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations.

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

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

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

In [ ]:
```
ys
```

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

In [ ]:
```
# This will help us have the plot within this notebook output
%matplotlib inline
```

In [ ]:
```
import matplotlib.pyplot as plt
plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")
plt.draw()
```

In [ ]:
```
plt.close('all')
```

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 [ ]:
```
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])
```

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

This has multiple problems:

- First, it will **not be very fast for large arrays** (because all the work is being done in interpreted Python code).
- Second, it will **not work with multidimensional arrays**.

With `np.where` you can write this very concisely:

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

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

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

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

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

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

The arrays passed to `np.where` can be more than just equal-sized arrays or scalars.

Mathematical and Statistical Methods

A set of mathematical functions that compute statistics about an entire array or about the data along an axis are accessible as methods of the array class.

You can use aggregations (often called ***reductions***) like `sum` , `mean` , and `std` (standard deviation) either by calling the array instance method or using the top-level NumPy function.

Here, some normally distributed random data and some aggregate statistics based on the data, have been computed:

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

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

In [ ]:
```
np.mean(arr)  # Numpy
```

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

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

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

Here, `arr.mean(1)` means "compute mean across the columns" where `arr.sum(0)` means "compute sum down the rows."

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

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

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

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

In [ ]:
```
arr
```

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

See Table 5 below, which contains a full listing of these statistical methods.

Table 5: Basic array statistical methods.

Methods for Boolean Arrays

Boolean values are forced 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 [ ]:
```
arr = np.random.randn(100)
arr
```

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

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

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

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

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

These methods also work with non-boolean arrays, where non-zero elements evaluate to `True` .

Sorting

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

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

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

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

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

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

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

Unique and Other Set Logic

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

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

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

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

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

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

In [ ]:
```
values = np.array([6, 0, 0, 3, 2, 5, 6])
np.in1d(values, [2, 3, 6])
```

See Table 6 for a listing of set functions in NumPy.

Table 6: Array set operations

File Input and Output with Arrays

NumPy is able to save and load data to and from disk either in text or binary format. However, here, we will only discuss NumPy's built-in binary format, since most users will prefer pandas and other tools for loading text or tabular data.

`np.save` and `np.load` are the two workhorse functions for efficiently saving and loading array data on disk. Arrays are saved by default in an uncompressed raw binary format with file extension *.npy*:

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

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

If the file path does not already end in *.npy*, the extension will be appended. The array on disk can then be loaded with `np.load`:

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

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

In [ ]:
```
np.savez('./numpy/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 [ ]:
```
arch = np.load('./numpy/array_archive.npz')
arch
```

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

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

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

In [ ]:
```
# remove the created file
#!rm ./numpy/some_array.npy    # rm => remove (suitable for Linux)
                               # for windows use !del or !erase; i.e., "!erase .\nump
y\some_array.npy" or "!del .\numpy\some_array.npy"
!erase .\numpy\some_array.npy
```

In [ ]:
```
# remove the created file
!rm ./numpy/array_archive.npz  # for windows use !del or !erase ...
```

In [ ]:
```
# remove the created file
!rm ./numpy/arrays_compressed.npz # for windows use !del or !erase ...
```

Linear Algebra

Linear algebra, like matrix multiplication, decompositions, determinants, and other square matrix math, is an important part of any array library. Unlike some languages like MATLAB, multiplying two two-dimensional arrays with * is an element-wise product instead of a matrix dot product.

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

In [ ]:
```
x = np.array([[1., 2., 3.], [4., 5., 6.]]) # Matrix dimension 2 x 3
x
```

In [ ]:
```
y = np.array([[6., 23.], [-1, 7], [8, 9]]) # Matrix dimension 3 x 2
y
```

In [ ]:
```
x.dot(y)    # Matrix dimension [2 x 3] * [3 x 2] => [2 x 2]
```

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

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

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

In [ ]:
```
np.dot(x, np.ones(3)) # Matrix dimension [2 x 3] * [3 x 1] => [2 x 1]
```

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

In [ ]:
```
x @ np.ones(3)
```

`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 [ ]:
```
from numpy.linalg import inv, qr
X = np.random.randn(5, 5)
X
```

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

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

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

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

`qr` computes QR decomposition.

- QR decompositionis a decomposition of a matrix X into a product X = QR of an orthogonal matrix Q and an upper triangular matrix R.
- QR decomposition is often used to solve the linear least squares problem.

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

In [ ]:
```
q
```

In [ ]:
```
r
```

In [ ]:
```
x=np.arange(12).reshape((3,4))
x
```

See Table 7 for a list of some of the most commonly used linear algebra functions.

Table 7: Commonly used `numpy.linalg` functions.

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

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 [ ]:
```
from random import normalvariate
N = 1000000
```

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

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

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 [ ]:
```
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 [ ]:
```
rng = np.random.RandomState(124)
rng.randn(10)
```

See Table 8 for a partial list of functions available in `numpy.random`.

Table 8: Partial list of `numpy.random` functions.

References:


```
[1] Python for Data Analysis Data Wrangling with Pandas, NumPy, and IPython by Wes McK
inney, 2nd Edn, O'Reilly 2017.
```