Introduction to NumPy

NumPy (short for *Numerical Python*) provides an efficient interface to store and operate on dense data buffers.

NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size

NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python

Install numpy

```
import numpy
numpy.__version__
```

```
import numpy as np
```

Understanding Data Types in Python

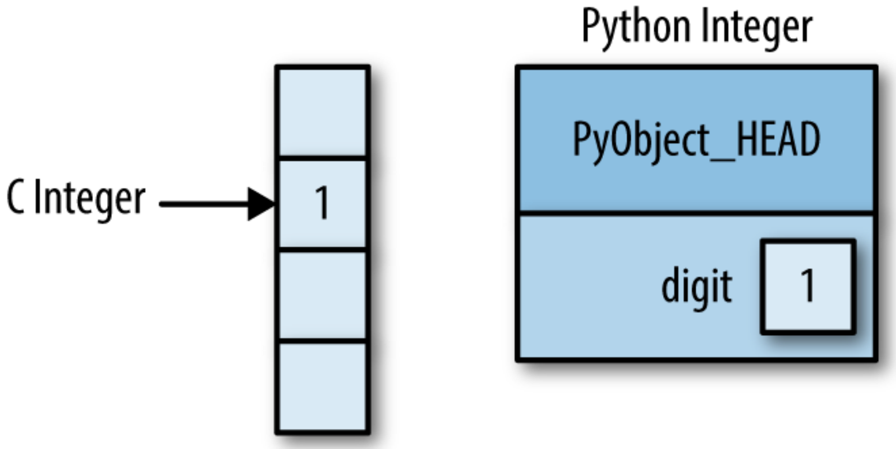
```
/* C code */
int result = 0;
for(int i=0; i<100; i++){
result += i;
}

# Python code
result = 0
for i in range(100):
result += i
```

```c
struct _longobject {
long ob_refcnt;
PyTypeObject *ob_type;
size_t ob_size;
long ob_digit[1];
};
```

A single integer in Python 3.4 actually contains four pieces:

- ob_refcnt, a reference count that helps Python silently handle memory allocation and deallocation
- ob_type, which encodes the type of the variable
- ob_size, which specifies the size of the following data members
- ob_digit, which contains the actual integer value that we expect the Python variable to represent

The difference between C and Python integers



A Python List Is More Than Just a List

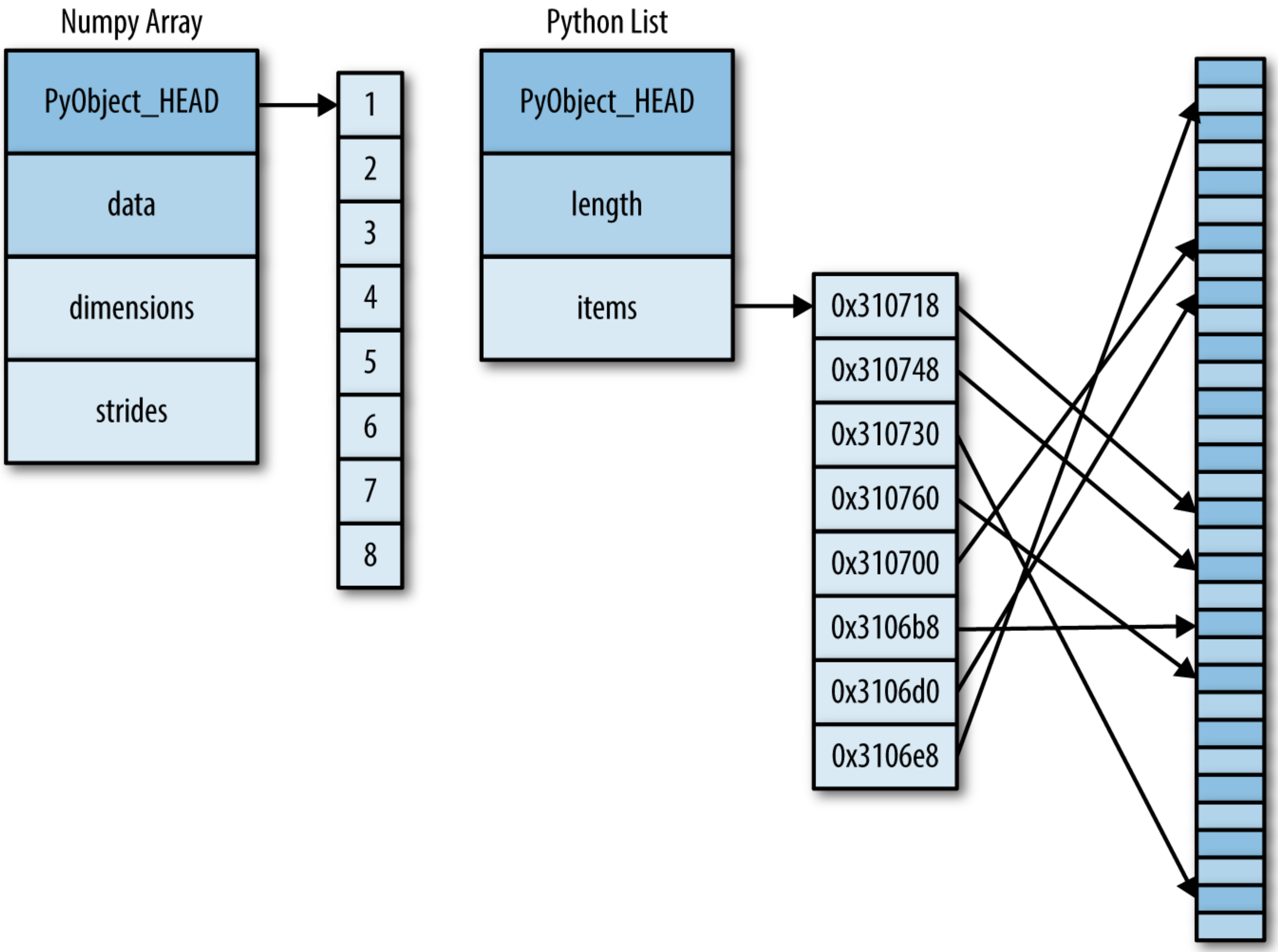
```
In[1]: L = list(range(10))
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In[2]: type(L[0])
Out[2]: int
```

Or, similarly, a list of strings:

```
In[3]: L2 = [str(c) for c in L]
Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
In[4]: type(L2[0])
Out[4]: str
```

Because of Python’s dynamic typing, we can even create heterogeneous lists:

```
In[5]: L3 = [True, "2", 3.0, 4]
[type(item) for item in L3]
Out[5]: [bool, str, float, int]
```


Numpy Array
PyObject_HEAD
data
dimensions
strides
1
2
3
4
5
6
7
8
Python List
PyObject_HEAD
length
items
0x310718
0x310748
0x310730
0x310760
0x310700
0x3106b8
0x3106d0
0x3106e8

Creating Arrays from Python Lists

```
In[8]: # integer array:
np.array([1, 4, 2, 5, 3])
Out[8]: array([1, 4, 2, 5, 3])

In[9]: np.array([3.14, 4, 2, 3])
Out[9]: array([ 3.14, 4. , 2. , 3. ])
In[10]: np.array([1, 2, 3, 4], dtype='float32')
Out[10]: array([ 1., 2., 3., 4.], dtype=float32)

In[11]: # nested lists result in multidimensional arrays
np.array([range(i, i + 3) for i in [2, 4, 6]])
Out[11]: array([[2, 3, 4],
[4, 5, 6],
[6, 7, 8]])
```

Creating Arrays from Scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines
built into NumPy.

```
In[12]: # Create a length-10 integer array filled with zeros
np.zeros(10, dtype=int)
Out[12]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In[13]: # Create a 3x5 floating-point array filled with 1s
np.ones((3, 5), dtype=float)
Out[13]: array([[ 1., 1., 1., 1., 1.],
[ 1., 1., 1., 1., 1.],
[ 1., 1., 1., 1., 1.]])
In[14]: # Create a 3x5 array filled with 3.14
np.full((3, 5), 3.14)9578071257
```

The Basics of NumPy Arrays

NumPy array manipulation to access data
and subarrays, and to split, reshape, and join the arrays.

Attributes of arrays

Determining the size, shape, memory consumption, and data types
of arrays

Indexing of arrays

Getting and setting the value of individual array elements

Slicing of arrays

Getting and setting smaller subarrays within a larger array

Reshaping of arrays

Changing the shape of a given array

NumPy Array Attributes

a one-dimensional, two-dimensional, and three-dimensional array.

```
In[1]: import numpy as np
np.random.seed(0) # seed for reproducibility
x1 = np.random.randint(10, size=6) # One-dimensional array
x2 = np.random.randint(10, size=(3, 4)) # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5)) # Three-dimensional array
```

Each array has attributes ndim (the number of dimensions), shape (the size of each dimension), and size (the total size of the array):

```
In[2]: print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
x3 ndim: 3
x3 shape: (3, 4, 5)
x3 size: 60
```

Array Indexing: Accessing Single Elements

In a one-dimensional array, you can access the *i*th value (counting from zero) by specifying the desired index in square brackets, just as with Python lists:

```
In[5]: x1
Out[5]: array([5, 0, 3, 3, 7, 9])
In[6]: x1[0]
Out[6]: 5
In[7]: x1[4]
Out[7]: 7
```

To index from the end of the array, you can use negative indices:

```
In[8]: x1[-1]
Out[8]: 9
In[9]: x1[-2]
Out[9]: 7
```

In a multidimensional array, you access items using a
comma-separated tuple of
indices:

```
In[10]: x2
Out[10]: array([[3, 5, 2, 4],
[7, 6, 8, 8],
[1, 6, 7, 7]])
In[11]: x2[0, 0]
Out[11]: 3
```

Array Slicing: Accessing Subarrays

To access subarrays with the *slice* notation, marked by the colon (:) character.
The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array x, use this:

x[start:stop:step]

If any of these are unspecified, they default to the values start=0, stop=*size of dimension*, step=1. One-dimensional subarrays

```
In[16]: x = np.arange(10)
x
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In[17]: x[:5] # first five elements
Out[17]: array([0, 1, 2, 3, 4])
In[18]: x[5:] # elements after index 5
Out[18]: array([5, 6, 7, 8, 9])
In[19]: x[4:7] # middle subarray
Out[19]: array([4, 5, 6])
```

Multidimensional subarrays

Multidimensional slices work in the same way, with multiple slices separated by commas.

For example:

```
In[24]: x2
Out[24]: array([[12, 5, 2, 4],
[ 7, 6, 8, 8],
[ 1, 6, 7, 7]])
In[25]: x2[:2, :3] # two rows, three columns
Out[25]: array([[12, 5, 2],
[ 7, 6, 8]])
In[26]: x2[:3, ::2] # all rows, every other column
Out[26]: array([[12, 2],
[ 7, 8],
[ 1, 7]])
```

Finally, subarray dimensions can even be reversed together:

```
In[27]: x2[::-1, ::-1]
Out[27]: array([[ 7, 7, 6, 1],
[ 8, 8, 6, 7],
[ 4, 2, 5, 12]])
```

Accessing array rows and columns.

One commonly

Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the reshape() method. For example, if you want to put the numbers

1 through 9 in a 3×3 grid, you can do the following:

```
In[38]: grid = np.arange(1, 10).reshape((3, 3))
print(grid)
[[1 2 3]
[4 5 6]
[7 8 9]]
```

You can do this with the reshape

method, or more easily by making use of the newaxis keyword within a slice operation:

```
In[39]: x = np.array([1, 2, 3])
# row vector via reshape
x.reshape((1, 3))
Out[39]: array([[1, 2, 3]])
```

Array Concatenation and Splitting

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished through the routines np.concatenate, np.vstack, and np.hstack. np.concatenate takes a tuple or list of arrays as its first argument

```
In[43]: x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
Out[43]: array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

```
In[44]: z = [99, 99, 99]
print(np.concatenate([x, y, z]))
[ 1 2 3 3 2 1 99 99 99]
```

np.concatenate can also be used for two-dimensional arrays:

```
In[45]: grid = np.array([[1, 2, 3],
[4, 5, 6]])
In[46]: # concatenate along the first axis
np.concatenate([grid, grid])
Out[46]: array([[1, 2, 3],
[4, 5, 6],
```

Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions np.split, np.hsplit, and np.vsplit. For each of these, we can pass a list of indices giving the split points:

```
In[50]: x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
[1 2 3] [99 99] [3 2 1]
```

Notice that *N* split points lead to *N + 1* subarrays. The related functions np.hsplit and np.vsplit are similar:

```
In[51]: grid = np.arange(16).reshape((4, 4))
grid
Out[51]: array([[ 0, 1, 2, 3],
[ 4, 5, 6, 7],
[ 8, 9, 10, 11],
[12, 13, 14, 15]])
In[52]: upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
[[0 1 2 3]
[4 5 6 7]]
```

Computation on NumPy Arrays: Universal Functions

Computation on NumPy arrays can be very fast, or it can be very slow. The key to making it fast is to use *vectorized* operations, generally implemented through NumPy's *universal functions* (ufuncs).

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Absolute value

Just as NumPy understands Python's built-in arithmetic operators, it also understands

Python's built-in absolute value function:

In[11]: x = np.array([-2, -1, 0, 1, 2])

abs(x)

Out[11]: array([2, 1, 0, 1, 2])

The corresponding NumPy ufunc is np.absolute, which is also available under the alias np.abs:

In**[12]: np.absolute(x)**

Out[12]: array([2, 1, 0, 1, 2])

In[13]: np.abs(x)

Out[13]: array([2, 1, 0, 1, 2])

This ufunc can also handle complex data, in which the absolute value returns the magnitude:

In[14]: x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])

np.abs(x)

Out[14]: array([5., 5., 2., 1.])

Trigonometric functions

NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions. We'll start by defining an array of angles:

```
In[15]: theta = np.linspace(0, np.pi, 3)
```

Now we can compute some trigonometric functions on these values:

```
In[16]: print("theta = ", theta)
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("tan(theta) = ", np.tan(theta))
theta = [ 0. 1.57079633 3.14159265]
sin(theta) = [ 0.00000000e+00 1.00000000e+00 1.22464680e-16]
cos(theta) = [ 1.00000000e+00 6.12323400e-17 -1.00000000e+00]
tan(theta) = [ 0.00000000e+00 1.63312394e+16 -1.22464680e-16]
```

Exponents and logarithms

Another common type of operation available in a numpy ufunc are the exponentials:

```
In[18]: x = [1, 2, 3]
Print("x =", x)
Print("e^x =", np.Exp(x))
Print("2^x =", np.Exp2(x))
Print("3^x =", np.Power(3, x))
X = [1, 2, 3]
E^x = [ 2.71828183 7.3890561 20.08553692]
2^x = [ 2. 4. 8.]
3^x = [ 3 9 27]
```

The inverse of the exponentials, the logarithms, are also available. The basic np.Log Gives the natural logarithm; if you prefer to compute the base-2 logarithm or the Base-10 logarithm, these are available as well:

```
In[19]: x = [1, 2, 4, 10]
Print("x =", x)
Print("ln(x) =", np.Log(x))
Print("log2(x) =", np.Log2(x))
Print("log10(x) =", np.Log10(x))
X = [1, 2, 4, 10]
Ln(x) = [ 0. 0.69314718 1.38629436 2.30258509]
Log2(x) = [ 0. 1. 2. 3.32192809]
```