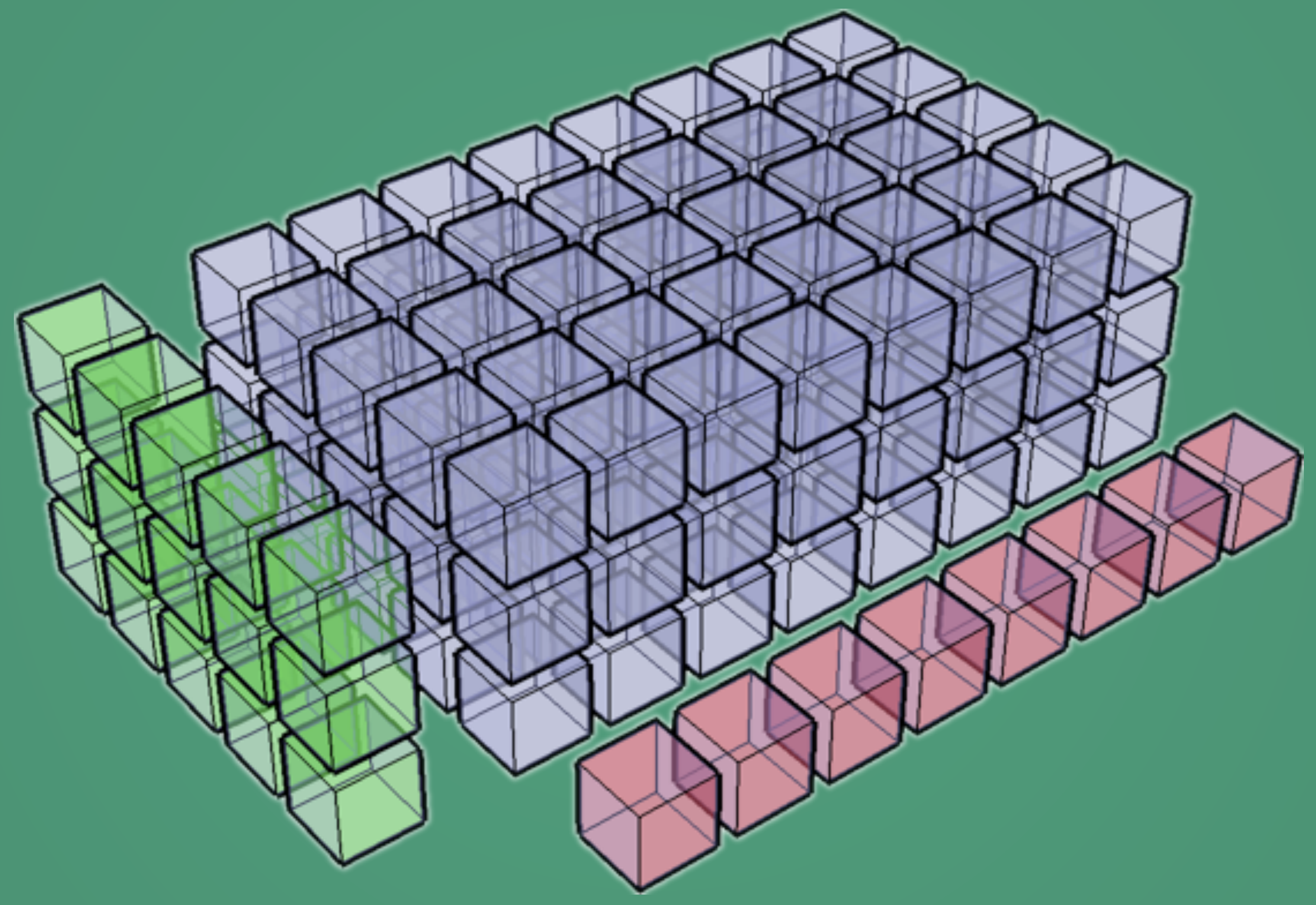
Overview of NumPy in Python



KnoxPy Meeting - knoxpy.org - April 7th, 2016

Gavin Wiggins

Scientific Python stack

NumPy - numerical computation package, defines operations on array and matrix types

SciPy - numerical libraries and toolboxes for signal processing, optimization, statistics, etc.

Matplotlib - a 2-D and 3-D plotting package

Pandas - high performance, easy to use data structures

SymPy - perform symbolic math and algebra

iPython - an interactive Python shell, Jupyter notebook formally iPython notebook

History of NumPy

- Python language not initially designed for numerical computing
- Group called Matrix-SIG formed in 1995 to define an array computing package where Jim Fulton created the Numeric matrix package
- Numarray (large arrays) written as more flexible version of Numeric (small arrays)
- Travis Oliphant developed NumPy in 2005 as a unified package containing features from Numarray and Numeric
- NumPy released in 2006 as part of the SciPy package

Installing NumPy

ANACONDA

Modern open source analytics platform
powered by Python

- Anaconda by Continuum Analytics is by far the easiest way to install Python 3 and the SciPy stack on Windows, Mac, and Linux machines

- https://www.continuum.io/downloads

Arrays

ndarray - multidimensional array class in NumPy

ndarray.shape - dimensions of the array with n rows and m columns such as (n, m)

ndarray.size - total number of elements in the array

ndarray.dtype - describes type of element in array

```python
import numpy as np

a = np.array([1, 2, 3, 4, 5])
# array([1, 2, 3, 4, 5])

b = np.array([[1, 2, 3],[4, 5, 6]])
# array([[1, 2, 3],
#         [4, 5, 6]])

b.size    # 6

b.shape   # (2, 3)
```

Arrays

np.zeros( )
creates an array full of zeros

np.ones( )
creates and array full of ones

np.empty( )
an empty array of random content

```
import numpy as np

np.zeros((3, 4))
# array([[ 0.,  0.,  0.,  0.],
#        [ 0.,  0.,  0.,  0.],
#        [ 0.,  0.,  0.,  0.]])

np.ones((2, 3, 4))
# array([[[ 1, 1, 1, 1],
#          [ 1, 1, 1, 1],
#          [ 1, 1, 1, 1]],
#         [[ 1, 1, 1, 1],
#          [ 1, 1, 1, 1],
#          [ 1, 1, 1, 1]]])

np.empty((2, 3))
# array([[ 3.73603959e-262, 6.02658058e-154, 6.55490914e-260],
#        [ 5.30498948e-313, 3.14673309e-307, 1.00000000e+000]])
```

Array vs List

np.array( )

- efficient memory usage
- vector and matrix operations
- built in functionality for FFTs, linear algebra, searching, statistics, etc.

list [ ]

- general purpose containers
- don't support vector operations
- type information stored for every element thus Python must execute type check for every operation

Array vs Matrix

np.array( )

- N-dimensional
- element-wise operations
- use np.dot() for matrix multiplication

np.matrix( )

- strictly two-dimensional
- matrix multiplication

```
array = np.array([1, 2, 3, 4, 5])
# array([1, 2, 3, 4, 5])

list = [1, 2, 3, 4, 5]
# [1, 2, 3, 4, 5]

matrix = np.matrix([1, 2, 3, 4, 5])
# matrix([[1, 2, 3, 4, 5]])
```

Range vs Arange vs Linspace

range( )

- immutable sequence type
- integers only

np.arange( )

- returns array of numbers
- integers and floats
- uses a step size

np.linspace( )

- returns array of numbers
- integers and floats
- uses number of samples

```
list(range(10))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

list(range(1, 10, 2))
# [1, 3, 5, 7, 9]

range(1.5, 9.5)
# error

np.arange(10)
# array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

np.arange(1, 10, 2)
# array([1, 3, 5, 7, 9])

np.arange(1.5, 9.5)
# array([ 1.5,  2.5,  3.5,  4.5,  5.5,  6.5,  7.5,  8.5])

np.linspace(1, 10, 5)
# array([  1.  ,   3.25,   5.5 ,   7.75,  10.  ])
```

Operations

- Arithmetic operations on arrays apply to each element
- Product operator * operates on each element of a NumPy array
- Use **np.dot( )** to calculate the matrix product
- Operations such as **+=** and ***=** act in place to modify an existing array rather than create a new one
- The **axis** parameter applies operation to a specific axis of the array

```
a = np.array([[1,2,3,4],[5,6,7,8]])
# array([[1, 2, 3, 4],
#        [5, 6, 7, 8]])

a.sum(axis=0)
# array([ 6,  8, 10, 12])

a.sum(axis=1)
# array([10, 26])
```

Indexing, slicing, iterating

- Arrays can be indexed, sliced, iterated much like lists and other sequence types in Python
- As with Python lists, slicing in NumPy can be accomplished with the colon (**:**) syntax
- Colon instances (**:**) can be replaced with dots (**...**)

```
a = np.array([1, 2, 3, 4, 5])
# array([1, 2, 3, 4, 5])

a[1:3]
# array([2, 3])

a[-1]
# 5

a[0:2] = 9

a
# array([9, 9, 3, 4, 5])
```

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

b[:, 1]
# array([ 2,  6, 10])

b[-1]
# array([ 9, 10, 11, 12])

b[-1, :]
# array([ 9, 10, 11, 12])

b[-1, ...]
# array([ 9, 10, 11, 12])

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

Copies and views

- Assignments make no copy of array objects or of their data
- Mutable objects are passed as references, so function calls make no copy
- The **view** method creates a new array object from the original array data
- The **copy** method makes a complete copy (deep copy) of the array and its data

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

b = a

b[0] = 99

b
# array([[99, 99, 99, 99],
#        [ 5,  6,  7,  8]])

a
# array([[99, 99, 99, 99],
#        [ 5,  6,  7,  8]])
```

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

c = a.copy()

c[0] = 99

c
# array([[99, 99, 99, 99],
#        [ 5,  6,  7,  8]])

a
# array([[1, 2, 3, 4],
#        [5, 6, 7, 8]])
```

Why use NumPy?

+

What makes Python fast is what makes Python slow.

Interpreted, dynamically typed, high-level.

NumPy can speed up your code

universal functions (ufuncs)

broadcasting

aggregations

slicing, masking, indexing

Broadcasting

- Broadcasting is used in NumPy to decide how to handle different shaped arrays
- Broadcasting makes your code more concise and fast
- Functions that support broadcasting are known as universal functions (ufuncs)

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

b = np.array([10, 0, 10])
# array([10,  0, 10])

c = a + b
# array([[11,  2, 13],
#        [14,  5, 16],
#        [17,  8, 19],
#        [20, 11, 22]])
```

Universal functions (ufuncs)

- Ufuncs operate element-by-element on an array and produce an array as output
- A vectorized wrapper for a function
- Support array broadcasting and type casting
- Many ufuncs are implemented as compiled C code
- Math functions such as +, -, *, /, np.sin, np.cos, np.tan, np.log, np.exp, etc.
- Currently more than 60 ufuncs defined in NumPy

```
n = 1000000

a = list(range(n))

%timeit [x * 2 for x in a]
10 loops, best of 3: 96.8 ms per loop

b = np.array(a)

%timeit b * 2
1000 loops, best of 3: 1.44 ms per loop
```

NumPy speedup ~67x

Aggregations

- Aggregations are functions that summarize the values (elements) in an array
- Math routines such as np.sum( ), np.mean( ), np.min( ), np.max( ), np.prod( ), etc.

```
n = 1000000

a = list(range(n))

sum(a)
# 499999500000

%timeit sum(a)
100 loops, best of 3: 12.9 ms per loop

b = np.array(a)

b.sum()
# 499999500000

%timeit b.sum()
1000 loops, best of 3: 717 µs per loop
```

```
n = 1000000

a = [np.random.random() for i in range(n)]

min(a)
# 5.59600061511567e-07

%timeit min(a)
10 loops, best of 3: 25.4 ms per loop

b = np.array(a)

b.min()
# 5.5960006151156705e-07

%timeit b.min()
1000 loops, best of 3: 531 µs per loop
```

Slicing, masking, indexing

- NumPy arrays and Python lists support slicing and indexing
- Masking in NumPy is indexing with booleans, a boolean array
- Intricate (fancy) indexing possible with NumPy arrays

```
a = np.array([10, 11, 12, 13, 14])

x = [1, 3, 4]

a[x]
# array([11, 13, 14])
```

```
c = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
# array([[1, 2, 3, 4],
#        [5, 6, 7, 8]])

c[c.sum(axis = 1) > 4, 1:]
# array([[2, 3, 4],
#        [6, 7, 8]])
```

```
b = np.array([1, 2, 3, 4, 5, 6, 7])

mask = (b < 2) | (b > 5)
# array([ True, False, False, False, False,  True,  True])

b[mask]
# array([1, 6, 7])
```

Limitless possibilities! 🤩

Beyond NumPy with Dask

- Dask is a flexible parallel computing library
- Dask provides parallelized NumPy array and Pandas DataFrame objects
- Scales up to clusters of 100s of nodes or run multiple cores on a single laptop

```
import numpy as np
import dask.array as da
from multiprocessing import cpu_count

n = 10**8

a = np.random.rand(n)*10

%timeit a.sum()
10 loops, best of 3: 63.7 ms per loop

b = da.from_array(a, chunks=len(a)/cpu_count())

%timeit b.sum().compute()
10 loops, best of 3: 41.9 ms per loop
```

Further speed improvements

Cython - run C code within Python

Numba - compile just-in-time functions

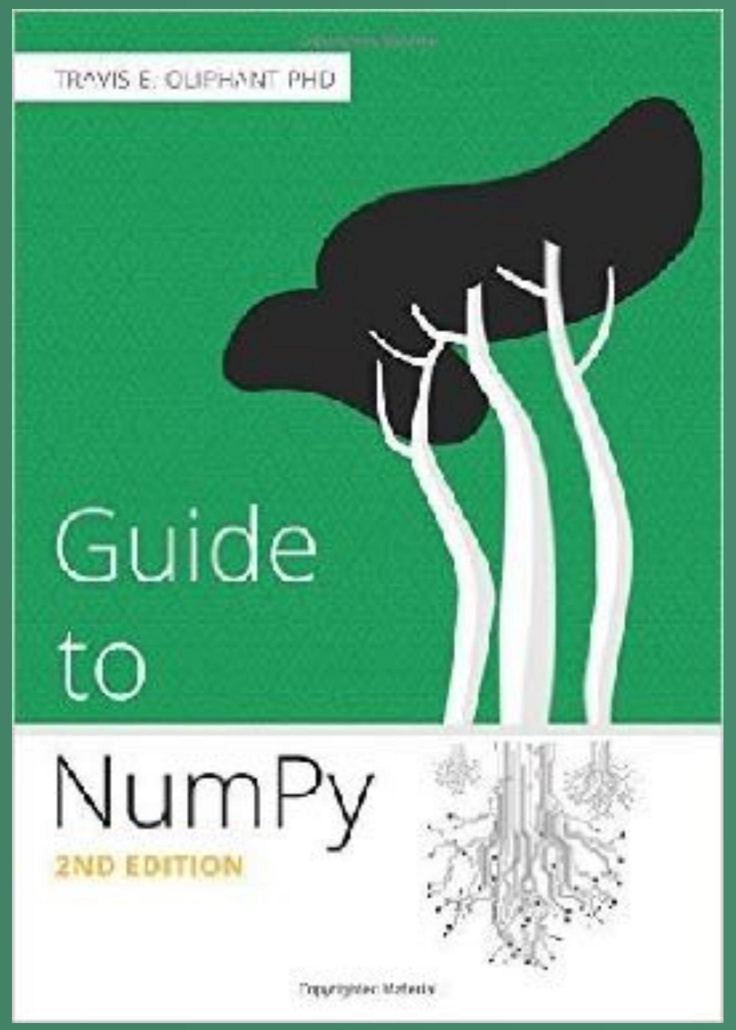
Blaze - high-level interface for databases

Dask - parallel computing, block algorithms

PyPy - just-in-time compiled Python

Blaze pypy

Resources



- Documentation and tutorials at numpy.org
- Python NumPy Tutorial from Stanford CS231n class
- Practical Numerical Methods with Python from GW Open edX MAE 6286 online course
- Guide to NumPy: 2nd edition book by Travis Oliphant
- Losing Your Loops: Fast Numerical Computing with NumPy from PyCon 2015 conference by Jake VanderPlas

"Why don't you use C++ instead of Python? It's so much faster!"

"Why don't you commute by airplane instead of by car? It's so much faster!"

from a tweet by Jake VanderPlas