Serially Fast Python

HPC Python

Todd Evans
rtevans@tacc.utexas.edu

April 17, 2015

Table of contents

NumPy

Python Objects

- High-level number objects: integers, floating point
- Containers: lists, dictionaries

NumPy

- Extension package for multi-dimensional arrays
- Closer to hardware $\rightarrow$ efficiency
- Designed for scientific computation

NumPy and Python List

Python List

```
In [1]: import numpy as np
In [2]: list = range(100000)

In [3]: %timeit [i**2 for i in list]
100 loops, best of 3: 6.43 ms per loop

In [4]: array = np.arange(100000)

In [5]: %timeit array**2
1000 loops, best of 3: 97.7 us per loop
```

Why so Slow?

- Dynamic typing requires lots of metadata around variables
- Potentially inefficient memory access
- Interpreted instead of compiled

What can you do?

- Make an object that has a single type and continuous storage
- Implement common functionality into that object to iterate in C

NumPy Features

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = np.array([2.0, 2.0, 2.0])
>>> a * b
array([ 2.,  4.,  6.])
```

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a*b
array([ 2.,  4.,  6.])
```

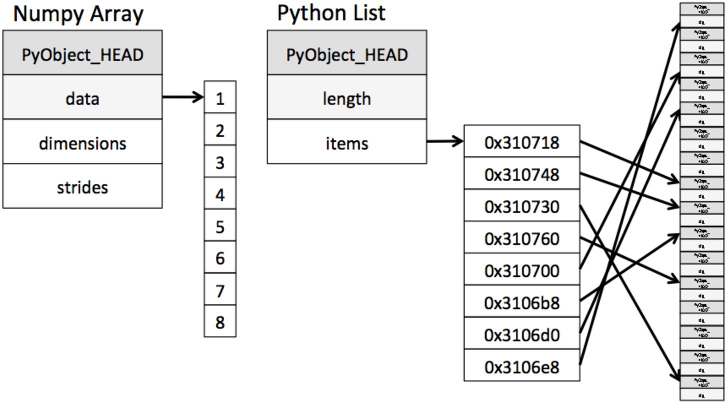
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

Array Object

What makes an array so much faster?

- Data layout
  - homogenous: every item takes up the same size block of memory
  - single data-type objects
  - powerful array scalar types
- universal function (ufuncs)
  - function that operates on ndarrays in an element-by-element fashion
  - vectorized wrapper for a function
  - built-in functions are implemented in compiled C code

Data Layout



- Numpy: contiguous data buffer of values
- Python: contiguous buffer of pointers

ufuncs

- function that operates on ndarrays in an element-by-element fashion
- vectorized wrapper for a function
- built-in functions are implemented in compiled C code

Python function - ufunc

```
In [1]: import numpy as np

In [2]: import math

In [3]: arr = np.arange(100000)

In [4]: %timeit[math.sin(i) for i in arr]
10 loops, best of 3: 18.3 ms per loop

In [5]: %timeit np.sin(arr)
100 loops, best of 3: 1.77 ms per loop

In [6]: %timeit[math.sin(i)**2 for i in arr]
10 loops, best of 3: 27.3 ms per loop

In [7]: %timeit np.sin(arr)**2
100 loops, best of 3: 1.83 ms per loop
```

Mathematical functions

How to Create an Array

examples/3_numpy/array.py

```
import numpy as np
a = np.array([2, 3, 12])  # Create from list
a = np.arange(10)        # 0, 1, 2, 3, 4,..., 9
b = np.arange (0,10,2) # start, end (exclusive), step. 0, 2, 4, 6, 8
#By number of points (start, end, num. points)
a = np.linspace(0,1,5) #0, 0.25, 0.50, 0.75, 1.0
a = np.linspace(0,1,5,endpoint=False) #0, 0.2, 0.4, 0.6, 0.8
#Useful arrays
a = np.ones((4,4))
a = np.zeros((3,3))
a = np.diag(np.ones(3))
a = np.eye(3)
#with random numbers
np.random.seed(1111)   #sets the random seed
a = np.random.rand(4)  #uniform in [0,1]
b = np.random.randn(4) #Gaussian
#uninitialized
a = np.empty((3,3))
#resize
a = np.zeros(10)
a = np.resize(a, 20)
```

Data Types

`bool` `string`	`int`	`float`	`complex`
	`int8`	`float16`	`complex64`
	`int16`	`float32`	`complex128`
	`int32`	`float64`	
	`int64`		
	`uint8`		
	`uint16`		
	`uint32`		
	`uint64`		

Data Types

Basic

```
In [1]: import numpy as np

In [2]: a = np.array([1, 2, 3])

In [3]: a.dtype
Out[3]: dtype('int64')

In [4]: b = np.array([1., 2., 3.])

In [5]: b.dtype
Out[5]: dtype('float64')
```

Other

```
In [6]: c = np.array([1, 2, 3], dtype=float)

In [7]: c.dtype
Out[7]: dtype('float64')

In [8]: d = np.array([True, False, True])

In [9]: d.dtype
Out[9]: dtype('bool')

In [10]: e = np.array([1+2j, 3+4j, 5+6*1j])

In [11]: e.dtype
Out[11]: dtype('complex128')

In [12]: f = np.array(['Bonjour', 'Hello', 'Hola'])

In [13]: f.dtype
Out[13]: dtype('S7')  #Strings of max. 7 characters
```

Linear Algebra

Linear Algebra dot Function

```
In [1]: import numpy as np

In [2]: np.dot(np.arange(3), np.arange(3))
Out[2]: 5

In [3]: np.dot(np.arange(9).reshape(3,3), np.arange(3))
Out[3]: array([ 5, 14, 23])

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

Automatic Offload (AO)

- Feature of Intel Math Kernel Library (MKL)[1]
  - growing list of computationally intensive functions
  - xGEMM and variants; also LU, QR, Cholesky
  - kicks in at appropriate size thresholds (e.g. SGEMM: (M,N,K) = (2048, 2048, 256))
  - Functions with AO
- Essentially no programmer action required
  - more than offload: work division across host and MIC
  - Tips for using MKL on Phi

[1] For more information refer to https://www.tacc.utexas.edu/resources/software/ao

Automatic Offload

Set at least three environment variables before launching your code

```
export MKL_MIC_ENABLE=1
export OMP_NUM_THREADS=16
export MIC_OMP_NUM_THREADS=240
```

- Other environment variables provide additional fine-grained control over host-MIC work division
- MKL documentation
- Intel MKL Automatic Offload enabled functions

Automatic Offload

`examples/3_offload/my_dgemm.py`

Important Variables

```
OMP_NUM_THREADS (1..16)
MKL_MIC_ENABLE (0, 1)
MIC_OMP_NUM_THREADS (1..240)
OFFLOAD_REPORT (0..2)
```

Data IO (broad categories)

- ascii
  - Simple structure
  - Human-readable
  - Maximum portability
  - Inefficent use of space
  - Examples: .csv, .txt, .dat, ...
- database
  - Complex (relational?) structure
  - Need special tools for write/read
  - Often have large computational/storage overheads
  - Examples: .xml, .db, .json, ...
- binary
  - Custom structure
  - Need special tools (+headers/information!) for write/read
  - Minimal computational/storage overheads
  - Examples: .bin, ...

Read/Write ascii

- Pure python
- Ascii data tools in numpy
- Binary tools in numpy

examples/31_numpy_data/data_creation.py

```
import numpy as np
import random as r

def data_creation(number_of_lines, array_length):
    data = []
    for k in range(number_of_lines):
        d = []
        for i in range(array_length):
            a = r.randint(1,10)
            d.append(a)
        data.append(d)
    return data

if __name__ == '__main__':
  number_of_lines = 6
  array_length = 4
  np_data = data_creation(number_of_lines,array_length)
```

Write ascii with numpy

- numpy.savetxt() makes it extremely easy to save tabular data

examples/31_numpy_data/file_writing_savetxt.py

```
import numpy as np
import random as r
from data_creation import data_creation

number_of_lines = 100000
array_length = 4
np_data = data_creation(number_of_lines,array_length)
np.savetxt("ascii_data_example.dat", np_data)
```

some file properties

```
In [1]: head -n 3 ascii_data_example.dat
4.000000000000000000e+00 2.000000000000000000e+00 7.000000000000000000e+00
      3.000000000000000000e+00
8.000000000000000000e+00 1.000000000000000000e+01 3.000000000000000000e+00
      4.000000000000000000e+00
...

In[2]: ls -lh ascii_data_example.dat
-rw-r--r--  1 alamas  staff   9.5M Apr  1 13:19 ascii_data_example.dat
```

Read ascii with numpy

- numpy.loadtxt() is the converse of savetxt()

examples/31_numpy_data/file_reading_loadtxt.py

```
import numpy as np

np_data = np.loadtxt("ascii_data_example.dat")
```

some data properties

```
In[1]: import file_reading_loadtxt as fr

In[2]: np_data = fr.np_data

In[3]: type(np_data)
Out[3]: numpy.ndarray

In[4]: np_data.shape
Out[4]: (100000, 4)
```

Write binary with numpy

- np.save(f,np_data) saves a numpy array in a numpy binary format

examples/31_numpy_data/file_writing_save.py

```
import numpy as np
import random as r
from data_creation import data_creation

number_of_lines = 100000
array_length = 4
np_data = data_creation(number_of_lines,array_length)
np.save("bin_data_example", np_data)
print type(np_data)
```

some file properties

```
In[1] head bin_data_example.npy
?NUMPYF{'descr': '<i8', 'fortran_order': False, 'shape': (100000, 4), }

In[2]: ls -lh bin_data_example.npy
-rw-r--r--  1 alamas  staff   3.1M Apr  1 14:26 bin_data_example.npy
```

Read binary with numpy

- np.load(f) is the converse of np.save()

examples/31_numpy_data/file_reading_load.py

```python
import numpy as np

np_data = np.loadtxt("ascii_data_example.dat")
```

some data properties

```
In[1]: import file_reading_load as fr

In[2]: np_data = fr.np_data

In[3]: type(np_data)
Out[3]: numpy.ndarray

In[4]: np_data.shape
Out[4]: (100000, 4)
```

Why bother with binary?

Almost identical effort to deal with ascii or binary . . . so why do it?

- Storage space
- Speed
- Load on file system

Let's have a closer look . . .

Profiling ascii vs binary

examples/31_numpy_data/profiling_output.txt

```
In[1] ./profiling_numpy_data.py
Generating data ...
Storing as a text file ...
         1000174 function calls in 4.690 seconds
(...)

Storing as a numpy binary with headers ...
         334 function calls in 0.050 seconds
(...)

Reading a txt file via numpy
         10000264 function calls (9000264 primitive calls) in 7.178 seconds
(...)

Reading a numpy binary without headers ...
         7831 function calls (5441 primitive calls) in 0.028 seconds
(...)
```

ascii vs binary

- The binary version is x100 faster!
- Ascii access requires x1000 more primitive calls!
- About x3 space savings

- Lots of python codes spend significant time in IO
- Reserve ascii IO for initial inputs and final outputs
- Keep intermediate results in efficient formats
- Use profiling tools not only for "pure computation"

License