Lose Your Loops with NumPy

Thoth Gunter

Python @ CERN

About me

- 3rd Graduate Student from Northwestern University
- working on CMS
- python fanatic

Contact me:

@xcopyfgc

Slides at:

https://github.com/TKGgunter/ (https://github.com/TKGgunter/)

Heavily based off of Jake VanderPlas' PyCon 2015

Losing your Loops Fast Numerical Computing with NumPy (https://www.youtube.com/watch?v=EEUXKG97YRw&list=LLABoYor0BYe94swDr9dQpMg&index=3)

Obligatory Python 3 shoutout

Python is Fast

No Really Fast

For Writing, Testing and Developing Code

```python
# Python
print( "Hello World!" )
```

```cpp
//C++
#include <iostream>

int main()
{
  std::cout << "Hello World!";
}
```

Why is python so fast?

Development speed is made possible by being high level, interpreted, and dynamically typed.

In [1]:
```
#timeit
array = [i for i in range( int(10E6) ) ]
def func( array ):
    for i in enumerate(array):
        array[i[0]] = i[1] + 1
%timeit func( array )
```

```
1 loops, best of 3: 2.1 s per loop
```

```
#include <iostream>
#include <ctime>
int main()
{
  int arr_len = 10E6;
  int array[arr_len];
  for(int i = 0; i < arr_len; i++){ array[i] = i; }

  clock_t begin = clock();
  for(int i = 0; i < arr_len; i++){ array[i] += 1; }
  clock_t end = clock();
  double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
  std::cout << elapsed_secs << std::endl;
}
```

0.0196 s = time

1.93 sec vs 0.0196 sec ~ 100x difference

Python's strength is also its weakness

high level, interpreted, dynamically typed...repeated overhead is costly in large loops

In [ ]:
```
for i in range( int(10E6) ):
    array[i] = array[i] + 1
```

So what do we do?

Where NumPy comes in

"NumPy is the fundamental package for scientific computing with Python." -- numpy.org

NumPy is Python's standard matrix computation library from which most other computational packaged are built on top of. -- Thoth Gunter

- It works by pushing repeated operations into a C++ compiled layer

Stategies to use NumPy its fullest

- Ufuncs
- Aggregations
- Broadcasting
- Slicing/Masking/Indexing

Ufunc

Ufuncs are universal functions that act on n-dimensional arrays in an element-by-element fashion.

All arithmatic functions are ufuncs. [+,-,*, /, //, %]

In [3]:
```
# With Python list
a =  [1 , 2, 3, 4, 5, 6, 7, 8, 9]
b = [ i + 1 for i in a ]
print( b )
```

```
[2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [4]:
```
# With NumPy arrays
import numpy as np
a = np.array( [1 , 2, 3, 4, 5, 6, 7, 8, 9] )
b = a + 1
print( b )
```

```
[ 2  3  4  5  6  7  8  9 10]
```

In [5]:
```
def func(array):
    array = array + 1

array = np.arange( 10E6 )
%timeit func( array )
```

```
10 loops, best of 3: 29.2 ms per loop
```

1.92 s per loop vs .0292 s ~ 70x

Ufunc

In addition NumPy supports:

- logs and exp
- trig
- bit-wise
- comparision

For even more scipy.special.*

In [6]:
```
a = np.arange( 6 ).reshape((2, 3))
np.sin( a )
```

Out[6]:
```
array([[ 0.        ,  0.84147098,  0.90929743],
       [ 0.14112001, -0.7568025 , -0.95892427]])
```

The full list can be found here http://docs.scipy.org/doc/numpy-1.10.1/reference/ufuncs.html (http://docs.scipy.org/doc/numpy-1.10.1/reference/ufuncs.html)

Aggregation

Aggregations are functions that summarize the values of an array.

Min, max, sum, mean are some examples.

In [7]:
```
from random import random

a = [random() for i in range(int(10E6))]

%timeit max(a)
```

```
1 loops, best of 3: 236 ms per loop
```

In [8]:
```
a = np.array(a)
%timeit np.max(a) # or a.max()
```

```
100 loops, best of 3: 5.16 ms per loop
```

~50x speed up

Using Aggregation with Multidimensional Arrays

In [9]:
```
a = np.random.randint( 0, 10, (2,3))
a
```

Out[9]:
```
array([[0, 8, 6],
       [0, 9, 5]])
```

In [10]:
```
a.sum(axis=0)
```

Out[10]:
```
array([ 0, 17, 11])
```

In [11]:
```
a.sum(axis=1)
```

Out[11]:
```
array([14, 14])
```

There are a number of aggregation functions

np.prod, np.sum, np.nansum, np.absolute, np.sign, np.gradient...

The full list can be found http://docs.scipy.org/doc/numpy-1.10.1/reference/routines.math.html (http://docs.scipy.org/doc/numpy-1.10.1/reference/routines.math.html)

Broadcasting

Broadcasting is the act of changing the dimensionality of your arrays with the goal of allowing arrays of different dimensionality to interact.

In [12]:
```
a = np.array([ 1, 2, 3])
np.add(a, 1)
```

Out[12]:
```
array([2, 3, 4])
```

In [13]:
```
# np.add(a, 1) is actually
b = np.array([1, 1, 1])
np.add(a , b)
```

Out[13]:
```
array([2, 3, 4])
```

So what happens with we have arrays of different dimensionality

In [14]:
```
a.reshape(3,1)
```

Out[14]:
```
array([[1],
       [2],
       [3]])
```

In [15]:
```
b = np.array([1, 5, 2])
b.reshape(1,3)
```

Out[15]:
```
array([[1, 5, 2]])
```

In [16]:
```
a.reshape(3,1) + b.reshape(1,3)
```

Out[16]:
```
array([[2, 6, 3],
       [3, 7, 4],
       [4, 8, 5]])
```

Rules of Broadcasting within NumPy

1. If arrays' dimensionality differ, then the dimensionality is changed so that they match
2. If an arrays' shapes are different then an error is thrown

In [17]:
```
np.array([1,2,3]) < np.array([5,6,7,8])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-17-23f89a86c5d0> in <module>()
      1 #
----> 2 np.array([1,2,3]) < np.array([5,6,7,8])

ValueError: operands could not be broadcast together with shapes (3,) (4,)
```

In [18]:
```
np.array([1,2,3]).reshape(3,1) < np.array([1,2,3,4]).reshape(1,4)
```

Out[18]:
```
array([[False,  True,  True,  True],
       [False, False,  True,  True],
       [False, False, False,  True]], dtype=bool)
```

Slicing/Masking/Indexing

Slicing/Masking/Indexing/ is the act of obtaining some subselection of an array

Slicing

Slicing is the act of obtaining some subsection of an array, retraining the original order and structure

In [19]:
```
a = np.array([[0, 1, 2],[3, 4, 5]])
a
```

Out[19]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [20]:
```
a[:]
```

Out[20]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [21]:
```
a[:,:2]
```

Out[21]:
```
array([[0, 1],
       [3, 4]])
```

Masking

Is the use of boolean arrays of equal dimensionality to the array of interest in order to obtain the values of said array where the boolean elements are true.

In [25]:
```
print( a )
mask = np.array([[True,False,True],[False,True,False]])
a[mask]
```

```
[[0 1 2]
 [3 4 5]]
```

Out[25]:
```
array([0, 2, 4])
```

In [22]:
```
print(a)
a < 3
```

```
[[0 1 2]
 [3 4 5]]
```

Out[22]:
```
array([[ True,  True,  True],
       [False, False, False]], dtype=bool)
```

In [23]:
```
a[a < 3]
```

Out[23]:
```
array([0, 1, 2])
```

Indexing

Is the act of passing a prefined list of array corrdinates to an array which then returns an array of stored values associated with those corrdinates.

In [26]:
```
array = np.random.randint(0,10, (10))
array
```

Out[26]:
```
array([4, 8, 1, 9, 3, 3, 9, 4, 4, 4])
```

In [27]:
```
array[[2,4,-1]]
```

Out[27]:
```
array([1, 3, 4])
```

In [57]:
```
array = array.reshape(2,5)
print( array )
array[[1,0],[1,3]]
```

```
[[9 0 7 7 4]
 [4 6 1 4 1]]
```

Out[57]:
```
array([6, 7])
```

Examples

You can mix and match for ... results

In [29]:
```
array = np.random.randint(0,10, (10)).reshape(2,5)
print(array)
array[:,[4,3]]
```

```
[[9 0 7 7 4]
 [4 6 1 4 1]]
```

Out[29]:
```
array([[4, 7],
       [1, 4]])
```

In [30]:
```
mask = (array > array.mean()) & (array < np.random.randint(5,15, (10)).reshape(2,5
))
mask
```

Out[30]:
```
array([[False, False, False,  True, False],
       [False,  True, False, False, False]], dtype=bool)
```

In [31]:
```
array[mask]
```

Out[31]:
```
array([7, 6])
```

Conclusion

Python is fast in development, but can also be fast computationally through the correct use of:

- Ufuncs
- Aggregations
- Broadcasting
- Slicing/Masking/Indexing