High Performance Python

Week 1: Profiling

The Strengths of Python

- Fast to Write
- Easy to Read
- Batteries Included

The Weaknesses of Python

- Slow to Execute
- Global Interpreter Lock (GIL)

How to Make Python Fast

- Array Computations
- Offloading Functions to C or Fortran
- Multiple Processors

Warning: Don't optimize before the code works
All of these techniques will make your code more difficult for yourself and others to read and understand.

We also should be (but aren't) unit testing

The Problem!

- How do we go about finding the slow parts of our code so we can speed them up?
- Once we find a slow part of the code, how to we make sure we change it so that it is faster?

The Answer

- Time functions using the built in time module

  (good)

- Time functions using ipython's magic timeit functionality (or using the built in timeit module)

  (better)

- Profile the codes execution time using a line profiler

  (best)

Example

```
def smooth1( ary,M ):
    #done in place
    for i in range(M):
        ary[1:-1] += ary[:-2]  + ary[2:]
        ary       /= 3
        ary[0]     = 2*ary[0]  + ary[1]/3
        ary[-1]    = 2*ary[-1] + ary[-2]/3
```

```
def smooth2( ary, M):
    #done in place
    for i in range(M):
        ary += np.append( ary[0] ,ary[:-1]) + \
               np.append( ary[1:],ary[-1] )
        ary /= 3
```

Using time.time

```
def smooth1( ary,M ):
    #done in place
    startTime = time.time()
    for i in range(M):
        ary[1:-1] += ary[:-2]  + ary[2:]
        ary       /= 3
        ary[0]     = 2*ary[0]  + ary[1]/3
        ary[-1]    = 2*ary[-1] + ary[-2]/3
    print time.time()-startTime


def smooth2( ary, M):
    #done in place
    startTime = time.time()
    for i in range(M):
        ary += np.append( ary[0] ,ary[:-1]) + \
               np.append( ary[1:],ary[-1] )
        ary /= 3
    print time.time()-startTime
```

iPython Shell

Using timeit in ipython

```
In [93]: %timeit( smooth.smooth1(x.copy(),M) )
1000 loops, best of 3: 1.78 ms per loop

In [94]: %timeit( smooth.smooth2(x.copy(),M) )
100 loops, best of 3: 2.18 ms per loop
```

Profiling

- For more complex problems, time.time and timeit quickly become limiting.
- This is where using a code profiler comes in.

cProfile

- Python has cProfile built in

```
$ python -m cProfile -s cumulative smooth.py | less

0.000411987304688
0.000672101974487
         11157 function calls (10968 primitive calls) in 0.068 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.069    0.069 smooth.py:2(<module>)
        1    0.001    0.001    0.067    0.067 __init__.py:106(<module>)
        1    0.000    0.000    0.035    0.035 add_newdocs.py:9(<module>)
        2    0.002    0.001    0.028    0.014 __init__.py:1(<module>)
        1    0.001    0.001    0.023    0.023 __init__.py:15(<module>)
        2    0.004    0.002    0.019    0.010 __init__.py:2(<module>)
```

More info can be found at http://docs.python.org/2/library/profile.html

cProfile Switches

Argument	Meaning
'calls'	Call count
'cumulative', 'cumtime'	Cumulative time
'file', 'filename', 'module'	File name
'ncalls'	Call count
'pcalls'	Primitive call count
'line'	Line number
'name'	Function name
'nfl'	Name/file/name
'stdname'	Standard name
'time', 'tottime'	Internal time

A Better Way (line_profiler)

Step 1: Add decorators

```
@profile
def smooth1( ary,M ):
```

Step 2: Profle

```
$ kernprof.py -l smooth.py
Wrote profile results to smooth.py.lprof
$ python -m line_profiler smooth.py.lprof | less
```

A Better Way (line_profiler)

```
Timer unit: 1e-06 s

File: smooth.py
Function: smooth1 at line 5
Total time: 0.00058 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     5                                           @profile
     6                                           def smooth1( ary,M ):
     7                                               #done in place
     8         1            7      7.0      1.2      startTime = time
     9        17           15      0.9      2.6      for i in range(M
    10        16          184     11.5     31.7          ary[1:-1
    11        16          188     11.8     32.4          ary
    12        16           86      5.4     14.8          ary[0]
    13        16           58      3.6     10.0          ary[-1]
    14         1           42     42.0      7.2      print time.time(
```

Exercise, cluster

- cluster.py works, but is slow
- Run line_profiler on cluster.py
- Find any bottle necks, and fix them

cluster_solution.py is my solution to the problem

We'll use this script again to further optimize it