# System Development with Python: Week 8

Christopher Barker

UW Continuing Education

May 14, 2013

# Table of Contents

Performance Testing

"Premature optimization is the root of all evil"

– Donald Knuth

## Optimizing

Optimizing procedure:

1. Get it right.
2. Test it's right.
3. Profile if slow.
4. Optimise (the bits that matter)
5. Repeat from 2 (until it's good enough)

## Optimizing

Anecdote:

I once optimized a portion of the start-up routine for a oil spill
modeling project – numpy+custom C extensions

The portion I worked on I sped up by a factor of 100

That made the start-up time twice as fast

But the start up took 20 seconds (was 40), and the whole run
took hours.

It was a total waste of my time (though I learned a lot)

## Profiling/timing

You can't optimize your code without knowing where the bottlenecks are.

Smarter people than me have said they they are almost always wrong when they try to logically determine where the slow code is. (I know I am)

... and how to speed it up

Profiling: finding where the bottlenecks are

Timing: Comparing differnet implimentations

## time.clock()

The really easy way:

```
import time

start = time.clock()
  ... do_some_stuff ...
print "It took %f seconds to run"%(time.clock - start)
```

It works, it's easy, and it gives a gross approximation

(use time.clock(), rather than time.time(), unless you want to time netwoek access, etc.)

## timeit

The good way:

```
import timeit

timeit.timeit( statement, setup=some_stuff)
```

It's kind of a pain, but gives meaningful results.
(can also be called on the command line)

```
http://docs.python.org/library/timeit.html
```

(code/timing.py)

## %timeit

The easy **and** good way:
iPython:

```
In [52]: import timing

In [53]: %timeit timing.primes_stupid(5)
100000 loops, best of 3: 10.9 us per loop
```

Takes care of the setup/namespace and number of iterations for you

(demo)

http://ipython.org/ipython-doc/dev/interactive/
tutorial.html

## timing

NOTE:

All these timing methods depend a lot on hardware, system load, etc.

So really only good for relative timing – comparing one method to another – on the same machine, under same load.

## Profiling

A profiler is a tool that describes the run time performance of a program, providing a variety of statistics

Helpful when you don't yet know where your bottlenecks are

The python profiler

```
python -m cProfile profile_example.py
```

spews some stats

http://docs.python.org/library/profile.html

## python profiler

What you get:

ncalls the number of calls.

tottime the total time spent in the given function (and excluding time made in calls to sub-functions),

percall the quotient of tottime divided by ncalls

cumtime the total time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.

percall the quotient of cumtime divided by primitive calls

(demo: python -m cProfile profile_example.py)

## python profiler

You can also dump to a file:

```
$ python -m cProfile -o profile_dump profile_example.py
```

This gives you a binary file you can examine with pstats:

```
demo: $ python -m pstats
```

## pstats

### Running pstats

```
$
$ python -m pstats
Welcome to the profile statistics browser.
% read profile_dump
profile_dump% stats
Wed Aug 29 16:21:39 2012    profile_dump

         51403 function calls in 0.032 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(fu
    51200    0.006    0.000    0.006    0.000 {method 'append' o
        1    0.000    0.000    0.032    0.032 profile_example.py
        1    0.001    0.001    0.032    0.032 profile_example.py
      100    0.022    0.000    0.027    0.000 profile_example.py
```

## pstats commands

Commands:

| | |
|---:|---|
| help | help on pstats or particular command |
| stats | print the profile statistics |
| sort | sort by various data fields |
| strip | strips the leading path info from file names |
| callers | Print callers statistics |
| callees | Print callees statistics |
| quit | quits |

Each has options to customize output

automating profile stats

cProfile and pstats are also modules

So you can script collection of profiles and stats

http://docs.python.org/library/profile.html

## "Run Snake Run"

For a visual look at your profiling results:

`http://www.vrplumber.com/programming/runsnakerun/`

(pretty cool stuff!)

## line profiler

```
$ pip install line_profile
```

decorate the function you want to profile:

```
@profile
def primes_stupid(N):
    ...
```

decorate the function you want to profile:
```
$ kernprof.py -l -v line_prof_example.py
```

http://pythonhosted.org/line_profiler/

http:
//www.huyng.com/posts/python-performance-analysis/

## "big O" notation

Computer scientists describe the efficency of an
algorithm in "big O" notation.
It describes how much longer a algorithm takes as a
function of how much data it is working with

$O(1)$ time stays the same regardless of how much data
(adding to dicts)

$O(n)$ goes up linearly with number of items
(searching lists)

$O(n^2)$ goes up quadratically with number of items

$O(log(n))$ goes up with the log of how many items
(bisection search)

Choosing the right data structure / algorithm is key.
http://wiki.python.org/moin/TimeComplexity

## Performance Tips

Some common python performance issues:

- function calls can be slow(ish) – sometimes worth in-lining.
- looking up names – particular global ones:
  can be worth making local copies
- looping: list comps, maps, etc **can** be faster.

```
http:
//wiki.python.org/moin/PythonSpeed/PerformanceTips/
```

(some nifty profiling tools described there, too)

## Memory

Don't forget memory:

Processors are fast

It often takes longer to push the memory around than do the computation

So keep in in mind for big data sets:

- Use efficient data structures: `array.array`, `numpy`
- use generators, rather than lists: `xrange`, etc.
- only suck in the data you need from datbases, etc.

```
http:
//wiki.python.org/moin/PythonSpeed/PerformanceTips/
and:
http://www.python.org/doc/essays/list2str.html
```

## for loops redux...

# Are for loops really slow?

## From Linkedin Discussion:

For loop is extremely slow
I retrieved 2 million lines of data from the database (each line has two columns), and i would like to restructure the tuple-format data to a dict,
it's really unacceptable in case i use a for loop, it takes a lot of memory and very very slow.
finally after long time searching for the solution in the network, i replace the for loop with a map + lambda. i think it's hundreds of times faster.

demo: `for_loop_test.ipynb`

## LAB

### Profiling lab

- run `timeit` on some code of yours (or timing.py, or..)
- run iPython's %timeit on the same code.
- try to make the primes code in timing.py faster, and time the difference.
- write some code that tests one of the performance issues in:
  http://wiki.python.org/moin/PythonSpeed/PerformanceTips
  use one of the `timeits` to see if you can make a difference.
- try the profile tutorial at:
  http://pysnippet.blogspot.com/2009/12/profiling-your-python-code.html

## Wrap up

I hope you have an idea how to profile and time
your code.

Try it on a part of your project

## Next Week:

wxPython Desktop GUIs

## Project Time!

Profile your project

Performance tune part of it

Get ready to present

Anyone want a public code review?