Quick Python Performance Optimization: Part I

<http://infiniteloop.in/blog/quick-python-performance-optimization-part-i/>

This is part I of a two part series of blog post on performance optimization in python.

Aim is to just explain, the right way to do simple things which we use in day-to-day python programming, and has a very relevant performance impact.

1. %timeit (per line) and %prun (cProfile) in ipython interactive shell.

Profile your code while working on it, and try to find of where is the bottleneck. This is not contrary to the fact that premature optimization is the root of all evil. This is mean for the first level optimization and not a heavy optimization sequence.

For more on profiling python code, you should read this: http://www.huyng.com/posts/python-performance-analysis/

Another interesting library, line\_profiler is for line by line profiling https://bitbucket.org/robertkern/line\_profiler

2. Reduce the number of function calls. If you need a list to be manipluated, pass the entire list, rather than iterating over the list and passing each element to the function and returning.

3. Use xrange instead of range.

xrange is C implementation of range, with an eye on efficient memory usage.

4. For huge data, use numpy, its better than standard datastructures.

5. "".join(string) is better than + or +=

6. while 1 is faster than while True

7. list comphrension > for loop > while loop

list comprehension is faster than looping over the list, and while loop is the slowest, with an external counter with it.

8. use cProfile, cStringIO and cPickle

always use available C versions of the modules.

9. Use local variables.

local variables are faster than global variables, builtins or attribute lookups

10. list and iterators versions exist - iterators are memory efficient and scalable. Use itertools

Create generators and use yeild as much as posible. They are faster compared to the normal list way of doing it.

http://www.diveinto.org/python3/iterators.html

http://stackoverflow.com/questions/231767/the-python-yield-keyword-explained

Quick Python Performance Optimization: Part II

This is the Part II of Quick Python Performance Optimizations.

11. Use Map, Reduce and Filter instead of for loop where ever possible.

12. for checking 'a in b', dict or set is better than list/tuple.

13. while working with big amount of data, if possible use immutable datatypes, they are faster - tuples > list

14. insertion into a list in O(n) complexity.

15. if you need manipulations on both ends of a list, use deque.

16. del - delete the objects after use -

Python does it by itself. But make sure of that with the gc module or

by writing an \_\_del\_\_ magic function or

the simplest way, del after use.

17. time.clock()

18. GIL(http://wiki.python.org/moin/GlobalInterpreterLock) - GIL is a demon.

GIL allows only one python native thread to be run per process, preventing CPU level parallelism. Try using ctypes and native C libararies to overcome this. When even you reach the end of optimizing with python, always there exist an option of rewriting terribly slow functions in native C, and using it through python C bindings. Other libraries like gevent is also attacking the problem, and is successful to some extend.

TL,DR: While you write code, just give one round of thought on the data structures, the iteration constructs, builtins and create C extensions for tricking the GIL if need.