Contents

Python speed

People are often worried about the speed of their Python programs; doesn't using Python mean an unacceptable loss in performance? Some people just jump to the conclusion that "hey, it's an interpreted scripting language, and those all run very slow!" Other people have actually tried Python and have found it performs well enough. Sometimes, though, you have a program that just runs too slowly.

Why is raw speed important? Or isn't it?

Some people are inappropriately obsessed with speed and think that just because C can provide better performance for certain types of problem, it must therefore be a better language for all purposes. Other people think that *speed of development* is far more important, and choose Python even for those applications where it will run slower. Often, they are surprised to find Python code can run at quite acceptable speeds, and in some cases even faster than what they could get from C/C++ with a similar amount of development time invested.

Usually it is not the absolute speed that is important, you should think about what would be an *acceptable* speed of execution. Optimisations beyond achieving this acceptable speed are wasteful of resources (usually: your time. And thus: money.).

Techniques for Improving Performance and Scalability

Here are some coding guidelines for applications that demand peak performance (in terms of memory utilization, speed, or scalability).

Use the best algorithms and fastest tools

- Membership testing with sets and dictionaries is much faster, O(1), than searching sequences, O(n). When testing "a in b", b should be a set or dictionary instead of a list or tuple.
- String concatenation is best done with `''.join(seq)` which is an O(n) process. In contrast, using the '+' or '+=' operators can result in an O(n**2) process because new strings may be built for each intermediate step. The CPython 2.4 interpreter mitigates this issue somewhat; however, `''.join(seq)` remains the best practice.
- Many tools come in both list form and iterator form (range and xrange, map and itertools.imap, list comprehensions and generator expressions, dict.items and dict.iteritems). In general, the iterator forms are more memory friendly and more scalable. They are preferred whenever a real list is not required.
- Many core building blocks are coded in optimized C. Applications that take advantage of them can make substantial performance gains. The building blocks include all of the builtin datatypes (lists, tuples, sets, and dictionaries) and extension modules like array, itertools, and collections.deque.
- Likewise, the builtin functions run faster than hand-built equivalents. For example, map(operator.add, v1, v2) is faster than map(lambda x,y: x+y, v1, v2).
- Lists perform well as either fixed length arrays or variable length stacks. However, for queue applications using pop(0) or insert(0,v)), collections.deque() offers superior O(1) performance because it avoids the O(n) step of rebuilding a full list for each insertion or deletion.
- Custom sort ordering is best performed with Py2.4's key= option or with the traditional decorate-sort-undecorate technique. Both approaches call the key function just once per element. In contrast, sort's cmp= option is called many times per element during a sort. For example, sort(key=str.lower) is faster than

sort(cmp=lambda a,b: cmp(a.lower(), b.lower())).
See also TimeComplexity.

Take advantage of interpreter optimizations

- In functions, local variables are accessed more quickly than global variables, builtins, and attribute lookups. So, it is sometimes worth localizing variable access in inner-loops. For example, the code for random.shuffle() localizes access with the line, random=self.random. That saves the shuffling loop from having to repeatedly lookup self.random. Outside of loops, the gain is minimal and rarely worth it.
- The previous recommendation is a generalization of the rule to factor constant expressions out of loops. Likewise, constant folding needs to be done manually. Inside loops, write "x=3" instead of "x=1+2".
- Function call overhead is large compared to other instructions. Accordingly, it is sometimes worth in-lining code inside time-critical loops.
- List comprehensions run a bit faster than equivalent for-loops (unless you're just going to throw away the result).
- Starting with Py2.3, the interpreter optimizes "while 1" to just a single jump. In contrast "while True" takes several more steps. While the latter is preferred for clarity, time-critical code should use the first form.
- Multiple assignment is slower than individual assignment. For example "x,y=a,b" is slower than "x=a; y=b". However, multiple assignment is faster for variable swaps. For example, "x,y=y,x" is faster than "t=x; x=y; y=t".
- Chained comparisons are faster than using the "and" operator.
  Write "x < y < z" instead of "x < y and y < z".
- A few fast approaches should be considered hacks and reserved for only the most demanding applications. For example, "not not x" is faster than "bool(x)".

Take advantage of diagnostic tools

- The hotshot and profile modules help identify performance bottlenecks. Profile can distinguish between time spent in pure Python and time spent in C code.
- The timeit module offers immediate performance comparisons between alternative approaches to writing individual lines of code.

Performance can dictate overall strategy

- SAX is typically faster and more memory efficient than DOM approaches to XML.
- Use C versions of modules that are used frequently; e.g. cPickle instead of the pickle module, and cStringIO instead of StringIO. Note that on occasion, the C versions are less flexible, so be sure to read the module docs to know what you may be missing.

- Threading can improve the response time in applications that would otherwise waste time waiting for I/O.
- Select can help minimize the overhead for polling multiple sockets.

Consider external tools for enhancing performance

- Numpy is essential for high volume numeric work.
- Psyco and pyrex can help achieve the speed of native code. However, keep in mind the limitations of each, as neither supports all Python constructs.
- See the SciPy-related document "A beginners guide to using Python for performance computing" for an interesting comparison of different tools, along with some timing results.

More Performance Tips

More performance tips and examples can be found at PythonSpeed/PerformanceTips.

CategoryDocumentation

PythonSpeed (last edited 2012-01-31 20:20:08 by techtonik)