

Speeding Up Python

Asked 16 years, 2 months ago Modified 1 year, 8 months ago

Viewed 31k times



This is really two questions, but they are so similar, and to keep it simple, I figured I'd just roll them together:

47



- **Firstly:** Given an established python project, what are some decent ways to speed it up beyond just plain in-code optimization?
- **Secondly:** When writing a program from scratch in python, what are some good ways to greatly improve performance?

For the first question, imagine you are handed a decently written project and you need to improve performance, but you can't seem to get much of a gain through refactoring/optimization. What would you do to speed it up in this case short of rewriting it in something like C?

python

optimization

performance

Share

Improve this question

Follow

edited Oct 5, 2008 at 22:30



Blorgbeard

103k ● 50 ● 235 ● 276

asked Oct 5, 2008 at 21:46

akdom

33.1k ● 27 ● 74 ● 79

19 Answers

Sorted by:

Highest score (default)



47

Regarding "Secondly: When writing a program from scratch in python, what are some good ways to greatly improve performance?"



Remember the Jackson rules of optimization:



- Rule 1: Don't do it.
- Rule 2 (for experts only): Don't do it yet.

And the Knuth rule:

- "Premature optimization is the root of all evil."

The more useful rules are in the [General Rules for Optimization](#).

1. Don't optimize as you go. First get it right. Then get it fast. Optimizing a wrong program is still wrong.
2. Remember the 80/20 rule.
3. Always run "before" and "after" benchmarks. Otherwise, you won't know if you've found the 80%.
4. Use the right algorithms and data structures. This rule should be first. Nothing matters as much as algorithm and data structure.

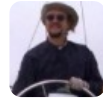
Bottom Line

You can't prevent or avoid the "optimize this program" effort. It's part of the job. You have to plan for it and do it carefully, just like the design, code and test activities.

Share Improve this answer

answered Oct 5, 2008 at 22:46

Follow



S.Lott

391k ● 82 ● 517 ● 788

@AbidRahmanK It's at archive.org:

web.archive.org/web/20101225202706/http://www.cs.cmu.edu/~jch/... – Tobias Kienzler Aug 10, 2012 at 14:28



30



Rather than just punting to C, I'd suggest:

Make your code count. Do more with fewer executions of lines:

- Change the algorithm to a faster one. It doesn't need to be fancy to be faster in many cases.
- Use python primitives that happens to be written in C. Some things will force an interpreter dispatch where some wont. The latter is preferable
- Beware of code that first constructs a big data structure followed by its consumption. Think the difference between range and xrange. In general it is often worth thinking about memory usage of the program. Using generators can sometimes bring $O(n)$ memory use down to $O(1)$.

- Python is generally non-optimizing. Hoist invariant code out of loops, eliminate common subexpressions where possible in tight loops.
- If something is expensive, then precompute or memoize it. Regular expressions can be compiled for instance.
- Need to crunch numbers? You might want to check `numpy` out.
- Many python programs are slow because they are bound by disk I/O or database access. Make sure you have something worthwhile to do while you wait on the data to arrive rather than just blocking. A weapon could be something like the `Twisted` framework.
- Note that many crucial data-processing libraries have C-versions, be it XML, JSON or whatnot. They are often considerably faster than the Python interpreter.

If all of the above fails for profiled and measured code, then begin thinking about the C-rewrite path.

Share Improve this answer

answered Oct 6, 2008 at 2:28

Follow



I GIVE CRAP
ANSWERS

18.8k ● 3 ● 44 ● 47

-
- 1 Nowadays pandas (pandas.pydata.org) should be added to that numbers crunching bit, i think. – [moooooeeep](#) Apr 3, 2013 at 17:26
-



25



The usual suspects -- profile it, find the most expensive line, figure out what it's doing, fix it. If you haven't done much profiling before, there could be some big fat quadratic loops or string duplication hiding behind otherwise innocuous-looking expressions.

In Python, two of the most common causes I've found for non-obvious slowdown are string concatenation and generators. Since Python's strings are immutable, doing something like this:

```
result = u""
for item in my_list:
    result += unicode (item)
```

will copy the *entire* string twice per iteration. This has been well-covered, and the solution is to use `"".join`:

```
result = "".join (unicode (item) for item in my_list)
```

Generators are another culprit. They're very easy to use and can simplify some tasks enormously, but a poorly-applied generator will be much slower than simply appending items to a list and returning the list.

Finally, **don't be afraid to rewrite bits in C!** Python, as a dynamic high-level language, is simply not capable of matching C's speed. If there's one function that you can't optimize any more in Python, consider extracting it to an extension module.

My favorite technique for this is to maintain both Python and C versions of a module. The Python version is written to be as clear and obvious as possible -- any bugs should be easy to diagnose and fix. Write your tests against this module. Then write the C version, and test it. Its behavior should in all cases equal that of the Python implementation -- if they differ, it should be very easy to figure out which is wrong and correct the problem.

Share Improve this answer

answered Oct 5, 2008 at 22:09

Follow




[John Millikin](#)

201k ● 41 ● 215 ● 227

String concatenation is not that bad in Python 2.5. Data point on generators: 2:1 (v2.5.2), 1,5:1,5 (v3.0) - list vs. generators for a processor-hungry function that I've tested yesterday (that is performance is the same on py3k, generators loose on v2.5) – [jfs](#) Oct 6, 2008 at 0:41

-
- 3 To add to that comment: the reason that string concatenation isn't that bad in 2.5 (and 2.6) is that there is a specific optimisation for this case in CPython (but not necessarily any other Python implementation). – [Tony Meyer](#) Oct 6, 2008 at 4:44

Tony, can you supply details of the optimization?
– [Seun Osewa](#) Feb 3, 2009 at 17:52

@John, do you convert the Python equivalent to C using Cython or do you just re-write in C? Does it make a difference? – [JustBeingHelpful](#) Apr 24 at 5:31 



First thing that comes to mind: [psyco](#). It runs only on x86, for the time being.

17



Then, [constant binding](#). That is, make all global references (and `global.attr`, `global.attr.attr...`) be local names inside of functions and methods. This isn't always successful, but in general it works. It can be done by hand, but obviously is tedious.

You said apart from in-code optimization, so I won't delve into this, but keep your mind open for typical mistakes (`for i in range(10000000)` comes to mind) that people do.

Share Improve this answer

Follow

edited Sep 3, 2011 at 3:40



[Jim Ferrans](#)

31k ● 12 ● 58 ● 83

answered Oct 5, 2008 at 21:51



[tzot](#)

95.8k ● 30 ● 149 ● 208



9



Cython and pyrex can be used to generate c code using a python-like syntax. Psyco is also fantastic for appropriate projects (sometimes you'll not notice much speed boost, sometimes it'll be as much as 50x as fast). I still reckon the best way is to profile your code (cProfile, etc.) and then just code the bottlenecks as c functions for python.

Share Improve this answer

Follow

answered Oct 5, 2008 at 22:03



[hacama](#)

291 ● 2 ● 5



7



I'm surprised no one mentioned ShedSkin:

<http://code.google.com/p/shedskin/>, it automatically converts your python program to C++ and in some benchmarks yields better improvements than psyco in speed.

Plus anecdotal stories on the simplicity:

<http://pyinsci.blogspot.com/2006/12/trying-out-latest-release-of-shedskin.html>

There are limitations though, please see [this](#)

Share Improve this answer

Follow

edited Apr 22, 2023 at 7:39



Koedlt

5,933 ● 9 ● 21 ● 45

answered Oct 5, 2008 at 22:54



torial

13.1k ● 9 ● 65 ● 89



5



I hope you've read:

<http://wiki.python.org/moin/PythonSpeed/PerformanceTips>

Resuming what's already there are usually 3 principles:

- write code that gets transformed in better bytecode, like, use locals, avoid unnecessary lookups/calls, use idiomatic constructs (if there's natural syntax for what you want, use it - usually faster. eg: don't do: "for key in some_dict.keys()", do "for key in some_dict")

- whatever is written in C is considerably faster, abuse whatever C functions/modules you have available
- when in doubt, import timeit, profile

Share Improve this answer

answered Dec 12, 2008 at 22:27

Follow

community wiki
[ionelmc](#)



Run your app through the Python profiler. Find a serious bottleneck. Rewrite that bottleneck in C. Repeat.

4

Share Improve this answer

answered Oct 5, 2008 at 22:02



Follow



[Vicent Marti](#)

7,295 ● 6 ● 32 ● 34



- 1 What if the main bottlenecks are due to library functions that are already written in the NumPy sort of optimized C style?
– [ely](#) Sep 13, 2012 at 12:58



4



People have given some good advice, but you have to be aware that when high performance is needed, the python model is: punt to c. Efforts like psyco may in the future help a bit, but python just isn't a fast language, and it isn't designed to be. Very few languages have the ability to do the dynamic stuff really well and still generate very fast



code; at least for the foreseeable future (and some of the design works against fast compilation) that will be the case.

So, if you really find yourself in this bind, your best bet will be to isolate the parts of your system that are unacceptable slow in (good) python, and design around the idea that you'll rewrite those bits in C. Sorry. Good design can help make this less painful. Prototype it in python first though, then you've easily got a sanity check on your c, as well.

This works well enough for things like numpy, after all. I can't emphasize enough how much good design will help you though. If you just iteratively poke at your python bits and replace the slowest ones with C, you may end up with a big mess. Think about exactly where the C bits are needed, and how they can be minimized and encapsulated sensibly.

Share Improve this answer

answered Oct 5, 2008 at 22:27

Follow



simon

7,022 ● 2 ● 30 ● 30

The above comment only makes sense if you can't use psyco and pyrex for some reason. (Pyrex is a particularly easy way to compile most python language features down to C code and declare variables whose types are compatible with C types, so that they will be handled as such in the compiled code.) – [Alex Coventry](#) Oct 5, 2008 at 23:03

Alex, psyco and pyrex doesn't give you a *general* solution for c-like speed now, and it may never get there. That being said,

getting too worried about execution speed is often bad: premature optimization. If you *know* you need it though, c is often the best way to get there in python. – [simon](#) Oct 6, 2008 at 2:58



4



This won't necessarily speed up any of your code, but is critical knowledge when programming in Python if you want to avoid slowing your code down. The "Global Interpreter Lock" (GIL), has the potential to drastically reduce the speed of your multi-threaded program if its behavior is not understood (yes, this bit me ... I had a nice 4 processor machine that wouldn't use more than 1.2 processors at a time). There's an introductory article with some links to get you started at [SmoothSpan](#).

Share Improve this answer

edited Oct 5, 2008 at 22:36

Follow

answered Oct 5, 2008 at 21:59



[Steve Moyer](#)

5,733 ● 1 ● 26 ● 34



4



It's often possible to achieve near-C speeds (close enough for any project using Python in the first place!) by replacing explicit algorithms written out longhand in Python with an implicit algorithm using a built-in Python call. This works because most Python built-ins are written in C anyway. Well, in CPython of course ;-)

<https://www.python.org/doc/essays/list2str/>



Share Improve this answer

edited Nov 14, 2014 at 1:43

Follow



twasbrillig

18.8k ● 9 ● 44 ● 69

answered Oct 29, 2008 at 17:06



sep332

1,049 ● 2 ● 13 ● 24



3



Share Improve this answer

answered Oct 5, 2008 at 22:45

Follow



Walter

7,961 ● 2 ● 32 ● 30



3



This is the procedure that I try to follow:

- `import psyco; psyco.full()`
- If it's not fast enough, run the code through a profiler, see where the bottlenecks are. (DISABLE psyco for this step!)
- Try to do things such as other people have mentioned to get the code at those bottlenecks as fast as possible.

- Stuff like `[str(x) for x in l]` or `[x.strip() for x in l]` is much, much slower than `map(str, x)` or `map(str.strip, x)`.
- After this, if I still need more speed, it's actually really easy to get PyRex up and running. I first copy a section of python code, put it directly in the pyrex code, and see what happens. Then I twiddle with it until it gets faster and faster.

Share Improve this answer

answered Oct 6, 2008 at 1:38

Follow



Claudiu

229k ● 173 ● 503 ● 697



2



The canonical reference to how to improve Python code is here: [PerformanceTips](#). I'd recommend against optimizing in C unless you really need to though. For most applications, you can get the performance you need by following the rules posted in that link.



Share Improve this answer

answered Oct 29, 2008 at 17:16



Follow



Jason Baker

198k ● 138 ● 382 ● 520



1



If using psyco, I'd recommend `psyco.profile()` instead of `psyco.full()`. For a larger project it will be smarter about the functions that got optimized and use a ton less memory.



I would also recommend looking at iterators and generators. If your application is using large data sets this will save you many copies of containers.

Share Improve this answer

answered Oct 6, 2008 at 17:24

Follow



[Peter Shinnars](#)

3,756 ● 26 ● 24



1



Besides the (great) [psyco](#) and the (nice) [shedskin](#), I'd recommend trying [cython](#) a great fork of [pyrex](#).

Or, if you are not in a hurry, I recommend to just wait.

Newer python virtual machines are coming, and [unladen-swallow](#) will find its way into the mainstream.



Share Improve this answer

edited May 14, 2009 at 15:15

Follow

answered May 14, 2009 at 14:32



[Davide](#)

17.7k ● 11 ● 55 ● 69



0



A couple of ways to speed up Python code were introduced after this question was asked:

- **Pypy** has a JIT-compiler, which makes it a lot faster for CPU-bound code.
- Pypy is written in [Rpython](#), a subset of Python that compiles to native code, leveraging the LLVM tool-





chain.

Share Improve this answer

answered Sep 17, 2013 at 12:20

Follow



Janus Troelsen

21.2k ● 14 ● 139 ● 207



0

For an established project I feel the main performance gain will be from making use of python internal lib as much as possible.



Some tips are here: <http://blog.hackerearth.com/faster-python-code>



Share Improve this answer

answered Jan 16, 2017 at 7:33

Follow



joydeep bhattacharjee

1,319 ● 4 ● 18 ● 43



0

There is also Python → 11l → C++ transpiler, which can be downloaded from [here](#).



Share Improve this answer

answered May 1, 2021 at 2:00

Follow



tav

617 ● 7 ● 10

