# Python is only slow
## if you use it wrong

Avery Pennarun <apenwarr@gmail.com>

Google Inc.

*This presentation absolutely, positively, in no way at all,
could ever possibly begin to reflect the opinion of my employer*



I mention Google here not because they endorse the content, but because if I mention Google, then this counts as “evangelism” and so going to conferences like this is totally a work-related expense. And it's actually sort of valuable too - I think a few people came away with positive feelings about Google after this one. Maybe Java lovers didn't though. But hold that thought.

## Who says?

- bup: backup software with 80 megs/sec thruput - in python
  - http://github.com/apenwarr/bup
- sshuttle: VPN software that easily handles 802.11g/n speeds - in python
  - http://github.com/apenwarr/sshuttle



This is where I plug my most recent awesome open source projects. The side note on 80 megs/sec is it's actually 80 megs/sec/core, so if your CPU has more cores then it could theoretically go faster. But it doesn't because nobody coded the multicore stuff yet.

## The Easiest Way to Use Python Wrong

TIGHT INNER LOOPS.

A line of python code is **80-100x slower** than a line of C code.

```
s = open('file').read()
for char in s:
   ...
```

3

7

“Don't do that then” is my primary advice here. In no circumstances should you attempt to process things character-by-character in any interpreted language if you care about performance.

## The Easiest Way to Make Python Fast

- Use regexes and C modules.
- No such thing as “100% pure python.”
- Forget about swig: writing C modules is easy.
- **python + C** together is, so far, the winning combination.
- C is simple; python is simple; pypy is hard.



There were some pypy people at the conference. pypy turns out to be pretty good (see later slides) but it's still a really complicated way to get not-as-good-as-C performance and memory usage. If you care about performance, use a C module.

The point I made about swig is just this: like all auto-generated code, it makes it look like what it's doing is rocket science. But python's C API is great! For most things all you need to know is there's something for unpacking tuples and another thing for packing and returning them.

Extremely lacking feature of pypy: it doesn't work with python's standard C API, so C modules don't work or (with their new crazy proxy thing) are super slow. So you'll have to continue ignoring pypy until they fix it.

## The Other Way to use Python Wrong

- Computation threads
  - Worthless because of GIL
- Threads are ok for I/O
- fork() works great for both
- C modules that use threads are fine

5

10

When I say “worthless because of GIL” I mean that whenever python is interpreting something, none of the other threads can be interpreting anything. So basically you get all the speed of one thread with all the annoyance of tracking your threads. And the GIL really isn't going anywhere soon, so you need a new strategy.

I/O bound processes (particularly disk) are okay for threads because python releases the GIL while they run.

Someone during the presentation pointed out the “multiprocess” module which does fork() for you and gives back the results. I heard it's pretty good, haven't tried it.

# Garbage Collection

## Refcounting... and threads

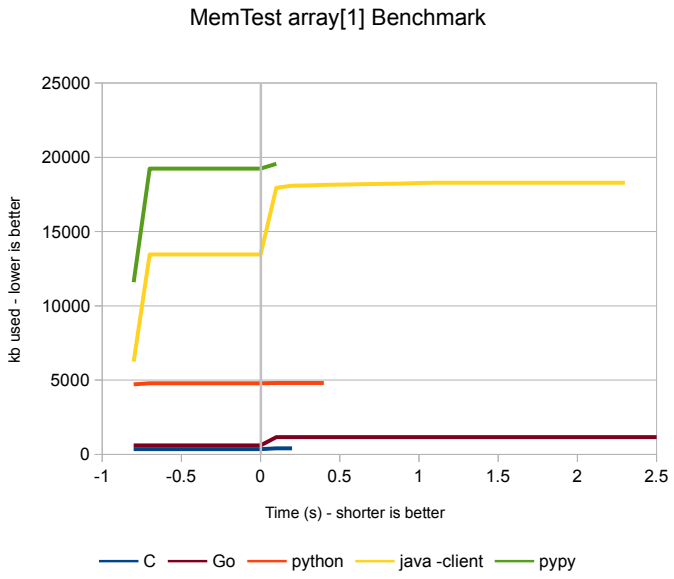
- A bad combo
- You would need locks around every single variable access
- One reason why removing the GIL is almost impossible
- There are tricks...



What you really need to know here is that the GIL is one big lock - and thanks to its existence, you don't need per-variable locks. This is why python's refcounting can be absurdly fast compared to doing the same thing in, say, just about any thread-enabled language. But if you drop the GIL, you have to add fine-grained locks for every variable access and that will *never* be fast... which is why the GIL isn't going anywhere unless someone invents a fundamental improvement to computer science. Why not just use GC instead of refcounting? That's coming up next.

The “tricks” I refer to are things like having smartpointer refcounts in addition to object-level refcounts. If your smartpointers don't cross threads, they don't need locks. C++ people do this sometimes and get good performance. But it's hard.

## Python is not a garbage collected language (*)




```
for i in xrange(1000000):
  a = '\0'*10000
```

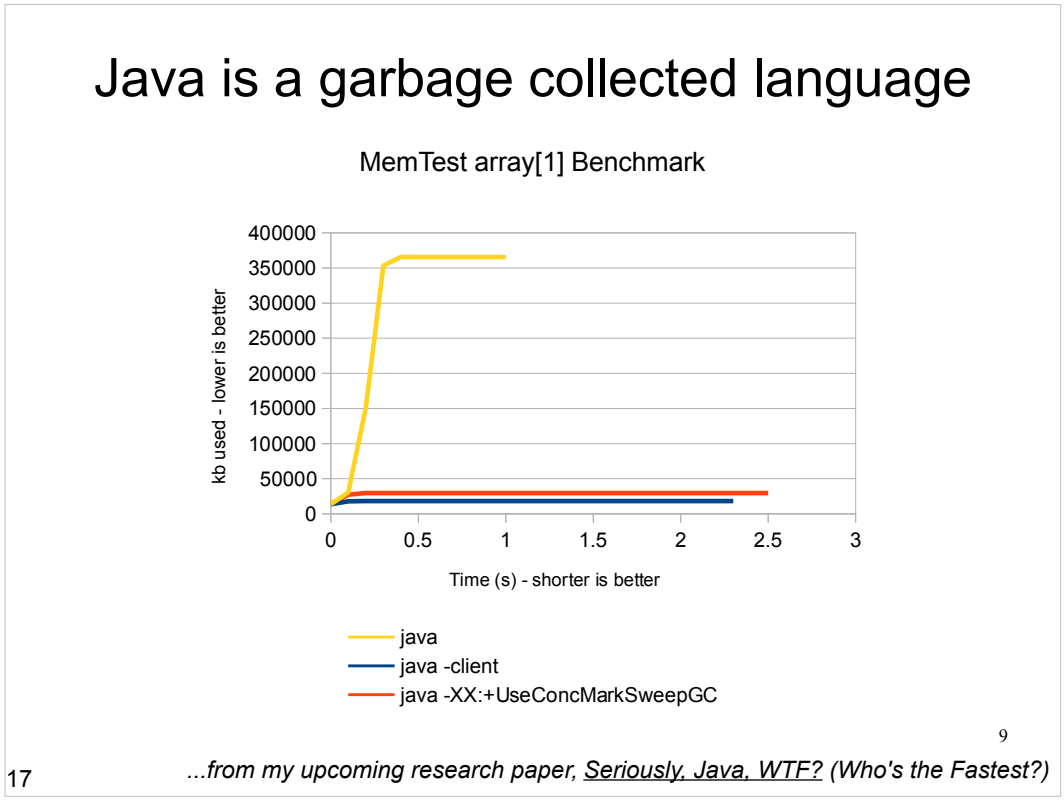
8

15

In this test, we just allocate a 10kbyte array a million times in a loop, throwing it away each time. Watch out for time=0 in the graph: that's when we actually start allocating. The time before that is “warmup” time for the test program (which sleeps a bit before starting the loop).

Notice how pypy's line is actually even shorter than C - a surprising result, but maybe they're using a fancy allocator (my C program just uses plain malloc()).

Note how both GC languages tested - pypy and java - suck memory like crazy after the program starts. Plain python has a baseline memory usage just for starting at all, but the 10k allocation makes basically no difference. Notice also that interpreted python is still much faster than java (!)



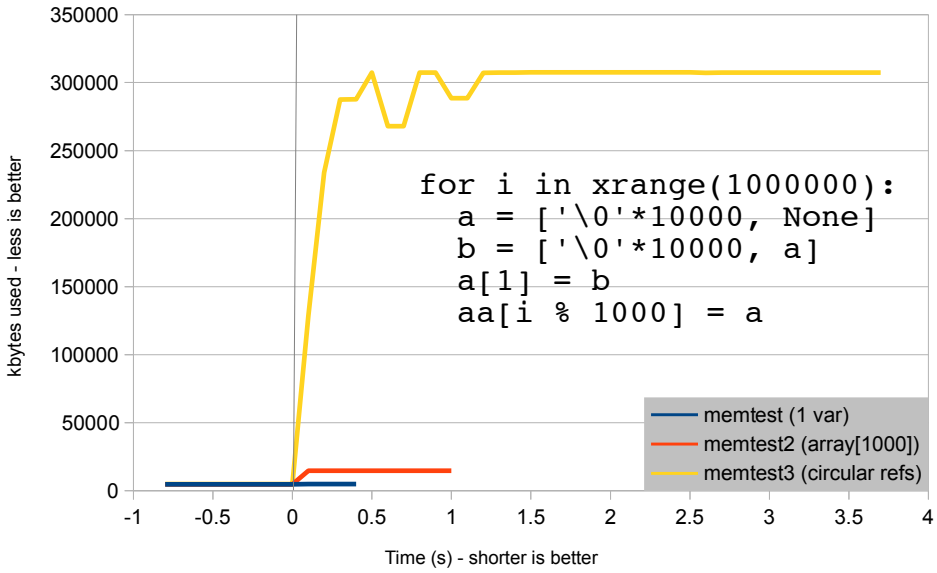

I found that Java not only used more memory than python in all my tests - it's also slower **in all cases!** Note also that the y-scale on this graph is different from the previous one.

The default “java” (on my 64-bit Linux system) is the same as “java -server”. Yes, I tried it. In this mode, it just sucks up all available heap space (-Xmx), then finally starts GC. Ever wonder why Java programs suck RAM? Because they **literally waste it all!**

java -client is much better on memory - it GCs frequently, but at a cost of 3x slower execution time. And no, the newfangled “concurrent gc” doesn't help: it's slower *and* uses more memory than -client.

BTW, default (-server) is the fastest, but still 2x slower than python.

## (*) Except sometimes python is a garbage collected language




19

I had to write a somewhat more complicated program in order to show that python actually shifts from refcounting to GC in some cases. This is needed because if you have objects with circular references, the refcount of no object in the circle will ever hit zero, so it will never be cleaned up.

Trivia: at least at one point (not sure if this is still true), perl simply didn't GC at all in that case, so you'd have a permanent memory leak. Python tries to save you from yourself in this case, but as you can see from the yellow graph here, it only barely succeeeds. Memory usage is roughly what java did on the previous slide :(

Conclusion: GC is hard.

## Getting the Most out of Python's GC

JUST AVOID IT AT ALL COSTS.

- Break circular references by hand when you're done.
- Better still: use the **weakref** module.

11



...and here's what you do about it: just don't use python's GC. Period. It's always avoidable, but you have to be careful.

The most common example I've seen of accidentally needing a GC is tree structures: a parent contains a list of children[] and each child has a pointer back to the parent. Easy for traversal, but a disaster for refcounting. Two options for fixing this sort of thing: delibrately break the circle when you're done with the tree (ie. set all parent pointers to None), or use weakref, which lets you make a pointer without incrementing the refcount.

The downside of a weakref is the pointed-to object might disappear if nobody else is using it. Not a big deal if you're just maintaining a tree though, as long as you're not doing anything crazy.

## Deterministic destructors

- Quiz: does this program work on win32?

```
open('file', 'w').write('hello')
open('file', 'w').write('world')
```

- With “real” gc, you have to manually manage resources:
  - files
  - database handles
  - sockets
  - locks

12



The quiz was fun because it's a double trick question: on win32, you can't open the same file for write twice at once unless you use special sharing options, so the second command could fail randomly if the first command hasn't GC'd the file pointer yet.

Except refcounting doesn't include the concept of “randomly.” It always cleans up immediately when the refcount goes to zero. So in plain python, the above program is guaranteed to work correctly (where “correctly” means overwriting 'hello' with 'world' for some reason).

Other python implementations, including ironically IronPython (usually run on Windows where you *really* care about file pointers) don't guarantee this. But that's stupid. We should have a protest.

## Ruffians & Vagabonds are trying to take away your Deterministic Destructors!

- Some people claim “real gc” is the “right solution”
- But what they mean is “it's the easiest way to do python on the java or .net runtime”
- “with” statement is powerful, but not good enough

13

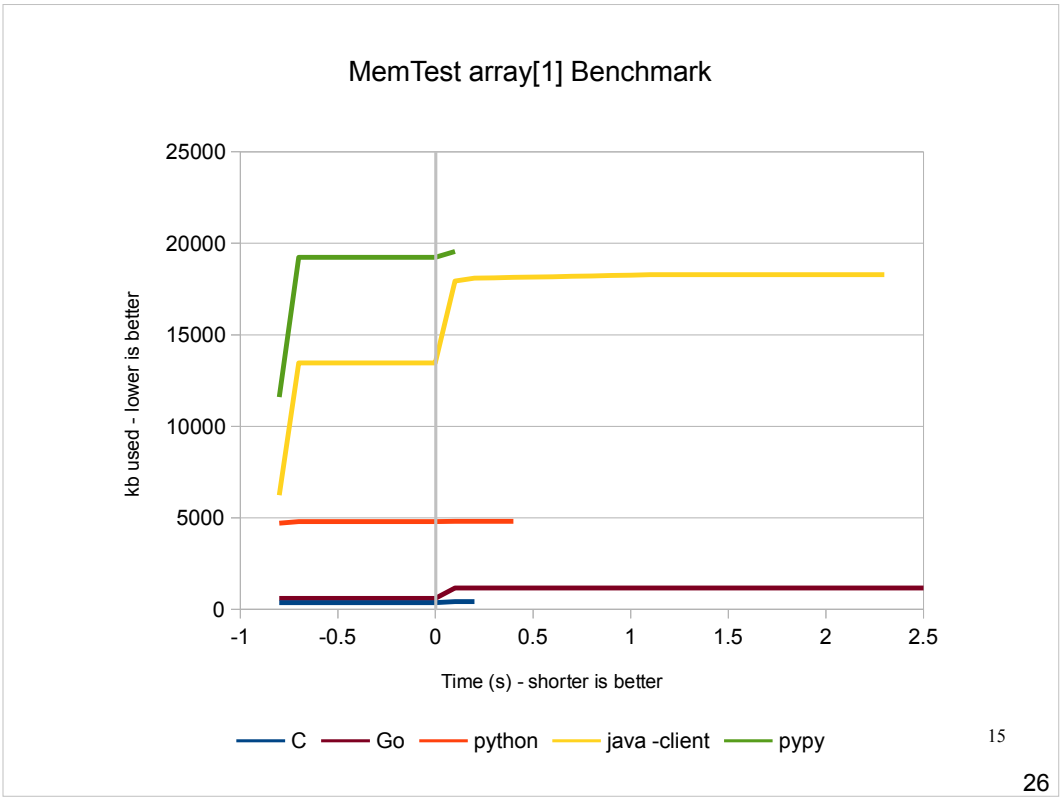
25

The real point of this presentation is to rail against the people who are trying to make python “better” by throwing away awesomely clean semantics - like refcounting - in exchange for slightly more speed.

It's just a bad move. Python is the last bastion of sane language semantics. If you want to throw away that in exchange for speed - then go use some language that does that! Please don't throw away python's best feature in exchange for one where it **will never win because C will always be faster and python already works great with C modules.**

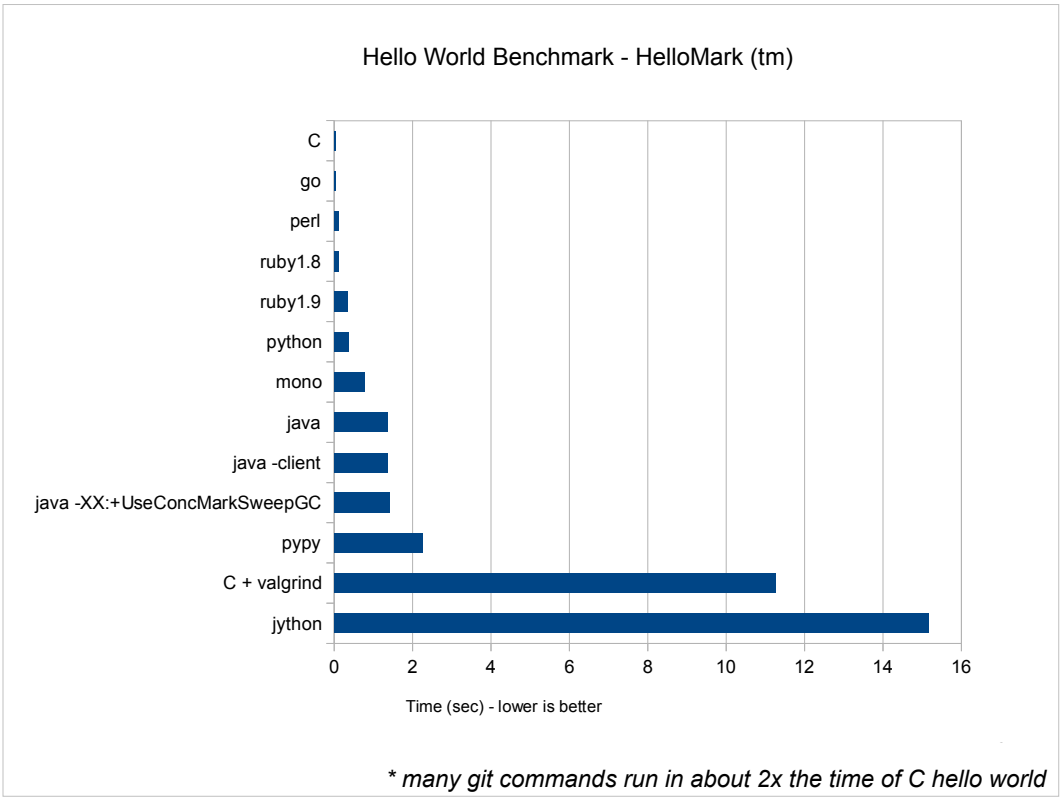
But don't bother switching to java if you want speed, as every benchmark I ran (and every real-life experience of everyone ever) clearly indicates.

# Interpreted vs. JIT




This is the same graph as earlier. The thing I want you to look at this time is the graph to the *left* of time zero. The jitted languages (java and pypy) show a sharp ramp-up in memory usage for a few milliseconds as they start up. (Sampling resolution here is 100ms, so don't take the timing too literally.)

That ramp-up is the JIT compiler. Which shows us two things: look how much memory JITting a tiny program can use (ugh!) and look how much time it adds to your process startup (double ugh!). And remember, a program that's *more* than two lines will take even more memory and take even longer.




My patent-pending HelloMark (tm) benchmark forks and execs a simple “hello world” process 20 times and tells you the total runtime.

A few things you can see from this: perl and ruby both start faster than python :( which ought to be eminently fixable if people care about it. mono is twice as fast as java, ha, suck it, java! pypy startup is slower still, the fixing of which could be an interesting challenge. And jython totally wins the suck prize here, coming in behind even c+valgrind (that's 20 valgrind startup/shutdown cycles!). Avoid.

Why does this matter? Because command-line tools rely fundamentally on startup time. I tried a few git commands, and typical runtime is ~2x my hello world test. That is, **git in C does real work faster than hello world in any other language** except go.

## .pyc files

- are awesome

17

29

I added this slide to retroactively justify one of the embarrassing bits in the previous slide: that perl and ruby both startup faster than python.

All these are non-representative microbenchmarks; you can't read too much into them. (But java really does always suck.) HelloMark is a serious misrepresentation because it doesn't use libraries.

In perl or ruby, importing a library requires re-interpreting every file in the library, because the languages are unhygienic. Python “compiles” each .py file into p-code the first time, so future runs don't have to parse anything. After the first run, library heavy python programs are *way* faster than ruby or perl. eg. django vs. rails: rails apps can take a full second to reparse all of rails when you change one file. django reloads near-instantly.

## Summary

- Love refcounting, hate gc
- Don't write tight inner loops: that's what C is for
- If you need a JIT, you're doing it wrong
  - ...even if the JIT is good
- Let's keep working on that startup time

http://github.com/apenwarr/avebench

18

30

"Even if the JIT is good" is a polite reference to pypy. To be honest, I had never used pypy before doing the benchmarks to do this presentation, but it actually amazed me with its speed. I mean, the memory allocator ran faster than C. Wow. Unfortunately, it uses a lot more RAM and has a much longer startup time than even plain python, so it doesn't work well in my personal common use case - systems-level programming.

Plain python is pretty great for systems programming, although I would love to see the startup time further improved. a 'strace python hello.py' makes me cry, but I bet it could be vastly improved.

That URL there is a copy of my (obviously trivial) benchmark code. Feel free to clone it and run the tests for yourself if you're curious.