Python performance characteristics

Maciej Fijałkowski

PyCon 2012

October 5, 2012

Who am I?

- Maciej Fijałkowski (yes this is unicode)
- PyPy core developer for I don't remember
- performance freak

What this talk is about?

- python performance (or lack of it)
- why does it matter
- what can we do about it
- how PyPy works

Python performance message

- according to Guido
- “Avoid overengineering datastructures. Tuples are better than objects (try namedtuple too though). Prefer simple fields over getter/setter functions.”
- “Built-in datatypes are your friends. Use more numbers, strings, tuples, lists, sets, dicts. Also check out the collections library, esp. deque.”
- “Be suspicious of function/method calls; creating a stack frame is expensive.”
- “The universal speed-up is rewriting small bits of code in C. Do this only when all else fails.”

What does it mean?

- don’t use abstractions
- don’t use Python

What does it mean?

- don’t use abstractions
- don’t use Python

But also

- measure!
- there are so many variables, you cannot care without benchmarks
- if you have no benchmarks, you don't care

But also

- measure!
- there are so many variables, you cannot care without benchmarks
- if you have no benchmarks, you don't care

This is not how I want to write software

- I like my abstractions
- I like Python
- I don't want to rewrite stuff for performance
- in C/C++

This is not how I want to write software

- I like my abstractions
- I like Python
- I don't want to rewrite stuff for performance

- in C/C++

Second best

- keep my abstractions
- do arcane voodoo to keep my programs fast
- but you have to understand the voodo in the first place

But Python performance!

- there is no such thing as language performance
- there is implementation performance
- the language might be easier or harder to optimize
- CPython performance characteristics is relatively straightforward

What is PyPy?

- PyPy is a Python interpreter (that's what we care about)
- PyPy is a toolchain for creating dynamic language implementations
- also, an Open Source project that has been around for a while

Compilers vs interpreters

- compilers compile language X (C, Python) to a lower level language (C, assembler) ahead of time
- interpreters compile language X to bytecode and have a big interpreter loop
- PyPy has a hybrid approach. It's an interpreter, but hot paths are compiled directly to assembler during runtime

Compilers vs interpreters

- compilers compile language X (C, Python) to a lower level language (C, assembler) ahead of time
- interpreters compile language X to bytecode and have a big interpreter loop
- PyPy has a hybrid approach. It's an interpreter, but hot paths are compiled directly to assembler during runtime

What is just in time (JIT) compilation?

- few different flavors
- observe runtime values
- compile code with agressive optimizations
- have checks if assumptions still stand

So what PyPy does?

- interprets a Python program
- the JIT observes python **interpreter**
- producing code through the path followed by the interpreter
- compiles loops and functions

Some properties

- the code speed **changes** over time
- hopefully from slow to fast
- you need to warm up things before they get fast

Some example

- integer addition!

Abstractions

- inlining, malloc removal
- abstractions are cheap
- if they don't introduce too much complexity

Abstractions

- inlining, malloc removal
- abstractions are cheap

- if they don't introduce too much complexity

Few words about garbage collection

- `CPython`: refcounting + cyclic collector
- `PyPy`: generational mark & sweep
- errr....

Few words about garbage collection

- `CPython`: refcounting + cyclic collector
- `PyPy`: generational mark & sweep

- errr....

The rest

- I'll explain various PyPy strategies
- ideally all this knowledge will be unnecessary
- this is the second best, how to please the JIT compiler

Allocations (PyPy)

- allocation is expensive
- for a good GC, short living objects don't matter
- it's better to have a small persistent structure and abstraction on allocation
- copying however is expensive
- we have hacks for strings, but they're not complete

Allocations (PyPy)

- allocation is expensive
- for a good GC, short living objects don't matter
- it's better to have a small persistent structure and abstraction on allocation

- copying however is expensive
- we have hacks for strings, but they're not complete

Calls

- Python calls are an incredible mess
- simple is better than complex
- simple call comes with no cost, the cost grows with growing complexity

Attribute access

- if optimized, almost as fast as local var access
- `dict` lookup optimized away
- class attributes considered constant
- meta programming is better than dynamism
- objects for small number of constant keys, dicts for large numbers of changing keys

Other sorts of loops

- there is more!
- `tuple(iterable)`, `map(iterable)`, `re.search`
- they're all jitted
- not all nicely

Summary

- we hope this knowledge will not be needed
- the more you care, the better you need to know

Questions?

- Thank you!
- `http://pypy.org`
- `http://baroquesoftware.com`