

Speed up your Python code

EuroPython 2006

CERN, Geneva

Stefan Schwarzer

sschwarzer@sschwarzer.com

SSchwarzer.com

2006-07-05

Overview

- Optimization mindset
- When to optimize
- Optimization process
- Finding the bottlenecks
- General optimization strategies
- Analyzing algorithms with big-O notation
- Optimization garden
- Conclusions

Not in this talk:

- Optimization of the Python interpreter
- Other optimization of C code

Optimization mindset

Optimization is costly

Optimization is supposed to save time

Optimization mindset

Optimization is costly

Optimization is supposed to save time

But ... it also has costs:

Optimization mindset

Optimization is costly

Optimization is supposed to save time

But . . . it also has costs:

- Costly developer time spent on the optimization itself
- Optimization often makes the code more difficult to comprehend, thus . . .
- More costly developer time spent on bugfixes
- More costly developer time spent on adding features

Optimization mindset

Optimization is costly

Optimization is supposed to save time

But ... it also has costs:

- Costly developer time spent on the optimization itself
- Optimization often makes the code more difficult to comprehend, thus ...
- More costly developer time spent on bugfixes
- More costly developer time spent on adding features

Optimize only if necessary:

The program has to run fast enough, *but not faster!*

Optimization mindset

Premature optimization is evil

Premature optimization is the root of all evil.

C. A. R. Hoare
(often misattributed to D. Knuth)

- *Don't* “optimize as you go” —
most of the time, the bottlenecks are elsewhere
- In other words, don't waste your time with useless optimization

When to optimize

- Consider only realistic use cases
 - Slow startup may not matter for a rarely started program even if it's used for a long time after start
 - Speed may not matter that much for a program which is usually run as a nightly cronjob
- Consider actual user experience
 - Does the program feel slow?
To you / to one user / to several users?
 - How slow?
Noticable / annoying / unbearable?

Recommended optimization process

```
def optimize():  
    """Recommended optimization process."""  
    assert got_architecture_right(), "fix architecture"  
    assert made_code_work(bugs=None), "fix bugs"  
    while code_is_too_slow():  
        wbn = find_worst_bottleneck(just_guess=False,  
                                   profile=True)  
        is_faster = try_to_optimize(wbn,  
                                   run_unit_tests=True, new_bugs=None)  
    if not is_faster:  
        undo_last_code_changes()
```

Recommended optimization process

```
def optimize():
    """Recommended optimization process."""
    --> assert got_architecture_right(), "fix architecture"
    assert made_code_work(bugs=None), "fix bugs"
    while code_is_too_slow():
        wbn = find_worst_bottleneck(just_guess=False,
                                    profile=True)
        is_faster = try_to_optimize(wbn,
                                    run_unit_tests=True, new_bugs=None)
    if not is_faster:
        undo_last_code_changes()
```

Recommended optimization process

```
def optimize():
    """Recommended optimization process."""
    assert got_architecture_right(), "fix architecture"
    --> assert made_code_work(bugs=None), "fix bugs"
    while code_is_too_slow():
        wbn = find_worst_bottleneck(just_guess=False,
                                    profile=True)
        is_faster = try_to_optimize(wbn,
                                    run_unit_tests=True, new_bugs=None)
    if not is_faster:
        undo_last_code_changes()
```

Recommended optimization process

```
def optimize():
    """Recommended optimization process."""
    assert got_architecture_right(), "fix architecture"
    assert made_code_work(bugs=None), "fix bugs"
--> while code_is_too_slow():
        wbn = find_worst_bottleneck(just_guess=False,
                                    profile=True)
        is_faster = try_to_optimize(wbn,
                                    run_unit_tests=True, new_bugs=None)
    if not is_faster:
        undo_last_code_changes()
```

Recommended optimization process

```
def optimize():
    """Recommended optimization process."""
    assert got_architecture_right(), "fix architecture"
    assert made_code_work(bugs=None), "fix bugs"
    while code_is_too_slow():
-->         wbn = find_worst_bottleneck(just_guess=False,
-->                                     profile=True)
        is_faster = try_to_optimize(wbn,
                                    run_unit_tests=True, new_bugs=None)
    if not is_faster:
        undo_last_code_changes()
```

Recommended optimization process

```
def optimize():
    """Recommended optimization process."""
    assert got_architecture_right(), "fix architecture"
    assert made_code_work(bugs=None), "fix bugs"
    while code_is_too_slow():
        wbn = find_worst_bottleneck(just_guess=False,
                                    profile=True)
-->    is_faster = try_to_optimize(wbn,
-->                               run_unit_tests=True, new_bugs=None)
    if not is_faster:
        undo_last_code_changes()
```

Recommended optimization process

```
def optimize():
    """Recommended optimization process."""
    assert got_architecture_right(), "fix architecture"
    assert made_code_work(bugs=None), "fix bugs"
    while code_is_too_slow():
        wbn = find_worst_bottleneck(just_guess=False,
                                   profile=True)
        is_faster = try_to_optimize(wbn,
                                    run_unit_tests=True, new_bugs=None)
-->     if not is_faster:
-->         undo_last_code_changes()
```

Finding the bottlenecks

- Find out if the program speed is bound by I/O (local file system, network) or CPU
- Operating system tools (here: Unix)
 - Command line: `time`, `top`, `dstat`, ...
 - GUI: `gkrellm`, `xosview`, ...
- Python tools
 - `profile` (cProfile in Python 2.5)
 - `hotshot`
 - `print statements`

General optimization strategies

- Doing things faster (e. g. change algorithm)
- Doing things less often (e. g. caching)
- Both at the same time (e. g. DBMS instead of flat files)

Analyzing algorithms with big-O notation

Introduction

- Terminology to explain which algorithms are slower or faster
- Describes how an increase in the amount of data n affects the running time
- Written as $O(n)$, $O(n \ln n)$, $O(n^2)$, ...
- The term in parentheses (without constant factors) is the most significant, less significant terms are left out; i. e. instead of $O(2.3n^2 + 3n)$ it's just $O(n^2)$

Analyzing algorithms with big-O notation

Common big-Os

| Order | Said to be "... time" | Examples |
|--------------|--------------------------|--|
| $O(1)$ | constant | <code>key in dict</code> <code>dict[key] = value</code> <code>list.append(item)</code> |
| $O(\ln n)$ | logarithmic | Binary search |
| $O(n)$ | linear | <code>item in sequence</code> <code>str.join(list)</code> |
| $O(n \ln n)$ | | <code>list.sort()</code> |
| $O(n^2)$ | quadratic | Nested loops (with constant time bodies) |

Analyzing algorithms with big-O notation

Common big-Os

| Order | Said to be "... time" | Examples |
|--------------|--------------------------|--|
| $O(1)$ | constant | <code>key in dict</code> <code>dict[key] = value</code> <code>list.append(item)</code> |
| $O(\ln n)$ | logarithmic | Binary search |
| $O(n)$ | linear | <code>item in sequence</code> <code>str.join(list)</code> |
| $O(n \ln n)$ | | <code>list.sort()</code> |
| $O(n^2)$ | quadratic | Nested loops (with constant time bodies) |

Try to avoid $O(n^2)$ and slower algorithms if n is large

Analyzing algorithms with big-O notation

Example: Finding common items in two lists

```
def intersection1(seq1, seq2): # approx.  $O(n^2)$ 
    result = {}
    for item in seq1:
        if item in seq2: result[item] = True
    return result.keys()
```

Analyzing algorithms with big-O notation

Example: Finding common items in two lists

```
def intersection1(seq1, seq2): # approx.  $O(n^2)$ 
    result = {}
    for item in seq1:
        if item in seq2: result[item] = True
    return result.keys()
```

```
def intersection2(seq1, seq2): # approx.  $O(n)$ 
    result = {}
    dict2 = dict((item, True) for item in seq2)
    for item in seq1:
        if item in dict2: result[item] = True
    return result.keys()
```

Analyzing algorithms with big-O notation

Example: Finding common items in two lists

```
def intersection1(seq1, seq2): # approx.  $O(n^2)$ 
    result = {}
    for item in seq1:
        if item in seq2: result[item] = True
    return result.keys()
```

```
def intersection2(seq1, seq2): # approx.  $O(n)$ 
    result = {}
    dict2 = dict((item, True) for item in seq2)
    for item in seq1:
        if item in dict2: result[item] = True
    return result.keys()
```

```
def intersection3(seq1, seq2): # approx.  $O(n)$ 
    return list(set(seq1) & set(seq2))
```

Optimization garden

Introduction

- *Depending on the situation*, some of the hints on the next pages may help, some may not

Optimization garden

Introduction

- *Depending on the situation*, some of the hints on the next pages may help, some may not
- Strive for a good compromise for performance gain on one hand and
 - Ease of code changes
 - Maintainabilityon the other hand

Reach for “low-hanging fruit”

Optimization garden

Algorithms—Notes

- Changing algorithms is the most promising optimization approach—speedups of several hundred percent may be possible

Optimization garden

Algorithms—Notes

- Changing algorithms is the most promising optimization approach—speedups of several hundred percent may be possible
- The same holds for **changes of the architecture** which can be seen as algorithms at a higher level

Optimization garden

Algorithms—Notes

- Changing algorithms is the most promising optimization approach—speedups of several hundred percent may be possible
- The same holds for **changes of the architecture** which can be seen as algorithms at a higher level
- **Changing data structures** can also have a huge effect, since switching data structures (e. g. from lists to dictionaries) implies changing the algorithms for data storage and retrieval

Optimization garden

Algorithms—General

- Avoid nested loops (watch out for implicit loops)
- Move loop-invariant code out of loops
- Update only changed data in an object
- Divide and conquer (e. g. binary search)
- Cache instead of recompute or reload (may be error-prone)
- But don't exhaust memory, avoid swapping
- Use multithreading for I/O-bound code (e. g. web spiders)
- Consider replacing an algorithm instead of tuning it

Optimization garden

Algorithms—Files

- Read a file completely and then process it if it's small
- Read and process a file line by line if it's large
- Instead of flat files, use database software
(e. g. *dbm modules, sqlite, PostgreSQL or MySQL)

Optimization garden

Hardware

- Use faster computers
- Provide more memory
- Use faster hard disks
- Use faster network hardware

Optimization garden

Python-specific—Very Python-specific ;-)

- Use `python -O`
- Avoid `exec` and `eval`
- Avoid `from module import *`
- Shortcut namespace searches (e.g. `opj = os.path.join`)
- Use `list.append` and `str.join` to concatenate many strings
- Use list or generator comprehensions instead of `for` loops
- Avoid function/method calls; inline code

Optimization garden

Python-specific—Algorithm-related

- In `list.sort`, use `key` rather than `cmp` argument
- Use dictionaries or sets for containment tests
- Use sets to find all items in a container which are also in another (intersection)
- Use sets to find all items in a container which aren't in another (difference)
- Don't copy objects (lists, dictionaries) routinely (“just to be sure”) if they don't change
- Use own specialized code if a library function is too general

Optimization garden

Python-specific—Use C, code Python

- Change code to use C-coded objects (lists, tuples, dictionaries, sets)
- Use built-in functions coded in C that do the same as your Python code
- Try Psyco, PyInline, Pyrex or weave
- Use available C extension modules (e. g. scipy)

If **everything else fails**, convert slow Python code to a C/C++ extension module (SWIG and similar tools can help)

Conclusions

Conclusions

- Optimize only if necessary; don't waste development time

Conclusions

- Optimize only if necessary; don't waste development time
- When looking for bottlenecks, consider only real use cases and user experience

Conclusions

- Optimize only if necessary; don't waste development time
- When looking for bottlenecks, consider only real use cases and user experience
- Use a profiler to find bottlenecks, don't guess

Conclusions

- Optimize only if necessary; don't waste development time
- When looking for bottlenecks, consider only real use cases and user experience
- Use a profiler to find bottlenecks, don't guess
- To speed up your software, do things faster or less often

Conclusions

- Optimize only if necessary; don't waste development time
- When looking for bottlenecks, consider only real use cases and user experience
- Use a profiler to find bottlenecks, don't guess
- To speed up your software, do things faster or less often
- Big-O notation can help to distinguish fast and slow algorithms

Conclusions

- Optimize only if necessary; don't waste development time
- When looking for bottlenecks, consider only real use cases and user experience
- Use a profiler to find bottlenecks, don't guess
- To speed up your software, do things faster or less often
- Big-O notation can help to distinguish fast and slow algorithms
- There are lots of things you can try to speed up your Python code: Use faster algorithms or hardware, or use Python-specific tuning

Thank you for your attention! :-)

Questions? Discussion?