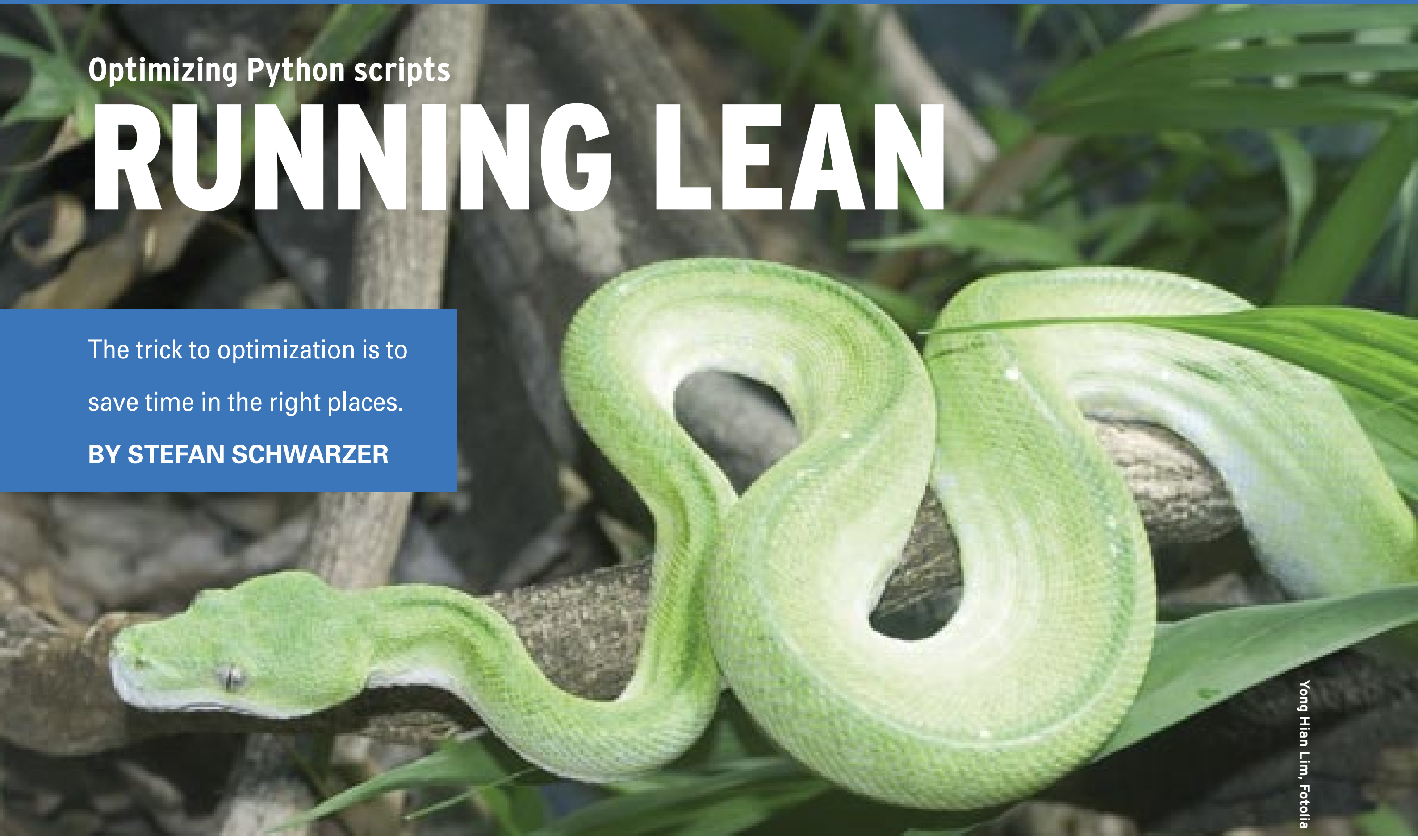
Optimizing Python scripts

RUNNING LEAN



The trick to optimization is to save time in the right places.

BY STEFAN SCHWARZER

Optimization saves execution time. Unfortunately, optimizing lengthens development cycles. The optimized source code is typically more complex than the original code, which increases the time for testing and debugging. Adding complexity also makes the code more difficult to maintain. Because the optimization process takes time and adds complexity, it is best to avoid optimizing code while you are writing it. Before you start optimizing, start with a stable program. Once your program is stable and complete, you can look for ways to improve performance. In this article, I describe some strategies for optimizing Python programs.

Listing 1: cProfile

```
>>> import cProfile
>>> import sys
>>> sys.argv.append("--search")
>>> sys.argv.append("python")
>>> f = open("/usr/bin/emerge")
>>> ef = f.read()
>>> f.close()
>>> cProfile.run(ef, "emerge.
stats")
Searching...
[ Results for search key : python
]
[ Applications found : 48 ]
...
```

Where to Optimize

From the developer's perspective, a program is never just slow or fast. Before you start accelerating code sections, it is important to discover exactly where the bottlenecks occur. The first step is to find out whether the CPU or I/O system is slowing down your software when a specific function is executed. It does not make sense to optimize the execution time for an algorithm by a factor of 100, only to discover that the hard disk or network is to blame.

To find out if a slow CPU, a slow hard disk, or some other hardware component is causing the problem, you can use a GUI-based tool such as Xosview [1] or Gkrellm [2]. Also, tools such as Dstat [3] give statistics for data transfers from and to specific partitions. To achieve plausible results, you must make sure the computer is not running any other processes that could cause additional load. As an alternative, *top* and *ps* give individual process performance data.

Figures 1 and 2 show screenshots of Gkrellm and Dstat for a process that is limited by CPU or data-transfer performance. The problem of data transfer is more difficult to identify than a CPU bottleneck because there is no hardware-independent upper threshold.

The best way to determine threshold values is to refer to hardware specifications or use benchmarks. Note that data transfer can refer to a CD-ROM drive, as in our example, but it can just as easily refer to a network interface, a tape autochanger on a backup system, and so on.

If 100 percent CPU load is slowing your program down, you will need to identify the sections of code that are causing the problem. The cProfile Python module can help you evaluate the results returned by the Pstats module.

As a practical example showing a Python profiler at work, consider the Gentoo Linux package management tool Emerge, which is written in Python. A search with *emerge --search python* takes just 10 seconds to execute on my computer. It wouldn't make sense to start optimizing this, unless you have a slow computer, but this is an example of how to approach the analysis phase.

Generating run-time statistics with Python's cProfile module is a bit complicated because the profiler's command-line interface doesn't see the need to pass in command-line parameters to the program you are calling, which is why I used the interactive interpreter (Listing 1). After importing the required Python modules, and preparing the parameters, I type *cProfile.run()* to start the test run. The Pstats module outputs a table with the run-time statistics (see Listing 2).

Although a genuine bottleneck does not occur here, you might optimize a couple of points. For example, Emerge does take 1.2 seconds of 9 to update the progress indicator (*update_twirl* method; you can see this in the *cumtime* [cumulative time] column in Listing 2). However, an Emerge option can switch off this display. About 1.2 seconds are used for time-intensive deep copying. If deep copies are not really needed, there is some scope for savings.

To accelerate code with a CPU bottleneck, do things faster or do things less often. If you replace your own flat-file data management system with a database such as SQLite [4], you can often achieve both goals. Of course, the goal is to achieve maximum, or at least sufficient, speed benefits with a minimum of development effort. And you still need to consider maintainability of the code.

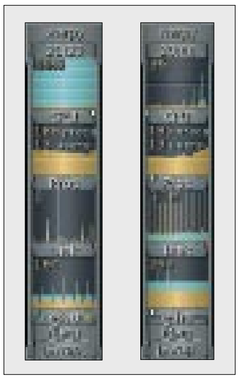


Figure 1: Gkrellm on the left shows 100 percent CPU load for a process with a CPU bottleneck. The program on the right shows the data throughput when copying a CD.

Before you start measuring the speed of your code, you need to make sure the code is as free of error as humanly possible. If your code has errors, the danger is that you might be "optimizing" code that only runs slowly because it is buggy. Automated tests, written with the *doctest* and the *unittest* Python modules, can help reduce errors when modifying code.

The next step is to start profiling to find the most important code sections for optimization. The best candidates for optimization will typically be the sections with the highest total run time – that is, sections in which the product of the run time and the frequency of execution is particularly high. It normally makes more sense to optimize a function that executes 10,000 times and takes a second per run than to optimize a function that runs just five times and takes 10 seconds. But you also should consider the extent to which the program run time affects the user experience. You might discover that the program simply seems to be lagging slightly in the first case, whereas the second case imposes a 10-second wait on the user.

Optimization Techniques

Replacing an algorithm with a more effective algorithm is one way to accelerate a program. Whereas most optimization techniques will speed up the code by a factor of 10 percent at the most, replacing an algorithm can achieve speed benefits of several hundred percent!

Big-O notation is used to describe the complexity of an algorithm. The "O"

Listing 2: Pstats

```
>>> import pstats
>>> s = pstats.Stats("emerge.stats")
>>> s.sort_stats('time')
<pstats.Stats instance at 0xb7d80eac>
>>> s.print_stats(10)
Sun Oct  1 23:12:36 2006    emerge.stats

         602508 function calls (586701 primitive calls) in 9.052 CPU seconds

   Ordered by: internal time
   List reduced from 609 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1240    1.022    0.001    1.022    0.001 {method 'readlines' of 'file' objects}
    11387    0.849    0.000    0.849    0.000 {method 'flush' of 'file' objects}
     1096    0.579    0.001    1.513    0.001 /usr/lib/portage/pym/portage.py:200(cacheddir)
14550/160    0.421    0.000    1.173    0.007 /home/schwa/python2.5/lib/python2.5/copy.py:144(deepcopy)
    76352    0.359    0.000    0.359    0.000 {method 'append' of 'list' objects}
        1    0.335    0.335    2.513    2.513 <string>:468(output)
66288/66173    0.316    0.000    0.317    0.000 {len}
    11383    0.256    0.000    1.225    0.000 <string>:94(update_twirl)
    36953    0.224    0.000    0.224    0.000 {method 'split' of 'str' objects}
```

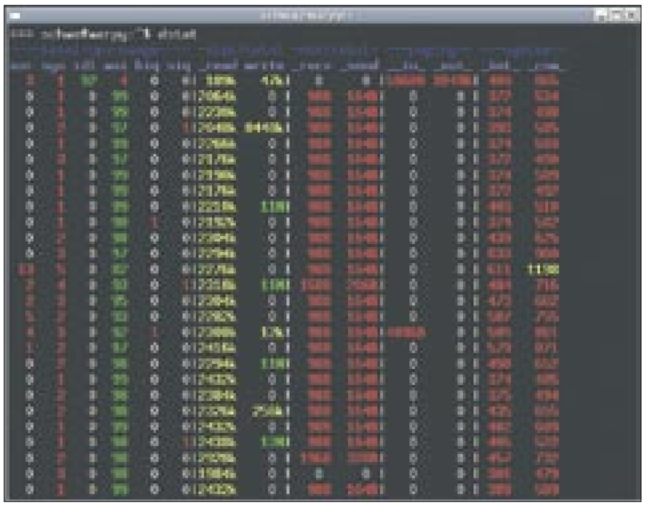


Figure 2: Dstat output for a process with a data transfer bottleneck (copying a CD).

here stands for the "order" of the algorithm. The expression in parentheses describes how the performance changes with respect to a change in the data input – the number of values in a list, or a string length, for example. An O(*n*) algorithm will take twice as long to handle twice the volume of data, whereas an O(n^2) algorithm would take four times as long for twice the volume of data.

Obviously, it is desirable for the expression in the parentheses to grow only little, despite an increasing value for *n*. Table 1 shows the performance for various Python algorithms. The performance decreases from top to bottom.

In multiply-nested loops, or for some combinatorial problems, the performance ratio can be greater than quadratic. In this case, even small values of *n* will lead to poor performance. If the code causes a bottleneck, you should look to avoid algorithms of more than quadratic order for larger values of *n*.

If you are running an algorithm against a smaller volume of data, a more complex algorithm can be faster. With some algorithms, the run time will only increase slowly as *n* grows; however, a longer preparatory step might be needed. In this case, a "slower" algorithm that avoids the need for a preparatory step could ultimately be faster.

Comparing algorithms on the basis of their order is useful in principle, but the technique might not be applicable in the wild, at least not for all possible volumes of data. For example, regardless of the theoretical efficiency of an algorithm, the performance could be drastically reduced if the list you need to sort does not fit into memory and the operating system has to start swapping memory out to disk. Effects like this also come into play if the memory management of the underlying C standard library has a major influence on the run time.

Optimized Set Intersection

Let's look at another example. Each of the Python functions I will be investigating finds the intersecting set of two lists – list elements that occur in both lists – and returns a new list with the results.

The first algorithm I use is of quadratic complexity given two lists with *n* elements (Listing 3). The outer loop iterates over all the elements in the first list, with linear complexity. The loop contains a second implicit loop, which is hidden in the *value in list2* test condition.

The search in *list2* is linear, so I need to consider the two occurrences of linear complexity. The nesting of the explicit outer loop and the implicit inner loop makes this an O(n^2) algorithm. Although the determination of the keys in the *return* instruction is linear, this is insignificant compared with the quadratic complexity of the previous algorithm with respect to *n*. In general, it is preferable to avoid nested loops. At best, this will give you quadratic performance; this said, optimization is probably not worthwhile for smaller volumes of input.

The algorithm in Listing 4 is a modified version of the previous algorithm with linear complexity. The code looks like the previous listing, but it creates a dictionary from the second list *before* entering the outer loop and then uses the dictionary in the loop.

value in dict2 shows constant performance, so the bottom line results in a linear outer loop. The last algorithm also results in linear performance (Listing 5). The operation that converts the first list to a set is linear, as is the generation of the resulting set by the *intersection* method and the conversion of the resulting set to a list. Although the syntax for these operations is nested, they actually run sequentially. Three sequential linear steps result in an O(*n*) algorithm.

Better Algorithms

The previous examples suggest a number of optimization rules you may know. For example, operations whose results do not change through multiple iterations of a loop should be moved in front of the loop, thus avoiding the need to execute them in each iteration.

The principle of divide and conquer might work fine with data. A well-known example of this is a binary search that requires presorted data but returns the results with a complexity of O(ln *n*), rather than O(*n*). However, if you are handling a small volume of input data, a trivial linear search will be just fine.

Instead of constantly reloading or recalculating, you can cache values. But consider the consequences and possible data inconsistency, especially on systems that use multiple threads or transactions.

Listing 3: intersection1

```
def intersection1(list1,
  list2):
    """Determine resulting set
  with O(n^2) algorithm."""
    result = {}
    for value in list1:
        if value in list2:
            result[value] =
  True
    return result.keys()
```

Table 1: Python Algorithm Performance

Order	Description	Examples
O(1)	Constant time	*key in dict, dict[key] = value, list.append(value)*
O(ln *n*)	Logarithmic time	Binary search
O(*n*)	Linear time	*value in list, str.join(list)*
O(*n* ln *n*)		*list.sort()*
O(n^2)	Quadratic time	Nested loops [for O(1) loop body]

Also, think about restricting the cache size to keep the system from swapping memory out to disk and thus negating any speed benefits. In scenarios in which caching makes sense, the use of a database server will often give you a major performance boost.

If you store an object on disk or transfer an object over the wire, you can accelerate the operation by just storing the changes instead of the whole object. On the downside, this kind of optimization can affect class abstractions or other code. Try to keep the interface abstract, even if you are optimizing internally.

The following rules apply to line-by-line text manipulation: if the files are short, it is typically easier, and faster, to read the text completely before going on to process the data. For longer files – logfiles are a typical example – it is better to read and process each line separately. Failure to do so could mean running out of memory, and continual swapping would freeze your system.

The choice of the right data structures is closely related to the choice of algorithm. In fact, your choice of a data structure will implicitly influence your choice of data access algorithms. As demonstrated earlier on, searching for a key in a dictionary is far quicker than a linear search for the same value in a list.

The architecture of a software system also impacts performance. You can regard the architecture as the algorithm that the whole system follows, and consider it before you start developing.

Python Tricks

Python-specific optimizations have different effects depending on the Python version. A new Python version might even make a piece of code slower, although this is an exception. The easiest way to optimize a Python script is to use the interpreter's *-O* option to automatically optimize the Python bytecode generated by the interpreter. Do not use *from module import ** in your scripts, which makes it impossible for the Python interpreter to perform some internal optimizations and also makes maintenance more difficult. Avoid lookup operations across multiple name-spaces by binding an object directly to to the local name-space. For example, after the line *opj = os.path.join*, you can access the *join* function more quickly than *opj*. This kind of optimization affects the readability of your code.

Avoid *exec* and *eval*. Python is flexible, so you should find a code variant that does not need these functions; in many cases, this practice actually improves readability. "In-lining" the function body can help to speed up code that executes functions with a short run time within loops, but this often leads to more redundancy and makes the software harder to maintain.

If you need to concatenate multiple strings, collect them in a list and join the list elements with *"".join(list)*. This method is faster than using the + operator. The *key* argument in *list.sort* leads to faster code than the *cmp* argument.

Listing 4: intersection2

```
def intersection2(list1,
  list2):
    """Determine resulting set
  with O(n) algorithm."""
    result = {}
    dict2 = dict((value, True)
  for value in list2)
    for value in list1:
        if value in dict2:
            result[value] =
  True
    return result.keys()
```

C Helps

It might be better to use highly optimized C code for some operations, but without sacrificing the benefits of Python. To do so, rewrite your code, or parts of it, to use Python's internal functions (e.g., *range* instead of a loop) or data types (lists, tuples, dictionaries, sets). Use C libraries for time-critical code; you can use the libxml2 library [5] to parse XML, and SWIG [6] and Ctypes [7] are useful for encapsulating existing C libraries. The latter became part of the standard distribution in Python 2.5.

PyInline [8] and Weave [9] give developers the ability to integrate C fragments into Python code. Pyrex [10], a language that is very similar to Python, lets developers encapsulate existing C libraries and define their own extensions (which are converted to C). The most flexible, but at the same time the most complicated, approach is the Python/C-API. As an alternative to basically just programming in C, you might like to try Psyco [11], a just-in-time compiler for Python that, unfortunately, is only available for 32-bit x86 systems.

Listing 5: intersection3

```
def intersection3(list1,
  list2):
    """Determine resulting set
  with O(n) algorithm."""
    return list(set(list1).
  intersection(list2))
@KE
```

Conclusions

Interpreted languages such as Python [12] are no hindrance to developing fast programs, but remember to test the program's speed and discover whether the program is fast enough for the intended application without optimization. If not, you should go on to find bottlenecks with the use of appropriate tools, and you should target your optimization efforts. Modifying algorithms and data structures, or simply replacing hardware, promises the biggest time savings.

Python-specific optimizations can also help. If possible, take the opportunity to use implicit C code in the form of Python code in the case of, for example, Python data structures or external C libraries. And always remember to keep maintainability in mind whenever you optimize your software. ■

INFO

[1] Xosview: *http://sourceforge.net/projects/xosview/*
[2] Gkrellm: *http://www.gkrellm.net/*
[3] Dstat: *http://dag.wieers.com/home-made/dstat/*
[4] SQLite: *http://www.sqlite.org/*
[5] Libxml2: *http://xmlsoft.org/*
[6] SWIG: *http://www.swig.org/*
[7] Ctypes: *http://starship.python.net/crew/theller/ctypes/*
[8] PyInline: *http://pyinline.sourceforge.net/*
[9] Weave: *http://www.scipy.org/Weave*
[10] Pyrex: *http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/*
[11] Psyco: *http://psyco.sourceforge.net/*
[12] Python: *http://www.python.org/*