



- » [PythonSpeed](#)
- » [PerformanceTips](#)
- » [PerformanceTips](#)
- » [FrontPage](#)
- » [RecentChanges](#)
- » [FindPage](#)
- » [HelpContents](#)
- » [PerformanceTips](#)

Page

- » **Immutable Page**
- » [Info](#)
- » [Attachments](#)
- »

User

- » [Login](#)

Contents

1. [Other Versions](#)
2. [Overview: Optimize what needs optimizing](#)
3. [Choose the Right Data Structure](#)
4. [Sorting](#)
5. [String Concatenation](#)
6. [Loops](#)
7. [Avoiding dots...](#)
8. [Local Variables](#)
9. [Initializing Dictionary Elements](#)
10. [Import Statement Overhead](#)
11. [Data Aggregation](#)
12. [Doing Stuff Less Often](#)
13. [Python is not C](#)
14. [Use xrange instead of range](#)
15. [Re-map Functions at runtime](#)
16. [Profiling Code](#)
 1. [Profiling](#)
 2. [The cProfile and Hotshot Modules](#)
 3. [Trace Module](#)
 4. [Visualizing Profiling Results](#)

This page is devoted to various tips and tricks that help improve the performance of your Python

programs. Wherever the information comes from someone else, I've tried to identify the source.

Python has changed in some significant ways since I first wrote my "fast python" page in about 1996, which means that some of the orderings will have changed. I migrated it to the Python wiki in hopes others will help maintain it.

You should always test these tips with your application and the specific version of the Python [implementation](#) you intend to use and not just blindly accept that one method is faster than another. See the [profiling](#) section for more details.

Also new since this was originally written are packages like [Cython](#), [Pyrex](#), [Psyco](#), [Weave](#), [Shed Skin](#) and [PyInline](#), which can dramatically improve your application's performance by making it easier to push performance-critical code into C or machine language.

Other Versions

» Russian: <http://omsk.lug.ru/wacko/PythonHacking/PerfomanceTips>

Overview: Optimize what needs optimizing

You can only know what makes your program slow after first getting the program to give correct results, then running it to see if the correct program is slow. When found to be slow, profiling can show what parts of the program are consuming most of the time. A comprehensive but quick-to-run test suite can then ensure that future optimizations don't change the correctness of your program. In short:

1. Get it right.
2. Test it's right.
3. Profile if slow.
4. Optimise.
5. Repeat from 2.

Certain optimizations amount to good programming style and so should be learned as you learn the language. An example would be moving the calculation of values that don't change within a loop, outside of the loop.

Choose the Right Data Structure

TBD.

Sorting

Sorting lists of basic Python objects is generally pretty efficient. The sort method for lists takes an optional comparison function as an argument that can be used to change the sorting behavior. This is quite convenient, though it can significantly slow down your sorts, as the comparison function will be called many times. In Python 2.4, you should use the key argument to the built-in sort instead, which should be the fastest way to sort.

Only if you are using older versions of Python (before 2.4) does the following advice from Guido van Rossum apply:

An alternative way to speed up sorts is to construct a list of tuples whose first element is a sort key that will sort properly using the default comparison, and whose second element is the original list element. This is the so-called [Schwartzian Transform](#), also known as [DecorateSortUndecorate](#) (DSU).

Suppose, for example, you have a list of tuples that you want to sort by the n-th field of each tuple. The following function will do that.

```
def sortby(somelist, n):
    nlist = [(x[n], x) for x in somelist]
    nlist.sort()
    return [val for (key, val) in nlist]
```

Matching the behavior of the current list sort method (sorting in place) is easily achieved as well:

```
def sortby_inplace(somelist, n):
    somelist[:] = [(x[n], x) for x in somelist]
    somelist.sort()
    somelist[:] = [val for (key, val) in somelist]
    return
```

Here's an example use:

```
>>> somelist = [(1, 2, 'def'), (2, -4, 'ghi'), (3, 6, 'abc')]
>>> somelist.sort()
>>> somelist
[(1, 2, 'def'), (2, -4, 'ghi'), (3, 6, 'abc')]
>>> nlist = sortby(somelist, 2)
>>> sortby_inplace(somelist, 2)
>>> nlist == somelist
True
>>> nlist = sortby(somelist, 1)
>>> sortby_inplace(somelist, 1)
>>> nlist == somelist
True
```

From Tim Delaney

From Python 2.3 sort is guaranteed to be stable.

(to be precise, it's stable in CPython 2.3, and guaranteed to be stable in Python 2.4)

Python 2.4 adds an optional key parameter which makes the transform a lot easier to use:

```
# E.g. n = 1
n = 1
import operator
nlist.sort(key=operator.itemgetter(n))
# use sorted() if you don't want to sort in-place:
```

```
# sortedlist = sorted(nlist, key=operator.itemgetter(n))
```

Note that the original item is never used for sorting, only the returned key - this is equivalent to doing:

```
# E.g. n = 1
n = 1
nlist = [(x[n], i, x) for (i, x) in enumerate(nlist)]
nlist.sort()
nlist = [val for (key, index, val) in nlist]
```

String Concatenation

The accuracy of this section is disputed with respect to later versions of Python. In CPython 2.5, string concatenation is fairly fast, although this may not apply likewise to other Python implementations. See [ConcatenationTestCode](#) for a discussion.

Strings in Python are immutable. This fact frequently sneaks up and bites novice Python programmers on the rump. Immutability confers some advantages and disadvantages. In the plus column, strings can be used as keys in dictionaries and individual copies can be shared among multiple variable bindings. (Python automatically shares one- and two-character strings.) In the minus column, you can't say something like, "change all the 'a's to 'b's" in any given string. Instead, you have to create a new string with the desired properties. This continual copying can lead to significant inefficiencies in Python programs.

Avoid this:

```
s = ""
for substring in list:
    s += substring
```

Use `s = "".join(list)` instead. The former is a very common and catastrophic mistake when building large strings. Similarly, if you are generating bits of a string sequentially instead of:

```
s = ""
for x in list:
    s += some_function(x)
```

use

```
slist = [some_function(elt) for elt in somelist]
s = "".join(slist)
```

Avoid:

```
out = "<html>" + head + prologue + query + tail + "</html>"
```

Instead, use


```
out = "<html>%s%s%s</html>" % (head, prologue, query, tail)
```

Even better, for readability (this has nothing to do with efficiency other than yours as a programmer), use dictionary substitution:

```
out = "<html>%(head)s%(prologue)s%(query)s%(tail)s</html>" % locals()
```

This last two are going to be much faster, especially when piled up over many CGI script executions, and easier to modify to boot. In addition, the slow way of doing things got slower in Python 2.0 with the addition of rich comparisons to the language. It now takes the Python virtual machine a lot longer to figure out how to concatenate two strings. (Don't forget that Python does all method lookup at runtime.)

Loops

Python supports a couple of looping constructs. The `for` statement is most commonly used. It loops over the elements of a sequence, assigning each to the loop variable. If the body of your loop is simple, the interpreter overhead of the `for` loop itself can be a substantial amount of the overhead. This is where the  `map` function is handy. You can think of `map` as a `for` moved into C code. The only restriction is that the "loop body" of `map` must be a function call. Besides the syntactic benefit of list comprehensions, they are often as fast or faster than equivalent use of `map`.

Here's a straightforward example. Instead of looping over a list of words and converting them to upper case:

```
newlist = []
for word in oldlist:
    newlist.append(word.upper())
```

you can use `map` to push the loop from the interpreter into compiled C code:

```
newlist = map(str.upper, oldlist)
```

List comprehensions were added to Python in version 2.0 as well. They provide a syntactically more compact and more efficient way of writing the above `for` loop:

```
newlist = [s.upper() for s in oldlist]
```

Generator expressions were added to Python in version 2.4. They function more-or-less like list comprehensions or `map` but avoid the overhead of generating the entire list at once. Instead, they return a generator object which can be iterated over bit-by-bit:

```
iterator = (s.upper() for s in oldlist)
```

Which method is appropriate will depend on what version of Python you're using and the characteristics of the data you are manipulating.

Guido van Rossum wrote a much more detailed (and succinct) examination of [loop optimization](#) that is definitely worth reading.

Avoiding dots...

Suppose you can't use `map` or a list comprehension? You may be stuck with the `for` loop. The `for` loop example has another inefficiency. Both `newlist.append` and `word.upper` are function references that are reevaluated each time through the loop. The original loop can be replaced with:

```
upper = str.upper
newlist = []
append = newlist.append
for word in oldlist:
    append(upper(word))
```

This technique should be used with caution. It gets more difficult to maintain if the loop is large. Unless you are intimately familiar with that piece of code you will find yourself scanning up to check the definitions of `append` and `upper`.

Local Variables

The final speedup available to us for the non-`map` version of the `for` loop is to use local variables wherever possible. If the above loop is cast as a function, `append` and `upper` become local variables. Python accesses local variables much more efficiently than global variables.

```
def func():
    upper = str.upper
    newlist = []
    append = newlist.append
    for word in oldlist:
        append(upper(word))
    return newlist
```

At the time I originally wrote this I was using a 100MHz Pentium running BSDI. I got the following times for converting the list of words in `/usr/share/dict/words` (38,470 words at that time) to upper case:

```
Version Time (seconds)
Basic loop 3.47
Eliminate dots 2.45
Local variable & no dots 1.79
Using map function 0.54
```

Initializing Dictionary Elements

Suppose you are building a dictionary of word frequencies and you've already broken your text up into a list of words. You might execute something like:

```
wdict = {}
for word in words:
    if word not in wdict:
        wdict[word] = 0
    wdict[word] += 1
```

Except for the first time, each time a word is seen the `if` statement's test fails. If you are counting a large number of words, many will probably occur multiple times. In a situation where the initialization of a value is only going to occur once and the augmentation of that value will occur many times it is cheaper to use a `try` statement:

```
wdict = {}
for word in words:
    try:
        wdict[word] += 1
    except KeyError:
        wdict[word] = 1
```

It's important to catch the expected [KeyError](#) exception, and not have a default `except` clause to avoid trying to recover from an exception you really can't handle by the statement(s) in the `try` clause.

A third alternative became available with the release of Python 2.x. Dictionaries now have a `get()` method which will return a default value if the desired key isn't found in the dictionary. This simplifies the loop:

```
wdict = {}
get = wdict.get
for word in words:
    wdict[word] = get(word, 0) + 1
```

When I originally wrote this section, there were clear situations where one of the first two approaches was faster. It seems that all three approaches now exhibit similar performance (within about 10% of each other), more or less independent of the properties of the list of words.

Also, if the value stored in the dictionary is an object or a (mutable) list, you could also use the `dict.setdefault` method, e.g.

```
4         wdict.setdefault(key, []).append(new_element)
```

You might think that this avoids having to look up the key twice. It actually doesn't (even in python 3.0), but at least the double lookup is performed in C.

Another option is to use the [!\[\]\(c444627dab9fee9a1550c053ffaaaae2_img.jpg\) `defaultdict`](#) class:


```
from collections import defaultdict

wdict = defaultdict(int)
```

```
for word in words:
    wdict[word] += 1
```

Import Statement Overhead

`import` statements can be executed just about anywhere. It's often useful to place them inside functions to restrict their visibility and/or reduce initial startup time. Although Python's interpreter is optimized to not import the same module multiple times, repeatedly executing an `import` statement can seriously affect performance in some circumstances.

Consider the following two snippets of code (originally from Greg McFarlane, I believe - I found it unattributed in a `comp.lang.python`  python-list@python.org posting and later attributed to him in another source):

```
def doit1():
    import string ##### import statement inside function
    string.lower('Python')

for num in range(100000):
    doit1()
```

or:

```
import string ##### import statement outside function
def doit2():
    string.lower('Python')

for num in range(100000):
    doit2()
```

`doit2` will run much faster than `doit1`, even though the reference to the `string` module is global in `doit2`. Here's a Python interpreter session run using Python 2.3 and the new `timeit` module, which shows how much faster the second is than the first:

```
>>> def doit1():
...     import string
...     string.lower('Python')
...
>>> import string
>>> def doit2():
...     string.lower('Python')
...
>>> import timeit
>>> t = timeit.Timer(setup='from __main__ import doit1', stmt='doit1()')
>>> t.timeit()
11.479144930839539
```



```
>>> t = timeit.Timer(setup='from __main__ import doit2', stmt='doit2()')
>>> t.timeit()
4.6661689281463623
```

String methods were introduced to the language in Python 2.0. These provide a version that avoids the import completely and runs even faster:

```
def doit3():
    'Python'.lower()

for num in range(100000):
    doit3()
```

Here's the proof from `timeit`:

```
>>> def doit3():
...     'Python'.lower()
...
>>> t = timeit.Timer(setup='from __main__ import doit3', stmt='doit3()')
>>> t.timeit()
2.5606080293655396
```

The above example is obviously a bit contrived, but the general principle holds.

Note that putting an import in a function can speed up the initial loading of the module, especially if the imported module might not be required. This is generally a case of a "lazy" optimization -- avoiding work (importing a module, which can be very expensive) until you are sure it is required.

This is only a significant saving in cases where the module wouldn't have been imported at all (from any module) -- if the module is already loaded (as will be the case for many standard modules, like `string` or `re`), avoiding an import doesn't save you anything. To see what modules are loaded in the system look in `sys.modules`.

A good way to do lazy imports is:

```
email = None

def parse_email():
    global email
    if email is None:
        import email
    ...
```

This way the `email` module will only be imported once, on the first invocation of `parse_email()`.

Data Aggregation

Function call overhead in Python is relatively high, especially compared with the execution speed of a

builtin function. This strongly suggests that where appropriate, functions should handle data aggregates. Here's a contrived example written in Python.

```
import time
x = 0
def doit1(i):
    global x
    x = x + i

list = range(100000)
t = time.time()
for i in list:
    doit1(i)

print "%.3f" % (time.time()-t)
```

vs.

```
import time
x = 0
def doit2(list):
    global x
    for i in list:
        x = x + i

list = range(100000)
t = time.time()
doit2(list)
print "%.3f" % (time.time()-t)
```

Here's the proof in the pudding using an interactive session:

```
>>> t = time.time()
>>> for i in list:
...     doit1(i)
...
>>> print "%.3f" % (time.time()-t)
0.758
>>> t = time.time()
>>> doit2(list)
>>> print "%.3f" % (time.time()-t)
0.204
```

Even written in Python, the second example runs about four times faster than the first. Had `doit` been written in C the difference would likely have been even greater (exchanging a Python `for` loop for a C `for` loop as well as removing most of the function calls).

Doing Stuff Less Often

The Python interpreter performs some periodic checks. In particular, it decides whether or not to let another thread run and whether or not to run a pending call (typically a call established by a signal handler). Most of the time there's nothing to do, so performing these checks each pass around the interpreter loop can slow things down. There is a function in the `sys` module, `setcheckinterval`, which you can call to tell the interpreter how often to perform these periodic checks. Prior to the release of Python 2.3 it defaulted to 10. In 2.3 this was raised to 100. If you aren't running with threads and you don't expect to be catching many signals, setting this to a larger value can improve the interpreter's performance, sometimes substantially.

Python is not C

It is also not Perl, Java, C++ or Haskell. Be careful when transferring your knowledge of how other languages perform to Python. A simple example serves to demonstrate:

```
% timeit.py -s 'x = 47' 'x * 2'
loops, best of 3: 0.574 usec per loop
% timeit.py -s 'x = 47' 'x << 1'
loops, best of 3: 0.524 usec per loop
% timeit.py -s 'x = 47' 'x + x'
loops, best of 3: 0.382 usec per loop
```

Now consider the similar C programs (only the add version is shown):

```
#include <stdio.h>

int main (int argc, char *argv[]) {
    int i = 47;
    int loop;
    for (loop=0; loop<500000000; loop++)
        i + i;
    return 0;
}
```

and the execution times:

```
% for prog in mult add shift ; do
< for i in 1 2 3 ; do
< echo -n "$prog: "
< /usr/bin/time ./$prog
< done
< echo
< done
mult: 6.12 real 5.64 user 0.01 sys
mult: 6.08 real 5.50 user 0.04 sys
mult: 6.10 real 5.45 user 0.03 sys
```

```
add: 6.07 real 5.54 user 0.00 sys
add: 6.08 real 5.60 user 0.00 sys
add: 6.07 real 5.58 user 0.01 sys

shift: 6.09 real 5.55 user 0.01 sys
shift: 6.10 real 5.62 user 0.01 sys
shift: 6.06 real 5.50 user 0.01 sys
```

Note that there is a significant advantage in Python to adding a number to itself instead of multiplying it by two or shifting it left by one bit. In C on all modern computer architectures, each of the three arithmetic operations are translated into a single machine instruction which executes in one cycle, so it doesn't really matter which one you choose.

A common "test" new Python programmers often perform is to translate the common Perl idiom

```
while (<>) {
    print;
}
```

into Python code that looks something like

```
import fileinput

for line in fileinput.input():
    print line,
```

and use it to conclude that Python must be much slower than Perl. As others have pointed out numerous times, Python is slower than Perl for some things and faster for others. Relative performance also often depends on your experience with the two languages.

Use xrange instead of range

This section no longer applies if you're using Python 3, where `range` now provides an iterator over ranges of arbitrary size, and where `xrange` no longer exists.

Python has two ways to get a range of numbers: `range` and `xrange`. Most people know about `range`, because of its obvious name. `xrange`, being way down near the end of the alphabet, is much less well-known.

`xrange` is a generator object, basically equivalent to the following Python 2.3 code:

```
def xrange(start, stop=None, step=1):
    if stop is None:
        stop = start
        start = 0
    else:
        stop = int(stop)
```

```
start = int(start)
step = int(step)

while start < stop:
    yield start
    start += step
```

Except that it is implemented in pure C.

`xrange` does have limitations. Specifically, it only works with `ints`; you cannot use `longs` or `floats` (they will be converted to `ints`, as shown above).

It does, however, save gobs of memory, and unless you store the yielded objects somewhere, only one yielded object will exist at a time. The difference is thus: When you call `range`, it creates a `list` containing so many number (`int`, `long`, or `float`) objects. All of those objects are created at once, and all of them exist at the same time. This can be a pain when the number of numbers is large.

`xrange`, on the other hand, creates *no* numbers immediately - only the range object itself. Number objects are created only when you pull on the generator, e.g. by looping through it. For example:

```
xrange(sys.maxint) # No loop, and no call to .next, so no numbers are
instantiated
```

And for this reason, the code runs instantly. If you substitute `range` there, Python will lock up; it will be too busy allocating `sys.maxint` number objects (about 2.1 billion on the typical PC) to do anything else. Eventually, it will run out of memory and exit.

In Python versions before 2.2, `xrange` objects also supported optimizations such as fast membership testing (`i in xrange(n)`). These features were removed in 2.2 due to lack of use.

Re-map Functions at runtime

Say you have a function

```
class Test:
    def check(self,a,b,c):
        if a == 0:
            self.str = b*100
        else:
            self.str = c*100

a = Test()
def example():
    for i in xrange(0,100000):
        a.check(i,"b","c")

import profile
profile.run("example()")
```

And suppose this function gets called from somewhere else many times.

Well, your check will have an if statement slowing you down all the time except the first time, so you can do this:

```
class Test2:
    def check(self,a,b,c):
        self.str = b*100
        self.check = self.check_post
    def check_post(self,a,b,c):
        self.str = c*100

a = Test2()
def example2():
    for i in xrange(0,100000):
        a.check(i,"b","c")

import profile
profile.run("example2()")
```

Well, this example is fairly inadequate, but if the 'if' statement is a pretty complicated expression (or something with lots of dots), you can save yourself evaluating it, if you know it will only be true the first time.

Profiling Code

The first step to speeding up your program is learning where the bottlenecks lie. It hardly makes sense to optimize code that is never executed or that already runs fast. I use two modules to help locate the hotspots in my code, `profile` and `trace`. In later examples I also use the `timeit` module, which is new in Python 2.3.



See the separate [profiling](#) document for alternatives to the approaches given below.

Profiling

There are a number of [profiling modules](#) included in the Python distribution. Using one of these to profile the execution of a set of functions is quite easy. Suppose your main function is called `main`, takes no arguments and you want to execute it under the control of the `profile` module. In its simplest form you just execute

```
import profile
profile.run('main()')
```

When `main()` returns, the `profile` module will print a table of function calls and execution times. The output can be tweaked using the `stats` class included with the module. From Python 2.4 `profile` has permitted the time consumed by Python builtins and functions in extension modules to be profiled as well.

A slightly longer description of profiling using the `profile` and `pstats` modules can be found here (archived version):

<http://web.archive.org/web/20060506162444/http://wingware.com/doc/howtos/performance-profiling-python-code>

The cProfile and Hotshot Modules

Since Python 2.2, the [hotshot package](#) has been available as a replacement for the `profile` module, although the `cProfile` module is now recommended in preference to `hotshot`. The underlying module is written in C, so using `hotshot` (or `cProfile`) should result in a much smaller performance hit, and thus a more accurate idea of how your application is performing. There is also a `hotshotmain.py` program in the distribution's `Tools/scripts` directory which makes it easy to run your program under `hotshot` control from the command line.

Trace Module

The [trace module](#) is a spin-off of the `profile` module I wrote originally to perform some crude statement level test coverage. It's been heavily modified by several other people since I released my initial crude effort. As of Python 2.0 you should find `trace.py` in the `Tools/scripts` directory of the Python distribution. Starting with Python 2.3 it's in the standard library (the `Lib` directory). You can copy it to your local `bin` directory and set the execute permission, then execute it directly. It's easy to run from the command line to trace execution of whole scripts:

```
% trace.py -t spam.py eggs
```

In Python 2.4 it's even easier to run. Just execute `python -m trace`.

There's no separate documentation, but you can execute "`pydoc trace`" to view the inline documentation.

Visualizing Profiling Results

[RunSnakeRun](#) is a GUI tool by Mike Fletcher which visualizes profile dumps from `cProfile` using square maps. Function/method calls may be sorted according to various criteria, and source code may be displayed alongside the visualization and call statistics. Currently (April 2016) `RunSnakeRun` supports Python 2.x only - thus it cannot load profile data generated by Python 3 programs.

An example usage:


```
runsnake some_profile_dump.prof
```

[Gprof2Dot](#) is a python based tool that can transform profiling results output into a graph that can be converted into a PNG image or SVG.

A typical profiling session with python 2.5 looks like this (on older platforms you will need to use actual script instead of the `-m` option):



```
python -m cProfile -o stat.prof MYSCRIPT.PY [ARGS...]
```

```
python -m pbp.scripts.gprof2dot -f pstats -o stat.dot stat.prof  
dot -ostat.png -Tpng stat.dot
```

 [PyCallGraph](#) pycallgraph is a Python module that creates call graphs for Python programs. It generates a PNG file showing an modules's function calls and their link to other function calls, the amount of times a function was called and the time spent in that function.

Typical usage:

```
pycallgraph scriptname.py
```

 [PyProf2CallTree](#) is a script to help visualize profiling data collected with the cProfile python module with the  [kcachegrind](#) graphical calltree analyser.

Typical usage:

```
python -m cProfile -o stat.prof MYSCRIPT.PY [ARGS...]  
python pyprof2calltree.py -i stat.prof -k
```

 [ProfileEye](#) is a browser-based frontend to  [gprof2dot](#) using  [d3.js](#) for decluttering visual information.

Typical usage:

```
python -m profile -o output.pstats path/to/your/script arg1 arg2  
gprof2dot -f pstats output.pstats | profile_eye --file-colon_line-colon-  
label-format > profile_output.html
```

 [SnakeViz](#) is a browser-based visualizer for profile data.

Typical usage:

```
python -m profile -o output.pstats path/to/your/script arg1 arg2  
snakeviz output.pstats
```

[CategoryDocumentation](#)

PythonSpeed/PerformanceTips (last edited 2016-04-17 10:40:47 by [LarsKruise](#))

- » [MoinMoin Powered](#)
- » [Python Powered](#)
- » [GPL licensed](#)
- » [Valid HTML 4.01](#)

[Unable to edit the page? See the FrontPage for instructions.](#)