Performance optimization All Versions

Examples

Code profiling

First and foremost you should be able to find the bottleneck of your script and note that no optimization can compensate for a poor choice in data structure or a flaw in your algorithm design. Secondly do not try to optimize too early in your coding process at the expense of readability/design/quality. Donald Knuth made the following statement on optimization:

> "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%"

To profile your code you have several tools: `cProfile` (or the slower `profile`) from the standard library, `line_profiler` and `timeit` . Each of them serve a different purpose.

`cProfile` is a determistic profiler: function call, function return, and exception events are monitored, and precise timings are made for the intervals between these events (up to 0.001s). The library documentation ([https://docs.python.org/2/library/profile.html][1]) provides us with a simple use case

```
import cProfile
def f(x):
    return "42!"
cProfile.run('f(12)')
```

Or if you prefer to wrap parts of your existing code:

```
import cProfile, pstats, StringIO
pr = cProfile.Profile()
pr.enable()
# ... do something ...
# ... long ...
pr.disable()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print s.getvalue()
```

This will create outputs looking like the table below, where you can quickly see where your program spends most of its time and identify the functions to optimize.

```
         3 function calls in 0.000 seconds

Ordered by: standard name
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <stdin>:1(f)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

The module `line_profiler` ([https://github.com/rkern/line_profiler][1]) is useful to have a line by line analysis of your code. This is obviously not manageable for long scripts but is aimed at snippets. See the documentation for more details. The easiest way to get started is to use the kernprof script as explained one the package page, note that you will need to specify manually the function(s) to profile.

```
$ kernprof -l script_to_profile.py
```

kernprof will create an instance of LineProfiler and insert it into the `__builtins__` namespace with the name profile. It has been written to be used as a decorator, so in your script, you decorate the functions you want to profile with `@profile` .

```
@profile
def slow_function(a, b, c):
    ...
```

The default behavior of kernprof is to put the results into a binary file `script_to_profile.py.lprof` . You can tell kernprof to immediately view the formatted results at the terminal with the [-v/--view] option. Otherwise, you can view the results later like so:

```
$ python -m line_profiler script_to_profile.py.lprof
```

Finally `timeit` provides a simple way to test one liners or small expression both from the command line and the python shell. This module will answer question such as, is it faster to do a list comprehension or use the built-in `list()` when transforming a set into a list. Look for the `setup` keyword or `-s` option to add setup code.

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
```

```
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.8187260627746582
```

from a terminal

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop
```

Syntax

Parameters

Remarks

When attempting to improve the performance of a Python script, first and foremost you should be able to find the bottleneck of your script and note that no optimization can compensate for a poor choice in data structures or a flaw in your algorithm design. Identifying performance bottlenecks can be done by profiling your script. Secondly do not try to optimize too early in your coding process at the expense of readability/design/quality. Donald Knuth made the following statement on optimization:

> “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”