Python Profiling

Amjith Ramanujam

UTOS

@amjithr

Profiling a Python Script

What is Profiling?

Where is the time spent?

Profiling a Python Script

What is Profiling?

Where is the time spent?

Why?

- Know the bottle-necks.
- Optimize intelligently.

In God we trust everyone else bring data.

Profiling Tools

Standard Library

- cProfile
- Profile (older pure python implementation)
- hotshot (deprecated)
- timeit

Third Party

- line_profiler
- memory_profiler

Commercial - Web Application

- New Relic

cProfile

Introduction

Let’s use cProfile

```
$ python -m cProfile lcm.py
  7780242 function calls in 4.474 seconds

 Ordered by: standard name

 ncalls  tottime  percall  cumtime  percall filenam
      1    0.000    0.000    4.474    4.474 lcm.py:
      1    2.713    2.713    4.474    4.474 lcm.py:
3890120    0.881    0.000    0.881    0.000 {max}
      1    0.000    0.000    0.000    0.000 {method
3890119    0.880    0.000    0.880    0.000 {min}
```

Lowest Common Multiplier

Problem

Given two numbers a,b find the lowest number c that is divisible by both a and b. eg: lcm(2,3) is 6

Lowest Common Multiplier

Problem

Given two numbers a,b find the lowest number c that is divisible by both a and b. eg: lcm(2,3) is 6

Algorithm:

```
1. Start i from the max(a,b)
2. If i is perfectly divisible by a and b
    i is the answer
3. Increment i by max(a,b). Goto Step 1.
```

Lowest Common Multiplier (ver 1)

```
# lcm.py
def lcm(arg1, arg2):
    i = max(arg1, arg2)
    while i < (arg1 * arg2):
        if i % min(arg1,arg2) == 0:
            return i
        i += max(arg1,arg2)
    return(arg1 * arg2)

lcm(21498497, 3890120)
```

Let’s Profile (ver 1)

```
$ python -m cProfile lcm.py
  7780242 function calls in 4.474 seconds

 Ordered by: standard name

 ncalls  tottime  percall  cumtime  percall filenam
      1    0.000    0.000    4.474    4.474 lcm.py:
      1    2.713    2.713    4.474    4.474 lcm.py:
3890120    0.881    0.000    0.881    0.000 {max}
      1    0.000    0.000    0.000    0.000 {method
3890119    0.880    0.000    0.880    0.000 {min}
```

Lowest Common Multiplier (ver 2)

```
# lcm.py
def ver_2(arg1, arg2):
    mx = max(arg1, arg2)
    mn = min(arg1, arg2)
    i = mx
    while i < (arg1 * arg2):
        if i % mn == 0:
            return i
        i += mx
    return(arg1 * arg2)
```

Let’s Profile (ver 2)

```
$ python -m cProfile lcm.py
  5 function calls in 0.774 seconds

Ordered by: standard name
ncalls  tottime  percall  cumtime  percall filename
     1    0.000    0.000    0.763    0.763 lcm.py:2
     1    0.763    0.763    0.763    0.763 lcm.py:2
     1    0.000    0.000    0.000    0.000 {max}
     1    0.000    0.000    0.000    0.000 {method
     1    0.000    0.000    0.000    0.000 {min}
```

pstats

cProfile

Large Programs

Profiling Large Programs

```
$ python -m cProfile shorten.py
 95657 function calls (93207 primitive calls) in 1

 Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename
    39    0.000    0.000    0.001    0.000 <string
     1    0.000    0.000    0.000    0.000 <string
     1    0.000    0.000    0.000    0.000 <string
     1    0.000    0.000    0.000    0.000 <string
     1    0.000    0.000    0.000    0.000 <string
     1    0.000    0.000    0.000    0.000 <string
     1    0.000    0.000    0.000    0.000 <string
     1    0.000    0.000    0.000    0.000 <string
     1    0.000    0.000    0.000    0.000 <string
     1    0.000    0.000    0.000    0.000 <string
     1    0.000    0.000    0.000    0.000 <string
```

Profiling Large Programs

Problem:

- Profiles of bigger programs are messy.
- Ordering by function name is useless.

Profiling Large Programs

Problem:

- Profiles of bigger programs are messy.
- Ordering by function name is useless.

Solution:

- Save the profile to a file.
- Reload the profile and analyze using pStat.

Save the Profile

Let’s save the profile to a file.

```
$ python -m cProfile -o shorten.prof shorten.py

$ ls
shorten.py shorten.prof
```

Analyze the Profile

```
>>> import pstats
>>> p  = pstats.Stats('script.prof')
>>> p.sort_stats('calls')
>>> p.print_stats(5)

    95665 function calls (93215 primitive calls) in

   Ordered by: call count
   List reduced from 1919 to 5 due to restriction <

   ncalls  tottime  percall  cumtime  percall filen
10819/10539    0.002    0.000    0.002    0.000 {le
      9432    0.002    0.000    0.002    0.000 {me
      6061    0.003    0.000    0.003    0.000 {is
      3092    0.004    0.000    0.005    0.000 /ho
      2617    0.001    0.000    0.001    0.000 {me
```

RunSnakeRun

Profile Viewer GUI

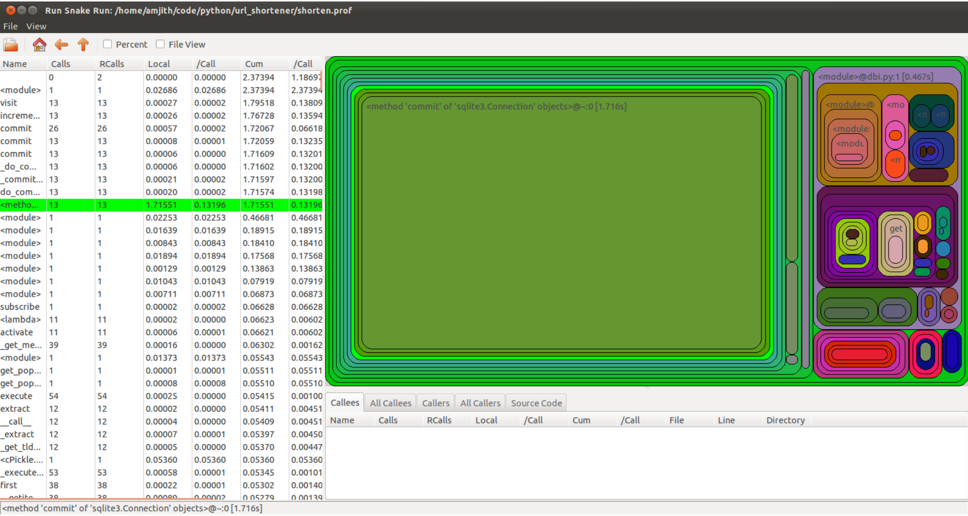
GUI Profile Viewer

- A GUI viewer for python profiles
- Shows the bigger picture
- Requires wxPython

```
$ pip install SquareMap RunSnakeRun
```

Smart Optimization

```
$ runsnake script.prof
```



Run Snake Run: /home/amjith/code/python/url_shortener/shorten.prof

Name	Calls	RCalls	Local	/Call	Cum	/Call
	0	2	0.00000	0.00000	2.37394	1.18697
<module>	1	1	0.02686	0.02686	2.37394	2.37394
visit	13	13	0.00027	0.00002	1.79518	0.13809
increme...	13	13	0.00026	0.00002	1.76728	0.13594
commit	26	26	0.00057	0.00002	1.72067	0.06618
commit	13	13	0.00008	0.00001	1.72059	0.13235
commit	13	13	0.00006	0.00000	1.71609	0.13201
_do_co...	13	13	0.00006	0.00000	1.71602	0.13200
_commit...	13	13	0.00021	0.00002	1.71597	0.13200
do_com...	13	13	0.00020	0.00002	1.71574	0.13198
<metho...	13	13	1.71551	0.13196	1.71551	0.13196
<module>	1	1	0.02253	0.02253	0.46681	0.46681
<module>	1	1	0.01639	0.01639	0.18915	0.18915
<module>	1	1	0.00843	0.00843	0.18410	0.18410
<module>	1	1	0.01894	0.01894	0.17568	0.17568
<module>	1	1	0.00129	0.00129	0.13863	0.13863
<module>	1	1	0.01043	0.01043	0.07919	0.07919
<module>	1	1	0.00711	0.00711	0.06873	0.06873
subscribe	1	1	0.00002	0.00002	0.06628	0.06628
<lambda>	11	11	0.00002	0.00000	0.06623	0.00602
activate	11	11	0.00006	0.00001	0.06621	0.00602
_get_me...	39	39	0.00016	0.00000	0.06302	0.00162
<module>	1	1	0.01373	0.01373	0.05543	0.05543
get_pop...	1	1	0.00001	0.00001	0.05511	0.05511
get_pop...	1	1	0.00008	0.00008	0.05510	0.05510
execute	54	54	0.00025	0.00000	0.05415	0.00100
extract	12	12	0.00002	0.00000	0.05411	0.00451
__call__	12	12	0.00004	0.00000	0.05409	0.00451
_extract	12	12	0.00007	0.00001	0.05397	0.00450
_get_tld...	12	12	0.00005	0.00000	0.05370	0.00447
<cPickle....	1	1	0.05360	0.05360	0.05360	0.05360
_execute...	53	53	0.00058	0.00001	0.05345	0.00101
first	38	38	0.00022	0.00001	0.05302	0.00140
__getite	38	38	0.00089	0.00002	0.05279	0.00139

Name	Calls	RCalls	Local	/Call	Cum	/Call	File	Line	Directory

<method 'commit' of 'sqlite3.Connection' objects>@~:0 [1.716s]

Clearly shows which parts are worth optimizing.

Profiling with Decorators

Fine grained control

Profiling Decorator

- Easy to use.
- Profiling specific functions in a larger program.

https://gist.github.com/1283366

Using Profiling Decorator

```
from profile_func import profile_func
@profile_func()
def convert_id_to_code(row_id):
    digits = []
    base = len(ALPHABET)
    while row_id > 0:
        digits.append(row_id % base)
        row_id = row_id / base
    digits.reverse()
    short_code = ''.join([ALPHABET[i] for i in digi
    return short_code
```

```
$ ls .profile
convert_id_to_code.profile
```

Line Profiler

Line Profiler

Fine Grain

Line Profiler

- What?
  - line-by-line stats on execution time.
- Why?
  - Sometimes function calls aren't enough information.
- How?
  - *$ pip install line_profiler*

Usage and Output

```
@profile
def compute(tokens):
    op_s = tokens[0]
    nums = map(int, tokens[1:])
    if op_s == "power":
        result = reduce(op.pow, nums)
    elif op_s == "plus":
        result = reduce(op.add, nums)
    return result
```

Usage and Output

```
$ kernprof.py -v -l compute.py data.txt

Line #      Hits         Time  Per Hit   % Time  Li
==================================================
     4                                           @p
     5                                           de
     6       606          843      1.4      2.9
     7       606         2607      4.3      8.9
     8       606          873      1.4      3.0
     9       101        20931    207.2     71.6
    10       505          624      1.2      2.1
    11       101          224      2.2      0.8
    12       606          794      1.3      2.7
```

Memory Profiler

Awesome - Experimental & Slow

Memory Profiler

- memory_profiler is a third party library for determining memory consumption.
- pip install memory_profiler
- line-by-line stats on cumulative memory usage.

Usage and Output

```
@profile
def func():
    a = [0] * 10
    b = [0] * 1000
    c = [0] * 10000000
    return a, b, c
```

Usage and Output

```
$ python -m memory_profiler -l -v mem_ex.py

Line #    Mem usage   Line Contents
===================================
     3                @profile
     4      6.65 MB   def func():
     5      6.66 MB       a = [0] * 10
     6      6.67 MB       b = [0] * 1000
     7     82.97 MB       c = [0] * 10000000
     8     82.97 MB       return a, b, c
```

Web Application Profiling

New Relic

Web Application Profiling

- New Relic is a commercial offering that specializes in web app performance monitoring.
- Provides *real-time* statistics on *production* servers.

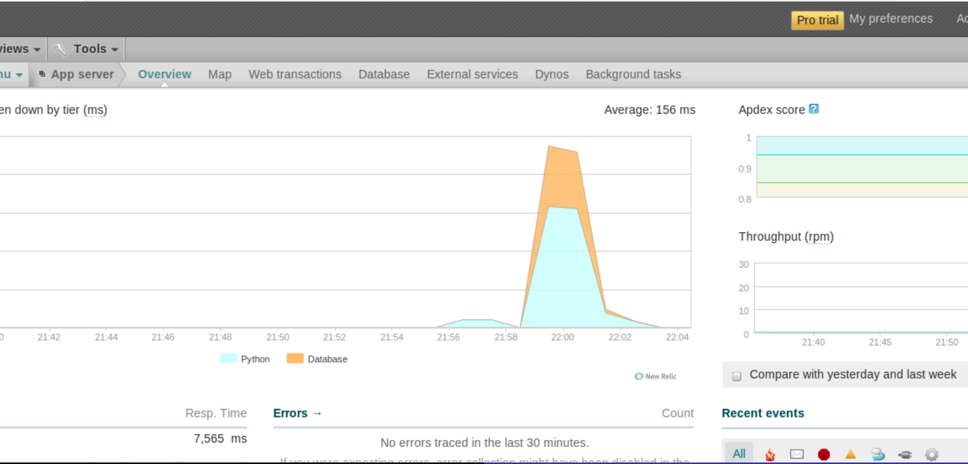
Metrics

- Time spent in Python vs Database.
- Slowest database queries.
- Water-fall graph of Web Transactions.
- etc...

Demo

New Relic Demo

http://productivemeter.herokuapp.com/productive



No errors traced in the last 30 minutes.

If you were expecting errors, error collection might have been disabled in the

Questions

Questions

slides : `http://github.com/User` twitter: @amjithr

Micro Benchmarks

timeit module

Micro Benchmarks with timeit

- *timeit* module can be used to profile individual statements or blocks in the code.
- Runs the code multiple times to collect more data points.
- Turns off Garbage Collector for accuracy.

```
$ python −m timeit ’range(0,1000)’
100000 loops, best of 3: 12 usec per loop
$ python −m timeit ’xrange(0,1000)’
1000000 loops, best of 3: 0.253 usec per loop
```