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}
```

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

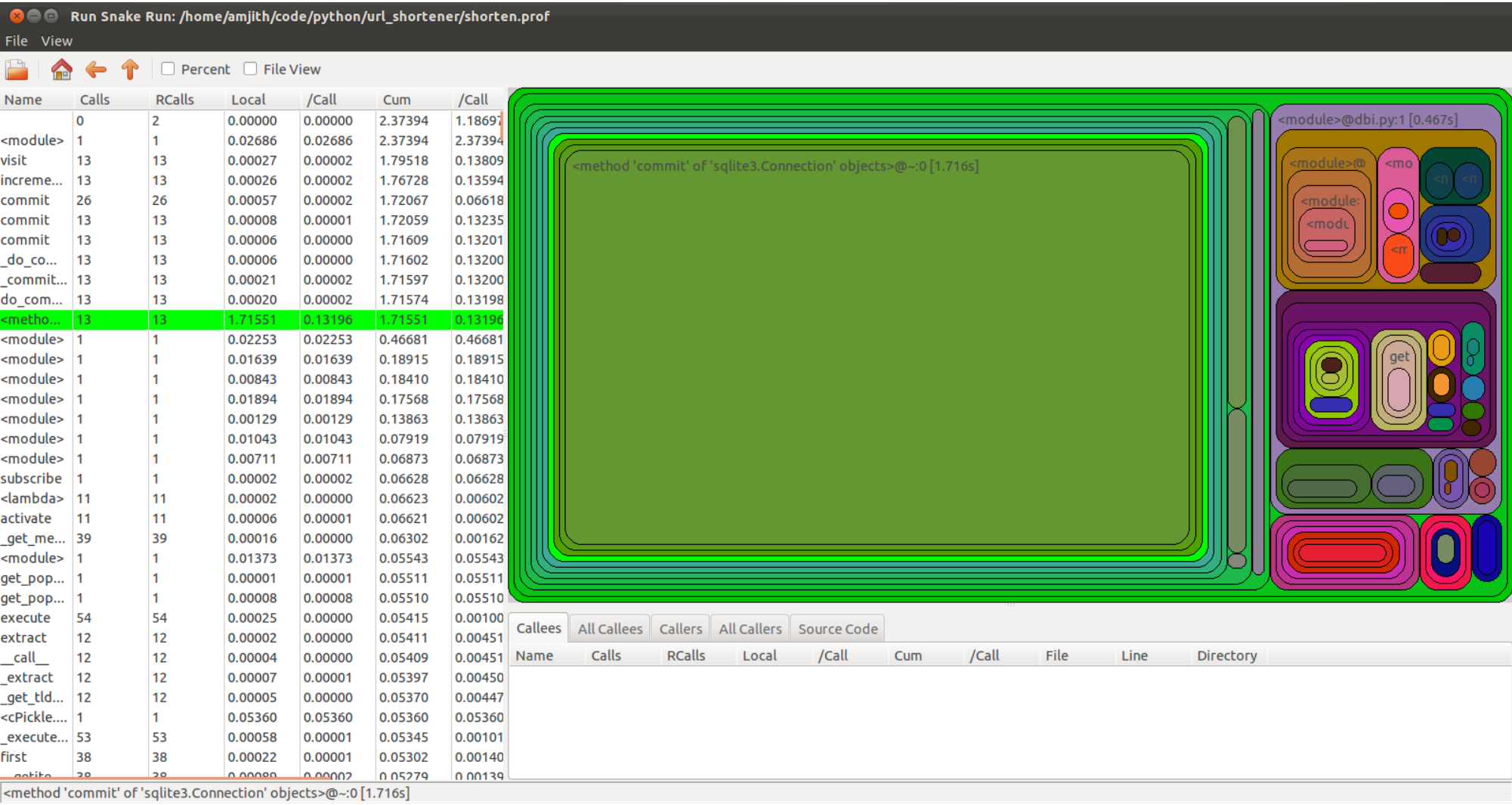
GUI Profile Viewer

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

```
$ pip install SquareMap RunSnakeRun
```

Smart Optimization

```
$ runsnake script.prof
```



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

Fine Grain

Line Profiler

- Why?
  - Sometimes function calls aren't enough information.
- What?
  - line-by-line stats on execution time.
- 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

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 Servers vs Database.
- Slowest database queries.

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
```

Questions