Premature optimization is a root of all evil

Проблемы при профилировании

Скорость работы зависит от множества факторов

- Статистическая обработка разультатов (делаем 5 тестов, отбрасываем самый медленный и самый быстрый)
- Контролируйте нагрузку на процессор и его тактовую частоту

Детерминированное и вероятностное профилирование

- Детерминированное - встраивание счетчиков в код
- Вероятностное - периодическая остановка программы и анализ времени останова (периодическая остановка по исчерпанию люого счетчика, а не только времени)

top/htop/atop/iotop/nettop/time & sysinternals

```
% time python some_script.py
real    0m1.028s
user    0m0.001s
sys     0m0.003s
```

timeit

Используется для профилирования быстрых конструкций

```
1 import timeit
2 print timeit.timeit("a + b", "a,b = 1,2") # 0.03747510910003
```

timeit

```
1       import timeit
2
3       for pow in range(6, 8):
4           print timeit.timeit("a + b",
5                               "a,b = 1,2",
6                               number=10 ** pow) / 10 ** pow
7
8       # 2.5 E-8
9       # 2.14E-8
10      # 2.1 E-8
```

```
1       zero_time = timeit.timeit("pass", "", number=number)
2       zero_time /=  number
3       timeit.timeit(..., number=number) / num - zero_time
```

profile & cProfile

```
1       import re
2       import cProfile
3
4       cProfile.run("re.compile('a|b|c' * 100 + 'c')")
```

```
4913 function calls (4712 primitive calls) in 0.002 seconds
Ordered by: standard name
ncalls tottime percall cumtime percall filename:lineno(function)
    1   0.000   0.000   0.002   0.002 <string>:1(<module>)
    1   0.000   0.000   0.002   0.002 re.py:188(compile)
    1   0.000   0.000   0.002   0.002 re.py:226(_compile)
    2   0.000   0.000   0.000   0.000 sre_compile.py:179(_compile_charset)
    2   0.000   0.000   0.000   0.000 sre_compile.py:208(_optimize_charset)
  408   0.000   0.000   0.000   0.000 sre_compile.py:25(_identityfunction)
    1   0.000   0.000   0.000   0.000 sre_compile.py:33(_compile)
    1   0.000   0.000   0.000   0.000 sre_compile.py:360(_compile_info)
    2   0.000   0.000   0.000   0.000 sre_compile.py:473(isstring)
    1   0.000   0.000   0.000   0.000 sre_compile.py:479(_code)
```

profile & cProfile

```
profile = cProfile.Profile()
try:
    profile.enable()
    result = func(*args, **kwargs)
    profile.disable()
    return result
finally:
    profile.print_stats()
    profile.dump_stats(fname)
```

```
import pstats
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

profile & cProfile

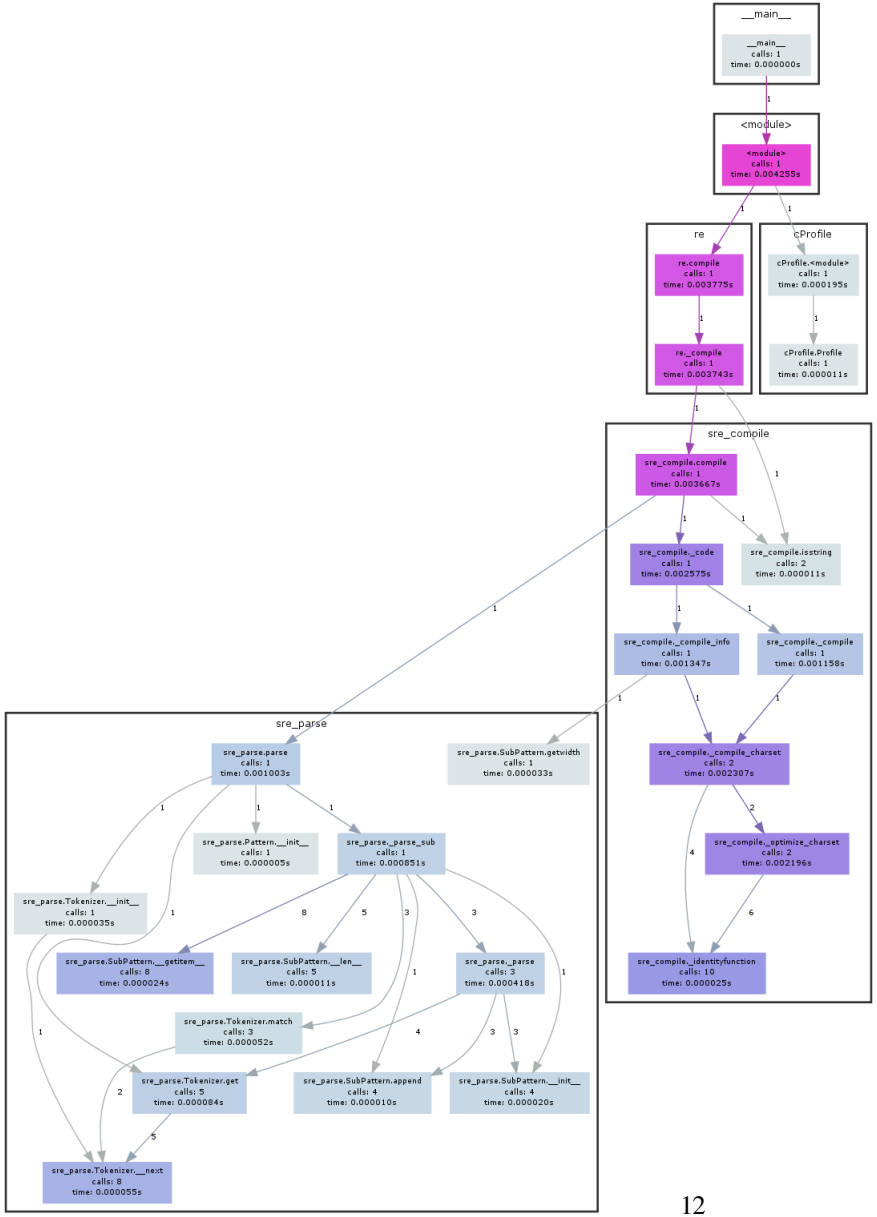
```
$ python -m cProfile [-o output_file] [-s sort_order] myscript.py
$ python -m cProfile --sort tottime 8_queen_task.py 12
[(7, 3), (6, 9), (5, 7), (11, 8), (8, 0), (3, 11), (10, 10), ...]
         4328231 function calls (4247059 primitive calls) in 1.802 seconds

Ordered by: internal time
 ncalls tottime percall cumtime percall filename:lineno(function)
  81173   1.237   0.000   1.528   0.000 8_queen_task.py:1(hit_to)
81173/1   0.274   0.000   1.802   1.802 8_queen_task.py:20(do_put_queens)
3516485   0.203   0.000   0.203   0.000 {method 'append' of 'list' objects}
 649397   0.088   0.000   0.088   0.000 {range}
      1   0.000   0.000   1.802   1.802 8_queen_task.py:1(<module>)
      1   0.000   0.000   1.802   1.802 8_queen_task.py:16(put_queens)
      1   0.000   0.000   0.000   0.000 {method 'disable' of '_lsprof.P'..}
```

pycallgraph

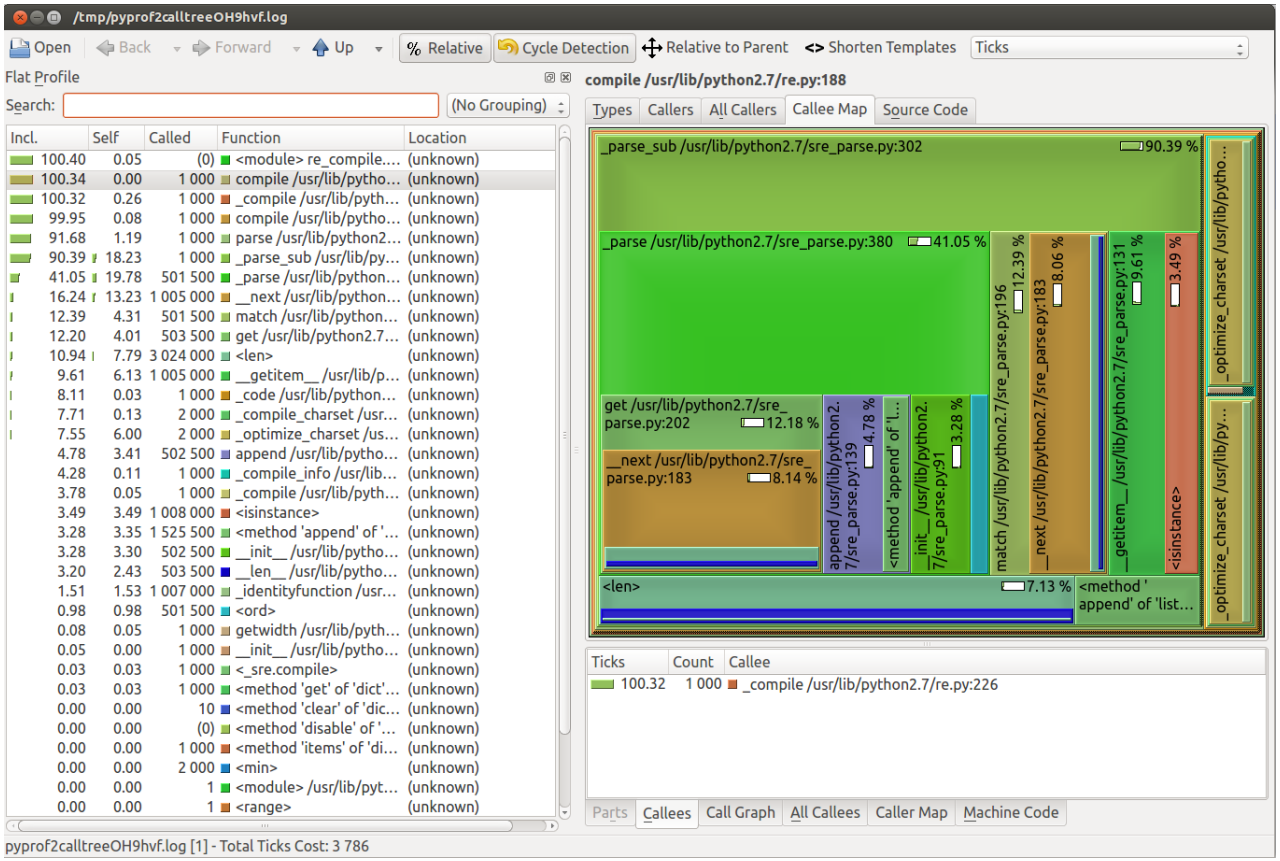
```
$ pip install pycallgraph
$ sudo apt-get install graphviz
$ pycallgraph graphviz -- ./mypythonscript.py
$ firefox pycallgraph.png
```

```
1 from pycallgraph import PyCallGraph
2 from pycallgraph.output import GraphvizOutput
3
4 with PyCallGraph(output=GraphvizOutput()):
5     code_to_profile()
```

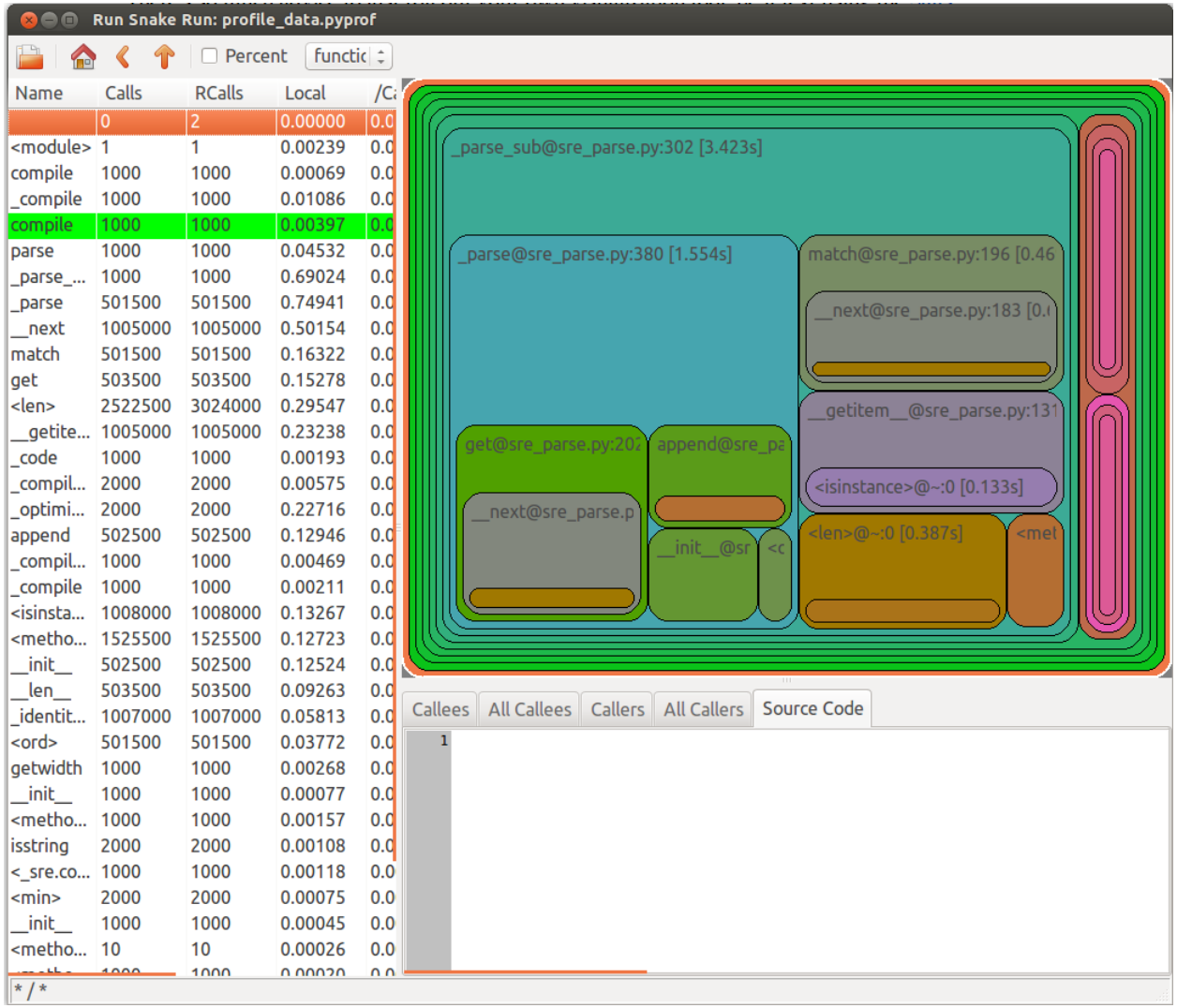

__main__
__main__
calls: 1
time: 0.000000s
<module>
<module>
calls: 1
time: 0.004255s
re
re.compile
calls: 1
time: 0.003775s
re._compile
calls: 1
time: 0.003743s
cProfile
cProfile.<module>
calls: 1
time: 0.000195s
cProfile.Profile
calls: 1
time: 0.000011s
sre_compile
sre_compile.compile
calls: 1
time: 0.003667s
sre_compile._code
calls: 1
time: 0.002575s
sre_compile.isstring
calls: 2
time: 0.000011s
sre_compile._compile_info
calls: 1
time: 0.001347s
sre_compile._compile
calls: 1
time: 0.001158s
sre_compile._compile_charset
calls: 2
time: 0.002307s
sre_compile._optimize_charset
calls: 2
time: 0.002196s
sre_compile._identityfunction
calls: 10
time: 0.000025s
sre_parse
sre_parse.parse
calls: 1
time: 0.001003s
sre_parse.SubPattern.getwidth
calls: 1
time: 0.000033s
sre_parse.Pattern.__init__
calls: 1
time: 0.000005s
sre_parse._parse_sub
calls: 1
time: 0.000851s
sre_parse.Tokenizer.__init__
calls: 1
time: 0.000035s
sre_parse.SubPattern.__getitem__
calls: 8
time: 0.000024s
sre_parse.SubPattern.__len__
calls: 5
time: 0.000011s
sre_parse._parse
calls: 3
time: 0.000418s
sre_parse.Tokenizer.match
calls: 3
time: 0.000052s
sre_parse.Tokenizer.get
calls: 5
time: 0.000084s
sre_parse.SubPattern.append
calls: 4
time: 0.000010s
sre_parse.SubPattern.__init__
calls: 4
time: 0.000020s
sre_parse.Tokenizer.__next
calls: 8
time: 0.000055s
Generated by Python Call Graph v1.0.1
http://pycallgraph.slowchop.com

kcachegrid & pyprof2calltree

```
$ python -m cProfile -o profile_data.pyprof re_compile.py
$ pyprof2calltree -i profile_data.pyprof -k
```



runsnakerun



line_profiler

```
$ sudo pip install line_profiler
$ kernprof.py -l -v queen_problem.py 12
```

```
1  @profile
2  def hit_to(pos, sz):
3      x, y = pos
4      cells = [pos]
5      for dx in (0, 1, -1):
6          for dy in (0, 1, -1):
7              if dx == 0 and dy == 0:
8                  continue
9
10             for i in range(sz):
11                 nx, ny = x + i * dx, y + i * dy
12                 if nx < 0 or nx > sz - 1 or ny > sz - 1 or ny <
13                     break
14                 cells.append((nx, ny))
15     return cells
```

line_profiler

```
....
#    Hits    Time PerHit %Time Line Contents
==============================================
1                               @profile
2                               def hit_to(pos, sz):
3   81173   34051    0.4   0.4     x, y = pos
4   81173   36006    0.4   0.4     cells = [pos]
5  324692  145524    0.4   1.6     for dx in (0, 1, -1):
6  974076  418874    0.4   4.7         for dy in (0, 1, -1):
7  730557  307418    0.4   3.4             if dx == 0 and dy == 0:
8   81173   31691    0.4   0.4                 continue
9
0 4165869 1771034    0.4  19.7             for i in range(sz):
1 4138314 2069723    0.5  23.0                 nx, ny = x + i * dx, y + i * dy
2 4138314 2164515    0.5  24.0                 if nx < 0 or nx > sz - 1 \
                                                      or ny > sz - 1 or ny < 0:
3  621829  293094    0.5   3.3                     break
4 3516485 1699301    0.5  18.9                 cells.append((nx, ny))
5   81173   30539    0.4   0.3     return cells
```

Профилирование потребления памяти

```
1 sys.getsizeof(1) == 24
2 sys.getsizeof("1213") == 41
3 sys.getsizeof([]) == 72
4 sys.getsizeof([1]) == 80
5 sys.getsizeof([1, 2]) == 88
6 sys.getsizeof({1, 2}) == 232
7 gc.get_referents({1,2,3}) == [1,2,3]
```

memory_profiler

```
$ pip install -U memory_profiler
$ python -m memory_profiler example.py
Line #    Mem usage  Increment   Line Contents
================================================
     3                           @profile
     4      5.97 MB    0.00 MB   def my_func():
     5     13.61 MB    7.64 MB       a = [1] * (10 ** 6)
     6    166.20 MB  152.59 MB       b = [2] * (2 * 10 ** 7)
     7     13.61 MB -152.59 MB       del b
     8     13.61 MB    0.00 MB       return a
```

Что еще

- yappi - performance
- Guppy-PE
- objgraph - memory, obj visualization
- PySizer - memory
- resource, psutil
- perf, oprofile, sar. perf top

ipython

```
%timeit a + b
%prun some_func
%load_ext memory_profiler
%load_ext line_profiler
%mprun
%lprun
```

Оптимизация

Скорость исполнения O(количество питон инструкций)

```
1 import dis
2
3 def mul2(x):
4     return (x + b) << 1
5 dis.dis(mul2)
```

```
  0 LOAD_FAST                0 (x)
  3 LOAD_GLOBAL              0 (b)
  6 BINARY_ADD
  7 LOAD_CONST               1 (1)
 10 BINARY_LSHIFT
 11 RETURN_VALUE
```

Низкоуровневая оптимизация

- Локальные переменные быстрее глобальных
- Взятие атрибута - дорогая операция
- Кешируйте, где можно
- Ищите подходящие встроенные методы
- cython/C++
- PerformanceTips
- optimizing-python-code

Локальные переменные быстрее глобальных

```
def c1(data):
   len(data); len(data); len(data); len(data); len(data)

%timeit c1(data)
1000000 loops, best of 3: 238 ns per loop

def c2(data):
   ln=len;ln(data); ln(data); ln(data); ln(data); ln(data)

%timeit c2(data)
10000000 loops, best of 3: 196 ns per loop
```

Доступ к атрибуту - дорогой

```
def r1():
    res = []
    for i in xrange(1000):
        res.append(i)

def r2():
    res = []
    a = res.append
    for i in xrange(1000):
        a(i)

%timeit r1()
10000 loops, best of 3: 56.9 µs per loop

%timeit r2()
10000 loops, best of 3: 33.1 µs per loop
```

Кешируйте, где можно

Шахматы

Ищите подходящие встроенные методы

- dict.setdefault
- dict.get
- sort
- heapq
- set operations