Профилирование и отладка Python, инструменты

Владимир

14 мин

109K

В предыдущей статье мы на практике разобрались, где и в каких случаях можно использовать ручное профилирование, а так же познакомились со статистическими профайлерами.

Сегодня мы познакомимся с основной и самой многочисленной группой инструментов — событийными профайлерами.

- Введение и теория — зачем вообще нужно профилирование, различные подходы, инструменты и отличия между ними
- Ручное и статистическое профилирование — переходим к практике
- **Событийное профилирование** — инструменты и их применение
- Отладка — что делать, когда ничего не работает

Задача для тренировки

В прошлой статье мы разбирали задачу 3 из Проекта Эйлера. Для разнообразия возьмём какой-нибудь другой пример, например, задачу 7 из этого же сборника задач:

> Выписав первые шесть простых чисел, получим 2, 3, 5, 7, 11 и 13. Очевидно, что 6-ое простое число — 13.
> Какое число является 10001-ым простым числом?

Пишем код:

```
"""Project Euler problem 7 solve"""
from __future__ import print_function
import math
import sys

def is_prime(num):
```

```
    """Checks if num is prime number"""
    for i in range(2, int(math.sqrt(num)) + 1):
        if num % i == 0:
            return False
    return True

def get_prime_numbers(count):
    """Get 'count' prime numbers"""
    prime_numbers = [2]
    next_number = 3

    while len(prime_numbers) < count:
        if is_prime(next_number):
            prime_numbers.append(next_number)
        next_number += 1

    return prime_numbers

if __name__ == '__main__':
    try:
        count = int(sys.argv[1])
    except (TypeError, ValueError, IndexError):
        sys.exit("Usage: euler_7.py number")
    if count < 1:
        sys.exit("Error: number must be greater than zero")

    prime_numbers = get_prime_numbers(count)
    print("Answer: %d" % prime_numbers[-1])
```

Помним, что код не идеален, и многие вещи можно сделать проще, лучше, быстрее. Именно в этом заключается цель данной статьи =)

В прошлой статье я оконфузился и не сделал самого важного: тестов. На самом деле поломать программу в процессе рефакторинга или оптимизации легче простого, и каждый цикл профилирования и переписывания кода должен в обязательном порядке сопровождаться тестированием функционала (как непосредственно затронутого изменениями, так и всего остального, ведь сайд-эффекты такие сайд-эффекты). Попробуем исправиться и добавим тесты в нашу программу. Самый простой и подходящий для такого простого скрипта вариант — использовать модуль doctest. Добавляем тесты и запускаем их:

Тесты

```
"""Project Euler problem 7 solve"""
from __future__ import print_function
import math
import sys
```

```
def is_prime(num):
    """
    Checks if num is prime number.

    >>> is_prime(2)
    True
    >>> is_prime(3)
    True
    >>> is_prime(4)
    False
    >>> is_prime(5)
    True
    >>> is_prime(41)
    True
    >>> is_prime(42)
    False
    >>> is_prime(43)
    True
    """
    for i in range(2, int(math.sqrt(num)) + 1):
        if num % i == 0:
            return False
    return True

def get_prime_numbers(count):
    """
    Get 'count' prime numbers.

    >>> get_prime_numbers(1)
    [2]
    >>> get_prime_numbers(2)
    [2, 3]
    >>> get_prime_numbers(3)
    [2, 3, 5]
    >>> get_prime_numbers(6)
    [2, 3, 5, 7, 11, 13]
    >>> get_prime_numbers(9)
    [2, 3, 5, 7, 11, 13, 17, 19, 23]
    >>> get_prime_numbers(19)
    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67]
    """
    prime_numbers = [2]
    next_number = 3

    while len(prime_numbers) < count:
        if is_prime(next_number):
            prime_numbers.append(next_number)
        next_number += 1
```

```
    return prime_numbers

if __name__ == '__main__':
    try:
        count = int(sys.argv[1])
    except (TypeError, ValueError, IndexError):
        sys.exit("Usage: euler_7.py number")
    if count < 1:
        sys.exit("Error: number must be greater than zero")

    prime_numbers = get_prime_numbers(count)
    print("Answer: %d" % prime_numbers[-1])
```

Запуск тестов и результат

```
➜ python -m doctest -v euler_7.py
Trying:
    get_prime_numbers(1)
Expecting:
    [2]
ok
Trying:
    get_prime_numbers(2)
Expecting:
    [2, 3]
ok
Trying:
    get_prime_numbers(3)
Expecting:
    [2, 3, 5]
ok
Trying:
    get_prime_numbers(6)
Expecting:
    [2, 3, 5, 7, 11, 13]
ok
Trying:
    get_prime_numbers(9)
Expecting:
    [2, 3, 5, 7, 11, 13, 17, 19, 23]
ok
Trying:
    get_prime_numbers(19)
Expecting:
    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67]
```

```
ok
Trying:
    is_prime(2)
Expecting:
    True
ok
Trying:
    is_prime(3)
Expecting:
    True
ok
Trying:
    is_prime(4)
Expecting:
    False
ok
Trying:
    is_prime(5)
Expecting:
    True
ok
Trying:
    is_prime(41)
Expecting:
    True
ok
Trying:
    is_prime(42)
Expecting:
    False
ok
Trying:
    is_prime(43)
Expecting:
    True
ok
1 items had no tests:
    euler_7
2 items passed all tests:
   6 tests in euler_7.get_prime_numbers
   7 tests in euler_7.is_prime
13 tests in 3 items.
13 passed and 0 failed.
Test passed.
```

Давайте посмотрим, насколько быстрый у нас получился код:

```
➜ python -m timeit -n 10 -s'import euler_7' 'euler_7.get_prime_numbers(10001)'
10 loops, best of 3: 1.27 sec per loop
```

Да, небыстро, есть что пооптимизировать =)

Инструменты

Стандартные библиотеки Python поражают своим разнообразием. В них, кажется, есть всё, что только может понадобится разработчику, и профайлеры не исключение. На самом деле их целых три «из коробки»:

- **cProfile** — относительно новый (с версии 2.5) модуль, написанный на C и оттого быстрый
- **profile** — нативная реализация профайлера (написан на чистом питоне), медленный, и поэтому не рекомендуется к использованию
- **hotshot** — экспериментальный модуль на си, очень быстрый, но больше не поддерживается и в любой момент может быть удалён из стандартных библиотек

cProfile

Какой разговор о профилировании питона обходится без описания cProfile — одного из стандартных модулей Python?
Уверен, каждый программист Python хоть раз пробовал запустить cProfile:

```
➜ python -m cProfile -s time euler_7.py 10001
         428978 function calls in 1.552 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   104741    0.955    0.000    1.361    0.000 euler_7.py:7(is_prime)
   104741    0.367    0.000    0.367    0.000 {range}
        1    0.162    0.162    1.550    1.550 euler_7.py:32(get_prime_numbers)
   104741    0.039    0.000    0.039    0.000 {math.sqrt}
   104742    0.024    0.000    0.024    0.000 {len}
    10000    0.003    0.000    0.003    0.000 {method 'append' of 'list' objects}
        1    0.001    0.001    1.552    1.552 euler_7.py:1(<module>)
        1    0.000    0.000    0.000    0.000 {print}
        1    0.000    0.000    0.000    0.000 __future__.py:48(<module>)
        7    0.000    0.000    0.000    0.000 __future__.py:75(__init__)
        1    0.000    0.000    0.000    0.000 __future__.py:74(_Feature)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Сразу заметим разницу во времени выполнения программы: без профайлера: 1.27 секунды, с профайлером: 1.55 секунд, то есть на

20% медленнее в нашем конкретном случае. И это ещё очень хороший результат!

Итак, мы видим, что самая долгая (по суммарному времени) операция — функция **is_prime**. Практически всё время программа проводит в этой функции. Следующий по «тяжести» вызов — функция **range**, которая как раз вызывается из функции is_prime. Читаем документацию и понимаем, что при вызове range в памяти создаётся список со всеми числами из заданного диапазона. С учётом того, что функция range вызывается 104741 раз, а верхняя граница диапазона при каждом вызове инкрементируется (перебираем числа последовательно), можно сделать вывод, что длина списка, создаваемого функцией range достигает сотни тысяч элементов к концу работы программы и список создаётся больше ста тысяч раз. Почитав ещё документацию мы узнаём, что нам следует использовать функцию **xrange** в этом цикле (внимательный читатель должен почувствовать сарказм в этом месте, ведь любой питонист знает про range VS xrange). Плюсом замены range на xrange будет так же явная экономия памяти (эту теорию мы проверим позже). Заменяем, запускаем тесты: всё ок, запускаем профайлер:

```
→ python -m cProfile -s time euler_7.py 10001
         324237 function calls in 1.010 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   104741    0.825    0.000    0.857    0.000 euler_7.py:7(is_prime)
        1    0.127    0.127    1.009    1.009 euler_7.py:32(get_prime_numbers)
   104741    0.032    0.000    0.032    0.000 {math.sqrt}
   104742    0.022    0.000    0.022    0.000 {len}
    10000    0.003    0.000    0.003    0.000 {method 'append' of 'list' objects}
        1    0.001    0.001    1.010    1.010 euler_7.py:1(<module>)
        1    0.000    0.000    0.000    0.000 {print}
        1    0.000    0.000    0.000    0.000 __future__.py:48(<module>)
        7    0.000    0.000    0.000    0.000 __future__.py:75(__init__)
        1    0.000    0.000    0.000    0.000 __future__.py:74(_Feature)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

1.010 секунды вместо 1.552, то есть в полтора раза быстрее! Нормально. Теперь самое узкое место в программе — функция **is_prime** сама по себе. Оптимизируем её позже, с использованием других инструментов.

Выводить результаты профилирования в консоль не всегда удобно, гораздо удобнее сохранять их в файл для дальнейшего анализа. Для этого можно воспользоваться ключом "-o":

```
→ python -m cProfile -o euler_7.prof euler_7.py 10001
→ ls
euler_7.prof      euler_7.py
```

или можно воспользоваться простейшим декоратором:

```
import cProfile

def profile(func):
    """Decorator for run function profile"""
    def wrapper(*args, **kwargs):
        profile_filename = func.__name__ + '.prof'
        profiler = cProfile.Profile()
        result = profiler.runcall(func, *args, **kwargs)
        profiler.dump_stats(profile_filename)
        return result
    return wrapper

@profile
def get_prime_numbers(count):
    ...
```

И тогда при каждом вызове функции foo будет сохранён файл с результатами профилирования («get_prime_numbers.prof» в нашем случае).

hotshot

hotshot — ещё один стандартный модуль Python, на данный момент не поддерживается и в любое время может быть удалён из стандартных библиотек. Но пока он есть, можно использовать его, ведь он очень быстрый и даёт минимальный оверхед при запуске программы под профайлером. Использовать его очень просто:

```
import hotshot

prof = hotshot.Profile("profile_name.prof")
prof.start()

# your code goes here

prof.stop()
prof.close()
```

Или в виде декоратора:

```
import hotshot

def profile(func):
    """Decorator for run function profile"""
    def wrapper(*args, **kwargs):
        profile_filename = func.__name__ + '.prof'
        profiler = hotshot.Profile(profile_filename)
        profiler.start()
```

```
        result = func(*args, **kwargs)
        profiler.stop()
        profiler.close()
        return result
    return wrapper

@profile
def get_prime_numbers(count):
    ...
```

Анализ результатов профилирования

Редко когда получается вывести результаты профилирования на экран сразу после запуска программы. Да и смысла в таких результатах немного: только простейшие скрипты и удастся изучить. Для просмотра и анализа данных лучше воспользоваться встроенным в Python модулем pstats (удобнее в сочетании с замечательной консолью iPython):

```
→ ipython

In [1]: import pstats

In [2]: p = pstats.Stats('get_prime_numbers.prof')

In [3]: p.sort_stats('calls').print_stats()

         324226 function calls in 1.018 seconds

   Ordered by: call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   104742    0.023    0.000    0.023    0.000 {len}
   104741    0.821    0.000    0.854    0.000 euler_7.py:19(is_prime)
   104741    0.034    0.000    0.034    0.000 {math.sqrt}
    10000    0.003    0.000    0.003    0.000 {method 'append' of 'list' objects}
        1    0.138    0.138    1.018    1.018 euler_7.py:44(get_prime_numbers)
```

Консоль, конечно, хорошая штука, но не очень наглядная. В особо сложных ситуациях, в программах с сотнями и тысячами вызовов анализ результатов представляется затруднительным. Так уж устроены люди, что графическую информацию нам (в отличие от компьютеров) воспринимать гораздо проще, чем текстовую. На помощь приходят различные инструменты для анализа результатов профилирования.

kcachegrind

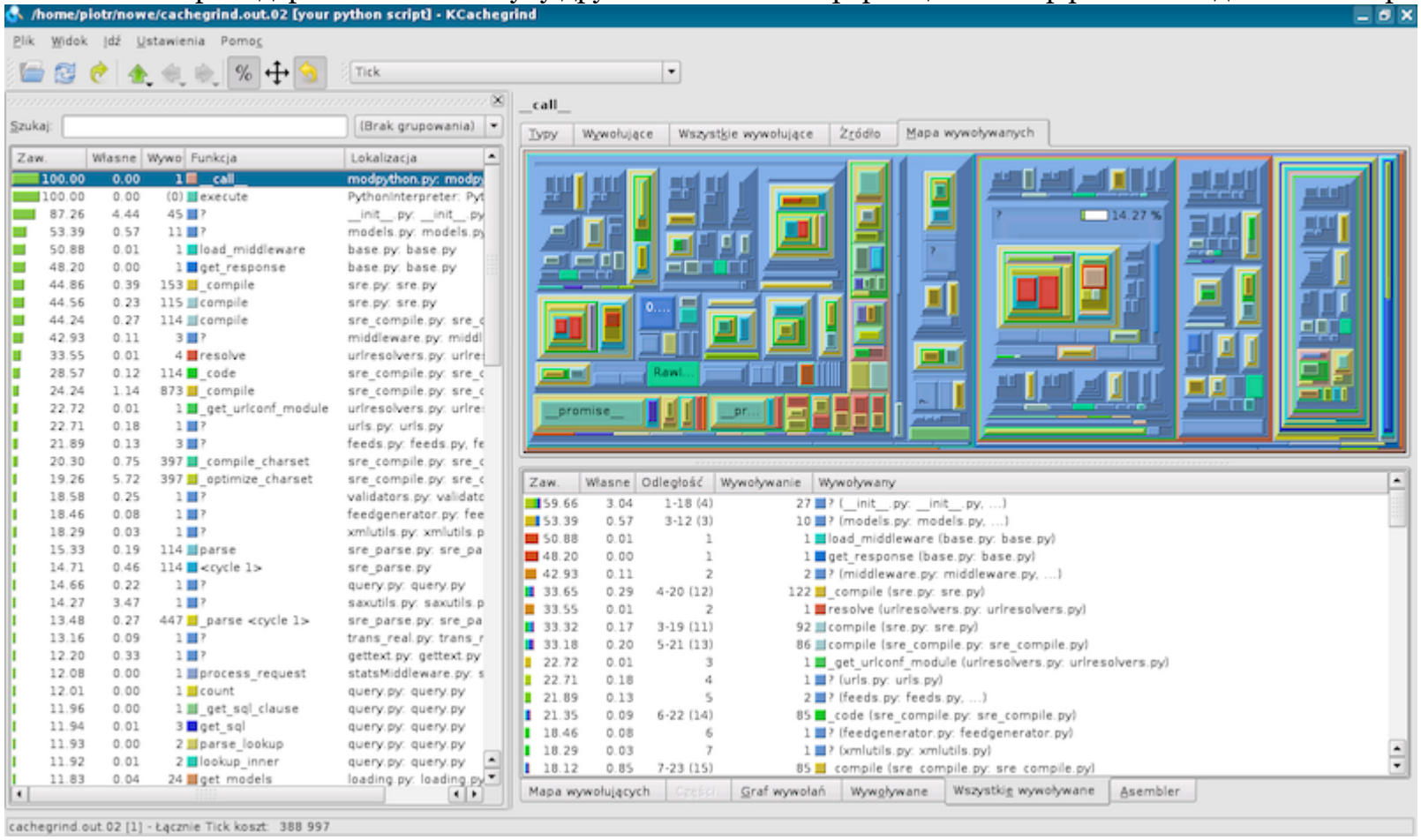
Начну, пожалуй, с такого известного инструмента, как kcachegrind, который, на самом деле, предназначен для визуализации результатов утилиты Callgrind, но переконвертировав результаты Python-профалера, их можно открыть в kcachegrind. Конвертирование выполняется с помощью утилиты pyprof2calltree:

```
➜ pip install pyprof2calltree
➜ pyprof2calltree -i profile_results.prof -o profile_results.kgrind
```

Можно сразу открыть результаты в kcachegrind, без сохранения в файл:

```
➜ pyprof2calltree -i profile_results.prof -k
```

Программа позволяет наглядно увидеть сколько времени занимает тот или иной вызов, а так же все вызовы внутри него. Так же можно посмотреть дерево вызовов и кучу другой полезной информации. Интерфейс выглядит таким образом:



RunSnakeRun

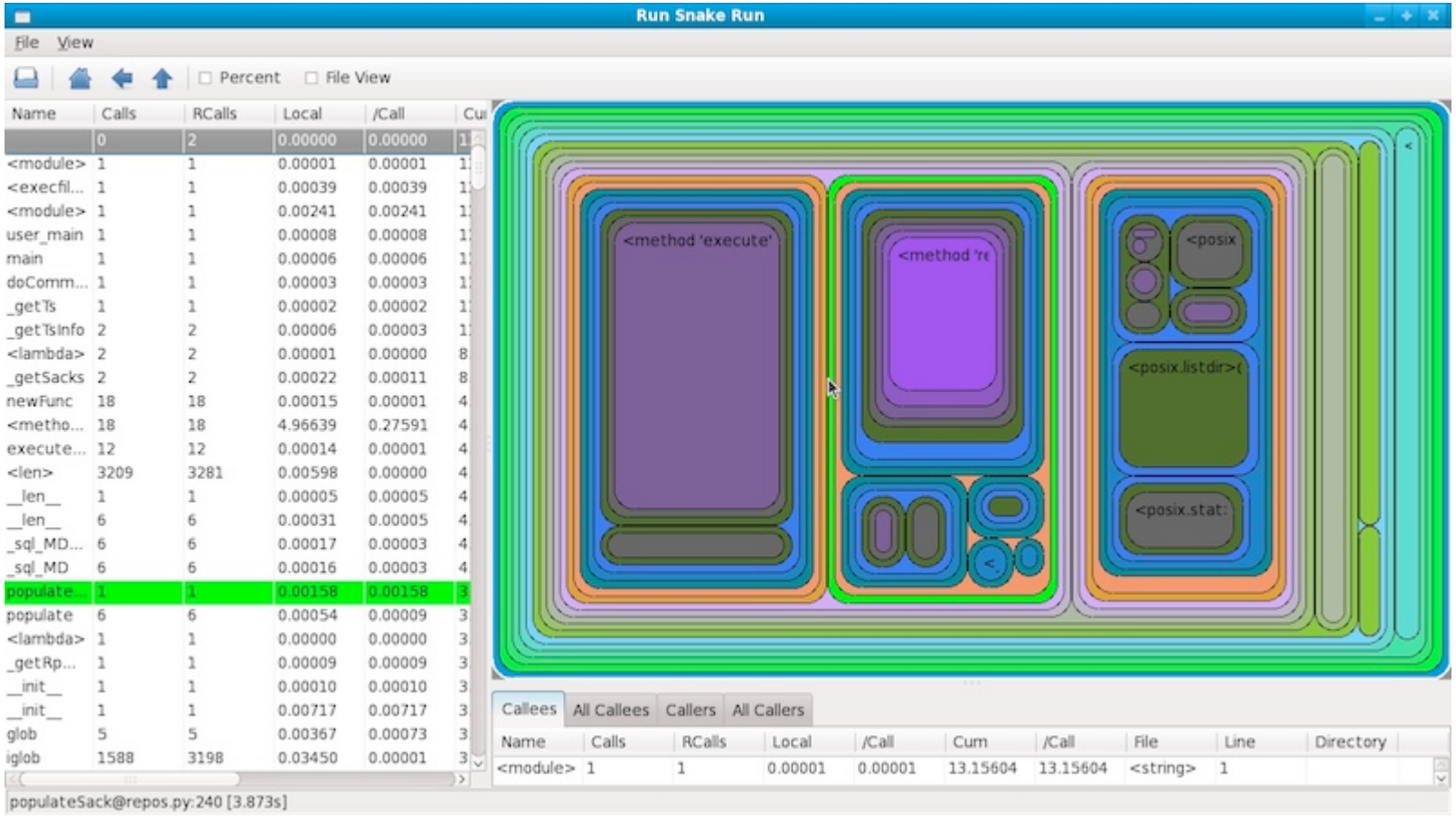
Ещё одна программа для визуализации результатов профайлинга RunSnakeRun была изначально написана для работы с профайлером питона (это видно из её названия). Похожа на kcachegrind, но, как говорят авторы, выгодно отличается более простым интерфейсом и функционалом. Установка не вызовет сложностей:

```
→ brew install wxwidgets
→ pip install SquareMap RunSnakeRun
```

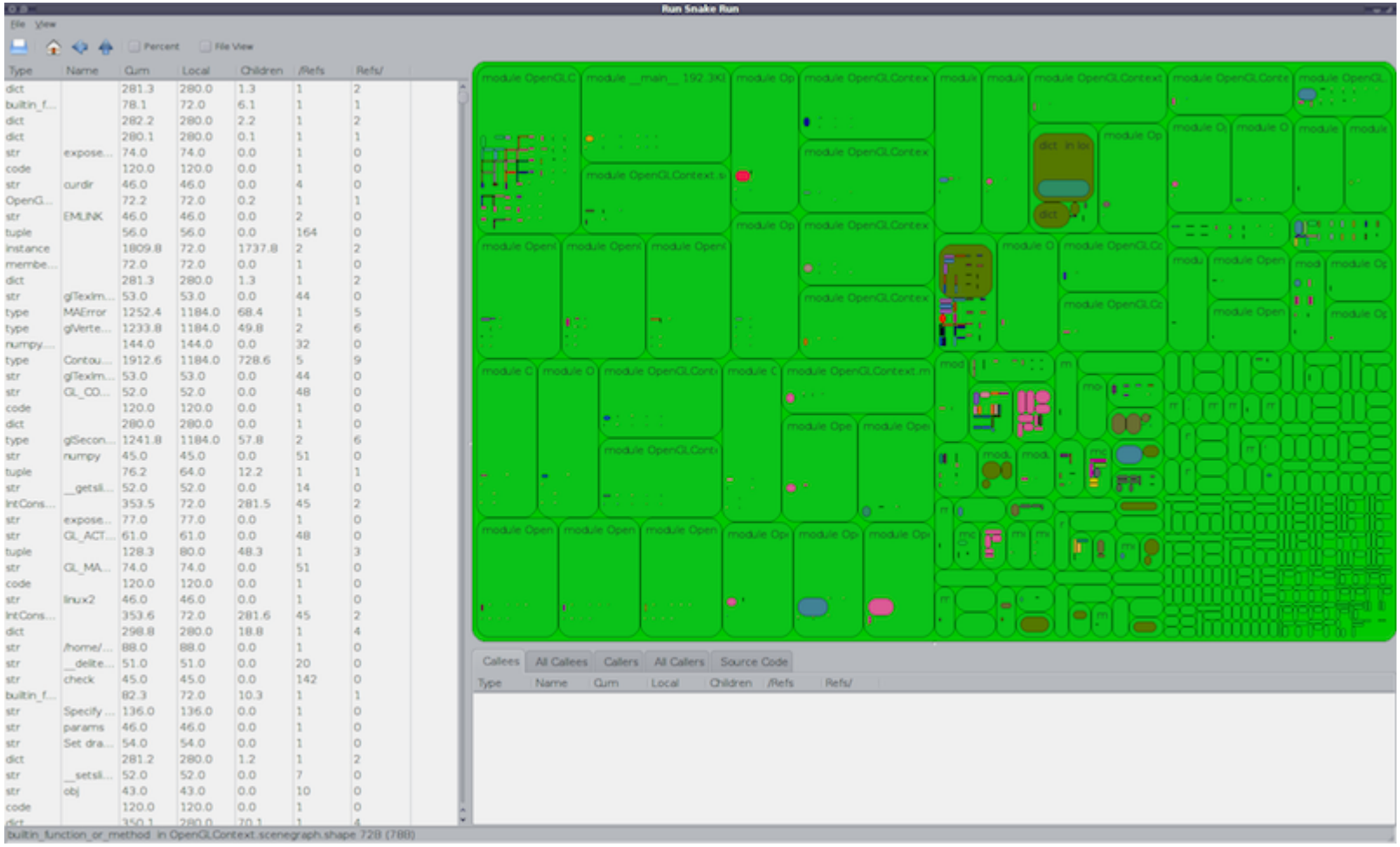
Использование тоже:

```
→ runsnake profile_results.prof
```

Точно так же видим карту квадратов: чем больше площадь квадрата, тем больше времени заняло выполнение соответствующей функции:



RunSnakeRun позволяет так же визуализировать результат профилирования потребления памяти с помощью утилиты Meliae:

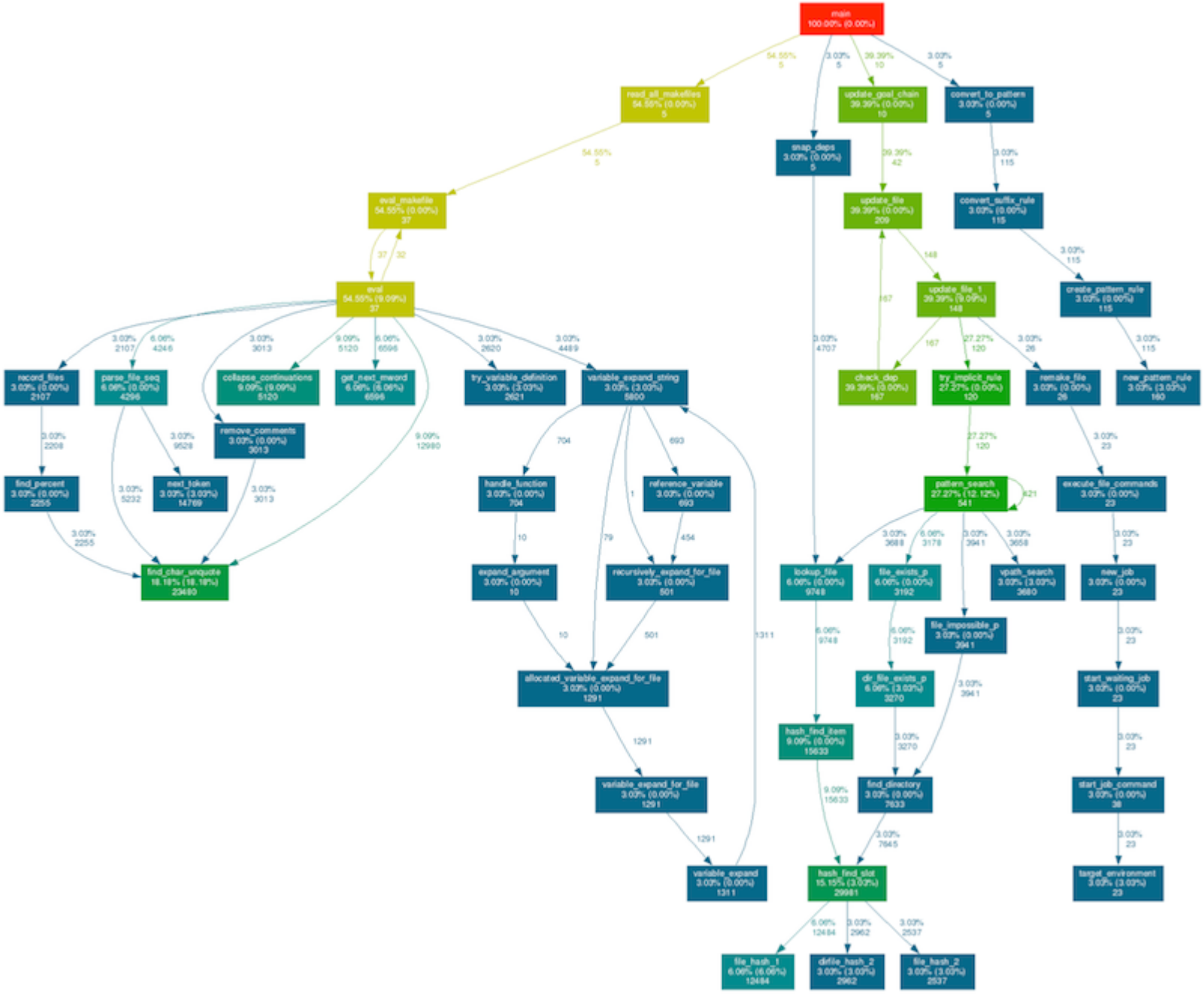


gprof2dot

Утилита gprof2dot генерирует картинку с деревом вызовов функций и информацией о времени их выполнения. В большинстве случаев этого достаточно для поиска узких мест в программе. Ставим и генерируем картинку:

```
➜ brew install graphviz
➜ pip install gprof2dot
➜ gprof2dot -f pstats profile_results.prof | dot -Tpng -o profile_results.png
```

Результат:


main
100.00% (0.00%)
54.55%
5
3.03%
5
39.39%
10
3.03%
5
read_all_makefiles
54.55% (0.00%)
5
update_goal_chain
39.39% (0.00%)
10
convert_to_pattern
3.03% (0.00%)
5
54.55%
5
snap_deps
3.03% (0.00%)
5
39.39%
42
3.03%
115
eval_makefile
54.55% (0.00%)
37
update_file
39.39% (0.00%)
209
convert_suffix_rule
3.03% (0.00%)
115
37
32
148
3.03%
115
eval
54.55% (9.09%)
37
167
update_file_1
39.39% (9.09%)
148
create_pattern_rule
3.03% (0.00%)
115
3.03%
4707
3.03%
2107
6.06%
4246
3.03%
3013
9.09%
5120
6.06%
6596
3.03%
2620
3.03%
4489
167
27.27%
120
3.03%
26
3.03%
115
record_files
3.03% (0.00%)
2107
parse_file_seq
6.06% (0.00%)
4296
collapse_continuations
9.09% (9.09%)
5120
get_next_mword
6.06% (6.06%)
6596
try_variable_definition
3.03% (3.03%)
2621
variable_expand_string
3.03% (3.03%)
5800
chack_dep
39.39% (0.00%)
167
try_implicit_rule
27.27% (0.03%)
120
remake_file
3.03% (0.00%)
26
new_pattern_rule
3.03% (3.03%)
160
3.03%
2208
3.03%
9528
remove_comments
3.03% (0.00%)
3013
9.09%
12980
704
693
27.27%
120
3.03%
23
find_percent
3.03% (0.00%)
2255
3.03%
5232
next_token
3.03% (3.03%)
14789
3.03%
3013
handle_function
3.03% (0.00%)
704
reference_variable
3.03% (0.00%)
693
pattern_search
27.27% (12.12%)
541
821
execute_file_commands
3.03% (0.00%)
23
3.03%
2255
1
79
10
454
3.03%
3688
6.06%
3178
3.03%
3941
3.03%
3658
3.03%
23
find_char_unquote
18.18% (18.18%)
23480
expand_argument
3.03% (0.00%)
10
recursively_expand_for_file
3.03% (0.00%)
501
lookup_file
6.06% (0.00%)
9748
file_exists_p
6.06% (0.00%)
3192
vpath_search
3.03% (3.03%)
3680
new_job
3.03% (0.00%)
23
file_impossible_p
3.03% (0.00%)
3941
10
501
1311
6.06%
9748
6.06%
3192
3.03%
23
allocated_variable_expand_for_file
3.03% (0.00%)
1291
dir_file_exists_p
6.06% (3.03%)
3270
start_waiting_job
3.03% (0.00%)
23
3.03%
3941
1291
hash_find_item
9.09% (0.00%)
15633
3.03%
3270
3.03%
23
variable_expand_for_file
3.03% (0.00%)
1291
find_directory
3.03% (0.00%)
7633
start_job_command
3.03% (0.00%)
38
9.09%
15633
3.03%
7645
3.03%
23
1291
variable_expand
3.03% (0.00%)
1311
hash_find_slot
15.15% (3.03%)
29981
target_environment
3.03% (3.03%)
23
6.06%
12484
3.03%
2962
3.03%
2537
file_hash_1
6.06% (6.06%)
12484
difile_hash_2
3.03% (3.03%)
2962
file_hash_2
3.03% (3.03%)
2537

Профилирование Django

Для профилирования Django удобно использовать модуль django-extensions, который, помимо кучи разных полезных вещей, имеет полезную команду «runprofileserver». Использовать его просто. Ставим:

```
→ pip install django-extensions
```

Добавляем application в settings.py:

```
INSTALLED_APPS += ('django_extensions',)
```

Запускаем:

```
→ python manage.py runprofileserver --use-cprofile --prof-path=/tmp/prof/
```

В директории /tmp/prof/ будет создан файл с результатами профилирования для каждого запроса в приложение:

```
→ ls /tmp/prof/
admin.000276ms.1374075009.prof
admin.account.user.000278ms.1374075014.prof admin.jsi18n.000185ms.1374075018.prof
favicon.ico.000017ms.1374075001.prof
root.000073ms.1374075004.prof
static.admin.css.base.css.000011ms.1374075010.prof
static.admin.css.forms.css.000013ms.1374075017.prof
static.admin.img.icon-yes.gif.000001ms.1374075015.prof
static.admin.img.sorting-icons.gif.000001ms.1374075015.prof
static.admin.js.core.js.000018ms.1374075014.prof
static.admin.js.jquery.js.000003ms.1374075014.prof
static.css.bootstrap-2.3.2.min.css.000061ms.1374074996.prof
static.img.glyphicons-halflings.png.000001ms.1374075005.prof
static.js.bootstrap-2.3.2.min.js.000004ms.1374074996.prof
static.js.jquery-2.0.2.min.js.000001ms.1374074996.prof
user.login.000187ms.1374075001.prof
```

Дальнейший анализ можно выполнить с помощью любого из инструментов, перечисленных выше: pstats, kcachegrind, RunSnakeRun или gprof2dot. Или любого другого =)

Помимо встроенного в Python профайлера имеется так же масса сторонних программ, простых и сложных, полезных и не очень.

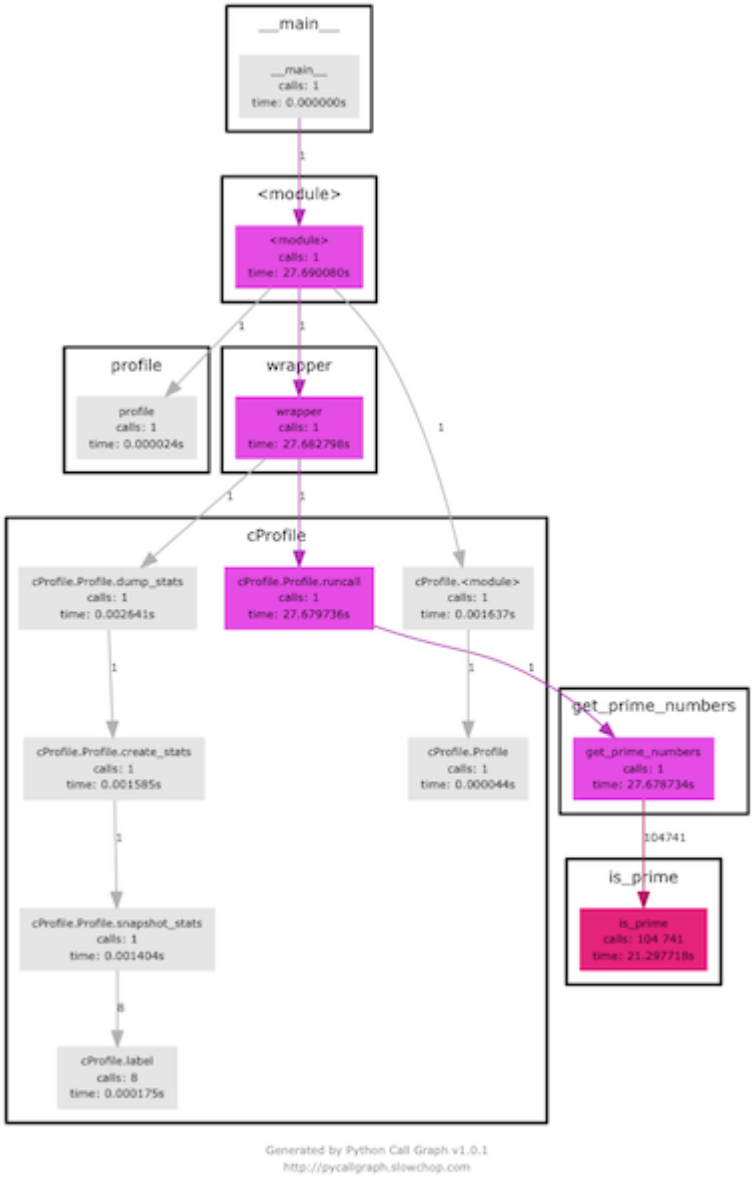
pycallgraph

pycallgraph позволяет строить дерево вызовов программы Python. Ставим:

```
➜ brew install graphviz
➜ pip install pycallgraph
```

Запускаем и смотрим результат:

```
➜ pycallgraph graphviz -- euler_7.py 10001
➜ open pycallgraph.png
```



line_profiler

line_profiler, как следует из его названия, позволяет построчно отпрофилировать нужные участки кода. Ставим:

```
→ pip install line_profiler
```

Добавляем в нужные места декоратор «profile» (я временно убрал докстринги для более компактного вывода результатов):

```
@profile
def is_prime(num):
    for i in xrange(2, int(math.sqrt(num)) + 1):
        if num % i == 0:
            return False
    return True

@profile
def get_prime_numbers(count):
    prime_numbers = [2]
    next_number = 3

    while len(prime_numbers) < count:
        if is_prime(next_number):
            prime_numbers.append(next_number)
        next_number += 1

    return prime_numbers
```

Запускаем профилирование:

```
→ kernprof.py -v -l euler_7.py 10001
Wrote profile results to euler_7.py.lprof
Timer unit: 1e-06 s

File: euler_7.py
Function: is_prime at line 7
Total time: 10.7963 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def is_prime(num):
     9   2935963      5187211      1.8     48.0      for i in xrange(2, int(math.sqrt(num)) + 1):
    10   2925963      5421919      1.9     50.2          if num % i == 0:
    11     94741       169309      1.8      1.6              return False
    12     10000        17904      1.8      0.2      return True

File: euler_7.py
Function: get_prime_numbers at line 15
Total time: 23.263 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
```

```
============================================================
    15                                           @profile
    16                                           def get_prime_numbers(count):
    17         1            5      5.0      0.0      prime_numbers = [2]
    18         1            3      3.0      0.0      next_number = 3
    19
    20    104742       208985      2.0      0.9      while len(prime_numbers) < count:
    21    104741     22843717    218.1     98.2          if is_prime(next_number):
    22     10000        22405      2.2      0.1              prime_numbers.append(next_number)
    23    104741       187927      1.8      0.8          next_number += 1
    24
    25         1            2      2.0      0.0      return prime_numbers
```

Сразу замечаем огромный оверхед: программа выполнялась больше 30 секунд, при том, что без профайлера она отрабатывает быстрее, чем за секунду.

Анализируя результаты можно сделать вывод, что наибольшее время программа тратит в строках 9 и 10, проверяя делители числа для определения его «простоты». И для каждого последующего числа происходят все те же самые проверки. Логичной оптимизацией программы является проверка в качестве делителей только тех чисел, которые ранее были определены как простые:

```
def is_prime(num, prime_numbers):
    """
    Checks if num is prime number.

    >>> is_prime(2, [])
    True
    >>> is_prime(3, [2])
    True
    >>> is_prime(4, [2, 3])
    False
    >>> is_prime(5, [2, 3])
    True
    >>> is_prime(41, [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37])
    True
    >>> is_prime(42, [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41])
    False
    >>> is_prime(43, [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41])
    True
    """
    limit = int(math.sqrt(num)) + 1
    for i in prime_numbers:
        if i > limit:
            break
        if num % i == 0:
            return False
```

```
    return True

def get_prime_numbers(count):
    """
    Get 'count' prime numbers.

    >>> get_prime_numbers(1)
    [2]
    >>> get_prime_numbers(2)
    [2, 3]
    >>> get_prime_numbers(3)
    [2, 3, 5]
    >>> get_prime_numbers(6)
    [2, 3, 5, 7, 11, 13]
    >>> get_prime_numbers(9)
    [2, 3, 5, 7, 11, 13, 17, 19, 23]
    >>> get_prime_numbers(19)
    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67]
    """
    prime_numbers = [2]
    next_number = 3

    while len(prime_numbers) < count:
        if is_prime(next_number, prime_numbers):
            prime_numbers.append(next_number)
        next_number += 1

    return prime_numbers
```

Запускаем тесты, убеждаемся, что всё отрабатывает правильно, замерим время выполнения программы:

```
→ python -m timeit -n 10 -s'import euler_7' 'euler_7.get_prime_numbers(10001)'
10 loops, best of 3: 390 msec per loop
```

Ускорили работу почти в три раза, неплохо. Запустим ещё разок профилирование:

```
→ kernprof.py -v -l euler_7.py 10001
Wrote profile results to euler_7.py.lprof
Timer unit: 1e-06 s

File: euler_7.py
Function: is_prime at line 7
Total time: 4.54317 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
```

```
     8                                           def is_prime(num, prime_numbers):
     9    104741       310160      3.0      6.8      limit = int(math.sqrt(num)) + 1
    10    800694      1296045      1.6     28.5      for i in prime_numbers:
    11    800692      1327770      1.7     29.2          if i > limit:
    12      9998        17109      1.7      0.4              break
    13    790694      1409731      1.8     31.0          if num % i == 0:
    14     94741       165761      1.7      3.6              return False
    15     10000        16599      1.7      0.4      return True

File: euler_7.py
Function: get_prime_numbers at line 18
Total time: 10.5464 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    18                                           @profile
    19                                           def get_prime_numbers(count):
    20         1            4      4.0      0.0      prime_numbers = [2]
    21         1            2      2.0      0.0      next_number = 3
    22
    23    104742       202443      1.9      1.9      while len(prime_numbers) < count:
    24    104741     10143489     96.8     96.2          if is_prime(next_number, prime_numbers):
    25     10000        22374      2.2      0.2              prime_numbers.append(next_number)
    26    104741       178074      1.7      1.7          next_number += 1
    27
    28         1            1      1.0      0.0      return prime_numbers
```

Как видим, программа стала выполняться гораздо быстрее.

memory_profiler

Для профилирования памяти можно использовать memory_profiler. Использовать его так же просто, как line_profiler. Ставим:

```
→ pip install psutil memory_profiler
```

Запускаем:

```
→ python -m memory_profiler euler_7.py 10001
Filename: euler_7.py

Line #    Mem usage    Increment   Line Contents
================================================
    18    8.441 MiB   -0.531 MiB   @profile
    19                             def get_prime_numbers(count):
```

```
    20    8.445 MiB    0.004 MiB       prime_numbers = [2]
    21    8.445 MiB    0.000 MiB       next_number = 3
    22
    23    8.973 MiB    0.527 MiB       while len(prime_numbers) < count:
    24                                     if is_prime(next_number, prime_numbers):
    25    8.973 MiB    0.000 MiB               prime_numbers.append(next_number)
    26    8.973 MiB    0.000 MiB           next_number += 1
    27
    28    8.973 MiB    0.000 MiB       return prime_numbers


Filename: euler_7.py

Line #    Mem usage    Increment   Line Contents
================================================
     7    8.973 MiB    0.000 MiB   @profile
     8                             def is_prime(num, prime_numbers):
     9    8.973 MiB    0.000 MiB       limit = int(math.sqrt(num)) + 1
    10    8.973 MiB    0.000 MiB       for i in prime_numbers:
    11    8.973 MiB    0.000 MiB           if i > limit:
    12    8.973 MiB    0.000 MiB               break
    13    8.973 MiB    0.000 MiB           if num % i == 0:
    14    8.973 MiB    0.000 MiB               return False
    15    8.973 MiB    0.000 MiB       return True
```

Как видим, никаких особых проблем с памятью у нас нет. Всё в пределах нормы.

Ещё инструменты

Перечислю ещё несколько инструментов для профилирования, к сожалению, статья и так получилась огромной, и разобрать все из них не предоставляется возможным.

- Dowser — отличнейший инструмент для профилирования памяти
- guppy — ещё одна программа для профилирования памяти
- Meliae — уже упоминавшийся ранее инструмент для профилирования памяти, которую можно использовать вместе с RunSnakeRun
- muppy — обнаружение утечек памяти
- memprof — снова профилирование памяти
- objgraph — интересный инструмент для исследования объектов

Мы познакомились с инструментами для профилирования кода на Python. Множество из них осталось за кадром, надеюсь, мои

коллеги дополнят меня в комментариях.

В следующей статье мы познакомимся с методами и инструментами для отладки Python-программ. Оставайтесь на связи!

Минутка статистики: в трёх статьях про профилирование питона я использовал слово «профилирование» больше ста раз.