ПРОФИЛИРОВАНИЕ Python

К курсу большие данные для магистрантов направления «Прикладная математика».

[Профилирование Python — почему и где тормозит ваш код / Хабр (habr.com)](https://habr.com/ru/companies/ruvds/articles/757336/)

Оглавление

[1. РАЗРАБОТКА ЧРЕЗВЫЧАЙНО БЫСТРЫХ ПРОГРАММ НА PYTHON 3](#_Toc178175542)

[2. ИЗМЕРЕНИЕ ВРЕМЕНИ И ПРОФИЛИРОВАНИЕ 3](#_Toc178175543)

[3. САМЫЙ ЛЁГКИЙ СПОСОБ «ПРОФИЛИРОВАНИЯ» КОДА 3](#_Toc178175544)

[3. САМЫЙ ТОЧНЫЙ СПОСОБ ПРОФИЛИРОВАНИЯ 4](#_Toc178175545)

[4. ИССЛЕДОВАНИЕ ВРЕМЕННЫХ ПОКАЗАТЕЛЕЙ ВЫПОЛНЕНИЯ КОНКРЕТНОЙ ФУНКЦИИ 4](#_Toc178175546)

[5. УСКОРЕНИЕ КОДА 5](#_Toc178175547)

[6. ИСПОЛЬЗУЙТЕ ВСТРОЕННЫЕ ТИПЫ ДАННЫХ 5](#_Toc178175548)

[7. ПРИМЕНЯЙТЕ КЭШИРОВАНИЕ (МЕМОИЗАЦИЮ) С ПОМОЩЬЮ LRU\_CACHE 5](#_Toc178175549)

[8. ИСПОЛЬЗУЙТЕ ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ 6](#_Toc178175550)

[9. ОБОРАЧИВАЙТЕ КОД В ФУНКЦИИ 7](#_Toc178175551)

[10. НЕ ОБРАЩАЙТЕСЬ К АТРИБУТАМ 7](#_Toc178175552)

[11. ОСТЕРЕГАЙТЕСЬ СТРОК 7](#_Toc178175553)

[12. ЗНАЙТЕ О ТОМ, ЧТО И ГЕНЕРАТОРЫ МОГУТ РАБОТАТЬ БЫСТРО 8](#_Toc178175554)

[13. ПРОФИЛИРОВАНИЕ PYTHON-ПРОГРАММ И АНАЛИЗ ИХ ПРОИЗВОДИТЕЛЬНОСТИ 8](#_Toc178175555)

[14. ИДЕНТИФИКАЦИЯ УЗКИХ МЕСТ 8](#_Toc178175556)

[15. БОЛЕЕ ГЛУБОКОЕ ИССЛЕДОВАНИЕ КОДА 10](#_Toc178175557)

[16. ПОДХОДЫ К РЕШЕНИЮ ПРОБЛЕМ 13](#_Toc178175558)

[ИТОГИ 15](#_Toc178175559)

# 1. РАЗРАБОТКА ЧРЕЗВЫЧАЙНО БЫСТРЫХ ПРОГРАММ НА PYTHON

Ненавистники Python всегда говорят, что одной из причин того, что они не хотят использовать этот язык, является то, что Python — это медленно. Но то, что некая программа, независимо от используемого языка программирования, может считаться быстрой или медленной, очень сильно зависит от разработчика, который её написал, от его знаний и от умения создавать оптимизированный и высокопроизводительный код.

Наша задача в улучшении производительность Python-программ и сделать их по-настоящему быстрыми.

# 2. ИЗМЕРЕНИЕ ВРЕМЕНИ И ПРОФИЛИРОВАНИЕ

Прежде чем приступить к оптимизации какого-либо кода, сначала надо выяснить то, какие его части замедляют всю программу. Иногда узкое место программы может быть очевидным, но, если программист не знает, где оно находится, он может воспользоваться некоторыми возможностями по его выявлению.

Ниже представлен код программы, который используется в демонстрационных целях. Он взят из документации к Python. Этот код возводит *e* в степень *x*:

# slow\_program.py

from decimal import \*

def exp(x):

    getcontext().prec += 2

    i, lasts, s, fact, num = 0, 0, 1, 1, 1

    while s != lasts:

        lasts = s

        i += 1

        fact \*= i

        num \*= x

        s += num / fact

    getcontext().prec -= 2

    return +s

exp(Decimal(150))

exp(Decimal(400))

exp(Decimal(3000))

# 3. САМЫЙ ЛЁГКИЙ СПОСОБ «ПРОФИЛИРОВАНИЯ» КОДА

Для начала рассмотрим самый простой способ профилирования кода. Так сказать, «профилирование для ленивых». Он заключается в использовании команды Unix time:

~ $ time python3.8 slow\_program.py

real 0m11,058s

user 0m11,050s

sys  0m0,008s

Такое профилирование вполне может дать программисту некие полезные сведения — в том случае, если ему нужно замерить время выполнения всей программы. Но обычно этого недостаточно.

# 3. САМЫЙ ТОЧНЫЙ СПОСОБ ПРОФИЛИРОВАНИЯ

На другом конце спектра методов профилирования кода лежит инструмент cProfile, который даёт программисту, надо признать, слишком много сведений:

~ $ python3.8 -m cProfile -s time slow\_program.py

         1297 function calls (1272 primitive calls) in 11.081 seconds

   Ordered by: internal time

   ncalls  tottime percall  cumtime percall filename:lineno(function)

        3   11.079    3.693 11.079    3.693 slow\_program.py:4(exp)

        1    0.000    0.000 0.002    0.002 {built-in method \_imp.create\_dynamic}

      4/1    0.000   0.000 11.081   11.081 {built-in method builtins.exec}

        6    0.000    0.000 0.000    0.000 {built-in method \_\_new\_\_ of type object at 0x9d12c0}

        6    0.000    0.000 0.000    0.000 abc.py:132(\_\_new\_\_)

       23    0.000   0.000 0.000    0.000 \_weakrefset.py:36(\_\_init\_\_)

      245    0.000   0.000 0.000    0.000 {built-in method builtins.getattr}

        2    0.000    0.000 0.000    0.000 {built-in method marshal.loads}

       10    0.000   0.000 0.000    0.000 <frozen importlib.\_bootstrap\_external>:1233(find\_spec)

      8/4    0.000   0.000 0.000    0.000 abc.py:196(\_\_subclasscheck\_\_)

       15    0.000   0.000 0.000    0.000 {built-in method posix.stat}

        6    0.000    0.000 0.000    0.000 {built-in method builtins.\_\_build\_class\_\_}

        1    0.000    0.000 0.000    0.000 \_\_init\_\_.py:357(namedtuple)

       48    0.000   0.000 0.000    0.000 <frozen importlib.\_bootstrap\_external>:57(\_path\_join)

       48    0.000   0.000 0.000    0.000 <frozen importlib.\_bootstrap\_external>:59(<listcomp>)

        1    0.000    0.000 11.081   11.081 slow\_program.py:1(<module>)

Тут мы запускаем исследуемый скрипт с использованием модуля cProfile и применяем аргумент time. В результате строки вывода упорядочены по внутреннему времени (cumtime). Это даёт нам очень много информации. На самом деле то, что показано выше, это лишь около 10% вывода cProfile.

Проанализировав эти данные, мы можем увидеть, что причиной медленной работы программы является функция exp (вот уж неожиданность!). После этого мы можем заняться профилированием кода, используя более точные инструменты.

# 4. ИССЛЕДОВАНИЕ ВРЕМЕННЫХ ПОКАЗАТЕЛЕЙ ВЫПОЛНЕНИЯ КОНКРЕТНОЙ ФУНКЦИИ

Теперь мы знаем о том месте программы, куда нужно направить наше внимание. Поэтому мы можем решить заняться исследованием медленной функции, не профилируя другой код программы. Для этого можно воспользоваться простым декоратором:

def timeit\_wrapper(func):

    @wraps(func)

    def wrapper(\*args, \*\*kwargs):

        start = time.perf\_counter()  # В качестве альтернативы тут можно использовать time.process\_time()

        func\_return\_val = func(\*args, \*\*kwargs)

        end = time.perf\_counter()

        print('{0:<10}.{1:<8} : {2:<8}'.format(func.\_\_module\_\_, func.\_\_name\_\_, end - start))

        return func\_return\_val

    return wrapper

Этот декоратор можно применить к функции, которую нужно исследовать:

@timeit\_wrapper

def exp(x):

    ...

print('{0:<10} {1:<8} {2:^8}'.format('module', 'function', 'time'))

exp(Decimal(150))

exp(Decimal(400))

exp(Decimal(3000))

Теперь после запуска программы мы получим следующие сведения:

~ $ python3.8 slow\_program.py

module     function time

\_\_main\_\_  .exp : 0.003267502994276583

\_\_main\_\_  .exp : 0.038535295985639095

\_\_main\_\_  .exp : 11.728486061969306

Тут стоит обратить внимание на то, какое именно время мы планируем измерять. Соответствующий пакет предоставляет нам такие показатели, как time.perf\_counter и time.process\_time. Разница между ними заключается в том, что perf\_counter возвращает абсолютное значение, в которое входит и то время, в течение которого процесс Python-программы не выполняется. Это значит, что на этот показатель может повлиять нагрузка на компьютер, создаваемая другими программами. Показатель process\_time возвращает только пользовательское время (user time). В него не входит системное время (system time). Это даёт нам только сведения о времени выполнения нашего процесса.

# 5. УСКОРЕНИЕ КОДА

А теперь переходим к самому интересному. Поработаем над ускорением программы. Я (по большей части) не собираюсь показывать тут всякие хаки, трюки и таинственные фрагменты кода, которые волшебным образом решают проблемы производительности. Я, в основном, хочу поговорить об общих идеях и стратегиях, которые, если ими пользоваться, могут очень сильно повлиять на производительность. В некоторых случаях речь идёт о 30% повышении скорости выполнения кода.

# 6. ИСПОЛЬЗУЙТЕ ВСТРОЕННЫЕ ТИПЫ ДАННЫХ

Использование встроенных типов данных — это совершенно очевидный подход к ускорению кода. Встроенные типы данных чрезвычайно быстры, в особенности — если сравнить их с пользовательскими типами, вроде деревьев или связных списков. Дело тут, в основном, в том, что встроенные механизмы языка реализованы средствами C. Если описывать нечто средствами Python — нельзя добиться того же уровня производительности.

# 7. ПРИМЕНЯЙТЕ КЭШИРОВАНИЕ (МЕМОИЗАЦИЮ) С ПОМОЩЬЮ LRU\_CACHE

Кэширование — популярный подход к повышению производительности кода. О нём [писано](https://martinheinz.dev/blog/4), но полагаю, что о нём стоит рассказать здесь:

import functools

import time

# кэширование до 12 различных результатов

@functools.lru\_cache(maxsize=12)

def slow\_func(x):

    time.sleep(2)  # Имитируем длительные вычисления

    return x

slow\_func(1)  # ... ждём 2 секунды до возврата результата

slow\_func(1)  # результат уже кэширован - он возвращается немедленно!

slow\_func(3)  # ... опять ждём 2 секунды до возврата результата

Вышеприведённая функция имитирует сложные вычисления, используя time.sleep. Когда её в первый раз вызывают с параметром 1 — она ждёт 2 секунды и возвращает результат только после этого. Когда же её снова вызывают с тем же параметром, оказывается, что результат её работы уже кэширован. Тело функции в такой ситуации не выполняется, а результат возвращается немедленно. [Здесь](https://martinheinz.dev/blog/4) можно найти примеры применения кэширования, более близкие к реальности.

# 8. ИСПОЛЬЗУЙТЕ ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

Применяя локальные переменные, мы учитываем скорость поиска переменной в каждой области видимости. Я говорю именно о «каждой области видимости», так как тут я имею в виду не только сопоставление скорости работы с локальными и глобальными переменными. На самом деле, разница в работе с переменными наблюдается даже, скажем, между локальными переменными в функции (самая высокая скорость), атрибутами уровня класса (например — self.name, это уже медленнее), и глобальными импортированными сущностями наподобие time.time (самый медленный из этих трёх механизмов).

Улучшить производительность можно, используя следующие подходы к присваиванию значений, которые несведущему человеку могут показаться совершенно ненужными и бесполезными:

#  Пример #1

class FastClass:

    def do\_stuff(self):

        temp = self.value  # это ускорит цикл

        for i in range(10000):

            ...  # Выполняем тут некие операции с `temp`

#  Пример #2

import random

def fast\_function():

    r = random.random

    for i in range(10000):

        print(r())  # здесь вызов `r()` быстрее, чем был бы вызов random.random()

# 9. ОБОРАЧИВАЙТЕ КОД В ФУНКЦИИ

Этот совет может показаться противоречащим здравому смыслу, так как при вызове функции в стек попадают некие данные и система испытывает дополнительную нагрузку, обрабатывая операцию возврата из функции. Однако эта рекомендация связана с предыдущей. Если вы просто поместите весь свой код в один файл, не оформив в виде функции, он будет выполняться гораздо медленнее из-за использования глобальных переменных. Это значит, что код можно ускорить, просто обернув его в функцию main() и один раз её вызвав:

def main():

    ...  # Весь код, который раньше был глобальным

main()

# 10. НЕ ОБРАЩАЙТЕСЬ К АТРИБУТАМ

Ещё один механизм, способный замедлить программу — это оператор точка (.), который используется для доступа к атрибутам объектов. Этот оператор вызывает выполнение процедуры поиска по словарю с использованием \_\_getattribute\_\_, что создаёт дополнительную нагрузку на систему. Как ограничить влияние этой особенности Python на производительность?

#  Медленно:

import re

def slow\_func():

    for i in range(10000):

        re.findall(regex, line)  # Медленно!

#  Быстро:

from re import findall

def fast\_func():

    for i in range(10000):

        findall(regex, line)  # Быстрее!

# 11. ОСТЕРЕГАЙТЕСЬ СТРОК

Операции на строках могут сильно замедлить программу в том случае, если выполняются в циклах. В частности, речь идёт о форматировании строк с использованием %s и .format(). Можно ли их чем-то заменить? Если взглянуть на недавний [твит](https://twitter.com/raymondh/status/1205969258800275456) Раймонда Хеттингера, то можно понять, что единственный механизм, который надо использовать в подобных ситуациях — это f-строки. Это — самый читабельный, лаконичный и самый быстрый метод форматирования строк. Вот, в соответствии с тем твитом, список методов, которые можно использовать для работы со строками — от самого быстрого к самому медленному:

f'{s} {t}'  # Быстро!

s + '  ' + t

' '.join((s, t))

'%s %s' % (s, t)

'{} {}'.format(s, t)

Template('$s $t').substitute(s=s, t=t)  # Медленно!

# 12. ЗНАЙТЕ О ТОМ, ЧТО И ГЕНЕРАТОРЫ МОГУТ РАБОТАТЬ БЫСТРО

Генераторы — это не те механизмы, которые, по своей природе, являются быстрыми. Дело в том, что они были созданы для выполнения «ленивых» вычислений, что экономит не время, а память. Однако экономия памяти может привести к тому, что программы будут выполняться быстрее. Как это возможно? Дело в том, что при обработке большого набора данных без использования генераторов (итераторов) данные могут привести к переполнению L1-кэша процессора, что значительно замедлит операции по поиску значений в памяти.

Если речь идёт о производительности, очень важно стремиться к тому, чтобы процессор мог бы быстро обращаться к обрабатываемым им данным, чтобы они находились бы как можно ближе к нему. А это значит, что такие данные должны помещаться в процессорном кэше.

Первое правило оптимизации заключается в том, что оптимизацией заниматься не нужно. Но если без оптимизации никак не обойтись, тогда я надеюсь, что советы, которыми я поделился, вам в этом помогут.

# 13. ПРОФИЛИРОВАНИЕ PYTHON-ПРОГРАММ И АНАЛИЗ ИХ ПРОИЗВОДИТЕЛЬНОСТИ

Профилирование — это неотъемлемая часть любых работ по оптимизации кода или производительности программ. Любой опыт, любые знания в сфере оптимизации производительности, которые уже у вас есть, не принесут особой пользы в том случае, если вы не знаете о том, где их применить. В результате оказывается, что поиск узких мест приложений может помочь в деле решения проблем производительности, поможет сделать это быстро и приложив не слишком много усилий.

В этом материале мы обсудим инструменты и методы работы, которые способны обнаруживать и конкретизировать проблемы с производительностью кода, связанные и с ресурсами процессора, и с потреблением памяти. Здесь же мы поговорим о том, как реализовывать (почти безо всяких усилий) простые механизмы, позволяющие бороться с проблемами производительности. Эти механизмы используются в тех случаях, когда даже точно просчитанные изменения кода больше не позволяют улучшить ситуацию.

# 14. ИДЕНТИФИКАЦИЯ УЗКИХ МЕСТ

В деле оптимизации производительности программ лениться — это хорошо. Вместо того чтобы пытаться понять то, какая именно часть кодовой базы замедляет приложение, можно просто воспользоваться инструментами профилирования кода. Они позволят найти те места приложения, на которые стоит обратить внимание, такие, которые нуждаются в более глубоком исследовании.

Самый распространённый инструмент, который используют для этих целей Python-разработчики — это cProfile. Это — стандартный модуль, который способен измерять время выполнения функций.

Рассмотрим следующую функцию, которая возводит (медленно) e в степень X:

# some-code.py¶

from decimal import \*¶

def exp(x):

    getcontext().prec += 2

    i, lasts, s, fact, num = 0, 0, 1, 1, 1

    while s != lasts:

        lasts = s

        i += 1

        fact \*= i

        num \*= x

        s += num / fact

    getcontext().prec -= 2

    return +s

exp(Decimal(3000))

Исследуем этот медленный код с помощью cProfile:

import cProfile

!python -m cProfile -s cumulative some-code.py

         1052 function calls (1023 primitive calls) in 2.765 seconds

   Ordered by: cumulative timek

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)

      5/1    0.000    0.000    2.765    2.765 {built-in method builtins.exec}

        1    0.000    0.000    2.765    2.765 some-code.py:1(<module>)

        1    2.764    2.764    2.764    2.764 some-code.py:3(exp)

      4/1    0.000    0.000    0.001    0.001 <frozen importlib.\_bootstrap>:986(\_find\_and\_load)

      4/1    0.000    0.000    0.001    0.001 <frozen importlib.\_bootstrap>:956(\_find\_and\_load\_unlocked)

      4/1    0.000    0.000    0.001    0.001 <frozen importlib.\_bootstrap>:650(\_load\_unlocked)

      3/1    0.000    0.000    0.001    0.001 <frozen importlib.\_bootstrap\_external>:842(exec\_module)

      5/1    0.000    0.000    0.001    0.001 <frozen importlib.\_bootstrap>:211(\_call\_with\_frames\_removed)

        1    0.000    0.000    0.001    0.001 decimal.py:2(<module>)

...

Тут мы воспользовались опцией -s cumulative для сортировки выходных данных по суммарному времени, затраченному на выполнение каждой из функций. Это упрощает поиск проблемных участков кода. Видно, что почти всё время (примерно 2,764 секунды) в ходе одного сеанса выполнения программы было потрачено в функции exp.

Профилирование подобного рода может принести пользу, но его, к сожалению, не всегда достаточно. cProfile снабжает нас информацией лишь о вызовах функций, но не об отдельных строках кода. Если вызвать какую-то особую функцию, вроде append, в разных местах, то сведения обо всех её вызовах будут собраны в одной строке отчёта cProfile. То же самое относится и к скриптам, вроде того, который мы исследовали выше. Он содержит единственную функцию, которая вызывается лишь один раз, в результате у cProfile оказывается не особенно много данных для формирования отчёта.

Иногда такая роскошь, как локальная отладка проблемного кода, программисту не доступна. Или бывает так, что нужно проанализировать проблему с производительностью, что называется, «на лету», когда она возникает в продакшн-окружении. В таких ситуациях можно воспользоваться пакетом py-spy. Это — профилировщик, способный исследовать программы, которые уже запущены. Например — приложения, работающие в продакшне, или на любой удалённой системе:

pip install py-spy

python some-code.py &

[1] 1129587

ps -A -o pid,cmd | grep python

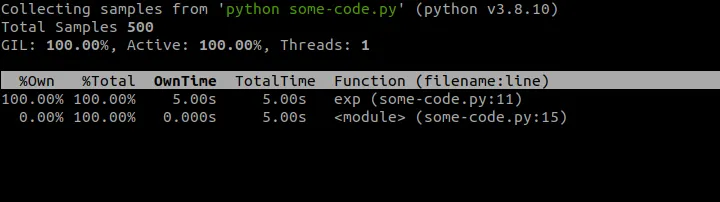
...

1129587 python some-code.py

1130365 grep python

sudo env "PATH=$PATH" py-spy top --pid 1129587

В этом фрагменте кода мы сначала устанавливаем py-spy, а потом, в фоне, запускаем программу, которая выполняется длительное время. Это приводит к автоматическому показу идентификатора процесса (PID), но если мы его не знаем, можно, для его выяснения, воспользоваться командой ps. И, наконец, мы запускаем py-spy в режиме top, передавая ему PID. Это ведёт к выводу данных, очень похожих на те, что выводит Linux-утилита top.



Данные, выводимые py-spy в режиме top

Тут, правда, в нашем распоряжении оказывается не так много данных, так как скрипт представляет собой всего лишь одну функцию, выполняющуюся длительное время. Но в реальных случаях, вероятнее всего, подобный отчёт будет содержать сведения о многих функциях, совместно использующих процессорное время. А это может помочь несколько прояснить ситуацию с существующими проблемами производительности программы.

# 15. БОЛЕЕ ГЛУБОКОЕ ИССЛЕДОВАНИЕ КОДА

Профилировщики, о которых мы только что говорили, должны помочь вам в деле обнаружения функций, которые вызывают проблемы, связанные с производительностью. Но если это не приведёт к обнаружению конкретных строк кода, которые надо доработать, это значит, что мы можем обратиться к профилировщикам, которые позволяют исследовать программы на более глубоком уровне.

Первый из таких инструментов представлен пакетом line\_profiler. Он, как можно судить по его названию, может использоваться для выяснения того, сколько времени уходит на выполнение каждой конкретной строки кода:

# https://github.com/pyutils/line\_profiler

pip install line\_profiler

kernprof -l -v some-code.py  # Это может занять некоторое время...

Wrote profile results to some-code.py.lprof

Timer unit: 1e-06 s

Total time: 13.0418 s

File: some-code.py

Function: exp at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents

     3                                           @profile

     4                                           def exp(x):

     5         1          4.0      4.0      0.0      getcontext().prec += 2

     6         1          0.0      0.0      0.0      i, lasts, s, fact, num = 0, 0, 1, 1, 1

     7      5818       4017.0      0.7      0.0      while s != lasts:

     8      5817       1569.0      0.3      0.0          lasts = s

     9      5817       1837.0      0.3      0.0          i += 1

    10      5817       6902.0      1.2      0.1          fact \*= i

    11      5817       2604.0      0.4      0.0          num \*= x

    12      5817   13024902.0   2239.1     99.9          s += num / fact

    13         1          5.0      5.0      0.0      getcontext().prec -= 2

    14         1          2.0      2.0      0.0      return +s

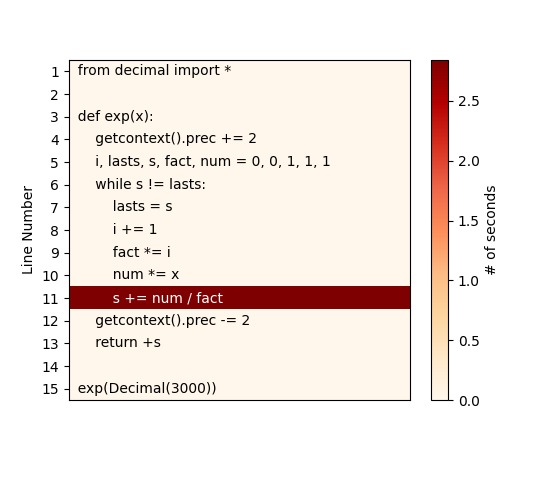
Библиотека line\_profiler распространяется вместе с интерфейсом командной строки kernprof (названным так в честь Роберта Керна), который используется для организации эффективного анализа результатов тестовых прогонов программ. Передача нашего кода этой утилите приводит к созданию .lprof-файла со сведениями об анализе кода. В нашем распоряжении, кроме того, оказывается отчёт, выводимый на экран (при использовании опции -v), подобный показанному выше. Тут чётко видны места функции, на выполнение которых уходит больше всего времени. Это очень сильно помогает в деле поиска и исправления проблем с производительностью. В выходных данных можно заметить декоратор @profile, добавленный к функции exp. Это — необходимое дополнение, которое позволяет line\_profiler узнать о том, какую именно функцию в файле мы хотим изучить.

Но даже если построчно проанализировать функцию, первоисточник проблем с производительностью можно и не обнаружить. Например, такое бывает в том случае, если в конструкциях while или if используются условия, составленные из множества выражений. В подобных случаях имеет смысл переписать проблемные фрагменты, разбить одну строку кода на несколько. Это позволит получить более полные и понятные результаты анализа.

Если же вы — по-настоящему ленивый разработчик (как я), и чтение текстового вывода в интерфейсе командной строки — это для вас уже слишком — тогда вот вам ещё один инструмент — pyheat. Это — профилировщик, основанный на pprofile, ещё одном построчном профилировщике, создатели которого черпали вдохновение из кода line\_profiler. Этот профилировщик генерирует тепловую карту для строк/областей кода, выполнение которых занимает основную долю времени выполнения программы:

pip install py-heat

pyheat some-code.py --out image\_file.png



Тепловая карта, построенная с помощью pyheat

Учитывая простоту кода нашего примера, отчёт, выводимый на экран с помощью kernprof, который мы видели, выглядит достаточно понятным. Но вышеприведённая тепловая карта ещё лучше идентифицирует узкое место нашей функции.

До сих пор мы говорили лишь о профилировании, имеющем отношение к ресурсам процессора. Но то, как программа пользуется CPU, не всегда является тем, что волнует разработчика. Оперативная память — дешёвый ресурс, поэтому программисты обычно не задумываются о её использовании. По крайней мере — до тех пор, пока программа не исчерпает доступную память.

Даже если ваша программа не попала в ситуацию, когда ей не хватает памяти, всё равно, то, как приложение пользуется памятью, стоит исследовать. Сделать это можно для того чтобы узнать, можно ли оптимизировать код с прицелом на экономию памяти, или можно ли дать программе больше памяти ради повышения её производительности. Для анализа использования памяти можно воспользоваться инструментом memory\_profiler. Он похож на уже известный вам line\_profiler:

# https://github.com/pythonprofilers/memory\_profiler

pip install memory\_profiler psutil

psutil is needed for better memory\_profiler performance

python -m memory\_profiler some-code.py

Filename: some-code.py

Line #    Mem usage    Increment  Occurrences   Line Contents

    15   39.113 MiB   39.113 MiB            1   @profile

    16                                          def memory\_intensive():

    17   46.539 MiB    7.426 MiB            1       small\_list = [None] \* 1000000

    18  122.852 MiB   76.312 MiB            1       big\_list = [None] \* 10000000

    19   46.766 MiB  -76.086 MiB            1       del big\_list

    20   46.766 MiB    0.000 MiB            1       return small\_list

Это испытание мы проводим на другом фрагменте кода.

Функция memory\_intensive создаёт и уничтожает большие Python-списки. На её примере мы способны оценить ту пользу, которую может принести нам memory\_profiler в деле анализа использования памяти. Так же, как и при kernprof-профилировании, функцию надо оснастить декоратором @profile. Он позволит memory\_profiler узнать о том, какой именно код мы хотим профилировать.

Тут видно, что для обычного списка, содержащего значения None, было выделено более 100 МиБ памяти. Анализируя эти данные, правда, надо учитывать то, что они отражают не реальное использование памяти, а то, сколько памяти было выделено при выполнении каждой из строк функции. В данном случае это значит, что переменные, хранящие списки, на самом деле, не занимают столько памяти. Здесь отражён лишь тот факт, что Python-объекты list, вероятнее всего, выделяют память с запасом, чтобы подстроиться под ожидаемый рост объёма данных, которые могут попасть в список.

Как мы уже видели, Python-списки часто потребляют сотни мегабайт или даже гигабайты памяти. Быстро улучшить ситуацию можно, прибегнув к оптимизации, которая заключается в переходе на обычные объекты array. Они эффективнее хранят данные примитивных типов, вроде int или float. Кроме того, можно ограничить использование памяти, выбирая типы с меньшей точностью, применяя параметр typecode. Воспользуйтесь командой help(array) чтобы посмотреть таблицу, в которой перечислены доступные варианты типов и их требования к памяти.

Если же даже подобные инструменты, дающие точную и детализированную информацию, не позволяют найти узкие места кода, можно попытаться дизассемблировать код и выйти на реальный байт-код, используемый интерпретатором Python. А если и дизассемблирование не помогает решить имеющуюся проблему — тогда полезным может оказаться выяснение и понимание того, какие операции выполняются в недрах Python при вызове некоей функции. Вооружившись знаниями, полученными в ходе таких исследований, в будущем вы сможете писать более производительный код.

Дизассемблированный вариант кода можно сгенерировать, воспользовавшись встроенным модулем dis, передав функцию методу dis.dis(...). Он сгенерирует и выведет список инструкций байт-кода, выполняемого при вызове функции.

from math import e

def exp(x):

    return e\*\*x  # math.exp(x)

import dis

dis.dis(exp)

В этом материале мы всё время исследовали очень медленную реализацию возведения e в степень X. Выше же представлена простейшая функция, которая решает эту задачу с высокой скоростью. Теперь мы можем сравнить результаты дизассемблирования быстрой и медленной функций. Результаты их дизассемблирования окажутся совершенно различными. Их изучение делает ещё более очевидным тот факт, что одна функция гораздо медленнее другой.

Вот как выглядит быстрая функция:

2 0 LOAD\_GLOBAL 0 (e)

2 LOAD\_FAST 0 (x)

4 BINARY\_POWER

6 RETURN\_VALUE

А вот — наша старая функция, которая работает медленно:

4 0 LOAD\_GLOBAL 0 (getcontext)

2 CALL\_FUNCTION 0

4 DUP\_TOP

6 LOAD\_ATTR 1 (prec)

8 LOAD\_CONST 1 (2)

10 INPLACE\_ADD

12 ROT\_TWO

14 STORE\_ATTR 1 (prec)

5 16 LOAD\_CONST 2 ((0, 0, 1, 1, 1))

18 UNPACK\_SEQUENCE 5

20 STORE\_FAST 1 (i)

22 STORE\_FAST 2 (lasts)

24 STORE\_FAST 3 (s)

26 STORE\_FAST 4 (fact)

28 STORE\_FAST 5 (num)

6 >> 30 LOAD\_FAST 3 (s)

32 LOAD\_FAST 2 (lasts)

34 COMPARE\_OP 3 (!=)

36 POP\_JUMP\_IF\_FALSE 80

...

100 RETURN\_VALUE

Для того чтобы лучше разобраться в том, что именно тут происходит — рекомендую взглянуть на [этот](https://stackoverflow.com/a/47529318) ответ со StackOverflow, в котором раскрывается смысл столбцов, по которым распределены эти данные.

# 16. ПОДХОДЫ К РЕШЕНИЮ ПРОБЛЕМ

Тот, кто занимается оптимизацией программы, рано или поздно доведёт её до такого состояния, когда изменения в коде или в алгоритмах начнут давать совсем небольшие улучшения. В этот момент хорошо будет обратить внимание на внешние инструменты, способные дать дополнительный прирост производительности.

Верный способ улучшить скорость работы кода заключается в компиляции его в виде C-программы. Это можно сделать, воспользовавшись различными инструментами. Например — PyPy или Cython. Первый из них — это JIT-компилятор, который можно использовать как непосредственную замену CPython. Он может дать, не требуя никаких усилий от программиста, значительный рост производительности кода. Его применение вполне может стать достойным решением некоей проблемы с производительностью. Для того чтобы воспользоваться PyPy — достаточно загрузить соответствующий архив, распаковать его и запустить с помощью PyPy свой код:

# Загрузить архив можно с https://www.pypy.org/download.html

tar -xjf pypy3.8-v7.3.7-linux64.tar.bz2

cd pypy3.8-v7.3.7-linux64/bin

./pypy some-code.py

Просто чтобы доказать то, что благодаря PyPy можно, не прилагая особых усилий, сразу же улучшить производительность программы, устроим небольшое испытание скрипта, запущенного с помощью CPython и PyPy:

time python some-code.py

real 0m2,861s

user 0m2,841s

sys     0m0,016s

time pypy some-code.py

real 0m1,450s

user 0m1,422s

sys     0m0,009s

PyPy, помимо вышеозначенных плюсов, отличается ещё и тем, что для его использования не нужно вносить в код никаких изменений. Он, кроме того, поддерживает все встроенные модули и функции Python.

Всё это звучит просто замечательно, но использование PyPy означает необходимость идти на кое-какие компромиссы. Этот инструмент поддерживает проекты, нуждающиеся в C-привязках, такие, как numpy, но это создаёт значительную дополнительную нагрузку на систему, что сильно замедляет соответствующие библиотеки, сводя на нет другие улучшения производительности. PyPy, кроме того, не решает проблем с производительностью в ситуациях, когда применяются внешние библиотеки, или в случаях, когда речь идёт о работе с базами данных. И, аналогично, если речь идёт о программах, производительность которых привязана к подсистеме ввода/вывода, не стоит ожидать значительной выгоды от применения PyPy.

Если PyPy вам не помогает — можете попробовать Cythoh. Это — компилятор, который использует C-подобные аннотации типов (не подсказки по типам, применяемые в Python) для создания компилируемых модулей расширения Python. Cython, кроме прочего, использует AOT-компиляцию, что может дать значительный прирост производительности благодаря уходу от холодного запуска приложений. Но использование Cython требует переработки существующего кода с использованием особого синтаксиса, что приводит к усложнению программ.

Если вы не против перейти на Python-синтаксис, немного отличающийся от обычного, тогда вам, возможно, интересно будет взглянуть на [prometeo](https://github.com/zanellia/prometeo" \t "_blank) — встраиваемый язык, отражающий специфику конкретной предметной области, основанный на Python. Он, в частности, ориентирован на научные вычисления. Программы, написанные на prometeo, транспилируются в чистый C-код. Их производительность сравнима со скоростью работы программ, изначально написанных на C.

Если же ни одно из представленных тут решений не позволит вам выйти на нужный уровень производительности, тогда вам, возможно, стоит писать свой оптимизированный код на C или Fortran, а для вызова этого кода из Python использовать EFI. Среди библиотек, которые способны вам в этом помочь, можно отметить ctypes и cffi для языка C, и f2py для Fortran.

# ИТОГИ

Первое правило оптимизации заключается в том, чтобы ничего не оптимизировать. Если же вам действительно это нужно — оптимизируйте то, что имеет смысл оптимизировать. Используйте инструменты для профилирования кода, о которых мы говорили — это позволит вам избежать пустой траты времени на улучшение малозначимых фрагментов программ. Ещё, занимаясь оптимизацией, полезно создавать воспроизводимые тесты производительности для улучшаемого фрагмента кода. Это позволит оценить реальное воздействие оптимизаций на производительность