Лекция 12

Ускорение Python ¶

- базовые принцип оптимизации
- инструменты для профилирования
- оптимизация средствами python
- AOT и JIT компиляция
- профилирование памяти

3 правила прежде чем приступить к оптимизации кода

1. НЕ ОПТИМИЗИРОВАТЬ, если нет конкретных критериев производительности.
2. НЕ ОПТИМИЗИРОВАТЬ ВСЕ РАВНО, время разработчика стоит дорого, дешевле обновить железо.
3. ПРОФИЛИРОВАТЬ, прежде чем приступать к какой-либо оптимизации.

Профилирование

Сбор характеристик работы программы, например, время выполнения отдельных фрагментов.

Общие советы:

- Использовать реальные данные и окружение, схожее с продакшеном
- Отключать антивирус
- Подготовить хороший набор тестов
- Понимать, что нужно профилировать

В качестве примера для изучения производительности Python будем использовать умножение матриц.

In [3]:
```
import random

class Matrix(list):
    @classmethod
    def zeros(cls, shape):
        n_rows, n_cols = shape
        return cls([[0] * n_cols for i in range(n_rows)])

    @classmethod
    def random(cls, shape):
        M, (n_rows, n_cols) = cls(), shape
        for i in range(n_rows):
            M.append([random.randint(-255, 255) for j in range(n_cols)])
        return M

    def transpose(self):
        n_rows, n_cols = self.shape
        return self.__class__(zip(*self))

    @property
    def shape(self):
        return ((0, 0) if not self else (len(self), len(self[0])))
```

In [4]:
```
def matrix_product(X, Y):
    xrows, xcols = X.shape
    yrows, ycols = Y.shape
    # верим, что с размерностями всё хорошо
    Z = Matrix.zeros((xrows, ycols))
    for i in range(xrows):
        for k in range(xcols):
            for j in range(ycols):
                Z[i][j] += X[i][k] * Y[k][j]
    return Z
```

Прежде чем что-то изменять, добавим тесты. Это поможет контролировать, что в процессе оптимизации не сломалась логика.

In [5]:
```
def test_matrix_product(product_func):
    X = Matrix([[1], [2], [3]])
    Y = Matrix([[4, 5, 6]])
    assert product_func(X, Y) == \
        [
            [4, 5,  6],
            [8, 10, 12],
            [12, 15, 18]
    ]
    print('ok')

test_matrix_product(matrix_product)
```

```
ok
```

Измерение времени

Модуль time

time.time() - зависит от системных часов, годится для получения текущего времени
_time.perf*counter() / time.monotonic()* - неубывающий таймер, включает время простоя текущего процесса
_time.process*time()* - включает только время работы процесса

In [6]:
```
from time import perf_counter

t_start = perf_counter()
[i for i in range(1_000_000)]
t_stop = perf_counter()
t_stop - t_start
```

Out[6]: 0.1112508579999485

Модуль timeit

Итеративно выполняет небольшие куски кода для большей точности.
Замеряет время с помощью _time.perf*counter()*.
На время измерений отключается сборщик мусора.

Программный интерфейс

In [7]:
```
def f():
    [i for i in range(100)]

if __name__ == '__main__':
    from timeit import timeit
    print(timeit('f()', setup='from __main__ import f', number=100_000))
```

```
0.3798427599999741
```

CLI

In [8]:
```
!python -m timeit "[i for i in range(100)]"
```

```
50000 loops, best of 5: 4.18 usec per loop
```

IPython magic

In [9]:
```
%timeit [i for i in range(100)]
```

```
4.24 µs ± 188 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [10]:
```
shape = 64, 64
X = Matrix.random(shape)
Y = Matrix.random(shape)
%timeit matrix_product(X, Y)
```

```
125 ms ± 2.02 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Подозрительно медленно... В чем может быть проблема?

Профилирование CPU

Опредлим вспомогательную функцию `benchmark`, которая генерирует случайные матрицы указанного размера, а затем n_iter раз умножает их в цикле.

In [11]:
```
def benchmark(shape=(64, 64), n_iter=16):
    X = Matrix.random(shape)
    Y = Matrix.random(shape)
    for iter in range(n_iter):
        matrix_product(X, Y)
```

Модуль cProfile

Позволяет профилировать код на Python с точностью до вызова функции или метода

In [12]:
```
import cProfile
cProfile.run('benchmark()', sort="tottime")
```

```
         41385 function calls in 2.205 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       16    2.175    0.136    2.175    0.136 <ipython-input-4-6cfe2080ded5>:1
(matrix_product)
     8192    0.010    0.000    0.020    0.000 random.py:174(randrange)
     8192    0.007    0.000    0.010    0.000 random.py:224(_randbelow)
      128    0.004    0.000    0.028    0.000 <ipython-input-3-324c60251be6>:1
3(<listcomp>)
     8192    0.004    0.000    0.024    0.000 random.py:218(randint)
     8211    0.002    0.000    0.002    0.000 {method 'getrandbits' of '_rando
m.Random' objects}
        1    0.002    0.002    2.205    2.205 <ipython-input-11-26c696dbb8ff>:
1(benchmark)
     8192    0.001    0.000    0.001    0.000 {method 'bit_length' of 'int' ob
jects}
        2    0.000    0.000    0.028    0.014 <ipython-input-3-324c60251be6>:9
(random)
       16    0.000    0.000    0.000    0.000 <ipython-input-3-324c60251be6>:7
(<listcomp>)
        1    0.000    0.000    2.205    2.205 <string>:1(<module>)
       16    0.000    0.000    0.000    0.000 <ipython-input-3-324c60251be6>:4
(zeros)
       32    0.000    0.000    0.000    0.000 <ipython-input-3-324c60251be6>:2
0(shape)
      128    0.000    0.000    0.000    0.000 {method 'append' of 'list' objec
ts}
        1    0.000    0.000    2.205    2.205 {built-in method builtins.exec}
       64    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Pr
ofiler' objects}
```

Результат предсказуемый и довольно бесполезный: >90% времени работы происходит в функции `matrix_product` .

Модуль line_profiler

Анализирует время работы с точностью до строки в исходном коде.

In [ ]:
```
!pip install line_profiler
```

В IPython/Jupiter доступна магическая команда lprun.
Чтобы воспользоваться ей, сначала нужно загрузить файл расширения

In [13]:
```
%load_ext line_profiler
```

In [14]:
```
%lprun -f matrix_product matrix_product(X, Y)
```

Заметим, что операция **list.__getitem__** не бесплатна. Переставим местами циклы **for** так, чтобы код делал меньше обращений по индексу.

In [15]:
```
def matrix_product_v1(X, Y):
    xrows, xcols = X.shape
    yrows, ycols = Y.shape
    Z = Matrix.zeros((xrows, ycols))
    for i in range(xrows):
        Xi = X[i]
        for j in range(ycols):
            acc = 0
            for k in range(xcols):
                acc += Xi[k] * Y[k][j]
            Z[i][j] = acc
    return Z

test_matrix_product(matrix_product_v1)
```

```
ok
```

In [16]:
```
X, Y = Matrix.random(shape), Matrix.random(shape)
%timeit matrix_product_v1(X, Y)
```

```
81.7 ms ± 14.7 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Почти в 2 раза быстрее, но всё равно слишком медленно: >30% времени уходит исключительно на итерацию!
В данном случае цикл for можно заменить на выражение-генератор.

In [18]:
```
def matrix_product_v2(X, Y):
    xrows, xcols = X.shape
    yrows, ycols = Y.shape
    Z = Matrix.zeros((xrows, ycols))
    for i in range(xrows):
        Xi = X[i]
        Zi = Z[i]
        for j in range(ycols):
            Zi[j] = sum(Xi[k] * Y[k][j] for k in range(xcols))
    return Z

test_matrix_product(matrix_product_v2)
```

```
ok
```

In [19]:
```
X, Y = Matrix.random(shape), Matrix.random(shape)
%timeit matrix_product_v2(X, Y)
```

```
80.3 ms ± 4.37 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Попробуем снова убрать лишние обращения по индексу из самого внутреннего цикла.

In [20]:
```
def matrix_product_v3(X, Y):
    xrows, xcols = X.shape
    yrows, ycols = Y.shape
    Z = Matrix.zeros((xrows, ycols))
    Yt = Y.transpose()
    for i in range(xrows):
        Xi = X[i]
        Zi = Z[i]
        for j in range(ycols):
            Ytj = Yt[j]
            Zi[j] = sum(Xi[k] * Ytj[k] for k in range(xcols))
    return Z

test_matrix_product(matrix_product_v3)
```

```
ok
```

In [21]:
```
X, Y = Matrix.random(shape), Matrix.random(shape)
%timeit matrix_product_v3(X, Y)
```

```
57.9 ms ± 3.31 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Сравним нашу функцию по производительности с numpy импелементацией

In [22]:
```
shape = 64, 64
X, Y = Matrix.random(shape), Matrix.random(shape)
%timeit matrix_product_v3(X, Y)
```

```
66.1 ms ± 20 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [23]:
```
import numpy as np
X = np.random.randint(-255, 255, shape)
Y = np.random.randint(-255, 255, shape)
%timeit X.dot(Y)
```

```
210 µs ± 4.53 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Как можно оптимизировать код средствами python

1. Различные техники кеширование:
    - в локальные переменные
    - предварительные вычисления
    - functools.lru_cache
2. inline-функции
3. built-in функции: sum, min, etc.
4. Использовать правильные структуры данных и алгоритмы (из стандартной библиотеки и внешние)

AOT и JIT компиляция

Дальнейшие способы ускорения кода на Python предполагают его преобразование в машинный код либо до, либо в момент его исполнения.

Ahead-of-time компиляция.

- Python C-API: пишем код на C и реализуем к нему интерфейс, понятный интерпретатору CPython.
- Пишем код на надмножестве Python и преобразуем его в код на C (Cython), использующий C-API интепретатора CPython.

Just-in-time компиляция: пишем код на Python и пытаемся сделать его быстрее в момент исполнения.

- PyPy: следим за исполнением программы и компилируем в машинный код наиболее частые пути в ней.
- Транслируем специальным образом помеченный код на Python в LLVM (Numba), а затем компилирует в машинный код.

Numba

Для использования Numba достаточно декорировать функцию с помощью 'numba.jit' (в теории).
В момент первого вызова функция будет транслирована в LLVM и скомпилирована в машинный код.
Numba не может эффективно оптимизировать любой код. Например, если код содержит вызовы Python функций, то ускорение от компиляции кода может быть незначительным.

Numba не работает с встроенными списками. Перепишем функцию `matrix_product` с использованием ndarray.

In [24]:
```
import numba
import numpy as np


@numba.jit
def jit_matrix_product(X, Y):
    xrows, xcols = X.shape
    yrows, ycols = Y.shape
    Z = np.zeros((xrows, ycols), dtype=X.dtype)
    for i in range(xrows):
        for j in range(ycols):
            for k in range(xcols):
                Z[i, j] += X[i, k] * Y[k, j]
    return Z
```

Посмотрим, что получилось.

In [25]:
```
shape = 64, 64
X = np.random.randint(-255, 255, shape)
Y = np.random.randint(-255, 255, shape)

%timeit -n100 jit_matrix_product(X, Y)
```

```
The slowest run took 38.20 times longer than the fastest. This could mean that
an intermediate result is being cached.
1.31 ms ± 2.52 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Cython

- типизированное расширение языка Python,
- оптимизирующий компилятор Python и Cython в код на C,
- использующий C-API интерпретатора CPython.

Для простоты мы будем работать с Cython из IPython/Jupiter.
"Магическая" команда cython компилирует содержимое ячейки с помощью Cython, а затем загружает все имена из скомпилированного модуля в глобальное пространство имён.

In [26]:
```
%load_ext Cython
```

Cython не может эффективно оптимизировать работу со списками, которые могут содержать элементы различных типов, поэтому перепишем `matrix_product` с использованием *ndarray*.

In [33]:
```
shape = 64, 64
X = np.random.randint(-255, 255, size=shape, dtype=np.int64)
Y = np.random.randint(-255, 255, size=shape, dtype=np.int64)
```

In [27]:
```
%%cython -a
import numpy as np

def cy_matrix_product(X, Y):
    n_xrows, n_xcols = X.shape
    n_yrows, n_ycols = Y.shape
    Z = np.zeros((n_xrows, n_ycols), dtype=X.dtype)
    for i in range(n_xrows):
        for k in range(n_ycols):
            for j in range(n_xcols):
                Z[i, k] += X[i, j] * Y[j, k]
    return Z
```

Out[27]:

Generated by Cython 0.29.20

Yellow lines hint at Python interaction.
Click on a line that starts with a " + " to see the C code that Cython generated for it.

```
+01: import numpy as np
 02:
+03: def cy_matrix_product(X, Y):
+04:     n_xrows, n_xcols = X.shape
+05:     n_yrows, n_ycols = Y.shape
+06:     Z = np.zeros((n_xrows, n_ycols), dtype=X.dtype)
+07:     for i in range(n_xrows):
+08:         for k in range(n_ycols):
+09:             for j in range(n_xcols):
+10:                 Z[i, k] += X[i, j] * Y[j, k]
+11:     return Z
```

In [28]:
```
%timeit cy_matrix_product(X, Y)
```

```
232 ms ± 14.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Избавимся от вызовов python кода, про аннотировав код типами.

In [29]:
```
%%cython -a
import numpy as np
cimport numpy as np

def cy_matrix_product(np.ndarray X, np.ndarray Y):
    cdef int n_xrows = X.shape[0]
    cdef int n_xcols = X.shape[1]
    cdef int n_yrows = Y.shape[0]
    cdef int n_ycols = Y.shape[1]
    cdef np.ndarray Z
    Z = np.zeros((n_xrows, n_ycols), dtype=X.dtype)
    for i in range(n_xrows):
        for k in range(n_ycols):
            for j in range(n_xcols):
                Z[i, k] += X[i, j] * Y[j, k]
    return Z
```

Out[29]:

Generated by Cython 0.29.20

Yellow lines hint at Python interaction.
Click on a line that starts with a " + " to see the C code that Cython generated for it.

```
+01: import numpy as np
 02: cimport numpy as np
 03:
+04: def cy_matrix_product(np.ndarray X, np.ndarray Y):
+05:     cdef int n_xrows = X.shape[0]
+06:     cdef int n_xcols = X.shape[1]
+07:     cdef int n_yrows = Y.shape[0]
+08:     cdef int n_ycols = Y.shape[1]
 09:     cdef np.ndarray Z
+10:     Z = np.zeros((n_xrows, n_ycols), dtype=X.dtype)
+11:     for i in range(n_xrows):
+12:         for k in range(n_ycols):
+13:             for j in range(n_xcols):
+14:                 Z[i, k] += X[i, j] * Y[j, k]
+15:     return Z
```

In [30]:
```
%timeit cy_matrix_product(X, Y)
```

```
253 ms ± 23.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Тело вложенного цикла Cython оптимизировать не смог. Fatality-time: укажем тип элементов в *ndarray*.

In [34]:
```
%%cython -a
import numpy as np
cimport numpy as np

def cy_matrix_product(np.ndarray[np.int64_t, ndim=2] X,
                      np.ndarray[np.int64_t, ndim=2] Y):
    cdef int n_xrows = X.shape[0]
    cdef int n_xcols = X.shape[1]
    cdef int n_yrows = Y.shape[0]
    cdef int n_ycols = Y.shape[1]
    cdef np.ndarray[np.int64_t, ndim=2] Z = \
        np.zeros((n_xrows, n_ycols), dtype=np.int64)
    for i in range(n_xrows):
        for k in range(n_ycols):
            for j in range(n_xcols):
                Z[i, k] += X[i, j] * Y[j, k]
    return Z
```

Out[34]:

Generated by Cython 0.29.20

Yellow lines hint at Python interaction.
Click on a line that starts with a " + " to see the C code that Cython generated for it.

```
+01: import numpy as np
 02: cimport numpy as np
 03:
+04: def cy_matrix_product(np.ndarray[np.int64_t, ndim=2] X,
 05:                       np.ndarray[np.int64_t, ndim=2] Y):
+06:     cdef int n_xrows = X.shape[0]
+07:     cdef int n_xcols = X.shape[1]
+08:     cdef int n_yrows = Y.shape[0]
+09:     cdef int n_ycols = Y.shape[1]
 10:     cdef np.ndarray[np.int64_t, ndim=2] Z = \
+11:         np.zeros((n_xrows, n_ycols), dtype=np.int64)
+12:     for i in range(n_xrows):
+13:         for k in range(n_ycols):
+14:             for j in range(n_xcols):
+15:                 Z[i, k] += X[i, j] * Y[j, k]
+16:     return Z
```

In [35]:
```
%timeit cy_matrix_product(X, Y)
```

```
1.92 ms ± 156 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Попробуем пойти дальше и отключить проверки на выход за границы массива и переполнение целочисленных типов.

In [36]:
```
%%cython -a
import numpy as np

cimport cython
cimport numpy as np

@cython.boundscheck(False)
@cython.overflowcheck(False)
def cy_matrix_product(np.ndarray[np.int64_t, ndim=2] X,
                      np.ndarray[np.int64_t, ndim=2] Y):
    cdef int n_xrows = X.shape[0]
    cdef int n_xcols = X.shape[1]
    cdef int n_yrows = Y.shape[0]
    cdef int n_ycols = Y.shape[1]
    cdef np.ndarray[np.int64_t, ndim=2] Z = \
        np.zeros((n_xrows, n_ycols), dtype=np.int64)
    for i in range(n_xrows):
        for k in range(n_ycols):
            for j in range(n_xcols):
                Z[i, k] += X[i, j] * Y[j, k]
    return Z
```

Out[36]:

Generated by Cython 0.29.20

Yellow lines hint at Python interaction.
Click on a line that starts with a " + " to see the C code that Cython generated for it.

```
+01: import numpy as np
 02:
 03: cimport cython
 04: cimport numpy as np
 05:
 06: @cython.boundscheck(False)
 07: @cython.overflowcheck(False)
+08: def cy_matrix_product(np.ndarray[np.int64_t, ndim=2] X,
 09:                       np.ndarray[np.int64_t, ndim=2] Y):
+10:     cdef int n_xrows = X.shape[0]
+11:     cdef int n_xcols = X.shape[1]
+12:     cdef int n_yrows = Y.shape[0]
+13:     cdef int n_ycols = Y.shape[1]
 14:     cdef np.ndarray[np.int64_t, ndim=2] Z = \
+15:         np.zeros((n_xrows, n_ycols), dtype=np.int64)
+16:     for i in range(n_xrows):
+17:         for k in range(n_ycols):
+18:             for j in range(n_xcols):
+19:                 Z[i, k] += X[i, j] * Y[j, k]
+20:     return Z
```

In [37]:
```
%timeit cy_matrix_product(X, Y)
```

```
1.18 ms ± 86.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Profit.

memory_profiler

In [38]:
```
%load_ext memory_profiler
```

In [39]:
```
%%writefile str_memory_consumption.py
from memory_profiler import profile

@profile
def compare_str_construction(n):
    phrase = 'repeat me'
    pmul = phrase * n
    pjoi = ''.join([phrase for x in range(n)])
    pinc = ''
    for x in range(n):
        pinc += phrase

    del pmul, pjoi, pinc

    return phrase


if __name__ == '__main__':
    compare_str_construction(100_000)
```

```
Overwriting str_memory_consumption.py
```

In [40]:
```
!python str_memory_consumption.py
```

```
Filename: str_memory_consumption.py

Line #    Mem usage    Increment   Line Contents
================================================
     3     43.7 MiB     43.7 MiB   @profile
     4                             def compare_str_construction(n):
     5     43.7 MiB      0.0 MiB       phrase = 'repeat me'
     6     44.6 MiB      0.9 MiB       pmul = phrase * n
     7     46.1 MiB      0.4 MiB       pjoi = ''.join([phrase for x in range
(n)])
     8     45.5 MiB      0.0 MiB       pinc = ''
     9     52.9 MiB      0.0 MiB       for x in range(n):
    10     52.9 MiB      0.7 MiB           pinc += phrase
    11
    12     44.4 MiB      0.0 MiB       del pmul, pjoi, pinc
    13
    14     44.4 MiB      0.0 MiB       return phrase
```

In [41]:
```
%%writefile slot_memory_consumption.py
from memory_profiler import profile
from random import uniform

class Particle():
    def __init__(self, x, y):
        self.x = x
        self.y = y


class ParticleSlot():
    __slots__ = ['x', 'y']

    def __init__(self, x, y):
        self.x = x
        self.y = y


@profile
def generate_particles(n):
    particles = [Particle(uniform(-1.0, 1.0), uniform(-1.0, 1.0)) for _ in ran
ge(n)]
    particles_sl = [ParticleSlot(uniform(-1.0, 1.0), uniform(-1.0, 1.0)) for _
in range(n)]


if __name__ == '__main__':
    generate_particles(100_000)
```

```
Overwriting slot_memory_consumption.py
```

In [42]:
```
!python slot_memory_consumption.py
```

```
Filename: slot_memory_consumption.py

Line #    Mem usage    Increment   Line Contents
================================================
    18     43.9 MiB     43.9 MiB   @profile
    19                             def generate_particles(n):
    20     70.6 MiB      0.5 MiB       particles = [Particle(uniform(-1.0, 1.
0), uniform(-1.0, 1.0)) for _ in range(n)]
    21     83.7 MiB      0.7 MiB       particles_sl = [ParticleSlot(uniform(-
1.0, 1.0), uniform(-1.0, 1.0)) for _ in range(n)]
```

Домашнее задание

Для следующих простых задач нужно привести 2-3 способа решения на python, сравнить между собой эти способы по затрачиваемым времени и памяти и объяснить с чем это может быть связано.

1. Чтение/запись в словарь. При попытке чтения несуществующего ключа возвращается None.
2. К каждому элементу списка применить какуе-либо преобразование (например, для числового списка - возвести в кавдрат, для строкового - привести к верхнему регистру, отфильтровать определенные символы, и т.д.).
3. Отсортировать список.
4. Распаковать вложенный список.

https://epa.ms/pyhomework (https://epa.ms/pyhomework)