Лекция 12: Быстрее Python, ещё быстрее

Сергей Лебедев

sergei.a.lebedev@gmail.com

30 ноября 2015 г.

```
[[4], [2]]
```

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

```
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

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

Пример: функция matrix_product

```
def matrix_product(X, Y):
    """Computes the matrix product of X and Y.

    >>> X = Matrix([[1], [2], [3]])
    >>> Y = Matrix([[4, 5, 6]])
    >>> matrix_product(X, Y)
    [[4, 5, 6], [8, 10, 12], [12, 15, 18]]
    >>> matrix_product(Y, X)
    [[32]]
    """
    n_xrows, n_xcols = X.shape
    n_yrows, n_ycols = Y.shape
    # верим, что с размерностями всё хорошо
    Z = Matrix.zeros((n_xrows, n_ycols))
    for i in range(n_xrows):
        for j in range(n_xcols):
            for k in range(n_ycols):
                Z[i][k] += X[i][j] * Y[j][k]
    return Z
```

Измерение времени работы

- Модуль timeit реализует одноимённую функцию timeit, которую можно использовать для измерения скорости работы кода на Python:

```
In [1]: import timeit

In [2]: setup = """
   ...: from faster_python_faster import Matrix, \
   ...:     matrix_product
   ...: shape = 64, 64
   ...: X = Matrix.random(shape)
   ...: Y = Matrix.random(shape)
   ...: """

In [3]: timeit.timeit("matrix_product(X, Y)", setup,
   ...:                       number=10)
Out[3]: 1.9958365359925665
```

- Функция timeit замеряет время с помощью функции time.perf_counter. На время измерений отключается сборщик мусора.

В IPython есть “магическая” команда timeit, которая упрощает работу с одноимённой функцией:

```
In [1]: from faster_python_faster import Matrix, \
   ...:     matrix_product

In [2]: shape = 64, 64

In [3]: X, Y = Matrix.random(shape), Matrix.random(shape)

In [4]: %timeit matrix_product(X, Y)
1 loops, best of 3: 198 ms per loop

In [5]: %timeit -n100 matrix_product(X, Y)
100 loops, best of 3: 198 ms per loop
```

- Умножение двух случайных матриц из целых чисел размера 64x64 занимает чуть меньше 200 миллисекунд.
- 5 умножений матриц в секунду. Подозрительно медленно.
- В чём может быть проблема?

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

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

```
def bench(shape=(64, 64), n_iter=16):
    X = Matrix.random(shape)
    Y = Matrix.random(shape)
    for iter in range(n_iter):
        matrix_product(X, Y)


if __name__ == "__main__":
    bench()
```

Модуль cProfile позволяет профилировать код на Python с точностью до вызова функции или метода:

```
In [1]: import cProfile

In [2]: source = open("faster_python_faster.py").read()

In [3]: cProfile.run(source, sort="tottime")
         41380 function calls in 3.209 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall          function
       16    3.172    0.198    3.173    0.198    matrix_product
   # ...
```

Для наших целей выглядит довольно бесполезно. Что делать?

- В отличие от cProfile модуль line_profiler анализирует время работы с точностью до строки в исходном коде.
  $ pip install line_profiler
- Модуль расширяет систему “магических” команд IPython командой lprun. Чтобы воспользоваться ей, сначала нужно загрузить файл расширения:

```
In [1]: %load_ext line_profiler

In [2]: from faster_python_faster import matrix_product,
   ...:     bench

In [3]: %lprun -f matrix_product bench()
#                ^                  ^
#                |                  |
#           имя функции        выражение
```

```
In [1]: %lprun -f matrix_product bench()
Timer unit: 1e-06 s

Total time: 9.08323 s
File: faster_python_faster.py
Function: matrix_product at line 24

% Time  Line Contents
==============================================
        def matrix_product(X, Y):
   0.0      n_xrows, n_xcols = X.shape
   0.0      n_yrows, ycols = Y.shape
   0.0      Z = Matrix.zeros((n_xrows, n_ycols))
   0.0      for i in range(n_xrows):
   0.4          for j in range(n_xcols):
  26.4              for k in range(n_ycols):
  73.2                  Z[i][k] += X[i][j] * Y[j][k]
   0.0      return Z
```

Операция `list.__getitem__` не бесплатна. Запомним значения `X[i]` и `Z[i]` и переставим местами циклы **`for`** так, чтобы код делал меньше обращений по индексу.

```
def matrix_product(X, Y):
    n_xrows, n_xcols = X.shape
    n_yrows, n_ycols = Y.shape
    Z = Matrix.zeros((n_xrows, n_ycols))
    for i in range(n_xrows):
        Xi = X[i]
        for k in range(n_ycols):
            acc = 0
            for j in range(n_xcols):
                acc += Xi[j] * Y[j][k]
            Z[i][k] = acc
    return Z
```

```
In [1]: %lprun -f matrix_product bench()
Timer unit: 1e-06 s

Total time: 7.5519 s
File: faster_python_faster.py
Function: matrix_product at line 36

% Time  Line Contents
==============================================
        def matrix_product(X, Y):
   0.0      n_xrows, n_xcols = X.shape
   0.0      n_yrows, n_ycols = Y.shape
   0.0      Z = Matrix.zeros((n_xrows, n_ycols))
   0.0      for i in range(n_xrows):
   0.0          Xi, Zi = X[i], Z[i]
   0.6          for k in range(n_ycols):
   0.5              acc = 0
  36.2              for j in range(n_xcols):
  61.7                  acc += Xi[j] * Y[j][k]
   0.8              Zi[k] = acc
   0.0      return Z
```

Больше 30% времени уходит на работу внутренней машинерии цикла **for**. В данном случае цикл **for** можно заменить на выражение-генератор.

```
def matrix_product(X, Y):
    n_xrows, n_xcols = X.shape
    n_yrows, n_ycols = Y.shape
    Z = Matrix.zeros((n_xrows, n_ycols))
    for i in range(n_xrows):
        Xi, Zi = X[i], Z[i]
        for k in range(n_ycols):
            Z[i][k] = sum(Xi[j] * Y[j][k]
                          for j in range(n_xcols))
    return Z
```

```
In [1]: %lprun -f matrix_product bench()
Timer unit: 1e-06 s

Total time: 3.7232 s
File: faster_python_faster.py
Function: matrix_product at line 50

% Time  Line Contents
==============================================================
        def matrix_product(X, Y):
   0.0      n_xrows, n_xcols = X.shape
   0.0      n_yrows, n_ycols = Y.shape
   0.0      Z = Matrix.zeros((n_xrows, n_ycols))
   0.0      for i in range(xrows):
   0.0          Xi, Zi = X[i], Z[i]
   1.8          for k in range(n_ycols):
  98.1              Zi[k] = sum(Xi[j] * Y[j][k]
   0.0                          for j in range(xcols))
   0.0      return Z
```

Почти всё время функция `matrix_product` проводит в самом внутреннем цикле. Попробуем убрать из него ещё одно обращение по индексу, транспонировав матрицу `Y`.

```
def matrix_product(X, Y):
    n_xrows, n_xcols = X.shape
    n_yrows, n_ycols = Y.shape
    Z = Matrix.zeros((n_xrows, n_ycols))
    Yt = Y.transpose()  # <--
    for i, (Xi, Zi) in enumerate(zip(X, Z)):
        for k, Ytk in enumerate(Yt):
            Zi[k] = sum(Xi[j] * Ytk[j]
                        for j in range(n_xcols))
    return Z
```

```
In [1]: %lprun -f matrix_product bench()
Timer unit: 1e-06 s

Total time: 2.72339 s
File: faster_python_faster.py
Function: matrix_product at line 76

% Time  Line Contents
==============================================================
         def matrix_product(X, Y):
   0.0       n_xrows, n_xcols = X.shape
   0.0       n_yrows, n_ycols = Y.shape
   0.0       Z = Matrix.zeros((n_xrows, n_ycols))
   0.0       Yt = Y.transpose()
   0.1       for i, (Xi, Zi) in enumerate(zip(X, Z)):
   2.9           for k, Ytk in enumerate(Yt):
  97.0               Zi[k] = sum(Xi[j] * Ytk[j]
   0.0                           for j in range(n_xcols))
   0.0       return Z
```

Измерение времени работы: резюме

- Измерить время работы функции можно с помощью функции `timeit` из модуля `timeit`.
- Найти узкое место в программе — с помощью модуля `cProfile`.
- Оценить вклад каждой строки кода в общее время работы функции — с помощью модуля `line_profiler`.
- В IPython есть “магические” команды для всех трёх типов измерений:
  - `%timeit`,
  - `%prun`,
  - `%lprun`.

- NumPy — библиотека для научных вычислений на Python.
- В основе библиотеки — *ndarray* — многомерный типизированный массив.
- Сравним результаты наших стараний с *ndarray*:

```
In [1]: shape = 64, 64

In [2]: X, Y = Matrix.random(shape), Matrix.random(shape)

In [3]: %timeit -n100 matrix_product(X, Y)
100 loops, best of 3: 57 ms per loop

In [4]: import numpy as np

In [5]: X = np.random.randint(-255, 255, shape)

In [6]: Y = np.random.randint(-255, 255, shape)

In [7]: %timeit -n100 X.dot(Y)
100 loops, best of 3: 321 µs per loop  # :-(
```

- 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 подмножестве Python и преобразуем его в код на C++ (**Pythran**) или 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**) или C++ (**HOPE**), а затем компилируем в машинный код.

Numba

- Поставить Numba с помощью `pip` может быть непросто. Рекомендуемый способ установки описан по ссылке: http://bit.ly/install-numba.
- На момент версии 0.21 Numba не умеет эффективно оптимизировать код со списками, поэтому нам придётся переписать функцию `matrix_product` с использованием *ndarray*:

```
import numba

@numba.jit
def jit_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
```

- Для использования Numba достаточно декорировать функцию с помощью `numba.jit`.
- В момент первого вызова функция будет транслирована в LLVM и скомпилирована в машинный код:

```
In [1]: %timeit -n100 jit_matrix_product(X, Y)
100 loops, best of 3: 332 µs per loop
```

- Декоратор `numba.jit` пытается вывести типы аргументов и возвращаемого значения декорируемой функции:

```
In [2]: jit_matrix_product.inspect_types()
jit_matrix_product (
    array(int64, 2d, C),
    array(int64, 2d, C))
```

- Напоминание: Numba не может эффективно оптимизировать любой код.
- Например, если код содержит вызовы Python функций, то ускорение от компиляции кода может быть незначительным:

```
In [1]: @numba.jit
   ...: def jit_matrix_product(X, Y):
   ...:     n_xrows, n_xcols = X.shape
   ...:     n_yrows, n_ycols = Y.shape
   ...:     Z = np.zeros((n_xrows, n_ycols))
   ...:     for i in range(n_xrows):
   ...:         for k in range(n_ycols):
   ...:             Z[i, k] = (X[i, :] * Y[:, k]).sum()
   ...:     return Z
   ...:

In [2]: %timeit -n100 jit_matrix_product(X, Y)
100 loops, best of 3: 1.3 ms per loop  # :-(
```

- Numba — это JIT компилятор для Python кода, основанный на трансляции в LLVM.
- В теории Numba не требует никаких изменений в коде, кроме использования декоратора `numba.jit`.
- На практике далеко не любой код поддаётся эффективной оптимизации.
- Мы **не** поговорили о:
  - явной аннотации типов,
  - интеграции с CUDA,
  - AOT компиляции кода, использующего Numba.
- Почитать об этом можно в документации проекта: `http://numba.pydata.org`.

Cython

- Cython — это:
  - типизированное **расширение**[1] 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:

```
In [1]: %load_ext cython

In [2]: %%cython
   ...: print("Hello, world!")
   ...:
Hello, world!
Out[2]: {}
```

- Узнать подробности о взаимодействии Cython с системой импортов и библиотекой `setuptools` можно на сайте проекта: http://bit.ly/compile-cython.

[1] Любой код на Python — это корректный код на Cython.

“Магическая” команда `cython` компилирует содержимое ячейки с помощью Cython, а затем загружает все имена из скомпилированного модуля в глобальное пространство имён.

```
In [1]: %%cython
   ...: def f():
   ...:     return 42
   ...: def g():
   ...:     return []
   ...:

In [2]: f
Out[2]: <function _cython_magic_[...].f>

In [3]: g
Out[3]: <function _cython_magic_[...].g>

In [4]: f(), g()
Out[4]: (42, [])
```

Скомпилируем функцию `cy_matrix_product` с помощью Cython и измерим её производительность.

```
In [1]: %%cython
   ...: from faster_python_faster import Matrix
   ...:
   ...: def cy_matrix_product(X, Y):
   ...:     n_xrows, n_xcols = X.shape
   ...:     n_yrows, n_ycols = Y.shape
   ...:     Z = Matrix.zeros((n_xrows, n_ycols))
   ...:     Yt = Y.transpose()
   ...:     for i, (Xi, Zi) in enumerate(zip(X, Z)):
   ...:         for k, Ytk in enumerate(Yt):
   ...:             Zi[k] = sum(Xi[j] * Ytk[j]
   ...:                         for j in range(n_xcols))
   ...:     return Z
   ...:

In [2]: %timeit -n100 cy_matrix_product(X, Y)
10 loops, best of 3: 34.3 ms per loop
```

- Компилятор Cython поддерживает опциональную типизацию.
- Посмотрим, что происходит в функции `cy_matrix_product`:

```
In [1]: %%cython -a
   ...: def cy_matrix_product(X, Y):
   ...:     n_xrows, n_xcols = X.shape
   ...:     n_yrows, n_ycols = Y.shape
   ...:     Z = Matrix.zeros((n_xrows, n_ycols))
   ...:     Yt = Y.transpose()
   ...:     for i, (Xi, Zi) in enumerate(zip(X, Z)):
   ...:         for k, Ytk in enumerate(Yt):
   ...:             Zi[k] = sum(Xi[j] * Ytk[j]
   ...:                         for j in range(n_xcols))
   ...:     return Z

Out[2]: <IPython.core.display.HTML at 0x108ebac18>
```

Результат аннотации функции cy_matrix_product

```
def cy_matrix_product(X, Y):
    n_xrows, n_xcols = X.shape
    n_yrows, n_ycols = Y.shape
    Z = Matrix.zeros((n_xrows, n_ycols))
    Yt = Y.transpose()
    for i, Xi in enumerate(X):
        for k, Ytk in enumerate(Yt):
            Z[i][k] = sum(Xi[j] * Ytk[j] for j in range(n_xcols))
    return Z
```

- Чем интенсивнее цвет, тем менее специфичен тип выражения и тем медленней выполняется фрагмент кода.
- Обилие желтого цвета намекает, что результат компиляции функции `cy_matrix_product` мало чем отличается от её версии на Python, потому что все объекты имеют тип `PyObject`.

К сожалению, списки в Python гетерогенны, поэтому, как и в случае с Numba, мы вынуждены перейти к использованию *ndarray*:

```
In [1]: %%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
   ...:

In [2]: %timeit -n100 cy_matrix_product(X, Y)
100 loops, best of 3: 182 ms per loop
```

```
In [1]: %%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
   ...:

In [2]: %timeit -n100 cy_matrix_product(X, Y)
100 loops, best of 3: 189 ms per loop  # :-(
```

```
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
```

Несмотря на то что все переменные имеют явный тип, тип элемента *ndarray* всё ещё не определён, поэтому тело самого вложенного цикла ярко-жёлтое.

```
In [1]: %%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 np.ndarray[np.int64_t, ndim=2] 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
   ...:

In [2]: %timeit -n100 cy_matrix_product(X, Y)
100 loops, best of 3: 877 µs per loop  # O_O
```

```
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
```

Всё прекрасно, но иногда хочется большего ;)

Cython позволяет пожертвовать безопасностью в пользу производительности, отключив проверки выхода за границы массива и проверки переполнения в целочисленных операциях.

```
In [1]: %%cython -a
   ...: import numpy as np
   ...: cimport numpy as np
   ...: cimport cython
   ...:
   ...: @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):
   ...:     # ...
   ...:

In [2]: %timeit -n100 cy_matrix_product(X, Y)
100 loops, best of 3: 611 µs per loop
```

- Cython — удобный инструмент для написания критичного по производительности кода на Python-подобном синтаксисе.
- Мы обсудили только самые основы использования Cython, в частности, мы **не** коснулись:
  - нюансов языка (C-функций и типов расширения),
  - взаимодействия Cython с кодом на C и C++*,
  - многопоточности,
  - профилирования и отладки.
- Обо всём этом и многом другом можно узнать из документации: `http://docs.cython.org`.

P.S.

```
In [3]: %timeit -n100 X.dot(Y)
100 loops, best of 3: 328 µs per loop
```