Лабы

Главная » Python

Python. Лабораторная работа №15

Повышение производительности Python

▼ Оглавление

Итак, ваш питонский код слишком медленный. Что же делать?

Конечно же, писать на C!

На этом вкладку можно закрывать, удачи вам, любви и терпения.

Baseline

Если серьёзно, то, во-первых — давайте подведём базу. Напишем питонский код, который решает задачу долго:

```
# pure_python.py
from math import cos, exp

def f(x):
    return cos(x + x * x * x) if x <= 1.0 else exp(-x * x) - x * x + 2.0 *


def integrate(a, b):
    n = 100_000_000
    h = (b - a) / n
    h2 = h * 0.5
    s = 0.0
    for i in range(n):
        s += f(a + i * h + h2)
    return s * h

if __name__ == '__main__':
    print(integrate(0.0, 2.0))
```

Запускаем с замером времени:

```
$ time python3 pure_python.py
1.435370044777462
python3 pure_python.py  50.03s user 0.06s system 99% cpu 50.454 total
```

Пока этот код выполняется, можно в полной мере прочувствовать процессы старения в организме… Замеры времени здесь и далее приведены только для пристрелочного сравнения масштабов времени, ничего предварительно не настраивалось, серьёзной статистики не собиралось, на фоне бубнил ютубчек, в

системе сидела куча лишних процессов и т.д. и т.п. — но некоторое общее понимание можно составить, мы тут не будем устраивать специальную олимпиаду и стараться выжимать сотые доли процента времени выполнения.

А теперь сишечное:

```c
// pure_c.c
#include <stdio.h>
#include <math.h>
#define N 100000000U

double f(double x)
{
    return x <= 1.0 ? cos(x + x * x * x) : exp(-x * x) - x * x + 2.0 * x;
}

double integrate(double a, double b)
{
    const double h = (b - a) / N;
    const double h2 = h * 0.5;
    double s = 0.0;
    for (register unsigned int i = 0; i < N; i++)
        s += f(a + i * h + h2);
    return s * h;
}

int main()
{
    printf("%lf\n", integrate(0.0, 2.0));
}
```

```
$ gcc pure_c.c -o pure_c -lm && time ./pure_c
```

```
1.435370
./pure_c  2.87s user 0.02s system 98% cpu 2.946 total
```

Довольно-таки неплохо. Очень приятно.. И это без каких-либо оптимизаций. Поэтому можно по-дешёвке сделать ещё лучше:

```
$ gcc pure_c.c -o pure_c -Ofast -march=native -lm && time ./pure_c
1.435370
./pure_c  1.60s user 0.00s system 99% cpu 1.600 total
```

Если это не “блейзингли фаст”, то уже где-то рядом с ним. Уверен, что вы сможете выжать больше, если ещё поиграете с флагами. Тут главное — не проиграть.

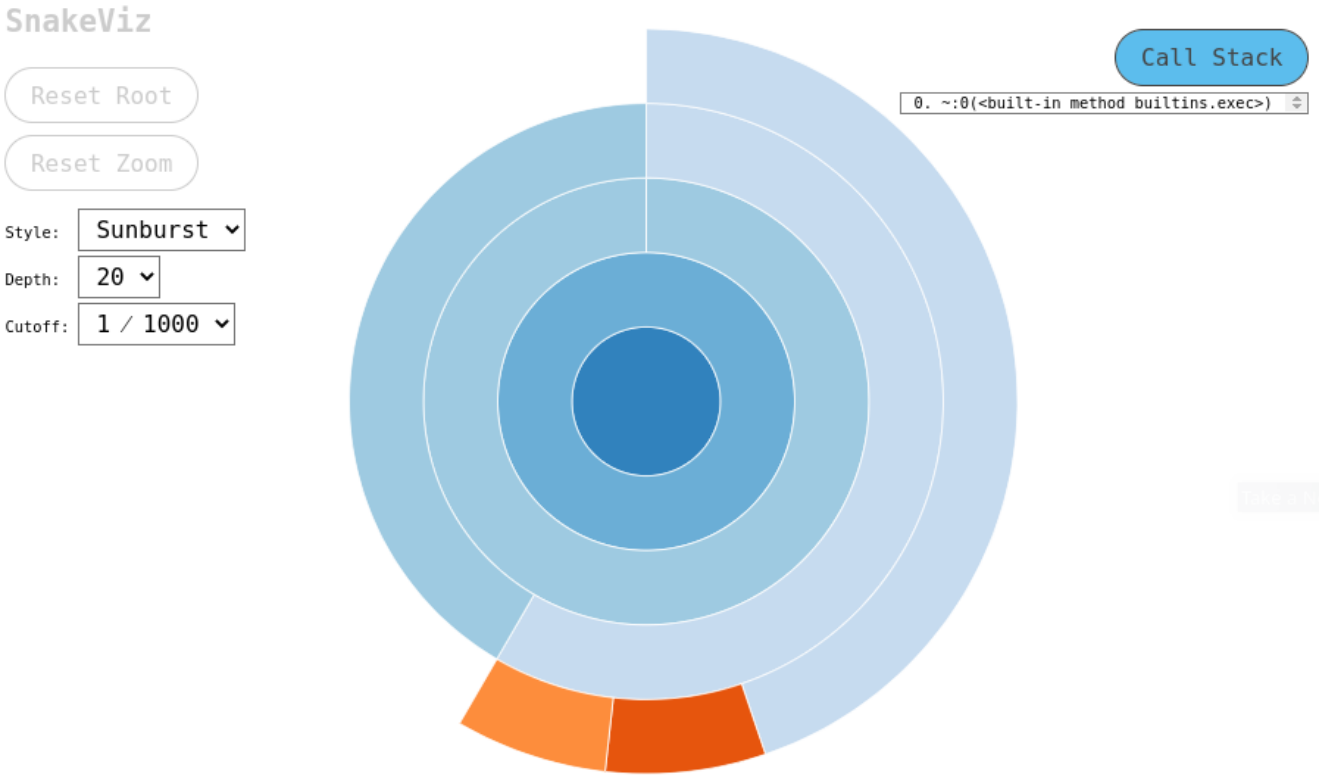
Посмотрим на вывод cProfile:

```
$ python3 -m cProfile -s time pure_python.py
1.435370044777462
         200000213 function calls in 83.776 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
100000000   35.877    0.000   49.500    0.000 pure_python.py:4(f)
        1   34.275   34.275   83.775   83.775 pure_python.py:8(integrate)
 50000000    7.140    0.000    7.140    0.000 {built-in method math.cos}
 50000000    6.483    0.000    6.483    0.000 {built-in method math.exp}
        1    0.000    0.000    0.000    0.000 {built-in method _imp.create
...
```

Можно получить красивое, если натравить на него snakeviz:

```
$ python3 -m cProfile -o pure_python.prof pure_python.py
```

```
$ snakeviz pure_python.prof
```



Search:

ncalls	tottime	percall	cumtime	percall	fi
100000000	35.28	3.528e-07	45.99	4.599e-07	pure_python.py:4(f)
1	32.77	32.77	78.76	78.76	pure_python.py:8(integrate)
50000000	5.491	1.098e-07	5.491	1.098e-07	~:0(<built-in method math.cos>)
50000000	5.215	1.043e-07	5.215	1.043e-07	~:0(<built-in method math.exp>)
1	0.0003025	0.0003025	0.0003025	0.0003025	~:0(<built-in method _imp.create_dyna
4	4.768e-05	1.192e-05	0.0001565	3.913e-05	<frozen importlib._bootstrap_external>

Для увеличения ширины, посмотрите какие ещё бывают питонские профилировщики, а ещё лучше - прогоните через какие-нибудь из них свой код.

Во-вторых, иногда бывает, что просто надо ускорить именно питонское. Стек у вас такой, или ТЗ, или просто хочется повыделываться — выбирайте сами.

Кроме того, давайте договоримся на берегу - здесь и далее подразумевается что **все сборки/установки пакетов/модулей/расширений происходят в** виртуальных окружениях.

Переходим собственно, к способам:

1. Компиляторы для Python кода

- **Nuitka**

Это т.н. “транспайлер” — модное слово, которое в данном случае означает, что питонский код сперва транслируется в C, а потом компилируется уже в исполняемый файл. Если вы помните, что, например, GCC-шная компиляция — это на самом деле двухуровневая трансляция, смысл крутого слова в итоге сводится к маркетингу, но оставим это на чьей-нибудь другой совести.

Попробуем (не забудьте про `venv`):

```
$ python3 -m nuitka pure_python.py
$ time ./pure_python.bin
1.435370044777462
./pure_python.bin  24.98s user 0.02s system 99% cpu 25.041 total
```

Лучше, но уж точно не быстро. Наверняка можно подкрутить - почитайте документацию.

- **Numba**

Numba - это уже JIT-компилятор, поэтому с ним нужно немножко допилить код. Хотя, в документации написано, как использовать его и для AOT компиляции, мы сейчас не будем в это углубляться. Зато из коробки есть возможность буквально в одну строку распоточить код, а это всегда приятно.

```
# run_numba.py
from math import cos, exp
from numba import njit, prange

@njit
def f(x):
    return cos(x + x * x * x) if x <= 1.0 else exp(-x * x) - x * x + 2.0 *
```

```
@njit(parallel=True)
def integrate(a, b):
    n = 100_000_000
    h = (b - a) / n
    h2 = h * 0.5
    s = 0.0
    for i in prange(n):
        s += f(a + i * h + h2)
    return s * h


if __name__ == '__main__':
    print(integrate(0.0, 2.0))
```

Смотрим:

```
$ time python3 run_numba.py
1.4353700447773279
python3 run_numba.py  4.95s user 0.22s system 210% cpu 2.455 total
```

И это вместе с компиляцией. Здорово!

- **PyPy**

Тоже JIT-компилятор. Самое халявное решение, потому что он жуёт чистый питонский код без изменений. Только посмотрите на это:

```
$ pypy pure_python.py
1.435370044777462
pypy pure_python.py  2.98s user 0.01s system 99% cpu 3.014 total
```

2. Специализированные питонские пакеты, написанные на C

- **numpy**

Numpy - это универсальный вычислительный пакет, очень популярный, в основном применяется для работы с массивами (матрицами, тензорами). Сделаем с ним:

```python
# run_numpy.py
import numpy as np

def f(x):
    return np.piecewise(
        x,
        [x <= 1, x > 1],
        [lambda x: np.cos(x + x * x * x), lambda x: np.exp(-x * x) - x * x
    )


def integrate(a, b):
    n = 100_000_000
    h = (b - a) / n
    h2 = h * 0.5
    mesh = np.linspace(a+h2, b-h2, num=n)
    res = f(mesh)
    return np.sum(res) * h


if __name__ == '__main__':
    print(integrate(0.0, 2.0))
```

Проверяем:

```
time python3 run_numpy.py
1.4353700447774913
python3 run_numpy.py  3.15s user 0.79s system 93% cpu 4.202 total
```

Весьма впечатляет! Но какой ценой:

2.5 Гб памяти - тоже впечатляет, но уже в другую сторону. Да, это определённо можно подрезать, но суть, думаю, ясна.

- **quadpy**

А это уже совсем-совсем специализированный пакет именно для численного интегрирования. Натравим его на нашу нумпишную функцию:

```
# run_quadpy.py
import numpy as np
import quadpy

def f(x):
    return np.piecewise(
        x,
        [x <= 1, x > 1],
        [lambda x: np.cos(x + x * x * x), lambda x: np.exp(-x * x) - x * x
    )


if __name__ == '__main__':
    val, err = quadpy.quad(f, 0.0, 2.0)
    print(val)
```

```
time python3 run_quadpy.py
1.4353700447774913
python3 run_quadpy.py  1.91s user 0.54s system 100% cpu 2.447 total
```

Теперь уже совсем хорошо, почти так же быстро как сишное. Вот только:

```
$ pip list
Package            Version
------------------ ------------
appdirs            1.4.4
certifi            2022.9.24
cffi               1.15.1
charset-normalizer 2.1.1
commonmark         0.9.1
contourpy          1.0.6
cplot              0.9.3
cryptography       38.0.3
cycler             0.11.0
fonttools          4.38.0
idna               3.4
kgt                0.4.7
kiwisolver         1.4.4
matplotlib         3.6.2
matplotx           0.3.10
mpmath             1.2.1
ndim               0.1.9
networkx           2.8.8
npx                0.1.1
numpy              1.23.4
orthopy            0.8.92
packaging          21.3
Pillow             9.3.0
```

```
pip                22.3.1
pycparser          2.21
Pygments           2.13.0
PyNaCl             1.5.0
pyparsing          3.0.9
pypng              0.20220715.0
python-dateutil    2.8.2
quadpy             0.16.16
requests           2.28.1
rich               12.6.0
scipy              1.9.3
setuptools         49.2.1
six                1.16.0
sympy              1.11.1
tomli              2.0.1
tomli_w            1.0.0
typing_extensions  4.4.0
urllib3            1.26.12
x21                0.3.6
```

Да, вы всё правильно поняли, это было голое окружение, куда я поставил только quadpy, т.е. почти все эти пакеты — его зависимости. В некоторых случаях это может быть проблемой.

We need to go deeper.

3. Динамическая библиотека на C, подключаемая к Python

Тут всё довольно просто - пишем на C, компилируем это в разделяемую библиотеку, потом используем в Python. Сишный код почти такой же, как и раньше, просто из него за ненадобностью выкидываем `stdio` и `main`.

Поехали:

```
$ gcc c_shared_lib.c -fPIC -shared -o c_shared_lib.so -lm -Ofast -march=na
```

Питонское:

```
# run_c_so_from_python.py
from ctypes import c_double, CDLL

if __name__ == '__main__':
    my_functions = CDLL("./c_shared_lib.so")
    my_functions.integrate.argtypes = [c_double, c_double]
    my_functions.integrate.restype = c_double
    print(my_functions.integrate(0.0, 2.0))
```

Никаких лишний движений в сишном коде, но в питонском мы должны знать и уметь в `ctypes`. В итоге это относительно просто решается написанием обёрточек и убиранием всего страшного в пакеты/модули, но остаётся проблема таскания с собой или пересборки на новом месте вашей so-шки.

Запускаем:

```
$ time python3 run_c_so_from_python.py
1.435370044777462
python3 run_c_so_from_python.py  1.59s user 0.02s system 99% cpu 1.613 tot
```

Почти так же быстро, как на чистом — оно и на самом деле почти так, оверхед у нас добавился по сути только на запуск интерпретатора, передачу туда-обратно аргументов/возврата и завершение.

Проблемки начнутся, если мы хотим сделать из нашего кода что-то более универсальное, например — поддержку питонских callables. Сдуваем пыль с

указателей на функции:

```c
// c_shared_lib.c
double integrate(double a, double b, double (*f)(double))
{
    // тут то же самое
}

// добавим ещё функцию для работы с массивами (последовательностями? колле
// сравнение производительности оставлю для вас. На самом деле потому что
int sum_array(unsigned int n, int a[n])
{
    int s = 0;
    for (unsigned int i = 0; i < n; i++)
        s += a[i];
    return s;
}
```

Питоновое:

```python
# run_c_so_from_python.py
from ctypes import c_int, c_uint, c_double, CDLL, CFUNCTYPE
from random import randint
from math import cos, exp


def f(x):
    return cos(x + x * x * x) if x <= 1.0 else exp(-x * x) - x * x + 2.0 *


if __name__ == '__main__':
    my_functions = CDLL("./c_shared_lib.so")
```

```
FXFUNC = CFUNCTYPE(c_double, c_double)
my_functions.integrate.argtypes = [c_double, c_double, FXFUNC]
my_functions.integrate.restype = c_double
print(my_functions.integrate(0.0, 2.0, FXFUNC(f)))

## это уже без меня
# n = 1000
# a = [randint(-n, n) for i in range(n)]
# print(a)
# print('python sum =', sum(a))

# n_ints_array = c_int * n
# my_functions.sum_array.argtypes = [c_uint, n_ints_array]
# my_functions.sum_array.restype = c_int
# print(f'c sum = {my_functions.sum_array(n, n_ints_array(*a))}')
```

Обращаю внимание на предварительные ласки питоновых коллбеков для их проброса в сишный код. И это ещё далеко не самый плохой вариант. Воздуха набрали?

```
$ time python3 run_c_so_from_python.py
1.435370044777462
python3 run_c_so_from_python.py  53.04s user 0.02s system 99% cpu 53.084 t
```

Чуть медленнее, чем просто на питоне 😂 Надеюсь, понятно почему.

Почти всегда есть варианты, как это победить. Далеко не самый лучший, но довольно интересный - прикрутить генерацию кода. Конечно, этот способ доступен не всегда, но нам повезло с примером. Используем пакет SymPy.

Генерация кода с SymPy

Чтобы фокус удался, нам нужно преобразовать нашу функцию в символьную:

```python
# build_and_run_sympy_with_so.py
from os import system
import sympy as sp
from sympy.utilities.codegen import codegen
from ctypes import c_double, CDLL


def write_file(filename, content):
    with open(filename, 'w') as f:
        f.writelines(content)


if __name__ == '__main__':
    # собственно, вот так вот. Дальше - дело техники
    x = sp.Symbol('x')
    f = sp.Piecewise(
        (sp.cos(x + x**3), x <= 1.0),
        (sp.exp(-x ** 2) - x ** 2 + 2.0 * x, x > 1.0)
    )

    f_name = 'f'
    # а вот тут-то всё и случится
    [(c_name, c_code), (h_name, h_code)] = codegen(
        (f_name, f),
        'C',
        'integral_func',
        header=False
    )

    # де-pow-ификация
    for i in range(2, 4):
        c_code = c_code.replace(f'pow({x.name}, {i})', ' * '.join(list(x.n
```

```
    write_file(c_name, c_code)
    write_file(h_name, h_code)

    so_name = 'c_shared_lib.so'
    system(f'gcc c_shared_lib.c {c_name} -fPIC -shared -o {so_name} -lm -O
    my_functions = CDLL(f'./{so_name}')
    my_functions.integrate.argtypes = [c_double, c_double]
    my_functions.integrate.restype = c_double
    print(my_functions.integrate(0.0, 2.0))
```

Сборку я написал максимально костыльно, можно было сделать хотя б мейкфайл, а вообще рекомендую копать в сторону `setuptools`. Зато сишное стало ещё проще:

```
// c_shared_lib.c
#include "integral_func.h"
#define N 100000000U

double integrate(double a, double b)
{
    const double h = (b - a) / N;
    const double h2 = h / 2.0;
    double s = 0.0;
    for (register unsigned int i = 0; i < N; i++)
        s += f(a + i * h + h2);
    return s * h;
}
```

Следите, чтобы имена файлов совпадали в коде на C и на Python. Пробуем:

```
$ time python3 build_and_run_sympy_with_so.py
1.435370044777462
```

```
python3 build_and_run_sympy_with_so.py  2.54s user 0.12s system 99% cpu 2.
```

Надеюсь, никто не удивлён, и да — это вместе с компиляцией, что по идее не совсем правильно.

4. SWIG

SWIG - simplified wrapper and interface generator. Позволяет генерировать обёртки сишного или плюсового кода для других языков, которых он поддерживает большое количество и, в том числе, наш Пайтон. Чтобы это сделать, нам понадобится т.н. интерфейсный файл `.i` :

```
/* my.i */
%module my
%{
   #define SWIG_FILE_WITH_INIT
   #include "my.h"
%}

%include "carrays.i"
%array_functions(int,intArray)
%include "my.h"
```

Если массивы не нужны, соотвествующие строки из файла можно убрать. Если нужны — копать сюда. Сишный код снова почти такой же:

```
// my.c
#include <math.h>
#define N 100000000U

static double f(double x)
{
```

```
    return x <= 1.0 ? cos(x + x * x * x) : exp(-x * x) - x * x + 2.0 * x;
}

double integrate(double a, double b)
{
    const double h = (b - a) / N;
    const double h2 = h * 0.5;
    double s = 0.0;
    for (register unsigned int i = 0; i < N; i++)
        s += f(a + i * h + h2);
    return s * h;
}

int sum_array(int *a, unsigned long int n)
{
    int s = 0;
    for (unsigned long int i = 0; i < n; i++)
        s += a[i];
    return s;
}
```

Добавляем заголовочник:

```
// my.h
#ifndef MY_H
#define MY_H
double integrate(double, double);
int sum_array(int *, unsigned long int);
#endif
```

Теперь можно сгенерировать обёртку:

```
$ swig -python my.i
```

И ещё нам понадобится питонячий файл для сборки питонского же модуля из сгенерированного SWIGом кода:

```
# setup.py
from distutils.core import setup, Extension
my_module = Extension('_my', sources=['my_wrap.c', 'my.c'])
setup(
    name = 'my',
    ext_modules = [my_module],
    py_modules = ["my"]
)
```

Собираем:

```
$ python3 setup.py build_ext --inplace
```

И последний файл, где мы работаем со свежеиспечённым модулем:

```
# run_swig.py
from my import integrate

if __name__ == '__main__':
    print(integrate(0.0, 2.0))

## это снова без меня
# from my import sum_array, new_intArray, intArray_setitem, intArray_getit
# n = 1000
# a = new_intArray(n)
# for i in range(n):
```

```
#     intArray_setitem(a, i, i + 1)
# l = [intArray_getitem(a, i) for i in range(n)]
# print(sum(l))
# print(sum_array(a, n))
```

Получается так:

```
$ time python3 run_swig.py
1.435370044777462
python3 run_swig.py  2.11s user 0.01s system 99% cpu 2.129 total
```

В эту сторону работает хорошо. Теперь сделаем то же самое, но с коллбеками. К сожалению, чтобы всё заработало, придётся писать на плюсах 🤢 Но сильно не пугайтесь, мы немношк.

Интерфейсник:

```
/* my.i */
%module(directors="1") my
%feature("director") Function;
%{
   #define SWIG_FILE_WITH_INIT
   #include "my.h"
%}
%include "my.h"
```

Крестовичок:

```
// my.cxx
#include "my.h"
#define N 100000000U
```

```
double integrate(double a, double b, Function *handler)
{
    const double h = (b - a) / N;
    const double h2 = h * 0.5;
    double s = 0.0;
    for (register unsigned int i = 0; i < N; i++)
        s += handler->handle(a + i * h + h2);
    return s * h;
}
```

Заголовочник:

```
// my.h
#ifndef MY_H
#define MY_H

struct Function
{
  virtual double handle(double x) = 0;
  virtual ~Function() {}
};

double integrate(double a, double b, Function *handler);

#endif
```

Довольно легко отделались - всего лишь структура с виртуальной функцией. Но, чтобы оно смогло-таки примотаться к питону, придётся в нём добавить класс-обёртку для этой самой структуры:

```
# function.py
from my import Function
```

```
class PythonFunction(Function):
    def __init__(self, f):
        Function.__init__(self)
        self._f = f

    def handle(self, x):
        return self._f(x)
```

Сетап:

```
# setup.py
from distutils.core import setup, Extension
my_module = Extension('_my', sources=['my_wrap.cxx', 'my.cxx'])
setup(
    name = 'my',
    ext_modules = [my_module],
    py_modules = ["my"],
)
```

И только теперь можно пользоваться:

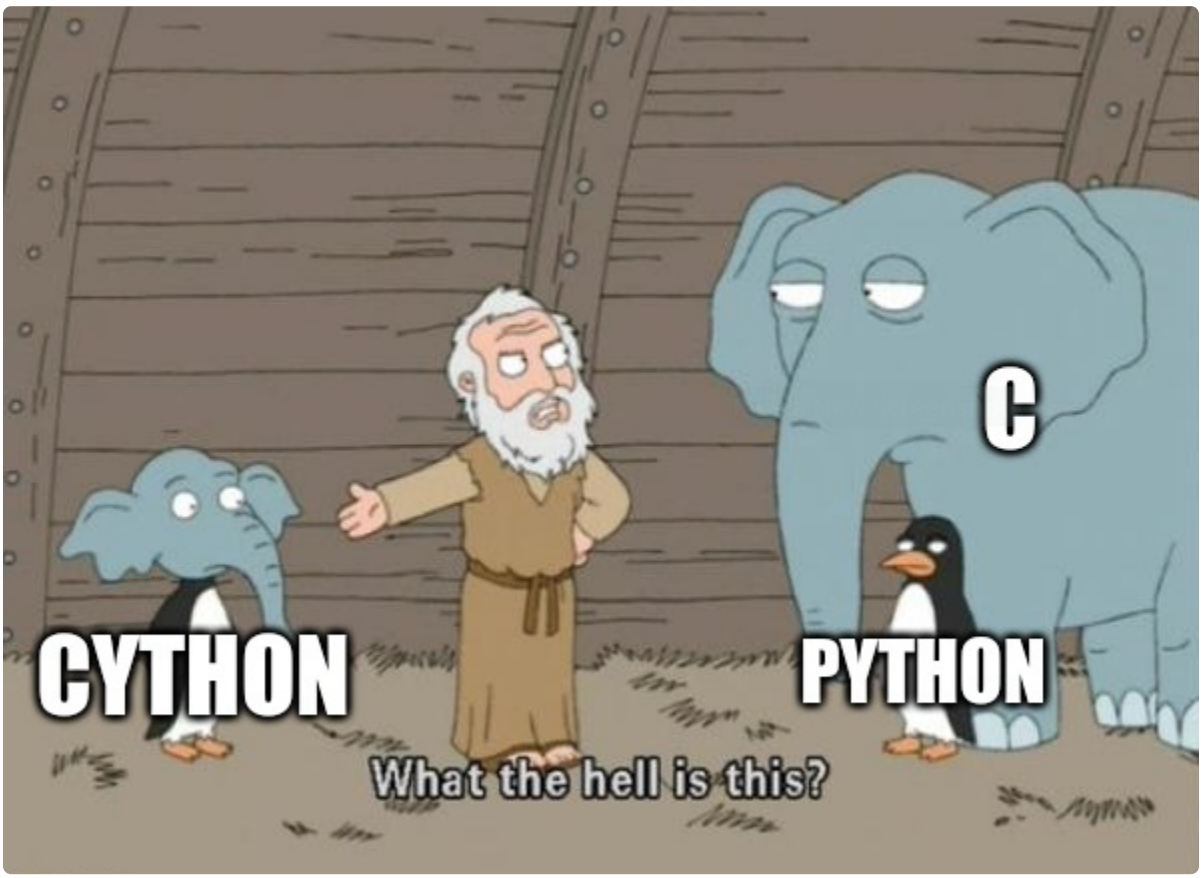
```
# run_swig.py
from my import integrate
from function import PythonFunction
from math import cos, exp

def f(x):
    return cos(x + x * x * x) if x <= 1.0 else exp(-x * x) - x * x + 2.0 *


if __name__ == '__main__':
```

```
print(integrate(0.0, 2.0, PythonFunction(f)))
```

Даже не буду приводить результаты запуска - вышло ещё медленее, чем в прошлый раз. Оно и понятно - у нас снаружи абстракция на абстракцию, а внутри абстракция, и гоняем мы это всё взад-назад литералли сто миллионов раз. Но лучше уметь, чем не уметь.

5. Cython

Объяснить что такое Cython проще всего так:



Поставим последнюю версию (на момент написания):

```
$ pip install Cython==3.0.0a11
```

Это нужно, потому что версия, которую пип ставит по умолчанию, не умеет работать с сишной математикой.

Напишем `.pyx` файлик с кодом. Обратите внимание на разницу в заголовках функций, ну и почитайте документацию на этот счёт:

```
# my.pyx
from cython.cimports.libc.math import cos, exp

cdef double f(double x):
    return cos(x + x * x * x) if x <= 1.0 else exp(-x * x) - x * x + 2.0 *


cpdef integrate(double a, double b):
    cdef int n = 100000000
    cdef double h = (b - a) / n
    cdef double h2 = h * 0.5
    cdef double s = 0.0
    for i in range(n):
        s += f(a + i * h + h2)
    return s * h

## это сами
# def sum_array(a):
#     cdef int s = 0
#     for i in a:
#         s += i
#     return s
```

setup.py :

```
# setup.py
from setuptools import setup
from Cython.Build import cythonize
```

```
setup(
    ext_modules=cythonize("my.pyx")
)
```

И запускающий питонский файл:

```
# run_cython.py
from my import integrate

if __name__ == '__main__':
    print(integrate(0, 2))
```

Собираем и запускаем:

```
$ python3 setup.py build_ext --inplace
$ time python3 run_cython.py
1.435370044777462
python3 run_cython.py  1.92s user 0.00s system 99% cpu 1.928 total
```

Замечательно. А теперь посмотрим, насколько плохо будет с питонским коллбеком. И тут у суетона есть ещё одно преимущество - не надо костылить их поддержку каким-то хитрым образом, а достаточно объявить функцию как:

```
cpdef integrate(double a, double b, f):
```

Тогда питонское станет таким:

```
# run_cython.py
from my import integrate
from math import cos, exp
```

```
def f(x):
     return cos(x + x * x * x) if x <= 1.0 else exp(-x * x) - x * x + 2.0


if __name__ == '__main__':
    print(integrate(0, 2, f))
```

A `setup.py` останется нетронутым. В итоге получим такое:

```
$ time python3 1.py
1.435370044777462
python3 1.py  27.76s user 0.00s system 99% cpu 27.764 total
```

Aaaaand we have a winner! Видно, что разработчики задумывались и об этом.

6. C расширения для Python

Максимальный хардкор. Хотя апишка написана очень хорошо и если уделить время документации, всё будет хорошо. Раскуриваемся и приступаем. Обратите внимание на реализацию подсчёта ссылок и парсинга переданных аргументов:

```
// c_module.c
#include <Python.h>
#include <math.h>
#define N 100000000U

static inline double f(double x)
{
    return x <= 1.0 ? cos(x + x * x * x) : exp(-x * x) - x * x + 2.0 * x;
}

static double integrate(double a, double b)
```

```
{
    const double h = (b - a) / N;
    const double h2 = h * 0.5;
    double s = 0.0;
    for (register size_t i = 0; i < N; i++)
        s += f(a + i * h + h2);
    return s * h;
}

static long int sum_array(size_t n, int a[n])
{
    register long int s = 0;
    for (register size_t i = 0; i < n; i++)
        s += a[i];
    return s;
}

static PyObject *py_integrate(PyObject *self, PyObject *args)
{
    double a, b;
    if (PyArg_ParseTuple(args, "dd", &a, &b))
    {
        double result = integrate(a, b);
        return Py_BuildValue("d", result);
    }
    Py_RETURN_NONE;
}

static PyObject *py_sum_sequence(PyObject *self, PyObject *args)
{
    if (PyTuple_Size(args) != 1)
    {
        PyErr_SetString(PyExc_TypeError, "wrong number of arguments");
```

```
    Py_RETURN_NONE;
}
PyObject *sequence = NULL;
if (!PyArg_ParseTuple(args, "O", &sequence))
{
    PyErr_SetString(PyExc_TypeError, "argument parsing failed");
    Py_RETURN_NONE;
}
sequence = PySequence_Fast(sequence, "argument must be iterable");
if (!sequence)
{
    PyErr_SetString(PyExc_TypeError, "all items must be integers");
    Py_RETURN_NONE;
}
size_t seq_len = PySequence_Fast_GET_SIZE(sequence);
int *array = malloc(seq_len * sizeof(int));
if (!array)
{
    Py_DECREF(sequence);
    return PyErr_NoMemory();
}
for (size_t i = 0; i < seq_len; i++)
{
    PyObject *item = PySequence_Fast_GET_ITEM(sequence, i);
    if (!item)
    {
        Py_DECREF(sequence);
        free(array);
        PyErr_SetString(PyExc_ValueError, "failed to get sequence item
        Py_RETURN_NONE;
    }
    PyObject *litem = PyNumber_Long(item);
    if (!litem)
```

```
            {
                Py_DECREF(sequence);
                free(array);
                PyErr_SetString(PyExc_TypeError, "all items must be integers")
                Py_RETURN_NONE;
            }
            array[i] = (int)PyLong_AsLong(litem);
            Py_DECREF(litem);
        }
        Py_DECREF(sequence);
        long int result = sum_array(seq_len, array);
        free(array);
        return Py_BuildValue("l", result);
}

static PyMethodDef mymodule_methods[] = {
    {"integrate", py_integrate, METH_VARARGS, "Function for numerical inte
    {"sum_sequence", py_sum_sequence, METH_VARARGS, "Function for counting
    {NULL, NULL, 0, NULL} // sentinel
};

static struct PyModuleDef mymodule = {
  PyModuleDef_HEAD_INIT,
  "my_module",
  "Example C module",
  -1, // no additional memory reqiurements
  mymodule_methods
};

PyMODINIT_FUNC PyInit_my_module(void)
{
    return PyModule_Create(&mymodule);
}
```

Уже привычный нам сетап:

```
# setup.py
from distutils.core import setup, Extension

setup(
    name="my_module",
    ext_modules=[Extension("my_module", ["c_module.c"])]
)
```

И запускающий питонский модулёк:

```
# run_c_ext.py
from my_module import integrate

if __name__ == '__main__':
    print(integrate(0.0, 2.0))
```

Собираем и наслаждаемся:

```
$ python3 setup.py build_ext --inplace
$ time python3 run_c_ext.py
1.435370044777462
python3 run_c_ext.py  1.93s user 0.01s system 99% cpu 1.934 total
```

Не знаю как вы, а я доволен. Не забудьте потестить функцию для суммирования последовательности.

Ну и традиционно — коллбеки. В сишном привожу, только то, что изменилось:

```
// c_module.c
```

```c
static PyObject *my_callback = NULL;

static double integrate(double a, double b)
{
    const double h = (b - a) / N;
    const double h2 = h * 0.5;
    double s = 0.0;
    PyObject *arglist, *result;
    for (register size_t i = 0; i < N; i++)
    {
        arglist = Py_BuildValue("(d)", a + i * h + h2);
        result = PyObject_CallObject(my_callback, arglist);
        Py_DECREF(arglist);
        s += PyFloat_AsDouble(result);
        Py_DECREF(result);
    }
    return s * h;
}

static PyObject *py_integrate(PyObject *self, PyObject *args)
{
    PyObject *temp;
    double a, b;
    if (PyArg_ParseTuple(args, "ddO:set_callback", &a, &b, &temp))
    {
        if (!PyCallable_Check(temp))
        {
            PyErr_SetString(PyExc_TypeError, "parameter must be callable")
            Py_RETURN_NONE;
        }
        Py_XINCREF(temp);
        Py_XDECREF(my_callback);
        my_callback = temp;
```

```
        double result = integrate(a, b);
        return Py_BuildValue("d", result);
    }
    Py_RETURN_NONE;
}
```

Обратите внимание на работу с коллаблом. Запускать это безобразие будем так:

```
# run_c_ext.py
from my_module import integrate
from math import cos, exp

def f(x):
    return cos(x + x * x * x) if x <= 1.0 else exp(-x * x) - x * x + 2.0 *


if __name__ == '__main__':
    print(integrate(0.0, 2.0, f))
```

Барабанная дробь…

```
$ time python3 run_c_ext.py
1.435370044777462
python3 1.py  31.08s user 0.01s system 99% cpu 31.097 total
```

Оркестр, туш! Почётное второе место, лучше выступил только сайтон.

Вот в принципе и все способы, которые я хотел показать. Наверняка на второй странице гугла будут ещё какие-то, но это уже совсем другая история…

Задания для самостоятельного выполнения

Вычислить $N(A)$, где: $A_{n\times m} = (a_{ij}) = \big(f(i) + g(j)\big); i = 1, \ldots, n; j = 1, \ldots, m.$

Задание выполнить 4 способами: чистый Python, чистый C и 2 способа по варианту. Протестировать на больших размерах матрицы, сравнить время выполнения.

Варианты заданий

1. $N(A) = \max\limits_j \sum_{i=1}^{n} |a_{ij}|,$

   $f(x) = \sin x - \cos x^{1.5},$

   $g(x) = x^2 - \frac{1}{\mathrm{tg}(x/99+0.01)}.$

   Способы:

   - PyPy,
   - C-расширение.

   Контрольное значение: $N(A_{10\times 10}) = 911.309862$

2. $N(A) = \sqrt{\max\limits_i\ |a_{ii}|},$

   $f(x) = e^{\sin x} - x,$

   $g(x) = \ln(1 + \sqrt{x}) - \cos x.$

   Способы:

   - Nuitka,
   - So.

   Контрольное значение: $N(A_{10\times 10}) = 2.674782$

3. $N(A) = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} a_{ij}^2},$

   $f(x) = \sin(1.3^x - 1.2),$

   $g(x) = \sin(x) - \frac{1}{1+x^2}.$

   Способы:

   - Numba,

- Cython.

Контрольное значение: $N(A_{10\times 10}) = 9.026581$

4. $N(A) = \max\limits_i \sum_{j=1}^{m} |a_{ij}|$,

   $f(x) = \cos(1.7^{x+1} - 2.7)$,

   $g(x) = 2^x + x^2 - 2$.

   Способы:

   - Numpy,
   - SWIG.

   Контрольное значение: $N(A_{10\times 10}) = 2420.820042$

5. $N(A) = \sqrt{nm}\max\limits_{i,j} |a_{ij}|$,

   $f(x) = \frac{5x^2-8}{x^3+1}$,

   $g(x) = \sqrt{x+1} - \sqrt{x} - \frac{1}{2}$.

   Способы:

   - PyPy,
   - So.

   Контрольное значение: $N(A_{10\times 10}) = 18.456529$

6. $N(A) = \left(\sum_{i=1}^{n}\sum_{j=1}^{m} a_{ij}^p\right)^{\frac{1}{p}}$,

   $f(x) = \frac{x^2}{5} - \sin\sqrt[3]{x} - 5$,

   $g(x) = \ln(x+1) - \sqrt{\frac{5}{(x+1)^2}}$.

   Способы:

   - Nuitka,
   - Cython.

   Контрольное значение: $N(A_{10\times 10}) = 38.838965, p = 3$

7. $N(A) = \sqrt{|\operatorname{Tr}(A^T A)|}$,

   $f(x) = x - 2\sin x - \cos 3x$,

$g(x) = \ln(x+2) - \sqrt{x}.$

Способы:

- Numba,
- SWIG.

Контрольное значение: $N(A_{10\times10}) = 59.918901$

8. $N(A) = \frac{\min\limits_{ij}|a_{ij}|+1}{\max\limits_{ij}|a_{ij}|+1} - 1,$

$f(x) = e^{-2x} - x,$

$g(x) = \sqrt{x} + \frac{1}{e^x}.$

Способы:

- Numpy,
- C-расширение.

Контрольное значение: $N(A_{10\times10}) = -0.895911$

9. $N(A) = \sqrt{\min\limits_{ij}|b_{ij}|}$, $B = A^{\mathrm{adT}}A$, где A^{adT} — A, транспонированная относительно побочной диагонали;

$f(x) = \cos\sqrt{x} - x,$

$g(x) = \frac{\cos x}{\sqrt{1+x}}.$

Способы:

- PyPy,
- Cython.

Контрольное значение: $N(A_{10\times10}) = 14.308647$

10. $N(A) = \sqrt{\sum_{i=1}^{n}|a_{ii}| + |a_{i,n-i}|},$

$f(x) = e^{-x^2} - \sin x,$

$g(x) = \frac{\sqrt{x}}{x^2+1} - 1.$

Способы:

- Nuitka,

- SWIG.

Контрольное значение: $N(A_{10\times10}) = 4.415591$

11. $N(A) = \frac{\sqrt{\sum_{i=1}^{n}|a_{i1}|+|a_{im}|+\sum_{j=2}^{m-1}|a_{1j}|+|a_{nj}|}}{n+m}$,

    $f(x) = e^{-x^2} - \cos x^2 - 2$,

    $g(x) = \frac{\ln x+1}{2}$.

    Способы:

    - Numba,
    - So.

    Контрольное значение: $N(A_{10\times10}) = 0.362621$

12. $N(A) = \frac{1}{nm}\sum_{i=1}^{n}\sum_{j=1}^{m}a_{ij}$

    $f(x) = \sqrt{x+2} - \sqrt{x} - \frac{3}{11}$,

    $g(x) = \frac{3x^2-11}{-x^3-2}$.

    Способы:

    - Numpy,
    - Cython.

    Контрольное значение: $N(A_{10\times10}) = 0.079255$

← ПРЕДЫДУЩАЯ

Python. Лабораторная работа №14
