# Accelerated Python

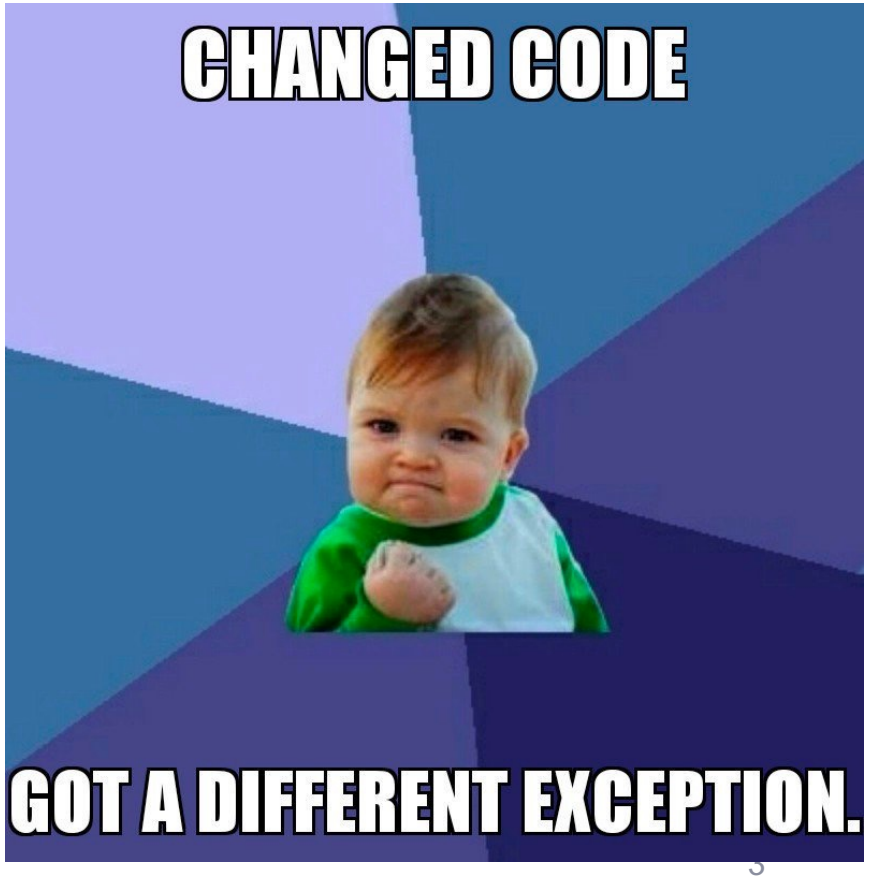
# Python

Быстро пишется, долго работает

# Python

Быстро пишется, долго работает


И не сложно учится



# C/C++

Долго пишется, быстро работает

# C/C++

Долго пишется, быстро работает

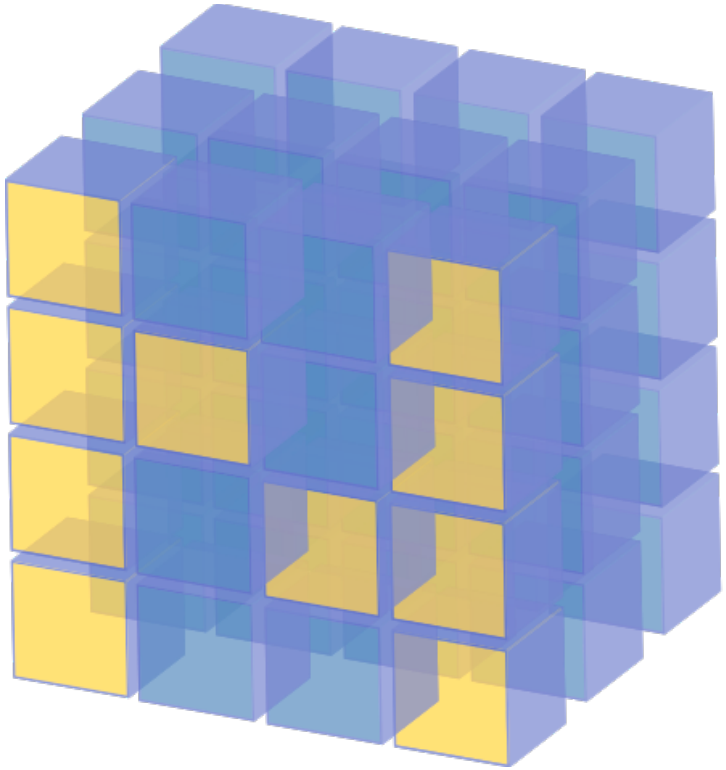

И...

# Тогда как объединить?

Скорость плюсов

Удобство питона

# Использовать numpy ☺



NumPy

# При подсчёте фичей

```
%%timeit
stat1 = 0
stat2 = 0
stat3 = 0
for i, elem in enumerate(a_list):
    if elem % 17 == 0:
        stat1 += i
    if elem % 19 == 0:
        stat2 += i
    if elem % 28 == 0:
        stat3 += i
```

executed in 12.0s, finished 22:11:47 2019-04-12

```
14.7 ms ± 272 µs per loop (mean ± std.
```

# При подсчёте фичей

```
%%timeit
stat1 = 0
stat2 = 0
stat3 = 0
for i, elem in enumerate(a_list):
    if elem % 17 == 0:
        stat1 += i
    if elem % 19 == 0:
        stat2 += i
    if elem % 28 == 0:
        stat3 += i
```

executed in 12.0s, finished 22:11:47 2019-04-12

```
14.7 ms ± 272 µs per loop (mean ± std.
```

```
%%timeit
np.where(a_array % 17 == 0)[0].sum()
np.where(a_array % 19 == 0)[0].sum()
np.where(a_array % 28 == 0)[0].sum()
```

executed in 2.10s, finished 22:11:54 2019-04-12

```
2.56 ms ± 47.9 µs per loop (mean ± std. dev.
```

# Когда нужны циклы

```
%%timeit
np.random.seed(42)
values = []
for _ in range(10000):
    values.append((np.random.randint(0, 2, size=10000) * 2 - 1).mean())
```

executed in 6.01s, finished 22:22:42 2019-04-12

```
750 ms ± 13.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

# Когда нужны циклы

```
%%timeit
np.random.seed(42)
values = []
for _ in range(10000):
    values.append((np.random.randint(0, 2, size=10000) * 2 - 1).mean())
```

executed in 6.01s, finished 22:22:42 2019-04-12

```
750 ms ± 13.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
np.random.seed(42)
values = (np.random.randint(0, 2, size=(10000, 10000)) * 2 - 1).mean(axis=1)
```

executed in 8.49s, finished 22:20:06 2019-04-12

```
1.06 s ± 15 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

# Когда нужны циклы

```
%%timeit
np.random.seed(42)
values = []
for _ in range(10):
    values.extend((np.random.randint(0, 2, size=(1000, 10000)) * 2 - 1).mean(axis=1))
```

executed in 6.30s, finished 22:22:17 2019-04-12

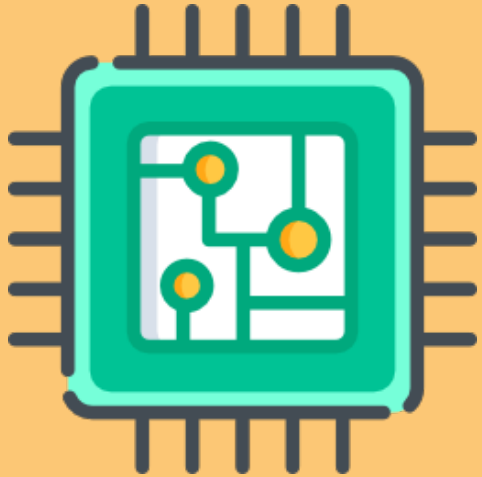
```
779 ms ± 14.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
np.random.seed(42)
values = []
for _ in range(100):
    values.extend((np.random.randint(0, 2, size=(100, 10000)) * 2 - 1).mean(axis=1))
```

executed in 5.11s, finished 22:22:22 2019-04-12

```
627 ms ± 10.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

# 2. C++ в Python



# Python.h

Обычно что это такое знают только те, кто словил ошибки с этим файлом

# Python.h

Обычно что это такое знают только те, кто словил ошибки с этим файлом

```c
#include "Python.h"

static PyObject *
keywdarg_parrot(PyObject *self, PyObject *args, PyObject *keywds)
{
    int voltage;
    char *state = "a stiff";
    char *action = "voom";
    char *type = "Norwegian Blue";

    static char *kwlist[] = {"voltage", "state", "action", "type", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, keywds, "i|sss", kwlist,
                                     &voltage, &state, &action, &type))
        return NULL;

    printf("-- This parrot wouldn't %s if you put %i Volts through it.\n",
           action, voltage);
    printf("-- Lovely plumage, the %s -- It's %s!\n", type, state);

    Py_INCREF(Py_None);

    return Py_None;
}

static PyMethodDef keywdarg_methods[] = {
    /* The cast of the function is necessary since PyCFunction values
     * only take two PyObject* parameters, and keywdarg_parrot() takes
     * three.
     */
    {"parrot", (PyCFunction)keywdarg_parrot, METH_VARARGS | METH_KEYWORDS,
     "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL}   /* sentinel */
};
```

# Python.h

Писать такой код – сомнительное удовольствие, в этом случае спасёт...

# В чём суть обёрток

1. Пишем код на удобном языке
2. Переводим код в **C/C++**
3. Компилируем
4. ...
5. PROFIT

# Boost.Python

Ссылка на проект

Following C/C++ tradition, let's start with the "hello, world". A C++ Function:

```
char const* greet()
{
   return "hello, world";
}
```

can be exposed to Python by writing a Boost.Python wrapper:

```
#include <boost/python.hpp>

BOOST_PYTHON_MODULE(hello_ext)
{
    using namespace boost::python;
    def("greet", greet);
}
```

That's it. We're done. We can now build this as a shared library. The resulting DLL is now visible to Python. Here's a sample Python session:

```
>>> import hello_ext
>>> print hello_ext.greet()
hello, world
```

# Pybind11

Ссылка на проект

```cpp
#include <pybind11/pybind11.h>

int add(int i, int j) {
    return i + j;
}

PYBIND11_MODULE(example, m) {
    m.doc() = "pybind11 example plugin"; // optional module docstring

    m.def("add", &add, "A function which adds two numbers");
}
```

# А в чём разница?

# А в чём разница?


Да ни в чём! Просто у Boost-а зависимость от boost, а ещё Pybind11 более лаконично реализован

# CMake

CMakeLists.txt

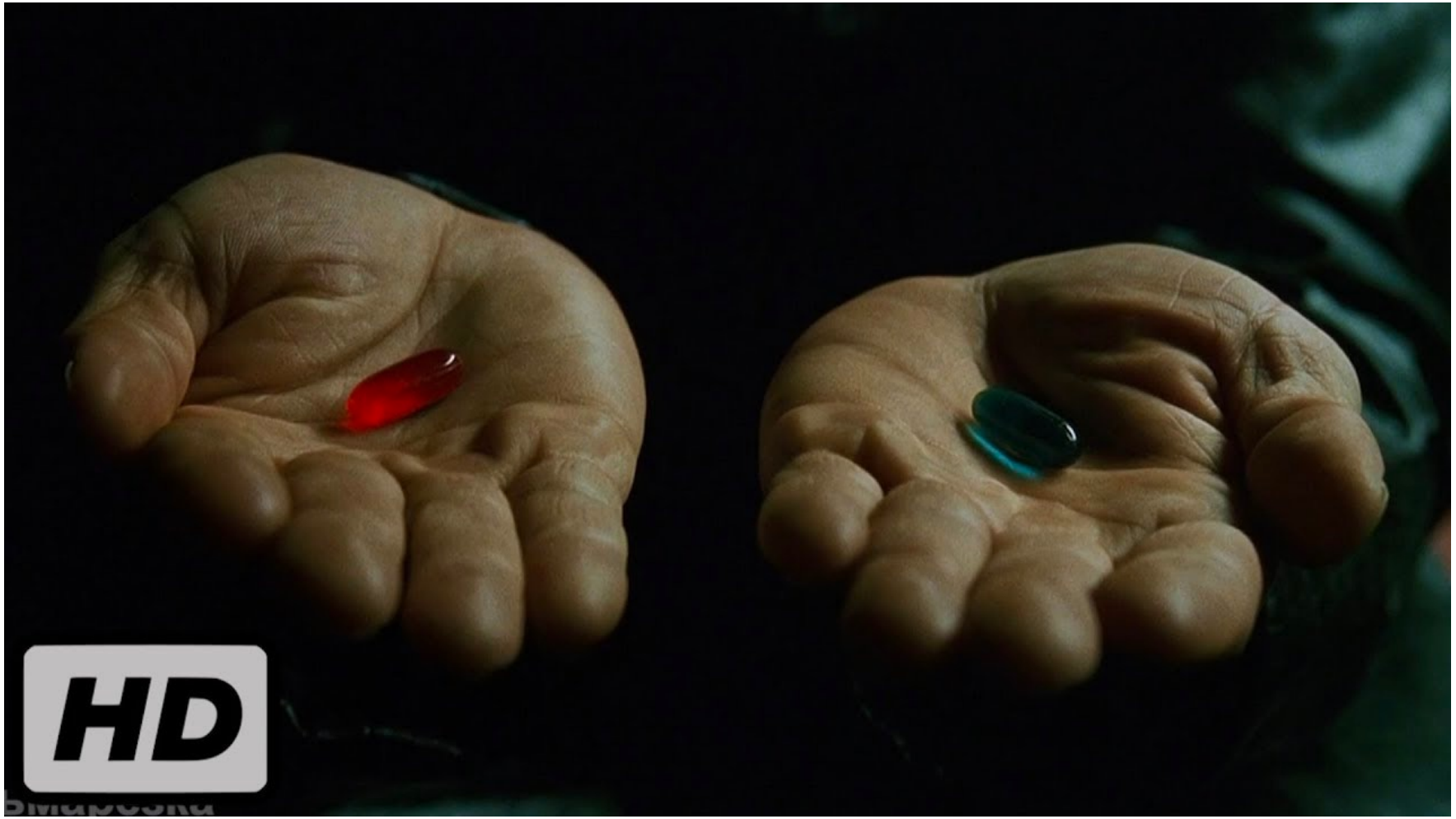
```
cmake_minimum_required(VERSION 3.5)
project(taxi_ml_cxx)

set(CMAKE_CXX_STANDARD 11)

if(NOT DEFINED CMAKE_LIBRARY_OUTPUT_DIRECTORY)
    set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR})
endif()

set(THIRD_PARTY_DIRS ${PROJECT_SOURCE_DIR}/third_party)

add_compile_options ("-fexceptions" "-g")
# build type specific
if (CMAKE_BUILD_TYPE MATCHES "Debug" OR CMAKE_BUILD_TYPE MATCHES "Test")
    add_compile_options ("-O0")
    add_definitions ("-DCONSOLE_LOGGER -DTEST_MOCK")
else ()
    add_compile_options ("-O3")
    add_definitions ("-DNDEBUG")
endif ()

find_package(PythonInterp 2.7 REQUIRED)
find_package(PythonLibs 2.7 REQUIRED)

set(CMAKE_POSITION_INDEPENDENT_CODE ON)
```

# А как это сделать частью пакета?

```
class CMakeBuild(build_ext):
    def run(self):
        try:
            subprocess.check_output(['cmake', '--version'])
        except OSError:
            raise RuntimeError(
                "CMake must be installed to build the following extensions: " +
                ", ".join(e.name for e in self.extensions))

        for ext in self.extensions:
            self.build_extension(ext)

    def build_extension(self, ext):
        ext_dir = os.path.abspath(
            os.path.dirname(self.get_ext_fullpath(ext.name)))
        cfg = 'Debug' if self.debug else 'Release'
        cmake_args = ['-DCMAKE_LIBRARY_OUTPUT_DIRECTORY=' + ext_dir,
                      '-DPYTHON_EXECUTABLE=' + sys.executable,
                      '-DCMAKE_BUILD_TYPE=' + cfg]
        build_args = ['--config', cfg, '--', '-j2']
        env = os.environ.copy()
        env['CXXFLAGS'] = '{} -DVERSION_INFO=\\"{}\\"'.format(
            env.get('CXXFLAGS', ''),
            self.distribution.get_version()
        )

        if not os.path.exists(self.build_temp):
            os.makedirs(self.build_temp)
        subprocess.check_call(['cmake', ext.source_dir] + cmake_args,
                              cwd=self.build_temp, env=env)
        subprocess.check_call(['cmake', '--build', '.'] + build_args,
                              cwd=self.build_temp)

        for dir_path, _, _ in os.walk(os.path.join(ext_dir, ext.name)):
            path = os.path.join(dir_path, '__init__.py')
            if not os.path.exists(path):
                open(path, 'w').close()
        print()  # Add an empty line for cleaner output
```

```
common_setup_kwargs = dict(
    name='yandex_taxi_ml_cxx',
    version=taxi_ml_cxx.__version__,
    description='C++ wrappers',
    packages=find_packages(include=('taxi_ml_cxx*',))
)

if os.environ.get('TAXI_ML_CXX_USE_BUILT'):
    setup(
        package_data={'': ['*.so']},
        **common_setup_kwargs
    )
else:
    setup(
        ext_modules=[CMakeExtension('taxi_ml_cxx')],
        cmdclass=dict(build_ext=CMakeBuild),
        **common_setup_kwargs
    )
```

# Альтернативный путь



# Cython

# В чём суть cython

1. Пишем код на языке, похожем на питон
2. Он переводится на C
3. Компилируем
4. ...
5. PROFIT

# Cython

[Ссылка на проект]

```python
# python_functions.py
# ----------------------
import math

def f(x):
    return math.exp(-(x ** 2))

def integrate_f(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f(a + i * dx)
    return s * dx

```

```python
# cython_functions.pyx
# ----------------------
from libc cimport math

cdef double f(double x):
    return math.exp(-(x ** 2))

def integrate_f(double a, double b, int N):
    cdef double s = 0
    cdef double dx = (b - a) / N
    cdef int i
    for i in range(N):
        s += f(a + i * dx)
    return s * dx

```

# Cython: до

```
import numpy as np

cimport numpy as np
cimport cython


@cython.boundscheck(False)
@cython.wraparound(False)
def memory_efficient_inner1d(
        cython.floating[:, :] fst_arr,
        cython.integral[:] fst_indices,
        cython.floating[:, :] snd_arr,
        cython.integral[:] snd_indices
):
    cdef int size = fst_indices.shape[0]
    cdef int width = fst_arr.shape[1]
    cdef np.ndarray[np.float64_t] result = np.zeros(size, dtype=np.float64)
    cdef int i, j, fst_index, snd_index
    cdef double sum
    for i in range(size):
        sum = 0
        fst_index = fst_indices[i]
        snd_index = snd_indices[i]
        for j in range(width):
            sum += fst_arr[fst_index, j] * snd_arr[snd_index, j]
        result[i] = sum

    return result
```

# Cython: после

```
28350   if (likely(PyLong_CheckExact(b))) {
28351     #if CYTHON_USE_PYLONG_INTERNALS
28352     const digit* digits = ((PyLongObject*)b)->ob_digit;
28353     const Py_ssize_t size = Py_SIZE(b);
28354     if (likely(__Pyx_sst_abs(size) <= 1)) {
28355         ival = likely(size) ? digits[0] : 0;
28356         if (size == -1) ival = -ival;
28357         return ival;
28358     } else {
28359       switch (size) {
28360          case 2:
28361            if (8 * sizeof(Py_ssize_t) > 2 * PyLong_SHIFT) {
28362              return (Py_ssize_t) (((((size_t)digits[1]) << PyLong_SHIFT) | (size_t)digits[0]));
28363              }
28364              break;
28365          case -2:
28366            if (8 * sizeof(Py_ssize_t) > 2 * PyLong_SHIFT) {
28367              return -(Py_ssize_t) (((((size_t)digits[1]) << PyLong_SHIFT) | (size_t)digits[0]));
28368              }
28369              break;
28370          case 3:
28371            if (8 * sizeof(Py_ssize_t) > 3 * PyLong_SHIFT) {
28372              return (Py_ssize_t) (((((((size_t)digits[2]) << PyLong_SHIFT) | (size_t)digits[1]) << PyLong_SHIFT) | (si
28373              }
28374              break;
28375          case -3:
28376            if (8 * sizeof(Py_ssize_t) > 3 * PyLong_SHIFT) {
28377              return -(Py_ssize_t) (((((((size_t)digits[2]) << PyLong_SHIFT) | (size_t)digits[1]) << PyLong_SHIFT) | (s
28378              }
28379              break;
28380          case 4:
28381            if (8 * sizeof(Py_ssize_t) > 4 * PyLong_SHIFT) {
28382              return (Py_ssize_t) (((((((((size_t)digits[3]) << PyLong_SHIFT) | (size_t)digits[2]) << PyLong_SHIFT) | (
28383              }
28384              break;
28385          case -4:
28386            if (8 * sizeof(Py_ssize_t) > 4 * PyLong_SHIFT) {
28387              return -(Py_ssize_t) (((((((((size_t)digits[3]) << PyLong_SHIFT) | (size_t)digits[2]) << PyLong_SHIFT) |
28388              }
28389              break;
28390       }
28391     }
28392     #endif
28393     return PyLong_AsSsize_t(b);
28394   }
28395   x = PyNumber_Index(b);
28396   if (!x) return -1;
28397   ival = PyInt_AsSsize_t(x);
28398   Py_DECREF(x);
28399   return ival;
28400 }
28401 static CYTHON_INLINE PyObject * __Pyx_PyBool_FromLong(long b) {
28402   return b ? __Pyx_NewRef(Py_True) : __Pyx_NewRef(Py_False);
28403 }
28404 static CYTHON_INLINE PyObject * __Pyx_PyInt_FromSize_t(size_t ival) {
28405     return PyInt_FromSize_t(ival);
28406 }
28407
```

```
28380          case 4:
28381            if (8 * sizeof(Py_ssize_
28382              return (Py_ssize_t) ((
28383            }
28384            break;
28385          case -4:
28386            if (8 * sizeof(Py_ssize_
28387              return -(Py_ssize_t) (
28388            }
28389            break;
28390       }
28391     }
28392     #endif
28393     return PyLong_AsSsize_t(b);
28394   }
28395   x = PyNumber_Index(b);
28396   if (!x) return -1;
28397   ival = PyInt_AsSsize_t(x);
28398   Py_DECREF(x);
28399   return ival;
28400 }
28401 static CYTHON_INLINE PyObject * __F
28402   return b ? __Pyx_NewRef(Py_True)
28403 }
28404 static CYTHON_INLINE PyObject * __F
28405     return PyInt_FromSize_t(ival);
28406 }
28407
```

```
28404
28405
28406
28407
```

# А как это сделать частью пакета?

```
#!/usr/bin/env python
from __future__ import print_function

try:
    from Cython.Build import build_ext
    import numpy as np
    np_include_dirs = np.get_include()
except ImportError:
    from setuptools.command.build_ext import build_ext
    np_include_dirs = ''

import platform

from distutils.core import setup
from distutils.extension import Extension
from setuptools import find_packages

included_packages = ('pyartm*',)

is_windows = platform.system() == 'Windows'
with open('requirements.txt', 'r') as req_file:
    requirements = [
        line.strip()
        for line in req_file
        if line.strip() and (is_windows or 'mysqlclient' not in line)
    ]

setup(
    setup_requires=requirements[:2],
    ext_modules=[Extension(
        "package.calculations.inner_product.cy_impl",
        sources=["package/calculations/inner_product/cy_impl.pyx"],
        include_dirs=[np_include_dirs],
    )],
    cmdclass={'build_ext': build_ext},
    install_requires=requirements,
    packages=find_packages(include=included_packages)
)

```

# Альтернативный путь 2

# Numba

# В чём суть numba

1. Код питона компилируется в байт код (*.pyc файлы)
2. Берём байт код функции
3. Перегоняем его в машинный код и заменяем реализацию
4. ...
5. PROFIT

# Numba

- Ссылка на проект

```python
from numba import jit
import numpy as np
import time

x = np.arange(100).reshape(10, 10)

@jit(nopython=True)
def go_fast(a): # Function is compiled and runs in machine code
    trace = 0
    for i in range(a.shape[0]):
        trace += np.tanh(a[i, i])
    return a + trace

# DO NOT REPORT THIS... COMPILATION TIME IS INCLUDED IN THE EXECUTION TIME!
start = time.time()
go_fast(x)
end = time.time()
print("Elapsed (with compilation) = %s" % (end - start))

# NOW THE FUNCTION IS COMPILED, RE-TIME IT EXECUTING FROM CACHE
start = time.time()
go_fast(x)
end = time.time()
print("Elapsed (after compilation) = %s" % (end - start))
```

This, for example prints:

```
Elapsed (with compilation) = 0.33030009269714355
Elapsed (after compilation) = 6.67572021484375e-06
```

# В итоге

1. Почти всегда можно обойтись numpy
2. Если надо одну простую функцию ускорить, и она работает с числами или массивами, то Numba в помощь
3. Если пишете библиотеку с большими встройками C/C++, то лучше использовать Cython
4. Если хотите писать на плюсах, ~~то пишите, питон тут причём~~, используя мощь проектов на новых стандартах, то это проще на pybind11 или Boost.Python