EQUIP YOUR PERFORMANCE TOOLBOX

CYTHON V.S. PYBIND11

Gavin Chan

Quant Developer

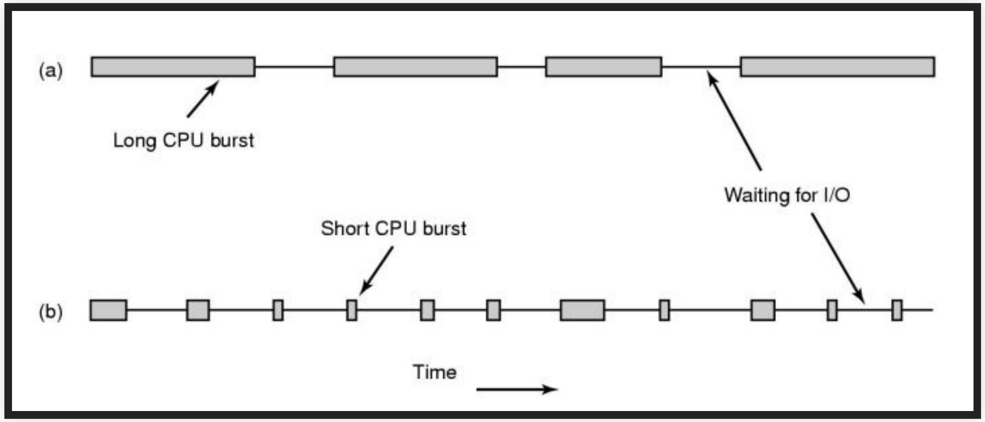
AXA Investment Managers Chorus Ltd

BACKGROUND

- AXA IM Chorus Ltd is a quantitative asset management fund.
- IT and quants work together to generate alpha signals, optimize and allocate the portfolio, and control the portfolio risk.

OBJECTIVE

- Stability and performance are equally important.
- The learning curve of Python is flat, but its performance is always a concern.
- Target on CPU bound problem.



SIMPLE EXAMPLE

- Consider a portfolio containing a number of instruments (e.g. 1000 instruments)
- Each instrument has its own category, e.g. currency
- Compute how much risk is exposed for each category, e.g. in every second

PROBLEM

```
# Labels: A series of labels
#  ╔═════════╤═════════╤═════════╤═════════╗
#  ║   USD   │   EUR   │   EUR   │   USD   ║
#  ╚═════════╧═════════╧═════════╧═════════╝
# Exposures: A series of portfolio weights,
#                     between -1 and 1
#  ╔══════════╤═════════╤═════════╤═════════╗
#  ║   -0.3   │   0.2   │   0.8   │   -0.7  ║
#  ╚══════════╧═════════╧═════════╧═════════╝
#
# => USD
#  ╔══════════╤═════╤═════╤═════════╗
#  ║   -0.3   │  0  │  0  │   -0.7  ║    => -1.0
#  ╚══════════╧═════╧═════╧═════════╝
#
# => EUR
#  ╔═════╤═════════╤═════════╤═════╗
#  ║  0  │   0.2   │   0.8   │  0  ║  => 1.0
#  ╚═════╧═════════╧═════════╧═════╝
```

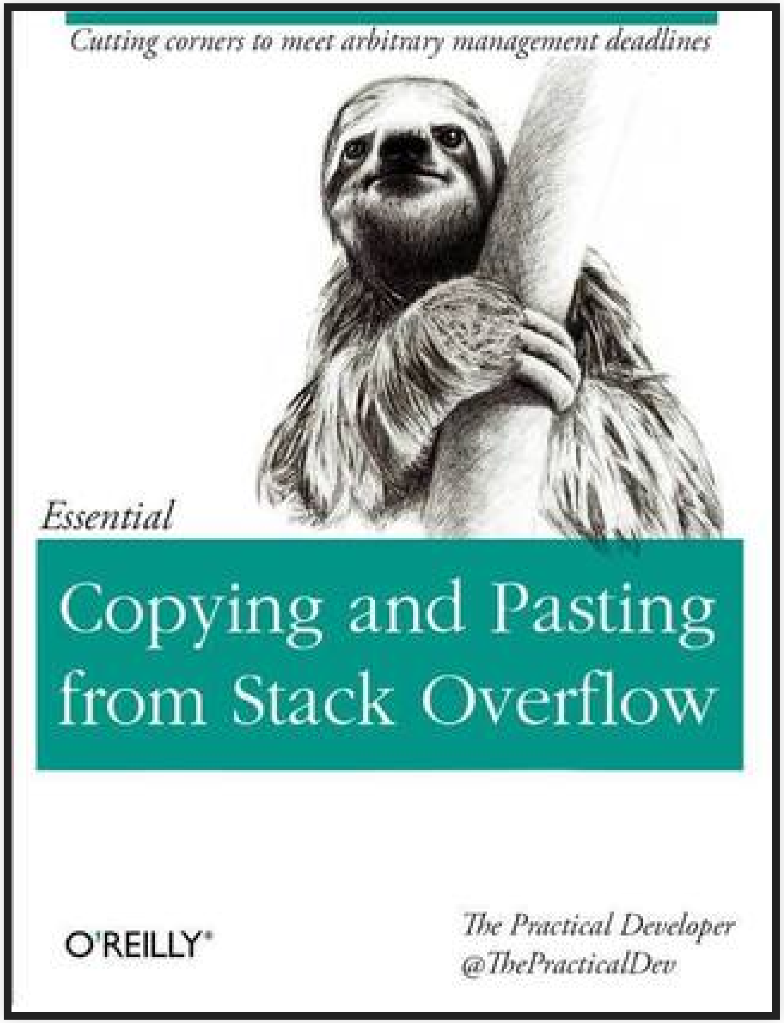
FOR LOOP

```python
# => USD
#  ┌────┬────┬────┬────┐   ┌──────┬─────┬─────┬──────┐
#  │ 1  │ 0  │ 0  │ 1  │ * │ -0.3 │ 0.2 │ 0.8 │ -0.7 │
#  └────┴────┴────┴────┘   └──────┴─────┴─────┴──────┘
#
# =>  ┌──────┬───┬───┬──────┐
#     │ -0.3 │ 0 │ 0 │ -0.7 │  => -1.0
#     └──────┴───┴───┴──────┘
#
# => EUR
#  ┌───┬─────┬─────┬───┐
#  │ 0 │ 0.2 │ 0.8 │ 0 │ => 1.0
#  └───┴─────┴─────┴───┘

def for_loop():
    label_exposures = []
    for label in range(LABEL_COUNT):
        label_exposure = ((labels == label) * exposures).sum()
        label_exposures.append(label_exposure)

    return label_exposures
```

LET'S GOOGLE IT



WARM-UP

```python
def list_comprehension():
    return [((labels == label) * exposures).sum()
            for label in range(LABEL_COUNT)]

def numpy_dot_product():
    return [(labels == label).dot(exposures)
            for label in range(LABEL_COUNT)]

def pandas_groupby():
    instmt_label_exposures = pd.DataFrame(
        np.array([labels, exposures], copy=False),
        index=['label', 'exposure'],
        columns=pd.Index(instruments, copy=False)).T
    return instmt_label_exposures.groupby('label')['exposure'].sum().values
```

RESULT

```
# Benchmark:
# Number of instruments: 1000
# Number of labels: 500
#
#
#           for loop     list comprehension
# -------   -----------  --------------------
# Average   430.54 msec  434.49 msec
# Std.Dev   8.73 msec    6.12 msec
# Minimum   422.90 msec  430.09 msec
#
#           numpy dot product    pandas groupby
# -------   ------------------   ----------------
# Average   225.44 msec          186.66 msec
# Std.Dev   11.88 msec           2.91 msec
# Minimum   211.54 msec          181.34 msec
```

COMPILED LANGUAGE

- Python compiles the source to bytecode on the runtime and the bytecode is run on a Python virtual machine.
- Python runtime is much slower than compiled languages (C / C++) and those using JIT (Java / C#).
- JIT solutions, e.g. PyPy and Numba, will not be covered.

WHY CYTHON AND PYBIND11?

- Cython is a static Python compiler, using C type declarations with Python-like syntax. Easy for Python developers to migrate to Cython.
- pybind11 is a modern library to create Python bindings to C++ code. Compared with the CPython API and Boost.Python, pybind11 is a lightweight header-only library.

BUILD

	Cython	**pybind11**
Platform	Windows, Linux, MacOS	Windows (VS2015+), Linux, MacOS
setup.py	Easy	Examples provided
Command	cython / cythonize	C++ compiler, e.g. gcc / clang
IPython	Yes	Yes with `ipybind`

FUNCTION

- Both Cython and pybind11 support well on binding the Python functions, including type casting and docstring.
- Cython can compile the native Python function (def), and also the C type syntax / function (cdef, cpdef).

FUNCTION

	Cython	pybind11
docstring	Yes	Yes
Overload	No	Yes
`*args`, `**kwargs`	Only in `def`	`py::args`, `py::kwargs`
Keyword arguments	No	No

CYTHON

```
cimport numpy as np

cpdef double cython_function(
    int a,
    double b,
    np.ndarray c):
    """Cython function."""
    cdef:
        double r = a + b

    for i in c:
        r += i

    return r
```

```
cython_function(1, 2.0, np.array([1, 2, 3]))  ## => 9.0
```

```
print(cython_function.__doc__)
### Output:
### Cython function.
```

PYBIND11

```
nclude <pybind11/pybind11.h>
#include <pybind11/numpy.h>
namespace py = pybind11;

double pybind11_function(int a) { return a; }
double pybind11_function(int a, double b, py::array_t<double> c, py::kwargs
    double r = a + b;
    py::buffer_info c_buf = c.request();
    double* c_ptr = (double *)c_buf.ptr;
    for (auto i = 0; i < c_buf.shape[0]; i++) r += c_ptr[i];
    py::print(kwargs["d"]);
    return r;
}

PYBIND11_MODULE(example, m) {
    m.def("pybind11_function",
          py::overload_cast<int>(&pybind11_function),
          "Pybind11 simple function")
     .def("pybind11_function",
          py::overload_cast<int, double, py::array_t<double>, py::kwargs>(
            &pybind11_function),
          "Pybind11 complex function.");
}
```

PYBIND11

```
pybind11_function(1)
# Output: 1.0
pybind11_function(1, 2.0, np.array([1, 2, 3]))
# Output:
# hello!
# 9.0
```

```
print(pybind11_function.__doc__)
# Output:
# pybind11_function(arg0: int, arg1: float, arg2: numpy.ndarray[float64]) ->
#
# pybind11 function.
```

TYPES

	Cython	pybind11
C types	All	All
Mixed types	Fused types	Function overload
C++ STL Containers	Yes	Yes

CYTHON

```
from cython.operator import dereference
from libcpp.vector cimport vector

cpdef void cython_print_types():
    cdef:
        int int_ctype = 1
        double double_ctype = 1.0
        double *double_ptr_ctype = &(double_ctype)
        double[5] double_array_ctype = [1.0, 2.0, 3.0, 4.0, 5.0]
        str string_ctype = "abcd"

    print(int_ctype.__class__.__name__)                          # Output: int
    print(double_ctype.__class__.__name__)                       # Output: float
    print(dereference(double_ptr_ctype).__class__.__name__)  # Output: float
    print(double_array_ctype.__class__.__name__)               # Output: list
    print(string_ctype.__class__.__name__)                       # Output: str

cpdef cython_vector():
    cdef:
        vector[int] a

    return a                                                       # Output: list
```

CLASS

	Cython	pybind11
Implementation	Extension / Binding C++	Binding C++
Property	Yes	Yes + static property
classmethod / staticmethod	Yes	`def_static`
Operator overloading	Yes	Yes

CLASS

	Cython	pybind11
Inheritance	Extension type: Yes, Binding C++: Limited	Yes
Dynamic Attribute	Declare `__dict__`	`py::dynamic_attr`

CYTHON

```
cdef class Order:
    """Order."""
    cdef:
        float _price
        float _quantity
        int _side

    def __init__(self, float price, float quantity, int side):
        self._price = price
        self._quantity = quantity
        assert side in [1, 2], "Side should be either 1 or 2"
        self._side = side

    @property
    def price(self):
        return self._price
```

CYTHON

```
cdef class StopOrder(Order):
    """Stop Order."""
    cdef:
        float _stop_price

    def __init__(self, float price, float quantity, int side, float stop_pri
        super().__init__(price, quantity, side)
        self._stop_price = stop_price

    @property
    def stop_price(self):
        return self._stop_price
```

```
order = StopOrder(price=1.0, quantity=1.0, side=1, stop_price=0.9)
order.price
# Output: 1.0
isinstance(order, Order)
# Output: True
StopOrder.__mro__
# Output: (cython_class.StopOrder, cython_class.Order, object)
```

Order.h

```
#ifndef ORDER_H
#define ORDER_H

class Order {
public:
    Order(double price, double quantity, int side) :
        price(price),
        quantity(quantity),
        side(side) {};

    double price, quantity;
    int side;
};

class StopOrder : public Order {
public:
    StopOrder(double price, double quantity, int side, double stop_price):
        Order(price, quantity, side),
        stop_price(stop_price) {}

    double stop_price;
};
```

PYBIND11

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

namespace py = pybind11;

PYBIND11_MODULE(example, m) {
    py::class_<Order>(m, "Order", "This is an order.")
        .def(py::init<double, double, int>(), "Constructor",
             py::arg("price"), py::arg("quantity"), py::arg("side"))
        .def_readonly("price", &Order::price);

    py::class_<StopOrder, Order>(m, "StopOrder", "This is a stop order.")
        .def(py::init<double, double, int, double>(), "Constructor",
             py::arg("price"), py::arg("quantity"), py::arg("side"), py::arg
        .def_readonly("stop_price", &StopOrder::stop_price);
}
```

```
order = StopOrder(price=1.0, quantity=1.0, side=1, stop_price=1.2)
order.price
# Output: 1.0
order.stop_price
# Output: 1.2
```

```
isinstance(StopOrder(price=1.0, quantity=1.0, side=1, stop_price=1.2),
           Order)
# Output: True
```

```
StopOrder.__mro__
# Output:
# (pybind11_586547e.StopOrder,
#  pybind11_2258990.Order,
#  pybind11_builtins.pybind11_object,
#  object)
```

```
StopOrder.__doc__
# Output: 'This is a stop order.'
```

CYTHON

```
# First the C++ class is imported
cdef extern from "Order.h":
    cdef cppclass Order:
        Order(float, float, int) except+
        float price, quantity
        int side

# Second wrap the class as a Python class
cdef class PyOrder:
    """Python order."""

    cdef Order* order

    def __cinit__(self, float price, float quantity, int side):
        self.order = new Order(price, quantity, side)

    @property
    def price(self):
        return self.order.price
```

```
PyOrder(price=1.0, quantity=1.0, side=1).price  # Output: 1.0
```

```
PyOrder.__doc__  # Output: 'Python order.'
```

ACCESS TO PYTHON MODULE / OBJECT

- As Cython supports compilation of native Python code, writting a function in hybrid of C and Python functions is straight-forward.
- Accessing the Python objects and their attributes / methods in C++ level is feasible in pybind11.

CYTHON

```
cimport numpy as np
import numpy as np

cpdef cython_l2_norm(np.ndarray vec):
    return np.dot(vec.T, vec)
```

PYBIND11

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

namespace py = pybind11;

double pybind11_l2_norm(py::object vec) {
    auto np = py::module::import("numpy");
    return np.attr("dot")(vec.attr("T"), vec).cast<double>();
}

PYBIND11_MODULE(example, m) {
    m.def("pybind11_l2_norm", &pybind11_l2_norm);
}
```

RETURN VALUE POLICY - PYBIND11

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

namespace py = pybind11;

class Data {
public:
    std::string val;
    Data() : val("hello world!") {}
};

class Container {
public:
    Data* data;
    Container() : data(new Data()) {}
    ~Container() { delete data; }
    Data* get_data() { return data; }
};
```

ACCESS TO DELETED POINTER

```
PYBIND11_MODULE(example, m) {
    py::class_<Data>(m, "Data", "Data")
        .def(py::init())
        .def_readonly("val", &Data::val);

    py::class_<Container>(m, "Container", "Container")
        .def(py::init())
        .def("return_default", &Container::get_data);
}
```

```
container = Container()
print(container.return_default().val)
print(container.return_default().val)

# Output:
# hello world!
# -------------------------------------------------------------------------
# UnicodeDecodeError                         Traceback (most recent call last
# <ipython-input-4-cb0eed65af98> in <module>
#       1 o = Container()
#       2 print(o.return_default().val)
# ----> 3 print(o.return_default().val)
#
# UnicodeDecodeError: 'utf-8' codec can't decode byte 0xd3 in position 1: in
```

RETAIN THE OWNERSHIP

```
PYBIND11_MODULE(example, m) {
    py::class_<Container>(m, "Container", "Container")
        .def(py::init())
        .def("return_reference", &Container::get_data,
             py::return_value_policy::reference);
}
```

```
o = Container()
print(o.return_reference().val)
print(o.return_reference().val)

# Output:
# hello world!
# hello world!
```

SMARTER WAY VIA SMART POINTERS

```
class SafeContainer {
public:
    std::shared_ptr<Data> data;
    SafeContainer() : data(std::make_shared<Data>()) {}
    std::shared_ptr<Data> get_data() { return data; }
};

PYBIND11_MODULE(example, m) {
    py::class_<SafeContainer>(m, "SafeContainer", "SafeContainer")
        .def(py::init())
        .def("return_default", &SafeContainer::get_data);
}
```

```
o = SafeContainer()
print(o.return_default().val)
print(o.return_default().val)

# Output:
# hello world!
# hello world!
```

BENCHMARK

- The C extension library has a great improvement on performance
- Although pybind11 performs better in some cases, cython is still surprisingly good to produce a significant improvement.

```
cpdef np.ndarray compute_exposures(
        np.ndarray labels, np.ndarray weight,
        int num_of_labels):
    """Compute exposures.
    """
    cdef:
        np.ndarray  exposures
        Py_ssize_t idx, label, weight_len = len(weight)
        double label_exposure

    exposures = np.zeros(weight_len)
    for idx in range(weight_len):
        exposures[labels[idx]] += weight[idx]

    return exposures
```

```
         pandas_groupby   cython       pybind11
-------  ---------------  -----------  -----------
Average  186.66 msec      39.11 msec   632.36 usec
Std.Dev  2.91 msec        484.74 usec  47.51 usec
Minimum  181.34 msec      38.76 msec   591.87 usec
```

CYTHON V.S. PYBIND11

- Cython can easily migrate your existing Python codebase and the performance gain is stunning and sufficient to impress everyone.
- However, pybind11 makes the binding of C++ to Python so native. And it supports C++11 so well.
- Choose the right tool according to your available resources, existing codebase and team expertise.

FUTURE PROSPECT

- The trend is moving towards pybind11
- Keep an eye on MYPYC, which compiles your Python code by type hints (PEP 484). Recommend reading: "Our journey to type checking 4 million lines of Python".
- Acceleration in GPU via cuDF (RAPIDS)

CONTACT

Github: @gavincyi

Email: gavincyi at gmail dot com