

2023

# Writing Python Bindings for C++ Libraries:

*Easy-to-use Performance*

Saksham Sharma

C++ now

# A QUICK BIO - YOURS TRULY

- Quantitative research developer at Tower Research Capital
  - High frequency trading firm based out of NYC
- Develop low latency trading systems (C++)
  - Nanoseconds matter
- Develop high throughput research systems (C++ and Python)
  - Data volume in terabytes
- Program analysis research and functional programming in a past life
- Love performance, software abstractions, and clean APIs

# WHY PYTHON? WHY C++?

## Why Python?

- Writing extensive APIs in Python – low boilerplate
- Familiar for domain experts
- Easy to use
  - Amazing interactive support out of the box (IPython)
  - Jupyter notebooks provide a great research environment
  - Very mature open source libraries for various domains

## Why C++?

- We're at CppNow :)

**PYTHON API**

**I NEED IT**

# WHY C++ AND PYTHON

- Why?
  - Avoid reimplementing complex code for Python
  - Performance
  - Back and forth with user's python code
  - Interoperability with data structures in Python – shared memory space
- How?
  - [Python/C API](#)
  - [Cython](#)
  - [Numba](#) / [PyPy](#)
  - [Boost::Python](#)
  - [Pybind11](#)
  - [cppyy](#)
  - Mix and match!

*Hello, world*

(not really)

# HELLO, WORLD

Different implementations of a simple increment function  
(takes an integer as argument, increments it and returns)

- C++
- Python
- C++ with pybind11
- C++ with boost::python

C++

```
int increment(int); // defined in a separate translation unit to prevent inlining

int main(int argc, char**argv) {
    int max_counter = std::stoi(std::string(argv[1]));
    int i = 0;
    auto start = high_resolution_clock::now();
    while (i < max_counter) {
        i = increment(i);
    }
    auto end = high_resolution_clock::now();
    std::cout << "Time taken per call: " << ... << std::endl;
}
```



# PYTHON

```
def increment(i):  
    return i + 1  
  
if __name__ == "__main__":  
    i = 0  
    max_i = 2**20  
    if len(sys.argv) > 1:  
        max_i = int(sys.argv[1])  
    start = time.time()  
    while i < max_i:  
        i = increment(i)  
    end = time.time()  
    print("Time taken: {:.2f}ms".format((end - start) * 1000))  
    print("Time taken per iteration: {:.2f}ns".format((end - start) * 1000000000 / i))
```

# PYBIND11

```
int cpp_increment(int i) { return i + 1; }
```

```
PYBIND11_MODULE(hello_world_pybind11, m) {
```

```
    m.doc() = "pybind11 incrementer";
```

```
    m.def("increment", &cpp_increment, "Incrementing function");
```

```
}
```

```
// After compiling above into hello_world_pybind11.so, in python
```

```
from hello_world_pybind11 import increment
```

```
while i < max_i:
```

```
    i = increment(i)
```

# BOOST::PYTHON

```
int cpp_increment(int i) { return i + 1; }
```

```
BOOST_PYTHON_MODULE(hello_world_bpy) {  
    using namespace boost::python;  
    def("increment", &cpp_increment, "Incrementing function");  
}
```

// After compiling above into hello\_world\_bpy.so, in python

```
from hello_world_bpy import increment
```

```
while i < max_i:  
    i = increment(i)
```

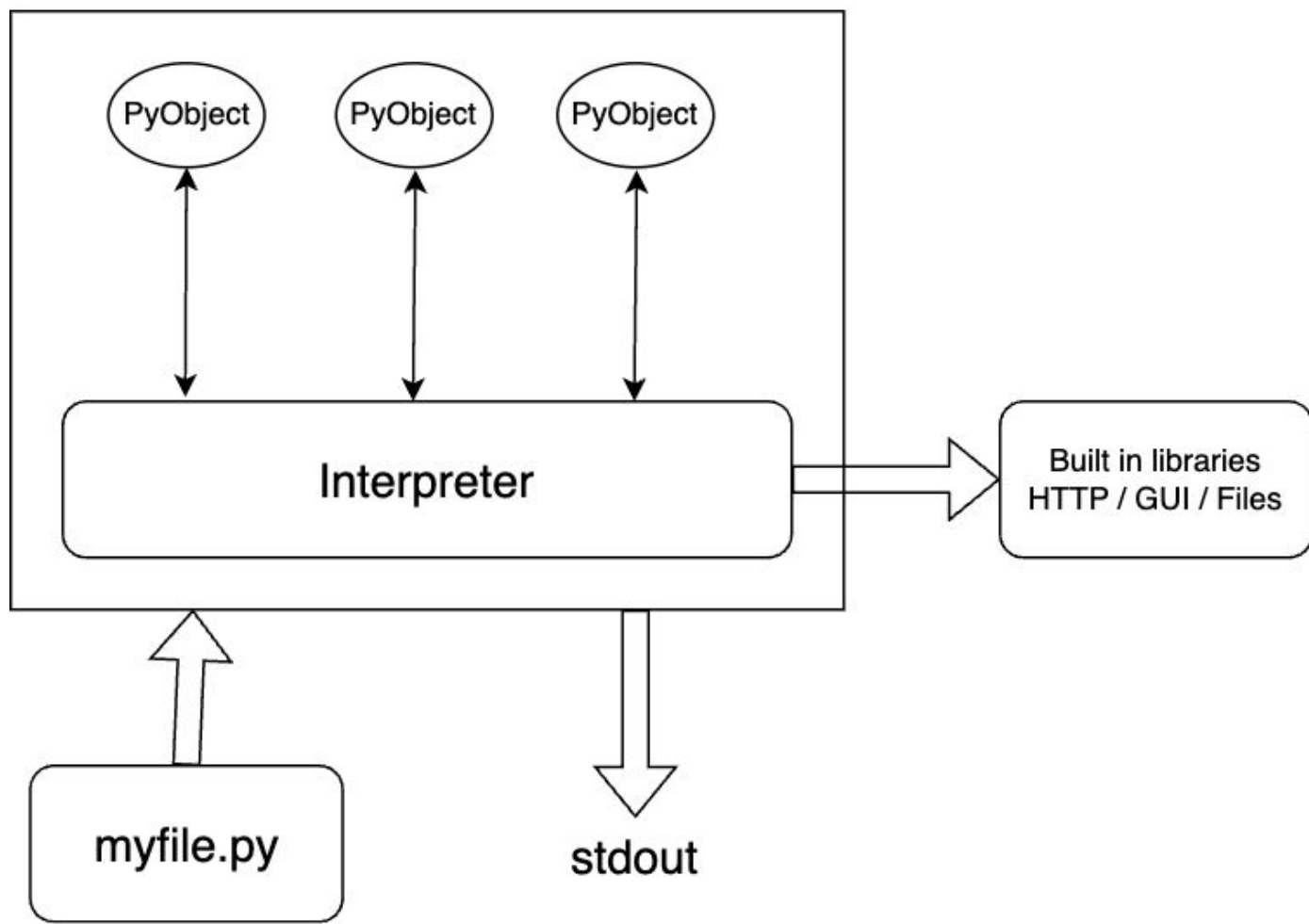
*Wait, what?*



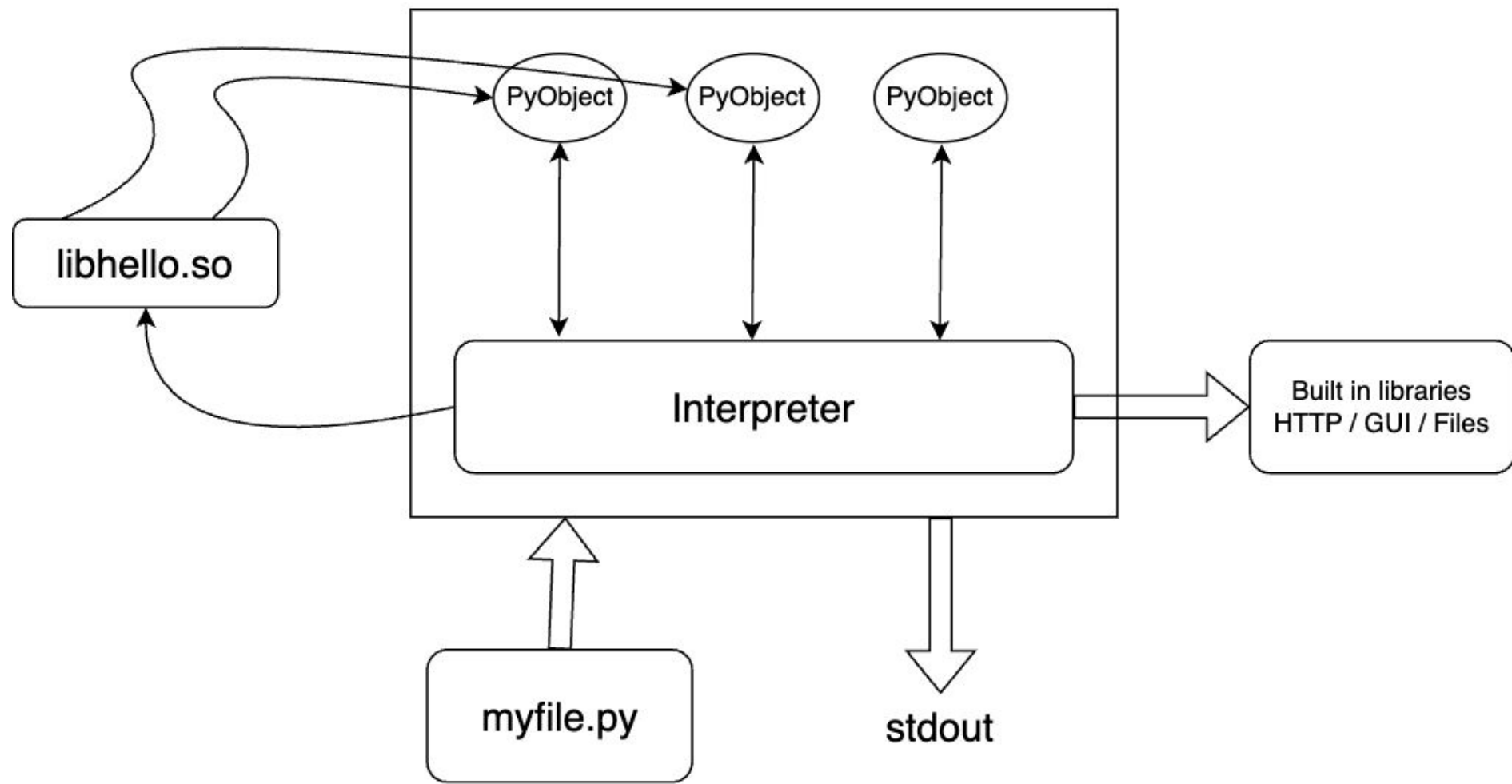
# ENTER THE CPYTHON INTERPRETER

- Reference implementation of the python interpreter
- A C program that interprets python instructions
  - Internally maintains state of your program (objects etc) as C objects derived from a struct called PyObject.
  - Each new python statement tells it how to modify these PyObject.
  - State of the program is maintained in these introspectable objects.

# CPython Runtime Core



# CPython Runtime Core





# ENTER THE CPYTHON INTERPRETER

- Loading a .so into this program's memory space allows us to interact with its internals.
- We can call methods (from <Python.h>) to change the state of the program
  - API documented nicely <https://docs.python.org/3/c-api/index.html>
- We can add “hooks” into the program to invoke our methods in certain situations.
- Such a module is called an “extension module”.

# PYTHON C API

```
#include <Python.h>

int cpp_increment(int i) { return i + 1; }

static PyObject* python_increment(PyObject* self, PyObject* args) {
    int i;
    if(!PyArg_ParseTuple(args, "i", &i)) {
        return NULL;
    }
    return PyLong_FromLong(cpp_increment(i));
}

static PyMethodDef HWMMethods[] = {
    {"increment", python_increment, METH_VARARGS, "Incrementing function"},
    {NULL, NULL, 0, NULL}
};

static struct PyModuleDef hello_world_python = {
    PyModuleDef_HEAD_INIT, "hello_world_python", "Simple python module", -1, HWMMethods
};

PyMODINIT_FUNC PyInit_hello_world_python(void) {
    return PyModule_Create(&hello_world_python);
}
```

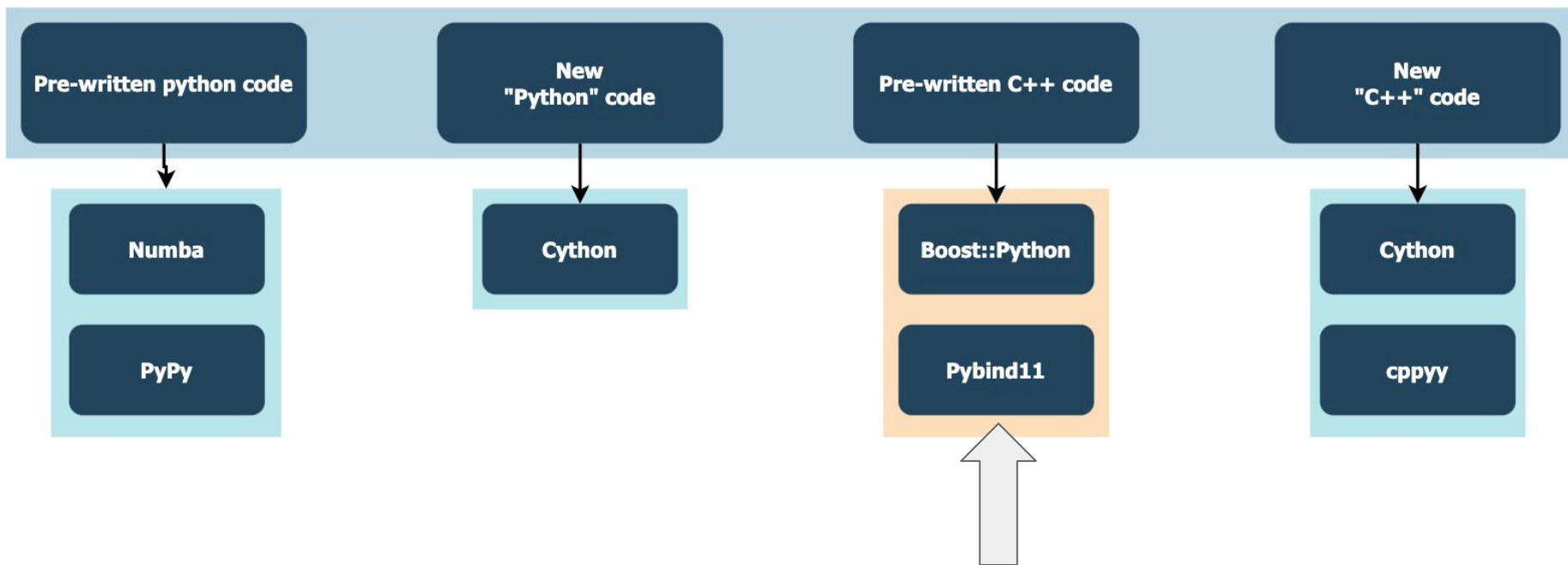
## DIGRESSION: JYTHON / IRONPYTHON / PYPY

<https://wiki.python.org/moin/PythonImplementations>

Easier to extend

Can import libraries / modules from the host language

# VARIOUS LIBRARIES TO "SPEED UP" PYTHON



Focus of this talk

*Performance*

# SOME PERF NUMBERS: PER INCREMENT RUNTIME

Take these numbers with a grain of salt



# SOME PERF NUMBERS: PER INCREMENT RUNTIME

Implementati on	1 run	10 runs	100 runs	1000 runs	Size
Python	953ns	143ns	71ns	67ns	NA
C++	85ns	11ns	3ns	1ns	91K*
Python C API	1192ns	214ns	83ns	74ns	14K
pybind11	9536ns	1525ns	634ns	552ns	279K
boost::python	1907ns	309ns	123ns	109ns	237K

\*: C++ program contains the `std::chrono` library to time, others use python to do that  
Everything compiled with `clang++12 -fPIC -O3`

# SOME PERF NUMBERS: PER INCREMENT RUNTIME

<https://news.ycombinator.com/item?id=34663930>

Out in the wild case of noticing high overheads in pysimdjson (they switched from pybind11 to cython)

Some GitHub issues:

<https://github.com/pybind/pybind11/issues/2005>

<https://github.com/pybind/pybind11/issues/1227>

Primary causes seem to be an `std::vector` to store arguments, and some extra argument type casting work.

This latency of calling a function may or may not be relevant to everyone. But is good to know.



*Classes 101*

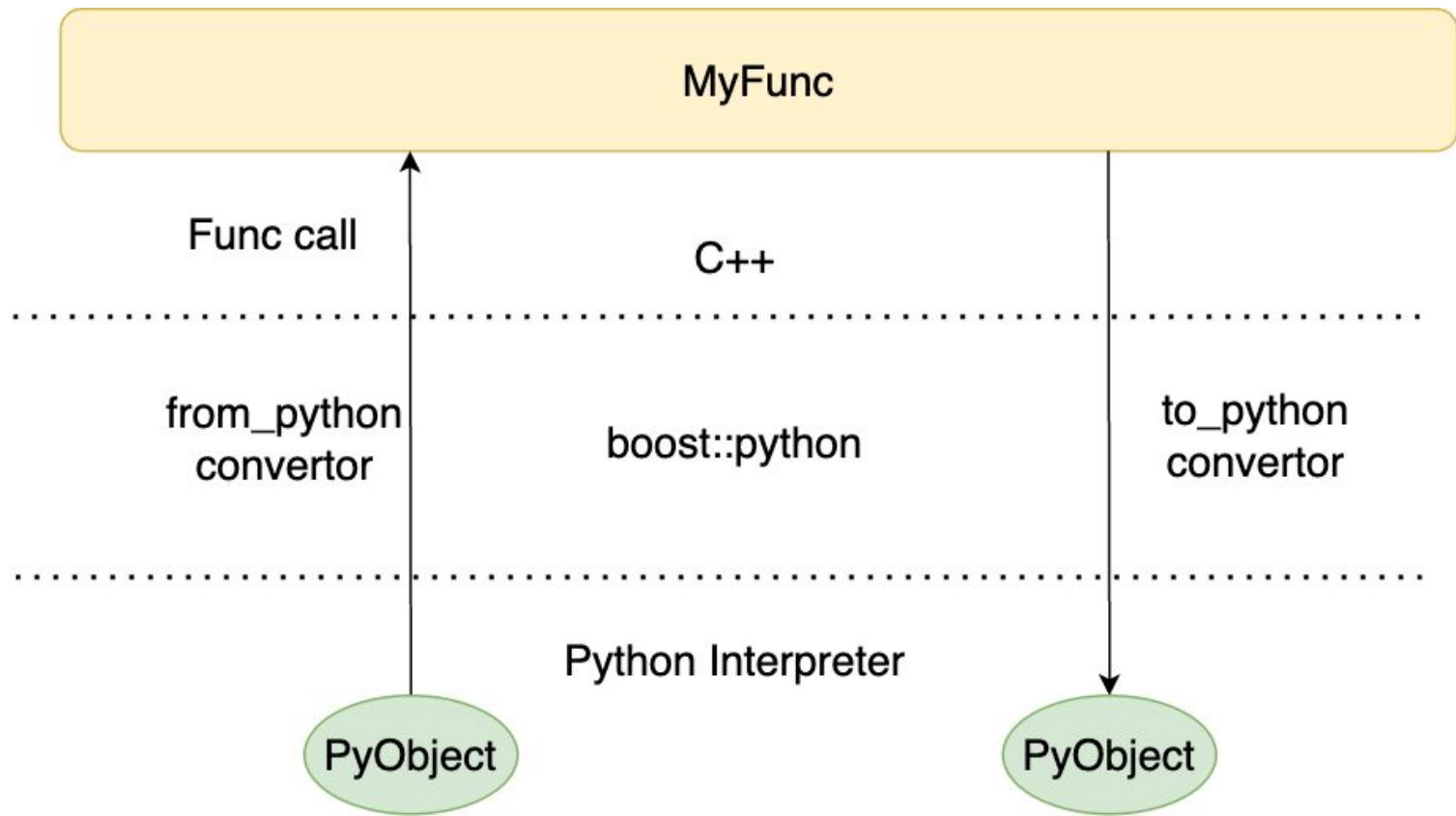
# CLASSES?

```
class RowReader : public BinaryListener {  
    std::vector<Row> rows_  
    BinaryReader reader_  
  
public:  
    RowReader(const std::string& filename); // somehow populates rows_ vector  
    void onData(int id, BData data) override; // BinaryListener virtual  
    std::string getIdName(int id) const;  
    auto getRows() { // can have variants that filter the rows  
        return rows_  
    }  
};
```

# CLASSES!

```
BOOST_PYTHON_MODULE(binary_reader_bpy) {  
    class_<binary_reader::Row>("Row", no_init)  
        .def("nanotime", &Row::nanotime)  
        .def("items", &Row::items, return_value_policy<reference_internal_object>());  
    class_<std::vector<Row>>("cpp::vector<Row>")  
        .def(vector_indexing_suite<std::vector<Row>>());  
    class_<RowReader>("RowReader", init<std::string>(arg("filename")))  
        .def("getRows", &RowReader::getRows)  
        .def("getIdName", &RowReader::getIdName);  
}
```

# CLASSES!



# CLASSES!

```
import binary_reader_bpy as mod
reader = mod.RowReader("./testfile")
rows = reader.getRows()
print("Number of rows: " + str(len(rows)))
for i, row in enumerate(rows):
    print("Row " + str(i) + " is " + str(row))
    for item in row.items():
        print("    - {}:{}".format(reader.getIdName(item.id), item.value()))
```

# WORKING EXAMPLE

```
$ python3 python/v1.py
```

```
Number of rows: 2
```

```
Row 0 is 1682563241000000000: 4 items
```

- trade\_size:3
- price:4086.2
- trade\_size:3
- price:4086.2

```
Row 1 is 1682563242000000000: 7 items
```

- trade\_size:5
- price:4086.2
- trade\_size:5
- price:4086.7
- signal:1.2
- order\_size:1
- price:4086.7

# *Software design with pybindings*

# BASIC THOUGHT PROCESS

- Define a user boundary for your library's API
- Expose methods at that boundary using python binding
- Try putting “hot path” code inside your library
  - Need to design a DSL for the user to be able to leverage your hot path capabilities written in C++
  - Numpy / Pandas / PyTorch are nice DSLs that provide “hot path” C++ functions
- Can allow python objects for constructors / non-critical operations
  - Simply for ease of integration



# WHAT FUNCTIONS TO EXPOSE IN THE API

- Init some data structures / load some data
- Configure your class' parameters
- Do a lot of complex work inside a “hot loop”
- Fetch internal state of your data structure

# ARGUMENTS TO FUNCTIONS

- Can be `int / float / std::vector<std::string>, ...`
  - Most libraries support all these common types' conversion from `PyObject*` and back
  - Should be optimal to let the library do the cast for you
- Can be instances types defined by your module
- Can be `PyObject* / boost::python::object`
  - `boost::python::object` is a smart pointer around a `PyObject*`
  - Only useful to use `bpy::object` if you're going to store it
  - Why would we want to do this?

# ARGUMENTS TO FUNCTIONS - PYOBJECTS

```
MyReader::MyReader(bpy::object filenames) {  
    std::vector<std::string> filenames_std;  
    bpy::extract<bpy::list> get_list(filenames);  
    if (get_list.check()) {  
        filenames_std = to_std_vector<std::string>(get_list());  
    } else {  
        bpy::extract<std::string> get_string(filenames);  
        if (get_string.check()) {  
            filenames_std.push_back(get_string());  
        } else {  
            // error  
        }  
    }  
}
```

# RETURN VALUES OF FUNCTIONS

- Easy enough to return standard types (`int`, `float`, ...)
  - Let `bindings` library do the heavy lifting
- Can return python objects
  - `boost::python::dict` / `boost::python::list` are common ones I use
  - `boost::python` just makes it easy to create refcounted `PyObject`s
  - Even more optimal if your code just populates the python data structure instead of doing a two pass
- What about larger objects?
  - Large vectors, matrices?
  - Python libraries aren't written to operate on `std::vector<float>`

# RETURN VALUES OF FUNCTIONS - NUMPY

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

namespace bnp = boost::python::numpy;

template <typename T>
inline bnp::ndarray to_boost_vector(const std::vector<T>& v) {
    Py_intptr_t shape[1] = {static_cast<long>(v.size())};
    bnp::ndarray result = bnp::zeros(1, shape, bnp::dtype::get_builtin<T>());
    std::copy(v.begin(), v.end(), reinterpret_cast<T*>(result.get_data()));
    return result;
}
```

# LIFETIME OF RETURNED VALUES

- Say you return a struct (`struct1`) containing references to another struct (`struct2`)
- Python will manage `struct1`'s instance as a refcounted object
  - Note: Python is a reference counted garbage collected language
- Python has no way to know `struct1` refers to `struct2` internally, and will happily garbage collect `struct2`
- Classic use for shared pointers, just good to remember

# LIFETIME OF RETURNED VALUES

```
class Item {  
    const Reader* reader_;  
    int id;  
    double value;  
public:  
    std::pair<std::string, double> getNameValue() {  
        return {reader_->getName(id), value};  
    }  
};
```

# LIFETIME OF RETURNED VALUES

```
class Item {  
    std::shared_ptr<const Reader> reader_;  
    int id;  
    double value;  
public:  
    std::pair<std::string, double> getNameValue() {  
        return {reader_->getName(id), value};  
    }  
};
```



# A NOTE ABOUT "SOFTWARE GLUE"

- Composing softwares requires a scripting "glue" at times
- The unix philosophy is to spin up lots of processes, and using pipes to compose softwares
  - Hard to do easy back and forth b/w softwares using OOP
  - Hard to shared typed data
  - Hard to optimize away the last few copies
- Python's ease of extension => powerful glue

EXAMPLE 1  
CUSTOM BINARY  
DATA FORMAT

# NAIVE APPROACH: CUSTOM BINARY LOGGING DATA

- Have existing C++ library to parse data
- Write C++ to serialize data as plaintext on stdout
- Write python library to process data items from stdout
- Write analysis code using python by iterating over all this data

Too slow! 🙄

# ENTER PYTHON BINDINGS

- Re-use C++ library
- Make class that stores / preprocesses data in memory
- Expose API methods to process that data and expose results in a memory layout that python understands
  - Write DSL to return numpy arrays for analysis

# WE'VE SEEN THIS BEFORE

```
import binary_reader_bpy as mod

print("Module version is " + str(mod.API_VERSION))

reader = mod.RowReader("./testfile")

rows = reader.getRows()

print("Number of rows: " + str(len(rows)))

for i, row in enumerate(rows):
    print("Row " + str(i) + " is " + str(row))
    for item in row.items():
        print("  - {}:{}".format(reader.getIdName(item.id), item.value()))
```


# READER CODE

```
class RowReader : public BinaryListener {
    std::vector<RowRef> rows_;
public:
    RowReader(const std::string& filename); // does the file IO to read file into memory
    void onData(int id, BData data, const IdMetadata& metadata) override {
        if (metadata.name == "timestamp")
            rows_.push_back(RowRef{.nanotime_ = std::get<int64_t>(data)});
        else
            rows_.back()->items_.push_back({id, data});
    }
    std::string getIdName(int id) const;
    std::vector<RowRef> getRows() const;
    std::vector<RowRef> getRowsWithSignalGreaterThan(double signal) const;
};
```

# BINDING CODE

```
BOOST_PYTHON_MODULE(binary_reader_bpy) {  
    class_<binary_reader::Item>("Item", no_init)  
        .def_readonly("id", &Item::id)  
        .def("value", &Item::getPyValue);  
    class_<binary_reader::RowRef>("Row", no_init)  
        .def("nanotime", &RowRef::nanotime)  
        .def("items", &RowRef::items, return_value_policy<return_internal_reference>());  
    class_<std::vector<RowRef>>("cpp::vector<RowRef>").def(vector_indexing_suite<std::vector<RowRef>>());  
    class_<std::vector<Item>>("cpp::vector<Item>").def(vector_indexing_suite<std::vector<Item>>());  
    class_<RowReader>("RowReader", init<std::string>(arg("filename")))  
        .def("getRows", &RowReader::getRows)  
        .def("getIdName", &RowReader::getIdName);  
    scope().attr("API_VERSION") = 1;  
    register_exception_translator<std::runtime_error>(translate_error);  
}
```

# (1) BINDING CODE - BASICS



```
BOOST_PYTHON_MODULE(binary_reader_bpy) {  
    class_<binary_reader::Item>("Item", no_init)  
        .def_readonly("id", &Item::id)  
        .def("value", &Item::getPyValue);  
    class_<binary_reader::RowRef>("Row", no_init)  
        .def("nanotime", &RowRef::nanotime)  
        .def("items", &RowRef::items, return_value_policy<return_internal_reference>());  
    class_<std::vector<RowRef>>("cpp::vector<RowRef>").def(vector_indexing_suite<std::vector<RowRef>>());  
    class_<std::vector<Item>>("cpp::vector<Item>").def(vector_indexing_suite<std::vector<Item>>());  
    class_<RowReader>("RowReader", init<std::string>(arg("filename")))  
        .def("getRows", &RowReader::getRows)  
        .def("getIdName", &RowReader::getIdName);  
    scope().attr("API_VERSION") = 1;  
    register_exception_translator<std::runtime_error>(translate_error);  
}
```



# (1) BINDING CODE - BASICS

```
using BData = std::variant<int64_t, double>;

struct Item {
    int id;
    BData value;
    PyObject* getPyValue() const;
    bool operator==(const Item& o) const { return o.id == id && o.value == value; }
};

// Declaration
class_<binary_reader::Item>("Item", no_init)
    .def_readonly("id", &Item::id)
    .def("value", &Item::getPyValue);
```

## (2) BINDING CODE - RETURN\_VALUE\_POLICY

```
BOOST_PYTHON_MODULE(binary_reader_bpy) {  
    class_<binary_reader::Item>("Item", no_init)  
        .def_readonly("id", &Item::id)  
        .def("value", &Item::getPyValue);  
    class_<binary_reader::RowRef>("Row", no_init)  
        .def("nanotime", &RowRef::nanotime)  
        .def("items", &RowRef::items, return_value_policy<return_internal_reference>());  
    class_<std::vector<RowRef>>("cpp::vector<RowRef>").def(vector_indexing_suite<std::vector<RowRef>>());  
    class_<std::vector<Item>>("cpp::vector<Item>").def(vector_indexing_suite<std::vector<Item>>());  
    class_<RowReader>("RowReader", init<std::string>(arg("filename")))  
        .def("getRows", &RowReader::getRows)  
        .def("getIdName", &RowReader::getIdName);  
    scope().attr("API_VERSION") = 1;  
    register_exception_translator<std::runtime_error>(translate_error);  
}
```

## (2) BINDING CODE - RETURN\_VALUE\_POLICY

```
struct Row {  
    std::vector<Item> items_;  
    const auto& items() const { return items_; }  
};  
  
// Declaration:  
.def("items", &RowRef::items, return_value_policy<return_internal_reference>());
```

Let's try to engineer the best possible semantics for managing references in python.

## (2) BINDING CODE -RETURN\_VALUE\_POLICY

```
.def("items", &RowRef::items, return_value_policy<return_existing_object>()); // UNSAFE!
```

```
// Unsafe python code:
```

```
def get_first_row_items(reader):  
    for row in reader.getRows():  
        return row.items()
```

- Return a PyObject that contains an unmanaged pointer
  - Unsafe, lifetime of “items” PyObject can exceed lifetime of Row

## (2) BINDING CODE - RETURN\_VALUE\_POLICY


```
class ItemsVectorWrapper {  
    const std::vector<Item> items_;  
    PyObject* row_;  
public:  
    Item operator[](size_t idx) { return items_[idx]; }  
    ItemsVectorWrapper(const std::vector<Item>& items, PyObject* row) : items_(items), row_(row) {  
        Py_IncRef(row);  
    }  
    ~ItemsVectorWrapper() { Py_DecRef(row_); }  
};
```

- Change return value to be a wrapped struct

## (2) BINDING CODE -RETURN\_VALUE\_POLICY

- Rather verbose and annoying to write
- Luckily, `return_value_policy<return_internal_reference>` is smart
- It does exactly what we discussed, but on its own

### (3) BINDING CODE - VECTOR\_INDEXING\_SUITE

```
BOOST_PYTHON_MODULE(binary_reader_bpy) {  
    class_<binary_reader::Item>("Item", no_init)  
        .def_readonly("id", &Item::id)  
        .def("value", &Item::getPyValue);  
    class_<binary_reader::RowRef>("Row", no_init)  
        .def("nanotime", &RowRef::nanotime)  
        .def("items", &RowRef::items, return_value_policy<return_internal_reference>());  
    class_<std::vector<RowRef>>("cpp::vector<RowRef>").def(vector_indexing_suite<std::vector<RowRef>>());  
     class_<std::vector<Item>>("cpp::vector<Item>").def(vector_indexing_suite<std::vector<Item>>());  
    class_<RowReader>("RowReader", init<std::string>(arg("filename")))  
        .def("getRows", &RowReader::getRows)  
        .def("getIdName", &RowReader::getIdName);  
    scope().attr("API_VERSION") = 1;  
    register_exception_translator<std::runtime_error>(translate_error);  
}
```

### (3) BINDING CODE - VECTOR\_INDEXING\_SUITE

```
class_<std::vector<Item>>("cpp::vector<Item>").def(vector_indexing_suite<std::vector<Item>>());
```

- C++ vectors have value semantics for operator[]
  - myvec[1].mutate() will not change value of myvec[1], unless?
- Python vectors have reference semantics for operator[]
- vector\_indexing\_suite tries to make c++ vectors behave like python vectors




### (3) BINDING CODE - VECTOR\_INDEXING\_SUITE

```
class_<std::vector<std::shared_ptr<Row>>>("cpp::vector<std::shared_ptr<Row>>")  
    .def(vector_indexing_suite<std::vector<std::shared_ptr<Row>>, true>());  
class_<RowReader>("RowReader", init<std::string>(arg("filename")))  
    .def("getRows", &RowReader::getRows, return_value_policy<reference_existing_object>())
```

- Will memory usage increase after calling getRows?
- No! (I checked 😊)
  - Does not copy the vector, just exposes an API around it

## (4) BINDING CODE - CONSTRUCTOR

```
BOOST_PYTHON_MODULE(binary_reader_bpy) {  
    class_<binary_reader::Item>("Item", no_init)  
        .def_readonly("id", &Item::id)  
        .def("value", &Item::getPyValue);  
    class_<binary_reader::RowRef>("Row", no_init)  
        .def("nanotime", &RowRef::nanotime)  
        .def("items", &RowRef::items, return_value_policy<return_internal_reference>());  
    class_<std::vector<RowRef>>("cpp::vector<RowRef>").def(vector_indexing_suite<std::vector<RowRef>>());  
    class_<std::vector<Item>>("cpp::vector<Item>").def(vector_indexing_suite<std::vector<Item>>());  
     class_<RowReader>("RowReader", init<std::string>(arg("filename")))  
        .def("getRows", &RowReader::getRows)  
        .def("getIdName", &RowReader::getIdName);  
    scope().attr("API_VERSION") = 1;  
    register_exception_translator<std::runtime_error>(translate_error);  
}
```

## (4) BINDING CODE - CONSTRUCTOR

```
class_<RowReader>("RowReader", init<std::string>(arg("filename")))
    .def("getRows", &RowReader::getRows)
    .def("getIdName", &RowReader::getIdName);
register_exception_translator<std::runtime_error>(translate_error);
```

Some nuances to consider when defining python construction:

- May throw an exception
- May do IO that we may want to parallelize across threads

## (4) BINDING CODE - EXCEPTIONS



```
void translate_error(const std::runtime_error& e) {  
    PyErr_SetString(PyExc_Exception, e.what());  
}  
register_exception_translator<std::runtime_error>(translate_error);
```

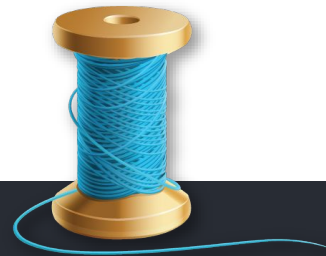
- Exceptions are thread local error flags.
- Existing libraries may have `::abort` / `::exit` calls in macros
- Use `#ifdef` macros to switch your exit macros to throw exceptions instead

## (4) BINDING CODE - EXCEPTIONS



```
#ifdef THROW_EXCEPTION_INSTEAD_OF_DYING
#define DIE(msg) MyNs::DieThrower(__FILE__, __LINE__, msg)
#else
#define DIE(msg) MyNs::DieExiter(__FILE__, __LINE__, msg)
#endif
```

## (4) BINDING CODE - THREADING

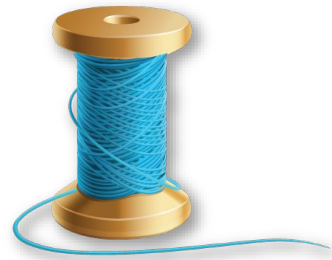


```
RowReader(const std::string& filename) : reader_(filename) {  
    reader_.addListener(this);  
    reader_.readAll();  
}
```

// Python API

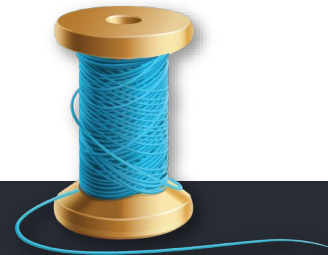
```
with ThreadPoolExecutor(max_workers=10) as executor:  
    futures = []  
    for f in sys.argv[1:]:  
        futures.append(executor.submit(mod.RowReader, f))  
    readers = [fut.result() for fut in futures]
```

## (4) BINDING CODE - THREADING



- Python has a Global Interpreter Lock (GIL)
- A blocking IO call that does not deal with any python objects should be allowed to release the lock
- Python's internal IO libraries already do this
- We'll discuss nuances of this in a later example

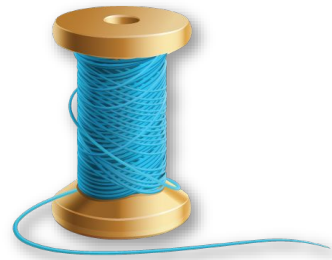
## (4) BINDING CODE - THREADING



```
class without_gil {  
    public: /* Source: https://stackoverflow.com/questions/41133001 */  
        without_gil() { state_ = PyEval_SaveThread(); }  
        ~without_gil() { PyEval_RestoreThread(state_); }  
        without_gil(const without_gil&) = delete;  
        without_gil& operator=(const without_gil&) = delete;  
    private:  
        PyThreadState* state_;  
};  
  
RowReader::RowReader(const std::string& filename) : reader_(filename) {  
    without_gil no_gil;  
    ....  
}
```



## (4) BINDING CODE - THREADING



Reading 3 161 MB files with and without releasing GIL

```
# without releasing GIL
$ /usr/bin/time python3 python/v2-threading.py testfile1 testfile2 testfile3
Module version is 1
Finished
10.63user 0.42system 0:11.11elapsed 99%CPU (0avgtext+0avgdata 1509900maxresident)k

# after releasing GIL
$ /usr/bin/time python3 python/v2-threading.py testfile1 testfile2 testfile3
Module version is 1
Finished
10.36user 0.52system 0:04.13elapsed 263%CPU (0avgtext+0avgdata 1509932maxresident)k
```

# EXAMPLE 2

## C++ LIBRARY WITH A DISPATCHER

# DISPATCHERS?

```
while (true) {  
  if (source1.has_data()) {  
    source1.listeners.inform();  
  }  
  if (source2.has_data()) {  
    source2.listeners.inform();  
  }  
}
```

# DISPATCHERS AND CALLBACKS

```
def print_packet(packet):  
    print(packet)  
  
my_network_library = MyNetworkLibrary()  
my_network_library.register_callback_on_packet(print_packet)  
my_network_library.listenAndBlock()
```

- Callbacks are a common theme in many dispatcher based applications
- Dispatching may happen in:
  - The main python thread, but blocked
  - A separate thread, thus letting the python thread run

# CALLBACKS SYNTAX

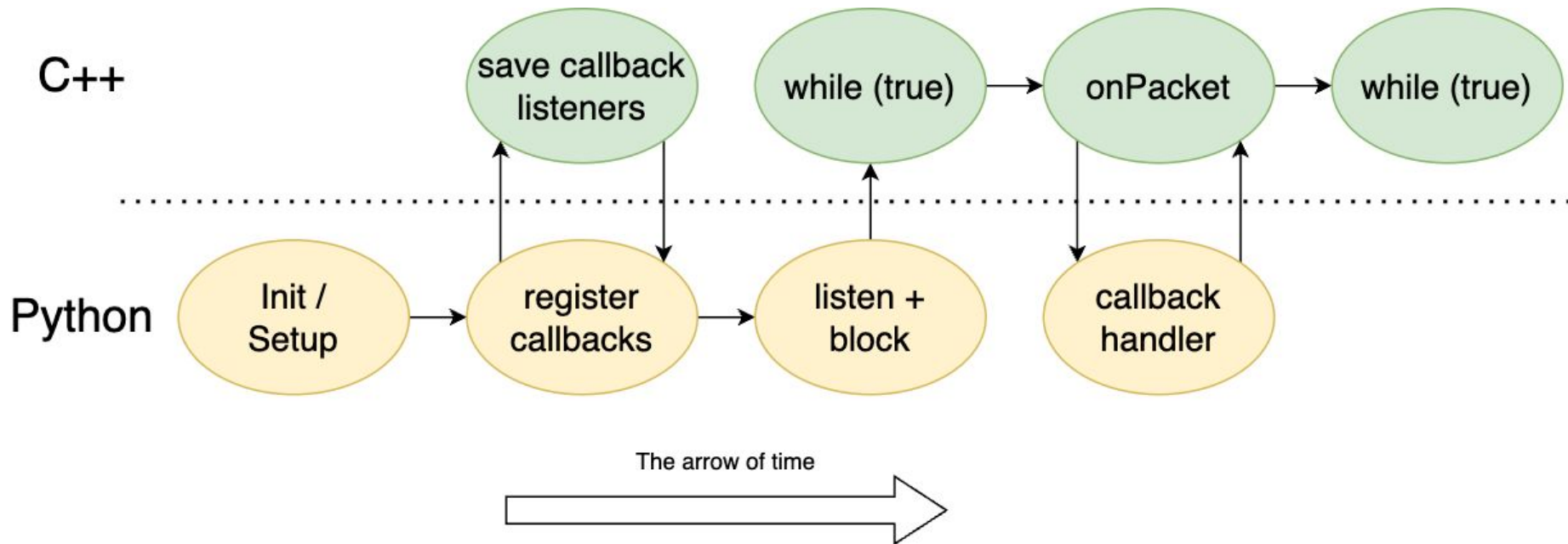
```
std::vector<boost::python::object> callbacks_;

void registerCallbackOnPacket(boost::python::object callback) {
    if (callback != boost::python::object() && !PyCallable_Check(callback.ptr())) {
        throw std::runtime_error("Callback has to be callable");
    }
    callbacks_.push_back(callback);
}

void sendInfoToPython(Info info) {
    for (auto& callback : callbacks_) {
        callback(info); // TODO: Need to grab GIL before we do this
    }
}
```

LET'S MAKE THIS COMPLICATED, ONE STEP AT A TIME

# COMPLEXITY LEVEL 1 : EVERYTHING IN THE SAME THREAD

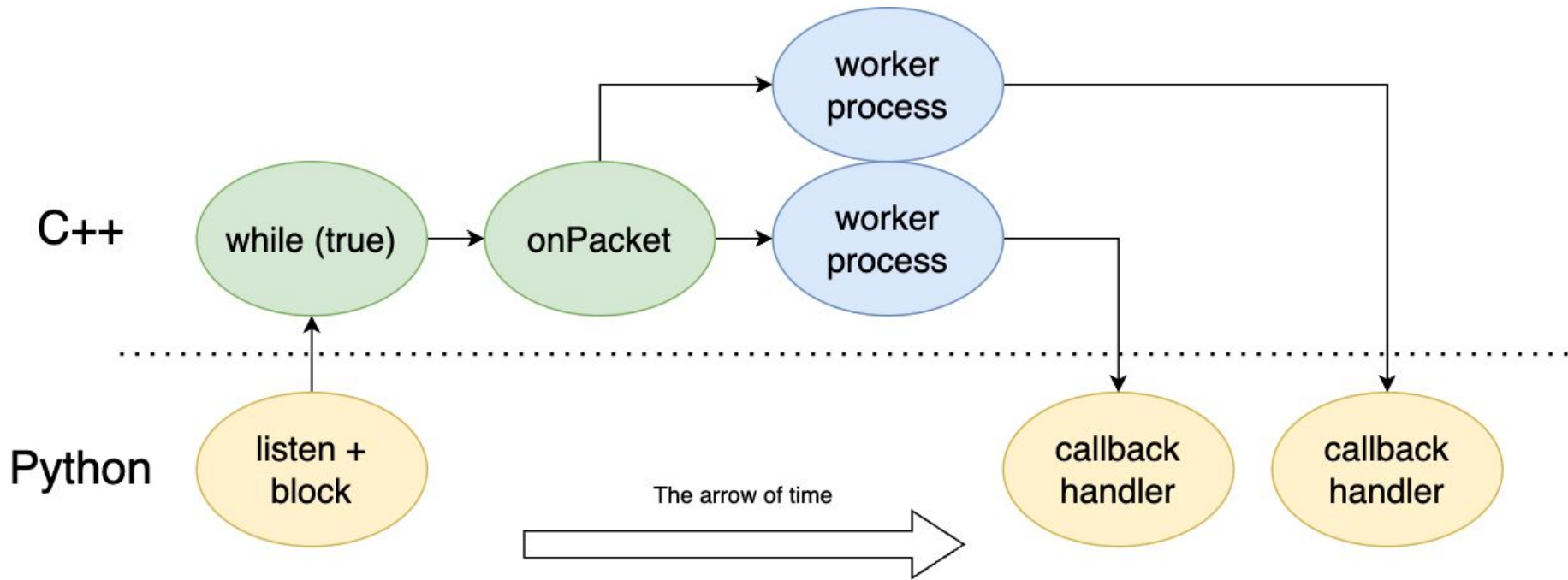


# COMPLEXITY LEVEL 1 : EVERYTHING IN THE SAME THREAD

- Very simple, single threaded operation
- No need to think about locking / threads
- Python threads cannot run in the background though
  - Maybe some GUI being run by python
  - Maybe some widgets / resource monitoring code
- Let's try to add more worker threads to this



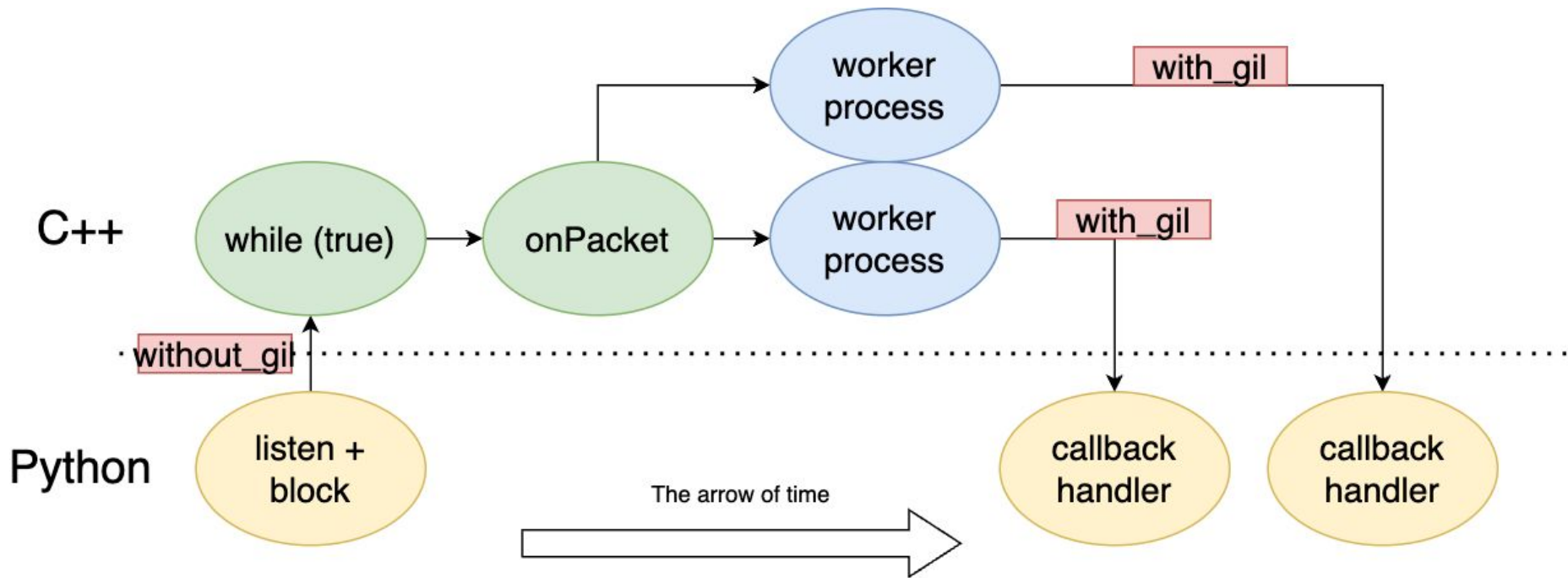
# COMPLEXITY LEVEL 2 : MULTIPLE WORKER THREADS



## COMPLEXITY LEVEL 2 : MULTIPLE WORKER THREADS

- Worker threads cannot call the callback handlers without acquiring the GIL
- The main watcher thread needs to release the GIL first!
- How do we resolve?
  - Main thread joins the worker threads and calls the callbacks?
    - Inefficient! Do you know why?
  - Main thread releases GIL before blocking, workers acquire GIL before callback

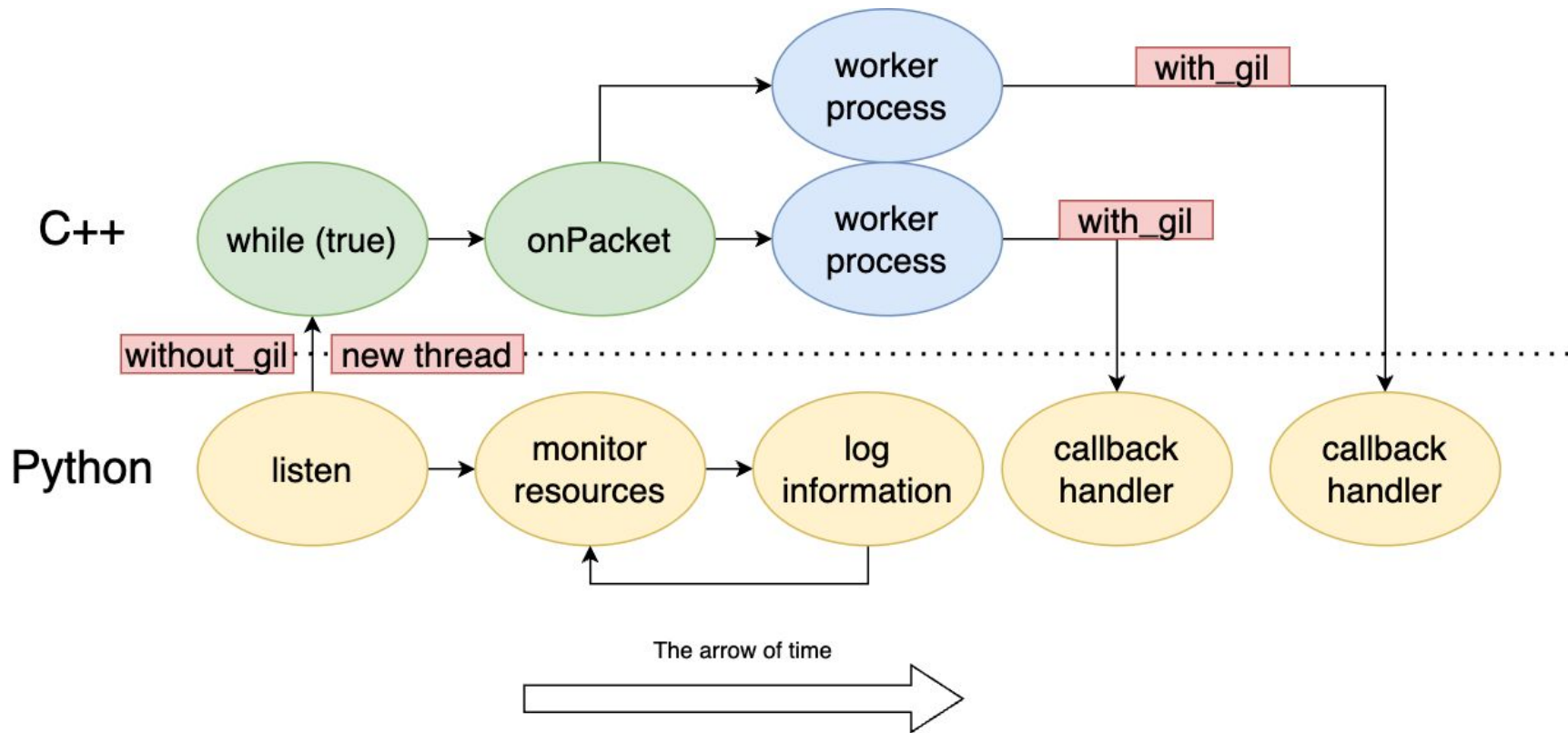
# COMPLEXITY LEVEL 2 : MULTIPLE WORKER THREADS



## COMPLEXITY LEVEL 2 : MULTIPLE WORKER THREADS

- Seems scalable
- C++ dispatching thread doesn't even need GIL
  - Python is not inherently blocked!

# COMPLEXITY LEVEL 3 : PYTHON BACKGROUND TASKS



# COMPLEXITY LEVEL 3 : PYTHON BACKGROUND TASKS

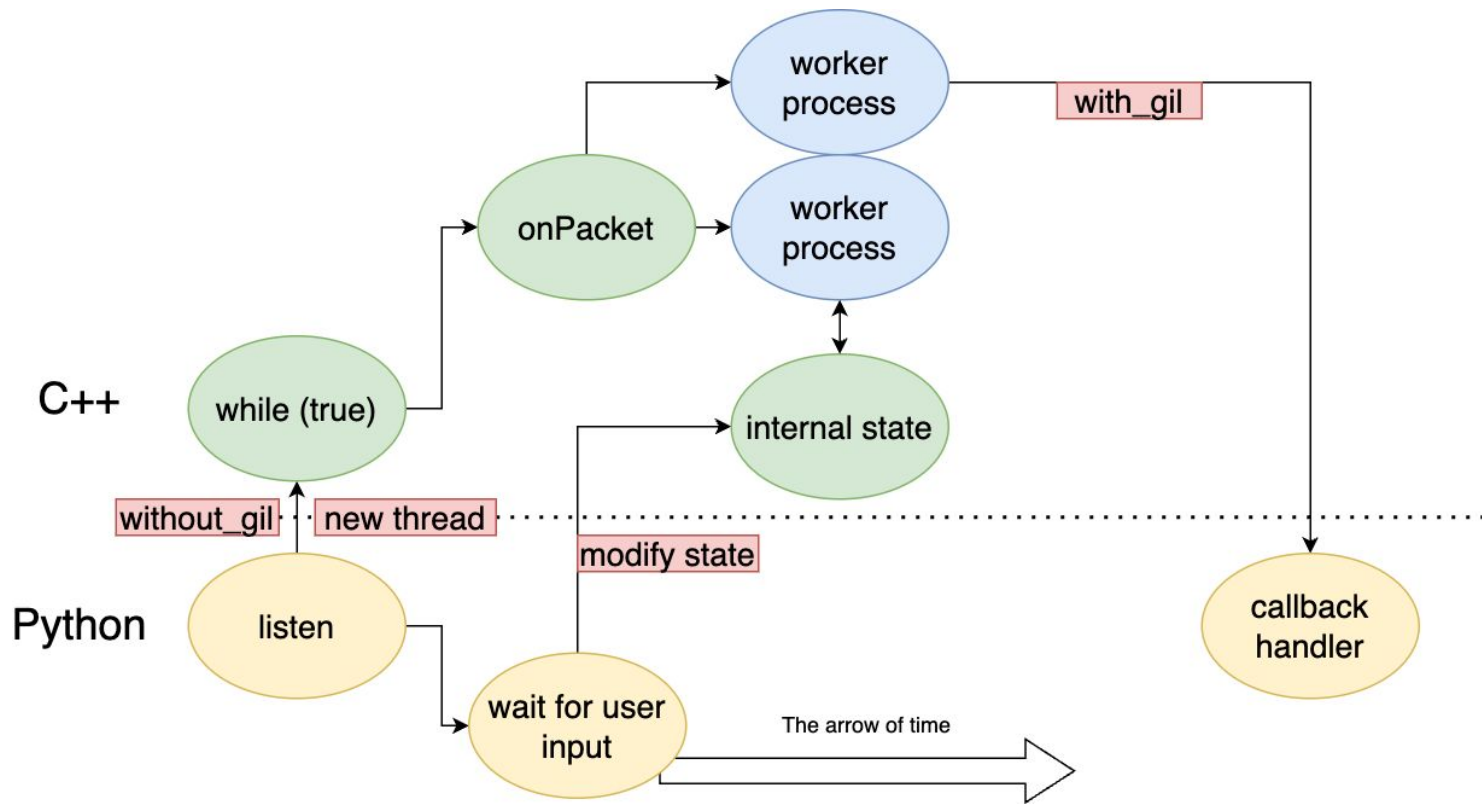
```
def print_packet(packet):  
    print(packet)  
  
my_network_library = MyNetworkLibrary()  
my_network_library.register_callback_on_packet(print_packet)  
my_network_library.listenInThread()  
  
while True:  
    print(get_resource_usage)  
    time.sleep(10)
```

Library is significantly more flexible now!

# COMPLEXITY LEVEL 4 : BACK AND FORTH

- Your network library maintains some connections to external processes (distributed computation)
- Python thread is used to interact with those external processes
  - Perhaps via some scripting
  - Perhaps the user wants to interactively interacting with the external processes

# COMPLEXITY LEVEL 4 : BACK AND FORTH





# COMPLEXITY LEVEL 4 : BACK AND FORTH

- Unsafe: python thread mutating C++ thread managed objects
- Use a lock in the C++ and python thread everywhere?
  - Pros and cons?
- Make C++ thread dispatcher wait for mutation instructions from python
  - How?

# COMPLEXITY LEVEL 4 : BACK AND FORTH

```
template <typename Ret>
Ret safe_dispatch(std::function<Ret(void)>&& fn) {
    cpp_chan->dispatch([fn, &done, &excep, &result]() mutable {
        try {
            result = boost::optional<Ret>{fn()};
        } // catch exceptions etc
    });
    while (!done) {    // done should be std::atomic<bool>
        usleep(1000);
    }
    return result.get();
}
```

# COMPLEXITY LEVEL 4 : BACK AND FORTH

```
bool MyPythonNetworkLibrary::subscribe(int listener) {  
    // Thread safe method on network library C++  
    return internals_->safe_dispatch<bool>(  
        [this]() { return internals_->subscribe(listener); }  
    );  
}
```

We have achieved `std::numeric_limits<PowerT>::max()`

# EXAMPLE 3

## OPTIMIZING PYTHON

### HOT LOOPS USING \*

# SOME ANALYSIS CODE

```
reader = mod.RowReader(f)
rows = reader.getRows()
sum_sig_change_by_trade_size, count, last_px = 0.0, 0, 0.0
for i, row in enumerate(rows):
    cur_trade_size = 0
    for item in row.items():
        id_name = reader.getIdName(item.id) # int => string
        ... some logic
print("average signal change is " + str(sum_sig_change_by_trade_size / count))
```

# SOME ANALYSIS CODE

```
if id_name == "timestamp":
    cur_trade_size = 0
if id_name == "trade_size":
    cur_trade_size += item.value()
if id_name == "signal":
    if last_px != 0:
        sig_change = abs(item.value() - last_px)
        if cur_trade_size > 0:
            sum_sig_change_by_trade_size += sig_change / cur_trade_size
        count += 1
    last_px = item.value()
    cur_trade_size = 0
```

# HOT LOOP OPTIMIZATION

- User wants to write arbitrary fast analysis
  - Could be written in Cython or Python
- You provide a pre-built .so for parsing files
- Maybe the user wants to write the hot loop themselves?
  - Perhaps we just failed as an API designer 😞?
  - Maybe the problem is just too generic
- Do we just give up?



# LET'S LOOK AT THIS AGAIN

```
reader = mod.RowReader(f)
rows = reader.getRows()
sum_sig_change_by_trade_size, count, last_px = 0.0, 0, 0.0
for i, row in enumerate(rows):
    cur_trade_size = 0
    for item in row.items():
        id_name = reader.getIdName(item.id) # int => string
        ... some logic
print("average signal change is " + str(sum_sig_change_by_trade_size / count))
```



# "TYPED" VS "UNTYPED" FOR LOOPS

```
for i, row in enumerate(rows):  
    cur_trade_size = 0  
    for item in row.items():  
        id_name = reader.getIdName(item.id) # int => string
```

- Python interpreter interprets some bytecode each time it hits the given loop
- Perhaps reduce the overhead by “compiling” this code?

# BRIEF DIGRESSION ABOUT CYTHON

- Cython compiles down your py script into C/C++ while retaining the exact same semantics
- The generated program can also do more things:
  - Deal with actual pointers and C++ data types
- The compiled program keeps most of the performance and dynamism of an interpreted language, and:
  - is now a C++ .so
  - is not an interpreted script

# CYTHON

```
/* for item in row.items(): */
__pyx_t_9 = 0;
if (unlikely(__pyx_v_row == Py_None)) {
    PyErr_Format(PyExc_AttributeError, "'NoneType' object has no attribute '%.30s'", "items");
    __PYX_ERR(0, 17, __pyx_L1_error)
}
__pyx_t_12 = __Pyx_dict_iterator(__pyx_v_row, 0, __pyx_n_s_items, (&__pyx_t_10), (&__pyx_t_11)); if (unlikely(!__pyx_t_12))
__PYX_ERR(0, 17, __pyx_L1_error)
__Pyx_GOTREF(__pyx_t_12);
__Pyx_XDECREF(__pyx_t_6);
__pyx_t_6 = __pyx_t_12;
__pyx_t_12 = 0;
while (1) {
    __pyx_t_13 = __Pyx_dict_iter_next(__pyx_t_6, __pyx_t_10, &__pyx_t_9, NULL, NULL, &__pyx_t_12, __pyx_t_11);
    if (unlikely(__pyx_t_13 == 0)) break;
    if (unlikely(__pyx_t_13 == -1)) __PYX_ERR(0, 17, __pyx_L1_error)
    .....
}
```

# CYTHON

```
while (1) {  
    __pyx_t_13 = __Pyx_dict_iter_next(  
        __pyx_t_6, __pyx_t_10, &__pyx_t_9, NULL, NULL, &__pyx_t_12, __pyx_t_11  
    );  
    if (unlikely(__pyx_t_13 == 0)) break;  
    if (unlikely(__pyx_t_13 == -1)) __PYX_ERR(0, 17, __pyx_L1_error)
```

- Performance does improve!
- Still has to do a lot of boilerplate to work around the lack of “type” information
- In my test, it went from 36.3s to 34.2s just in the hot loop (no IO included)

# CHEAPER FUNCTION CALL CONVENTIONS - TYPE INFORMED

```
for i, row in enumerate(rows):  
    cur_trade_size = 0  
    for item in row.items():  
        id_name = reader.getIdName(item.id) # int => string
```

- Writing python, interpreter interprets some bytecode each time it hits the given statement
  - It needs to handle a lot of possible eventualities
  - We profiled earlier, python function calls are roughly ~100ns
  - Can we inform the runtime that getIdName is always a simple function call?
- Can Cython do this?
  - No, just strips off the bytecode interpretation code

# POINTERS...AHEM...CAPSULES



- What if we had a function pointer that we could fetch early on, and call it in our hot loop?
- What if we could get typed objects out of our python module, and let cython generate type-informed code?

# POINTERS..AHM..CAPSULES



- PyCapsule\_New creates a new python object that wraps a void\* pointer and a string description of it.
- PyCapsule\_GetPointer takes a capsule object as input and returns the pointer
  - Only if you provide the same string description

```
PyObject *PyCapsule_New(void *pointer, const char *name);
```

```
void *PyCapsule_GetPointer(PyObject *capsule, const char *name);
```

# UGLY API TO EXPOSE POINTERS TO PYTHON

```
typedef std::string(get_id_name_ptr)(const void*, int);

boost::python::object getIdNameFunctionPtr() const {
    api::v1::get_id_name_ptr* fn(+)[](const void* self, int id) -> std::string {
        return static_cast<const RowReader*>(self)->getIdName(id);
    });
    PyObject* result = PyCapsule_New((void*)fn, "get_id_name_ptr_v1", NULL);
    return object(detail::new_reference(result));
}

boost::python::object getSelfPtr() const {
    PyObject* result = PyCapsule_New((void*)this, "row_reader_ptr", NULL);
    return object(detail::new_reference(result));
}
```



# FASTER FUNCTION CALLS USING TYPE INFORMATION

```
// user code
get_id_name_ptr = <string(*)(const void*, int) nogil> PyCapsule_GetPointer(
    reader.getIdNameFunctionPointer(), "get_id_name_ptr_v1")
self_ptr = <const void*> PyCapsule_GetPointer(
    reader.getSelfPtr(), "row_reader_ptr")
for i, row in enumerate(rows):
    cur_trade_size = 0
    for item in row.items():
        item_id = <int>item.id
        id_name = get_id_name_ptr(self_ptr, item_id)
        if id_name == string(<const char*>"timestamp"):
            cur_trade_size = 0
```

# CAPSULES

- As expected, performance is better
- 34.2s => 30.5s on a test file with 1 million rows:
  - Had 13000540 function calls
  - Saving ~300ns per function call, adds up!
- Can take this principle further for your highly latency sensitive applications
  - not useful everywhere, adds ugly pointers to python code

EXAMPLE 4  
OPEN SOURCE  
LIBRARIES

The NumPy logo consists of the word "NUMPY" in a white, hand-drawn, monospace-style font. The letters are slightly irregular and have a casual feel. The text is centered within a solid black rectangular background.

Optimized numeric operations for python using typed arrays

# BRIEF OVERVIEW OF NUMPY

- Abstraction over contiguous and multi-dimensional arrays exposed in Python (ndarray)
- Contains lots of optimized functions for common operations over such ndarrays
- Library's core written in C, the ndarray type and various common operations on it are implemented in C

```
In [7]: x = np.array([6, 7, 8]); print("{} , {}".format(type(x), x.dtype))  
<class 'numpy.ndarray'>, int64
```

```
In [8]: np.mean(x)  
Out[8]: 7.0
```

# BRIEF OVERVIEW OF NUMPY

- Source for subsequent slides:

<https://numpy.org/doc/stable/reference/c-api/types-and-structures.html>

- Example source code:

<https://github.com/numpy/numpy/blob/main/numpy/core/src/multiarray/arrayobject.c>

# LET'S DESIGN A N-DIMENSIONAL ARRAY (BRIEF)

How would you design a library exposing an API to an n-dimensional fixed size array?

- `void*` or `char*` to the start of an allocated memory region
- Store number and length of each dimension
- Store data type size
  - Use that to calculate and store some “strides” for each dimension
- Expose some n-d iterator API for it

# NUMPY TYPES

```
typedef struct PyArrayObject {  
    PyObject_HEAD  
    char *data;  
    int nd;  
    npy_intp *dimensions;  
    npy_intp *strides;  
    PyObject *base;  
    PyArray_Descr *descr;  
    int flags;  
    PyObject *weakreflist;  
    /* version dependent private members */  
} PyArrayObject;
```



# NUMPY TYPES

- The `PyArrayObject` is a valid `PyObject*` so can be constructed in python using pybinded methods
- Methods on this defined in `PyArray_Type` struct
- Various other C methods exposed by the library that can operate on these objects efficiently
- Usually constructed for simple C types (`float`, `double`, `int`, `int64`, etc)

# TAKEAWAYS

- NumPy implemented a complicated datatype and various methods in C
- Hot path is hidden behind a DSL using (potentially vectorizable) C methods
  - `np.mean`
  - `np.median`
  - `np.unique`
- What if we have a custom hot path and we want to loop over it in user written C++?
  - `boost::python::numpy` has a nice wrapper around numpy ndarrays!

# HOW DO I WRITE A PROGRAM THAT OPERATES ON NUMPY ARRAYS?

- Your library needs to interop with some ML library:
  - return a `std::vector<float>` and convert it to numpy array in python?
- Why not populate numpy arrays in your library?
  - `Boost::python` has easy constructors

```
#include <boost/python/numpy.hpp>
namespace bnp = boost::python::numpy;

Py_intptr_t shape[1] = {static_cast<long>(100)};
bnp::ndarray result = bnp::zeros(1, shape, bnp::dtype::get_builtin<T>());
// Example write operation (v is a source range)
std::copy(v.begin(), v.end(), reinterpret_cast<T*>(result.get_data()));
```

# PANDAS

Columnar data manipulation API built on top of numpy

# BRIEF OVERVIEW OF PANDAS

- Pythonic APIs built on top of numpy to add first class support for real world data types:
  - Enumerated values (called categoricals)
  - Timestamp values (datetime and timedeltas)
  - Arbitrary python objects
- Supports transformation of n-dimensional arrays:
  - Slicing out certain rows - creating views or a copied buffer
  - Slicing out certain columns - creating views or a copied buffer
- Think of it as a higher level DSL to optimize even more hot loops than numpy did.

# THE "HOW" OF PANDAS

```
cdef extern from "numpy/ndarrayobject.h":
    bint PyArray_CheckScalar(obj) nogil

@cython.wraparound(False)
@cython.boundscheck(False)
def maybe_booleans_to_slice(ndarray[uint8_t, ndim=1] mask):
    cdef:
        Py_ssize_t i, n = len(mask)
        Py_ssize_t start = 0, end = 0
        bint started = False, finished = False

    for i in range(n):
```

# THE “HOW” OF PANDAS

- Chose to write all code in Cython instead of C / C++
- Benefits:
  - Easy interop, Cython writes the bindings for your helper functions
  - Easy syntax, code looks very similar to python
  - Can import C / C++ constructs easily
- Why aren't we doing this?
  - Cannot do this with existing C++ code with complex build steps
  - But a pretty good idea for writing new code!
- Always good to know what is out there, even if not C++

# BUILDING EXTENSION MODULES



# GOING BACK TO NUMPY

- Any C++ code using numpy has to be built with a version compatible with the version in the python venv
- Python itself needs to be ABI compatible!
- Else, chaos!

```
#include <boost/python/numpy.hpp>
namespace bnp = boost::python::numpy;

Py_intptr_t shape[1] = {static_cast<long>(100)};
bnp::ndarray result = bnp::zeros(1, shape, bnp::dtype::get_builtin<T>());
// Example write operation (v is a source container)
std::copy(v.begin(), v.end(), reinterpret_cast<T*>(result.get_data()));
```

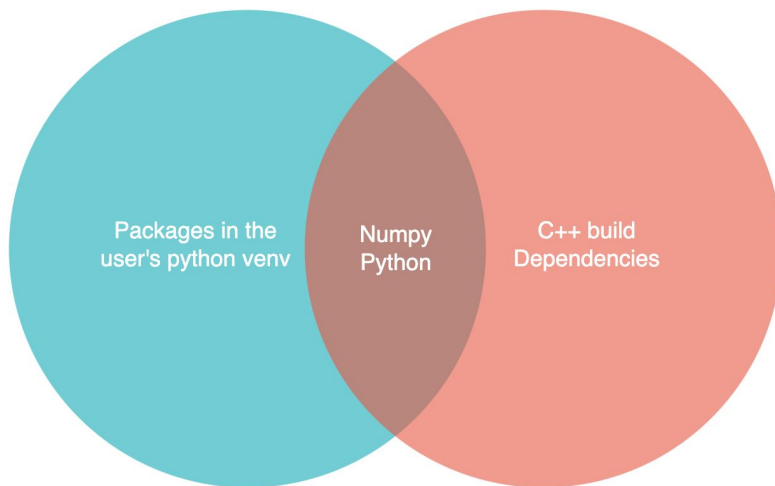
SO HOW DO WE RELEASE SUCH EXTENSION MODULES TO USERS?

# (1) WHEELS

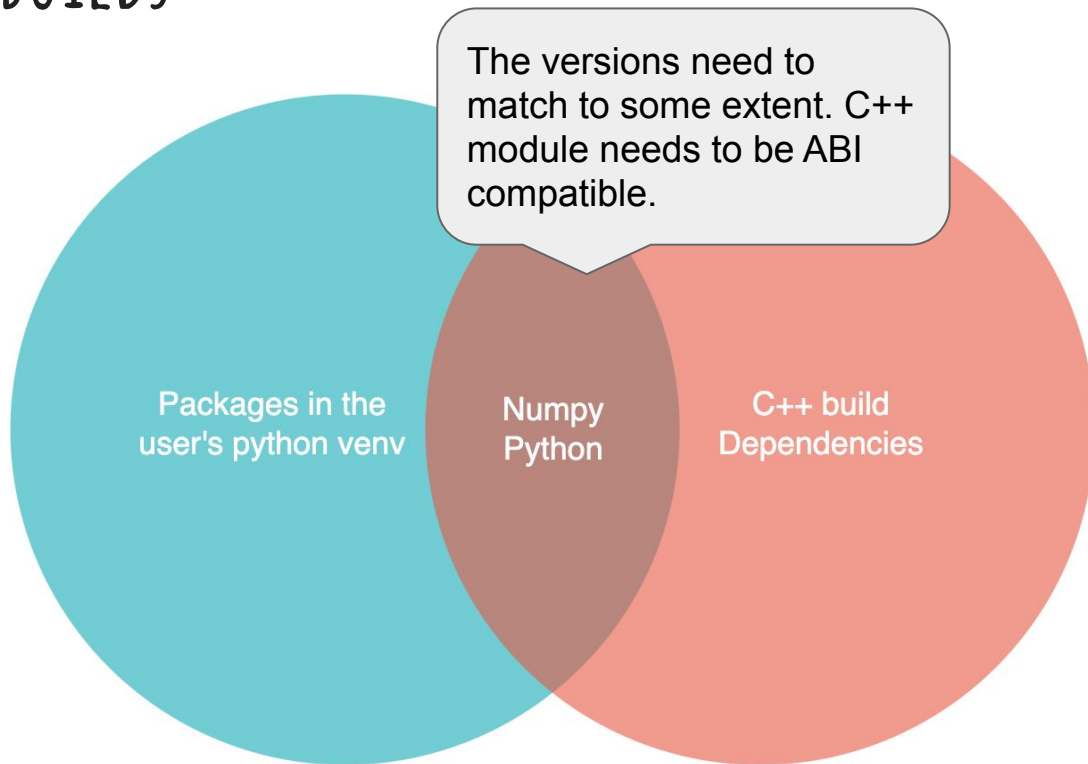
- Extension module is built with a minimum-stable-ABI requirement for CPython headers
- Can stay compatible for future python releases
- Discussion: <https://blog.trailofbits.com/2022/11/15/python-wheels-abi-abi3audit/>

## (2) SOURCE BUILDS

- The user builds the extension module with some context of the python environment it is supposed to run in
- Hard to keep python and C++ build dependencies consistent



## (2) SOURCE BUILDS



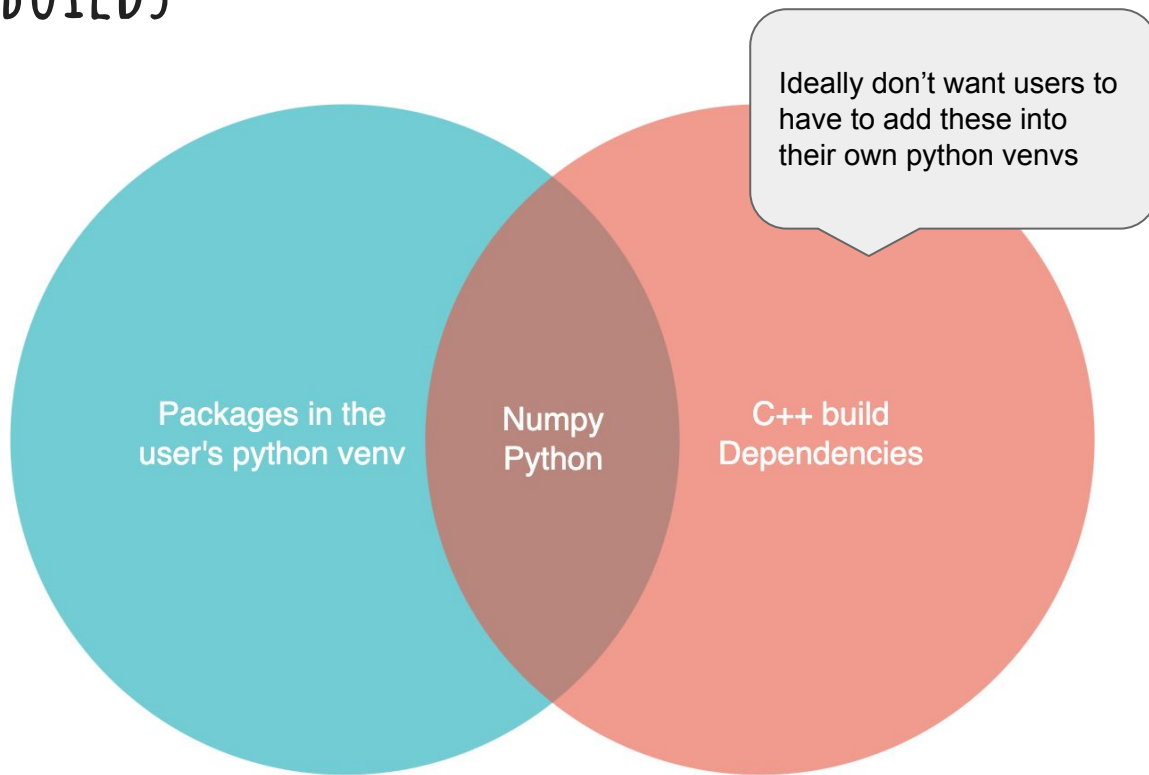
# GENERATE ENVIRONMENT SPEC FROM PYTHON VENV

- Your C++ library has certain dependencies
  - numpy, python, ...
- The user's python venv may have some of these libraries
- Bundle a script with your library that parses user's python venv
- Creates a build environment which has your C++ dependencies but pinned to versions already present in the user's python venv

# GENERATE ENVIRONMENT SPEC FROM PYTHON VENV

```
required_build_spec = [  
    "json={json}",  
    "llvm={llvm}",  
    "curl={libcurl}",  
    "lz4={lz4}",  
    "mongo-c-driver={mongo-c-driver}",  
    "numpy={numpy}", ....]  
  
def get_build_spec():  
    build_spec = []  
    versions = {entry["name"]: entry["version"] for entry in get_current_environment_spec()}  
    for b in required_build_spec:  
        b = b.format(**versions) # some extra stuff needs to be done to handle default versions  
        build_spec.append(b)
```

## (2) SOURCE BUILDS





# STATIC LINKING C++ DEPENDENCIES

- Build some libraries statically into your python binding
- Overall just reduces the number of jigsaw pieces we have to fit together to load the library into the python env
- For cmake:
  - Boost\_USE\_STATIC\_LIBS=ON, Protobuf\_USE\_STATIC\_LIBS=ON
- Some libraries are not that helpful, try this:

```
if ("${MYLIB_BUILD_STATIC}" STREQUAL "ON")  
  ADD_LIBRARY(MyLib::MyLib STATIC IMPORTED)  
  SET_TARGET_PROPERTIES(MyLib::MyLib PROPERTIES IMPORTED_LOCATION "${VENV_DIR}/lib/libmylib.a")  
endif()
```

# STATIC LINKING C++ DEPENDENCIES

- What if two extension modules use different versions of your statically linked libblah.a?
  - Potentially conflicting symbols loaded?
- What if:
  - extension module links statically to libblah.a
  - python virtual environment contains different version of libblah.so with potentially conflicting symbols

# STATIC LINKING C++ DEPENDENCIES

Two common ways of loading .so files into your program:

- `RTLD_GLOBAL`
  - The symbols defined by this library will be made available for symbol resolution of subsequently loaded libraries.
- `RTLD_LOCAL`
  - This is the converse of `RTLD_GLOBAL`, and the default if neither flag is specified. Symbols defined in this library are not made available to resolve references in subsequently loaded libraries.

(copied from `dlopen` manpage)

# STATIC LINKING C++ DEPENDENCIES

- Python import uses `RTLD_LOCAL` to load your dynamically built extension module
- `RTLD_GLOBAL` is pretty bad at diamond dependencies
  - Luckily for us we don't have to deal with this mostly
- Need static linking if:
  - Users' venv is not expected to have your library
  - Your library links to some library without an explicit rpath and users' venv may have a different version
- Need exact version match if:
  - Library is used on the API surface between C++ and Python

DISCUSSION / QUESTIONS