Writing Python Bindings for C++ Libraries:

Easy-to-use Performance

SAKSHAM SHARMA

Cppcon
The C++ Conference
2023
October 01 - 06

A quick bio - Yours truly

- Director, Quant Research Tech 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 CppCon :)

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
- Cython

C++

```cpp
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

```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

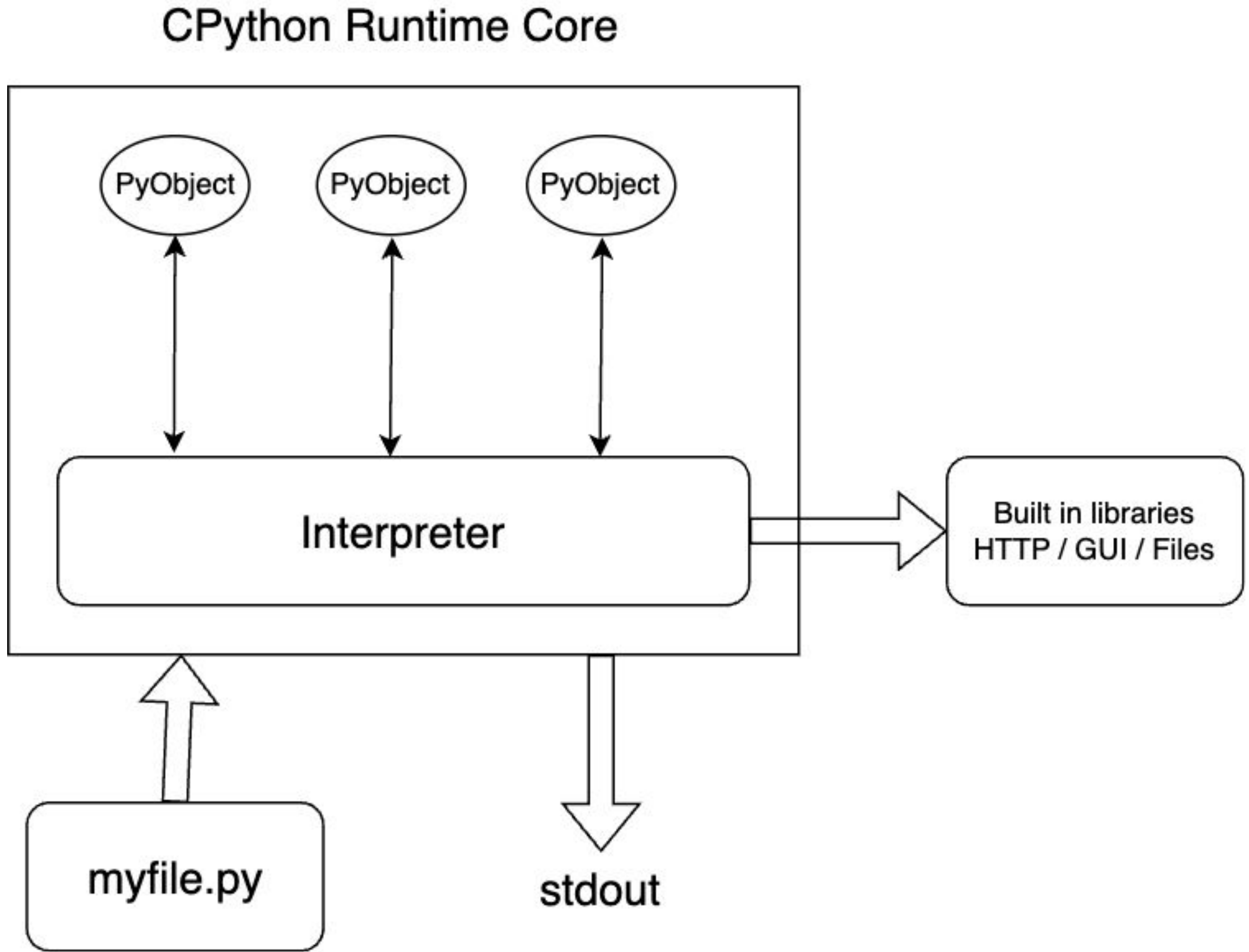
```
int cpp_increment(int i) { return i + 1; }

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)
```

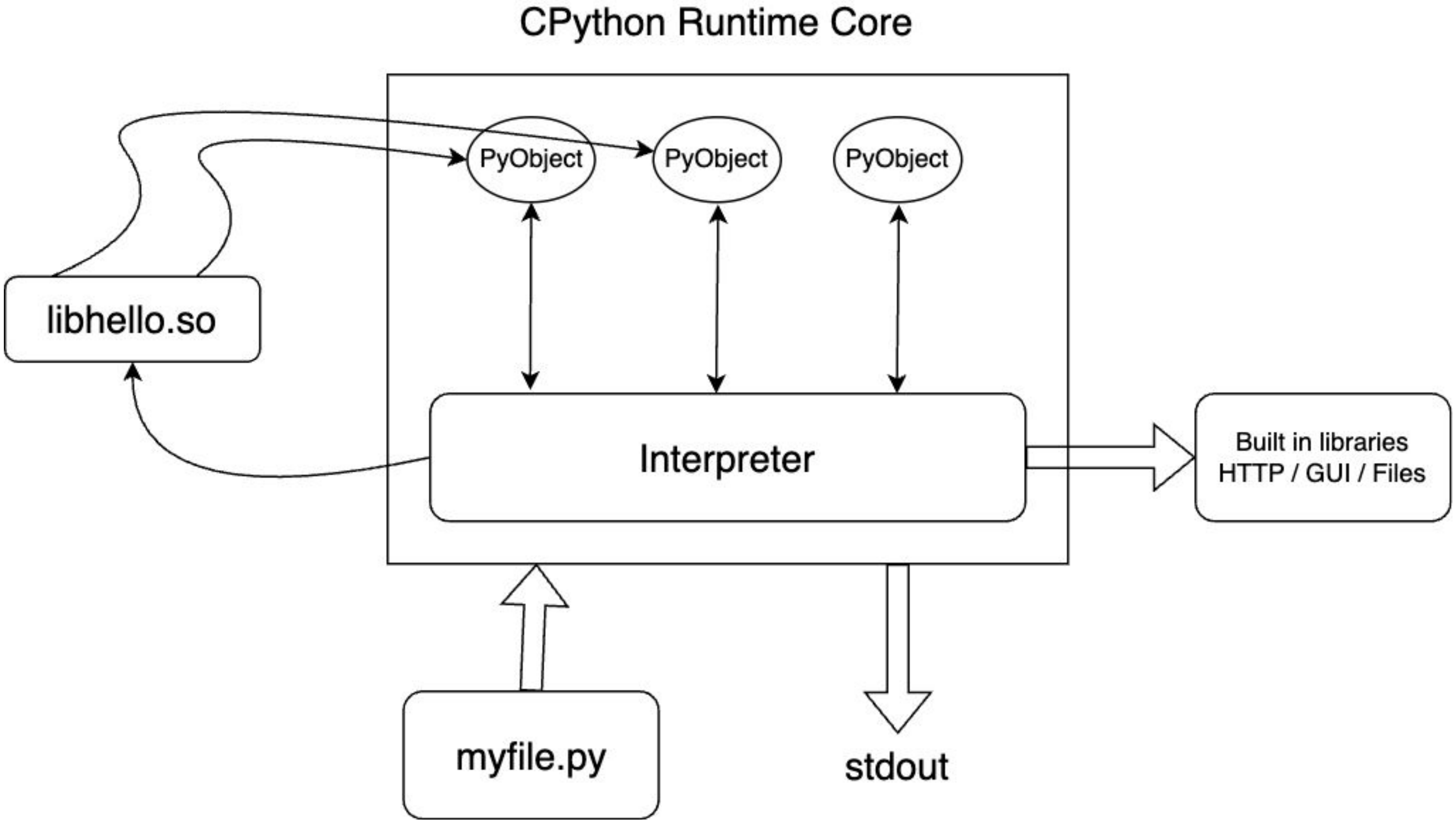
Wait, what?

Cython

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 PyObjects.
  - State of the program is maintained in these introspectable objects.


CPython Runtime Core
PyObject
PyObject
PyObject
Interpreter
Built in libraries
HTTP / GUI / Files
myfile.py
stdout


CPython Runtime Core
PyObject
PyObject
PyObject
libhello.so
Interpreter
Built in libraries
HTTP / GUI / Files
myfile.py
stdout

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".

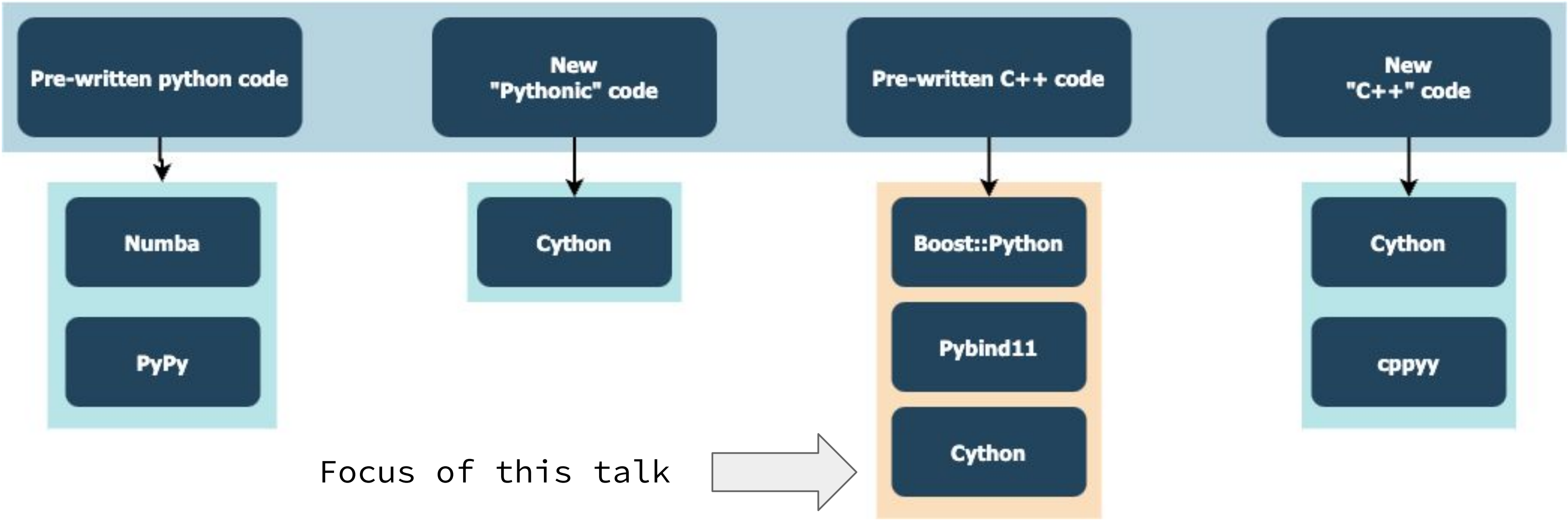
Python C API

```c
#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 HWMethods[] = {
    {"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, HWMethods
};
PyMODINIT_FUNC PyInit_hello_world_python(void) { return PyModule_Create(&hello_world_python); }
```

Cython

```
#cython: language_level=3
# distutils: language = c++

cdef extern from "cy.h":
    int cpp_increment(int x)

def cy_increment(x):
    return cpp_increment(x)
```

Various libraries to "speed up" python



Performance

Some perf numbers: per increment runtime

Take these numbers with a grain of salt

Some perf numbers: per increment runtime

Implementation	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
Cython	476ns	119ns	57ns	54ns	61K

*: 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 cause seems to be dynamic memory allocation 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.

Class 101

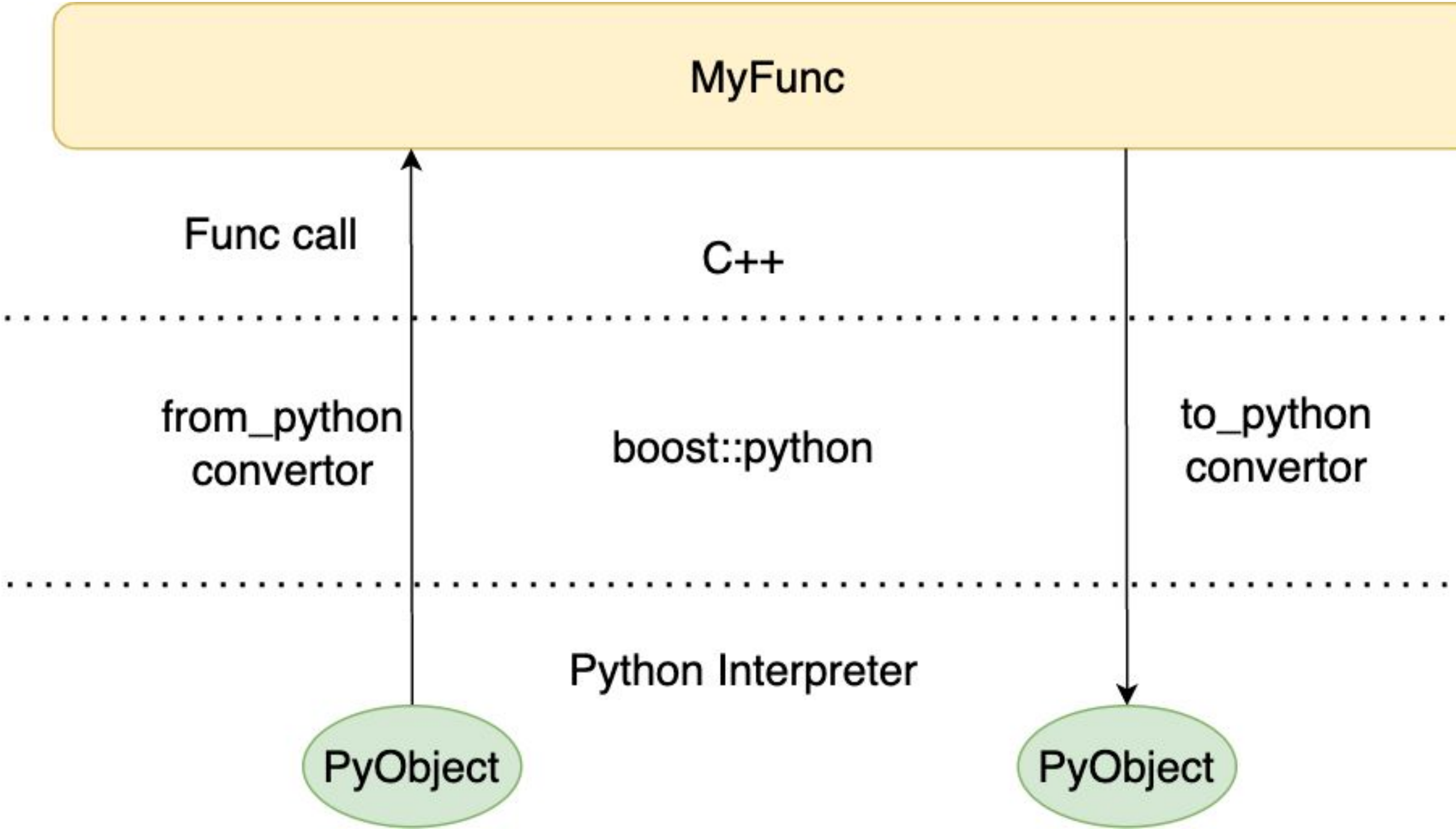
Classes?

```
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_internal_reference<>());
   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!


MyFunc
Func call
C++
from_python
convertor
boost::python
to_python
convertor
Python Interpreter
PyObject
PyObject

Classes!

```python
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

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 path”
- 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 keep it around
  - Why would we want to do this?

Arguments to functions - PyObjects

```cpp
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 useful
  - boost::python just makes it easy to create refcounted PyObjects
  - Even more optimal if your code just populates the python data structure instead of doing a conversion
- What about larger objects?
  - Large vectors, matrices?
  - Python libraries aren't written to operate on std::vector<float>

Return values of functions - numpy

```cpp
#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;
}
```

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

```python
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;
};
```

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;
};
```

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_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);
   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_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);
   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);
```

Object lifetime

(2) Binding code - lifetime of returned objects

```
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_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);
   register_exception_translator<std::runtime_error>(translate_error);
}
```

(2) Binding code - lifetime of returned objects

```
struct Row {
   std::vector<Item> items_;
   const auto& items() const { return items_; }
};

// Declaration:
.def("items", &RowRef::items, return_internal_reference<>());
```

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

(2) Binding code - lifetime of returned objects

```
.def("items", &RowRef::items, return_value_policy<reference_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
- This API is also unsafe in C++

(2) Binding code - lifetime of returned objects

```cpp
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 - lifetime of returned objects

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

Lists

(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_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);
   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[]
  - val = myvec[i]
  - myvec[i].mutate()
  - val == myvec[i] ??
- 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_internal_reference<>())
```

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

Constructors

(4) Binding code - Constructors

```
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);
   register_exception_translator<std::runtime_error>(translate_error);
}
```

(4) Binding code - Constructors

```
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

Exceptions

(5) Binding code - Exceptions

```cpp
void translate_error(const std::runtime_error& e) {
   PyErr_SetString(PyExc_RuntimeError, e.what());  // This much is already done by default
}
register_exception_translator<std::runtime_error>(translate_error);
```

- Exceptions are thread local error flags
- Exception handlers are at the interpreter level

(5) 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
```

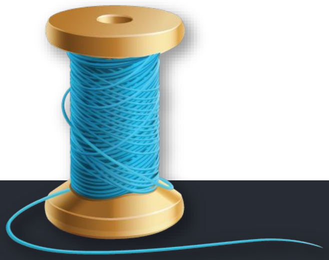
- Existing libraries may have ::abort / ::exit calls in macros
- Use #ifdef macros to switch your exit macros to throw exceptions instead

(5) Binding code - Exceptions

- Remember, exceptions are still expensive!
  - Especially C++ exceptions
- Use judiciously!
- In my tests:
  - Raising exception in boost::python, catching, and setting py exception is ~9-10us
  - Setting python exception by hand using Python C API is 250-500ns
  - Of course, these are microbenchmarks
- https://stackoverflow.com/questions/13835817/are-exceptions-in-c-really-slow

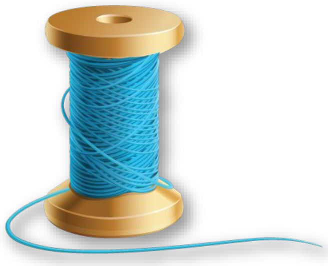
Threading

(6) Binding code - Threading



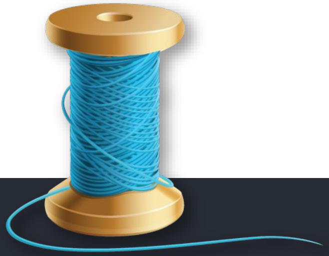
```
RowReader(const std::string& filename) : reader_(filename) {
    reader_.addListener(this);
    reader_.readAll();
}

// 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]
```

(6) Binding code - Threading



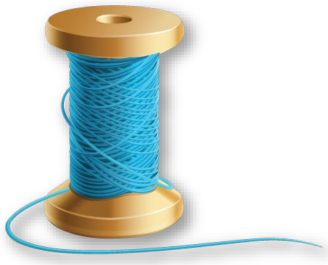
- Python has a Global Interpreter Lock (GIL)
- A blocking IO call that does not deal with any python objects should be allowed to release the lock
- Python's internal IO libraries already do this
- We'll discuss nuances of this in a later example

(6) 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;
  ....
}
```

(6) 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()
```

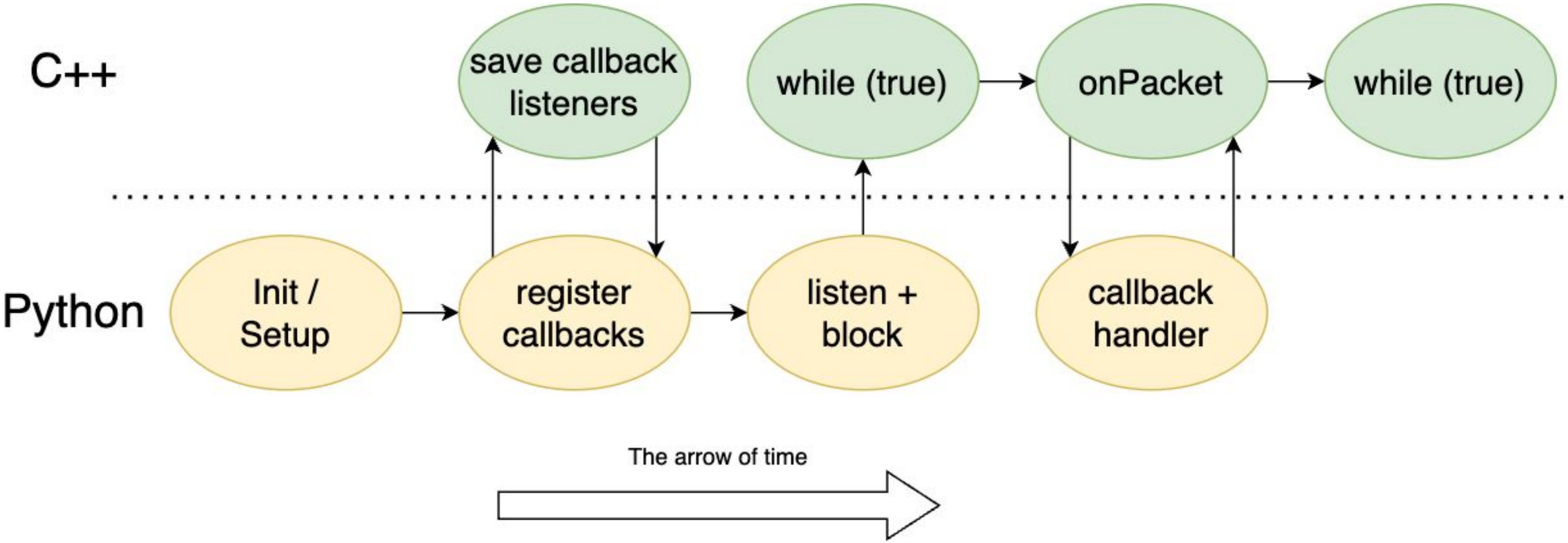
- Callbacks are a common theme in many dispatcher based applications
- Dispatching may happen in:
  - The main python thread, but blocked
  - A separate thread, thus letting the python thread run

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 dispatchPacketInfoToPython(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

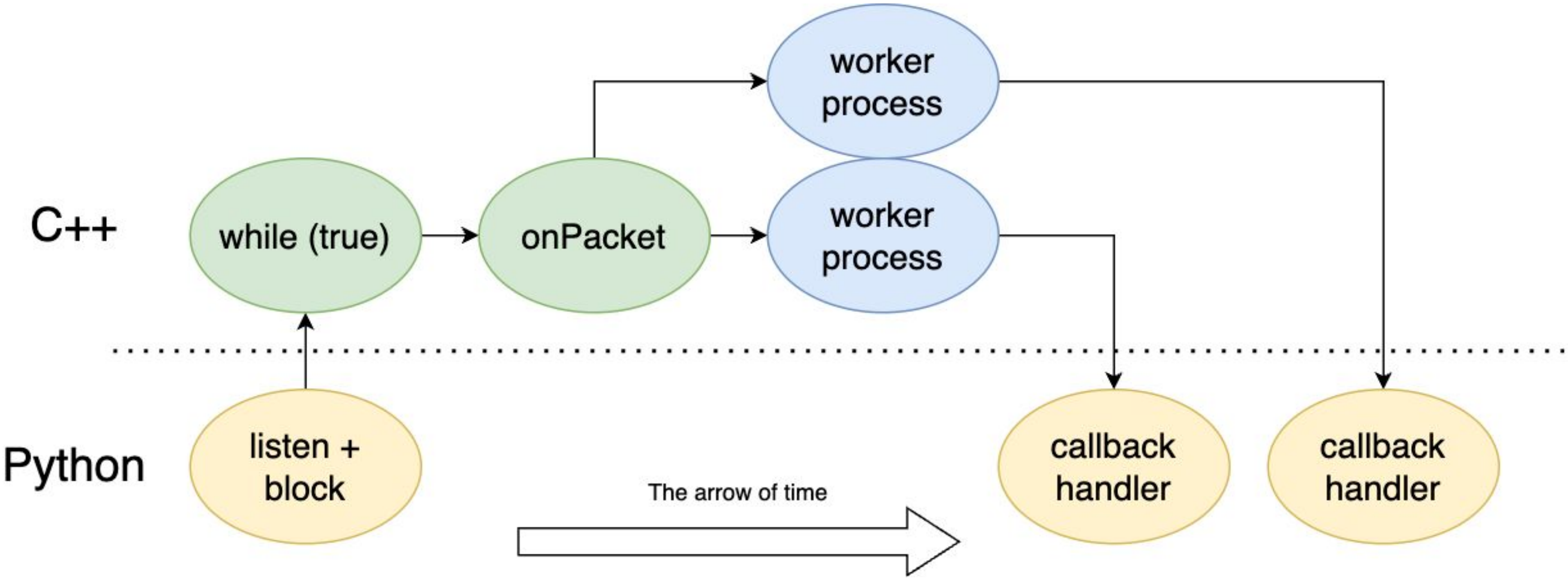
Complexity level 1 : Everything in the same thread



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

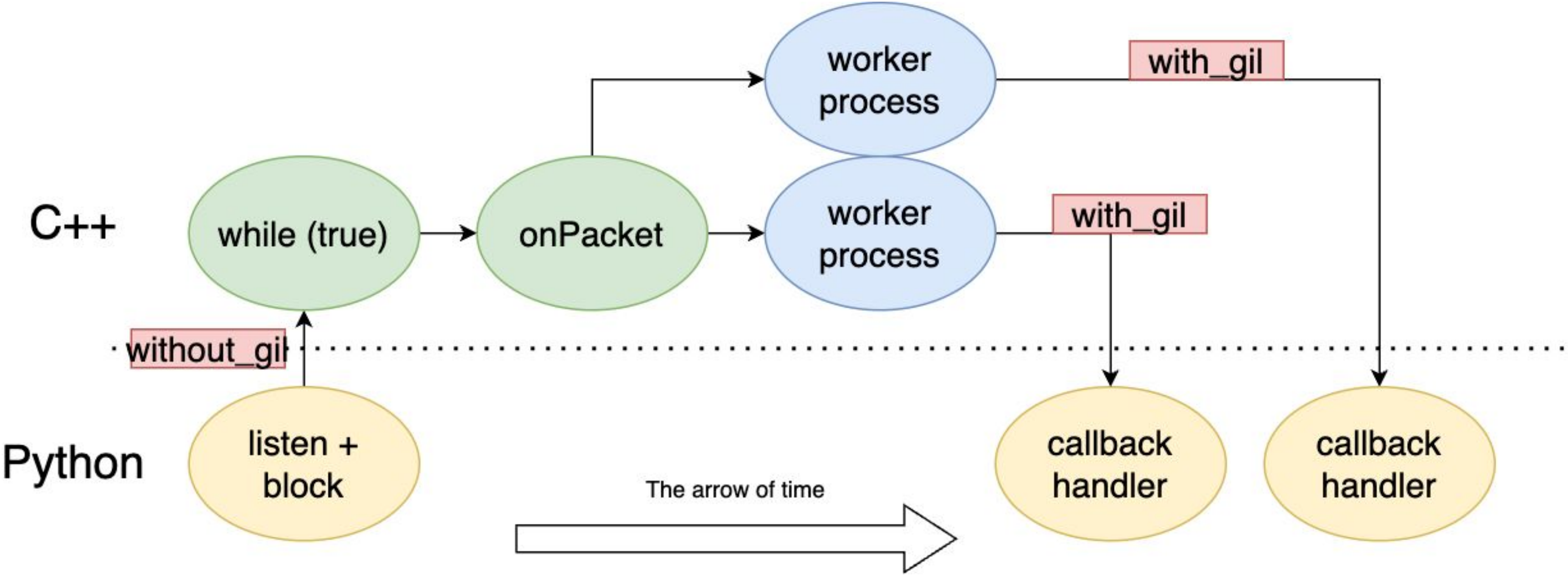
Complexity level 2 : Multiple worker threads



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

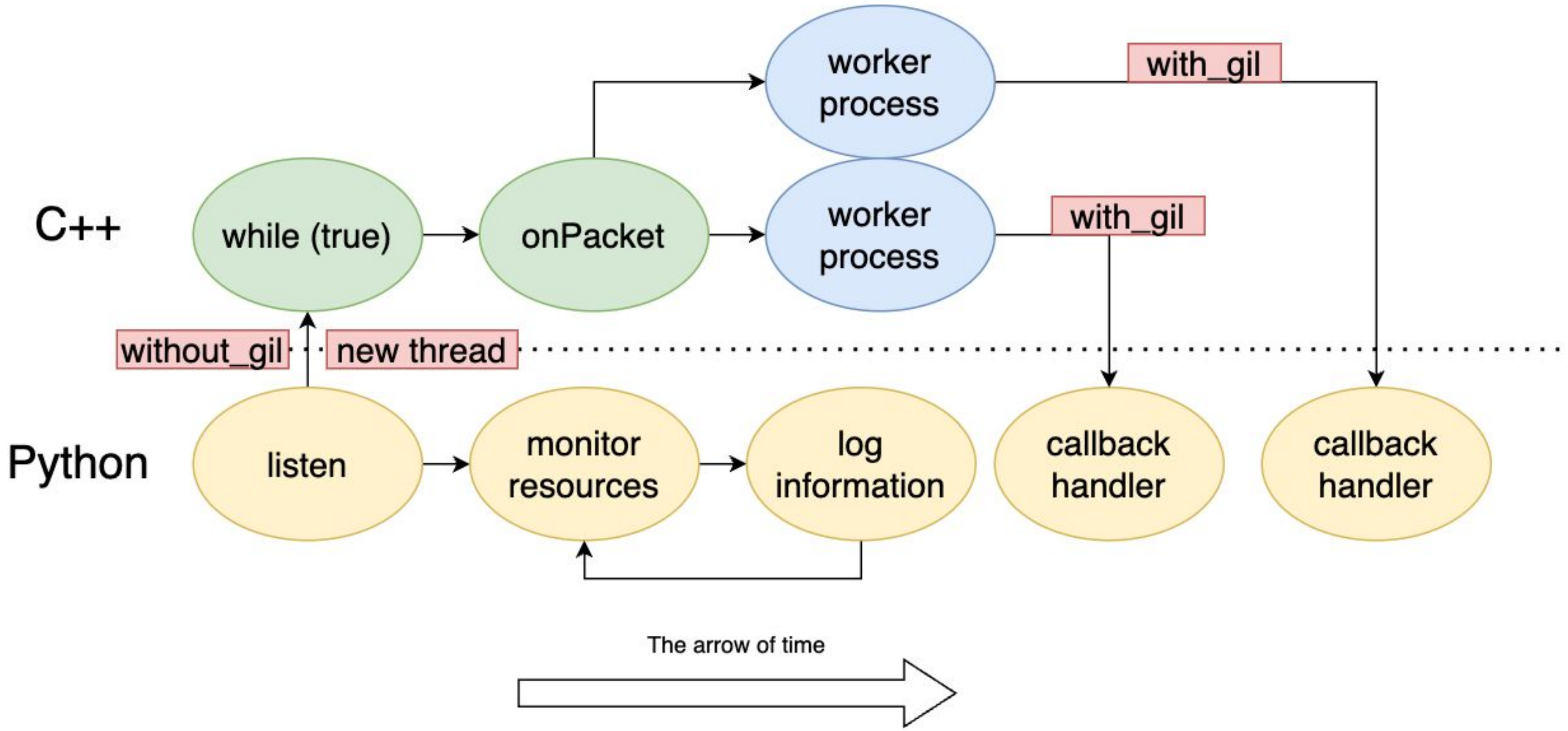
Complexity level 2 : Multiple worker threads



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

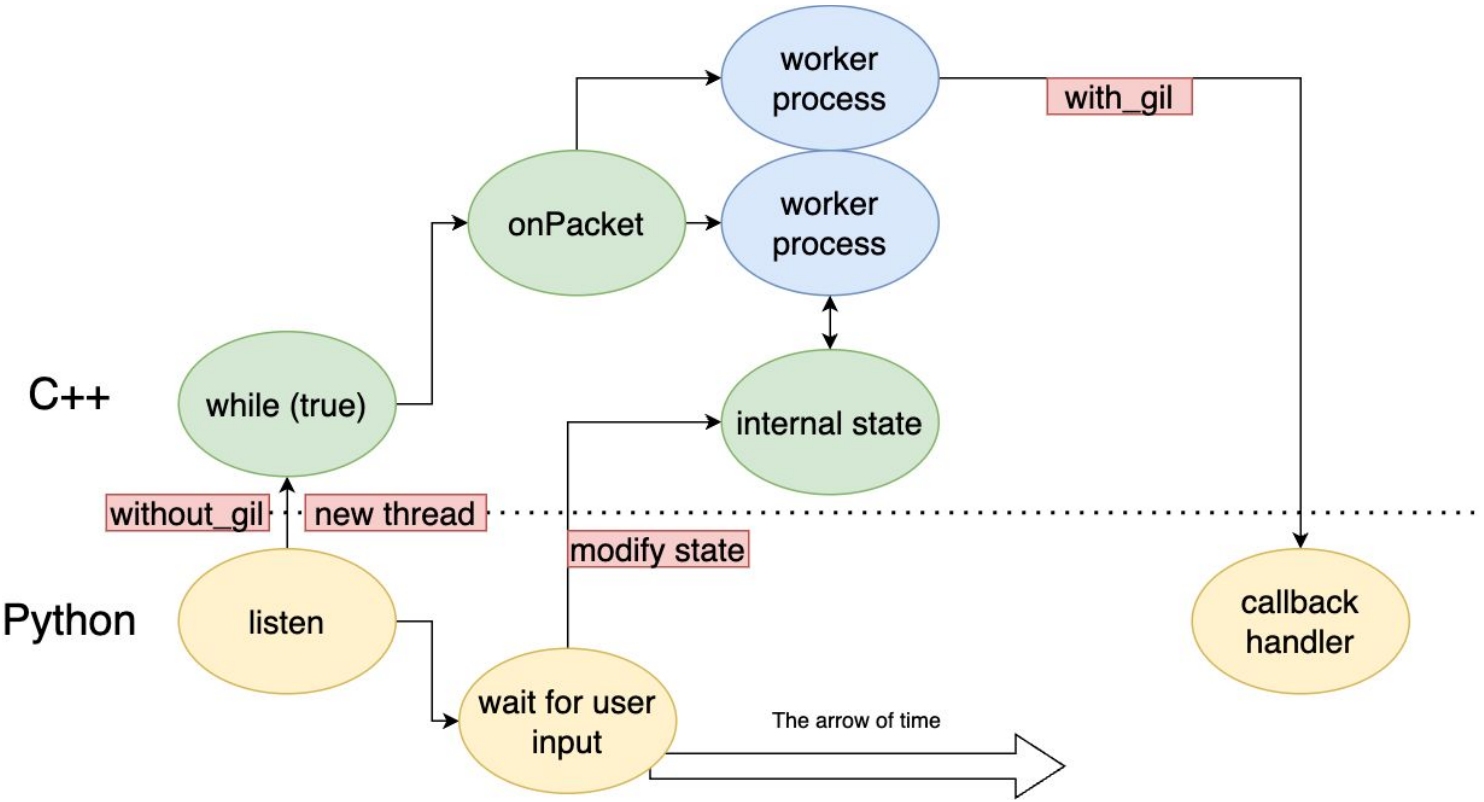
```python
def print_packet(packet):
  print(packet)
my_network_library = MyNetworkLibrary()
my_network_library.register_callback_on_packet(print_packet)
my_network_library.listenInThread()
while True:
    print(get_resource_usage)
    time.sleep(10)
```

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 wait for mutation instructions from python
    - How?

Complexity level 4 : back and forth

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

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

```python
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 path 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

```python
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 O(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?


SEGFAULTS!
I CHOOSE YOU!
imgflip.com

Pointers..ahem..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

NUMPY

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

Implementing PyObject interface for custom types

```
typedef struct PyCustomObject {
    PyObject_HEAD
    std::shared_ptr<CustomType> underlying_obj; /* could be held without shared_ptr too */
    /* py version dependent private members */
} PyCustomObject;
```

- Boost::python / pybind11 do this for us

Numpy types

```c
typedef struct PyArrayObject {
    PyObject_HEAD
    char *data;
    int nd;
    npy_intp *dimensions;
    npy_intp *strides;
    PyObject *base;
    PyArray_Descr *descr;
    int flags;
    PyObject *weakreflist;
    /* py 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 path loops than numpy did.

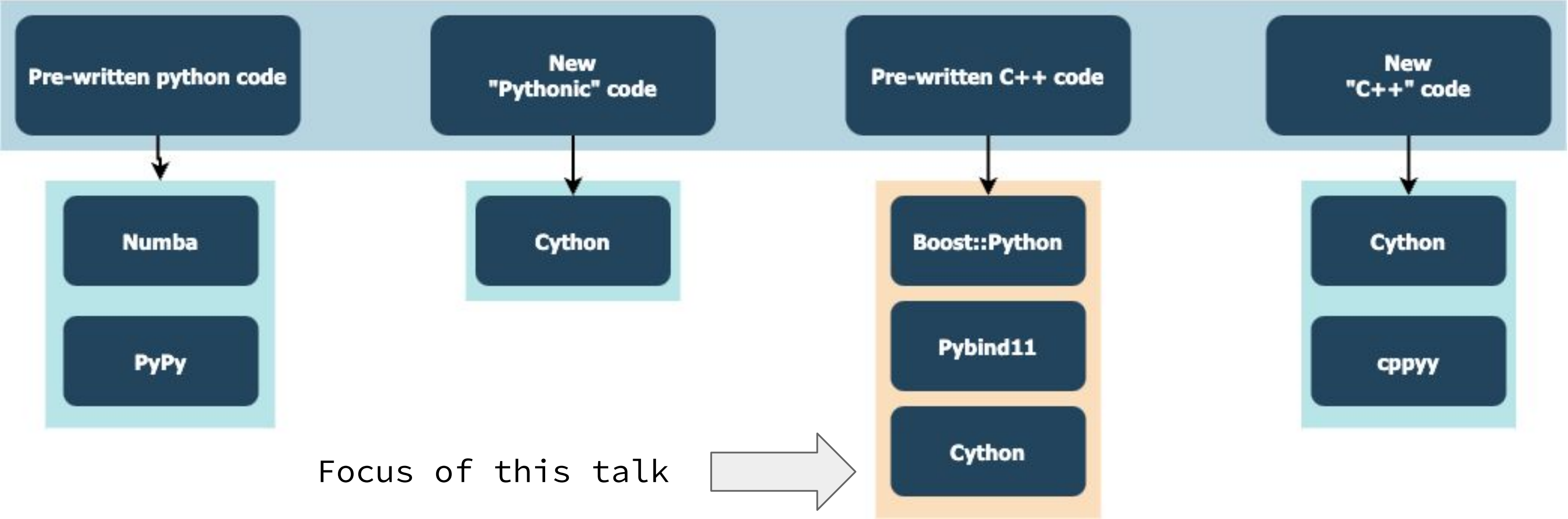
The "how" of pandas

```
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?
  - Adds a new sourcegen dependency into your build system
  - Learning a new "language" / different debugging experience
  - But a pretty good idea for writing new code!
- Always good to know what is out there, even if not C++

Various libraries to "speed up" python



Pros and cons

	C/C++ native	Documentation	Latency 101	Debugging
Boost::python	✅	😕	✅	✅
Pybind11	✅	✅	😕	✅
Cython	?	✅	✅	😕

References

- https://boostorg.github.io/python/doc/html/index.html
- https://erikerlandson.github.io/algorithm/libs/python/doc/v2/indexing.html
- https://stackoverflow.com/questions/27077518/
- https://pybind11.readthedocs.io/
- https://cython.org/
- https://docs.python.org/3/c-api/index.html

THANK YOU!

Discussion / questions