pybind11

Seamless operability between C++11 and Python

Ivan Smirnov

July 14, 2017

Susquehanna International Group

europython 2017

Introduction

Extension modules

CPython *extension module*: Python module not written in Python.

Most often written in C or C++.

Why bother?

- Interfacing with existing libraries
- Writing performance-critical code
- Mirroring library API in Python to aid prototyping
- Running tests for non-Python libraries in Python

Python C API

It is possible to write CPython extension modules in pure C, but...

- Manual refcounting
- Manual exception handling
- Boilerplate to define functions and modules
- High entry barrier, prone to programmer errors
- Differences in the API between Python versions

Cython

Cython: "let's write C extensions as if it was Python".

Why not?

- It's neither C nor Python
- A 2-line Cython module can be transpiled into 2K lines of C
- Two build steps (`.pyx` → `.c`, `.c` → `.so`); poor IDE support
- Limited C++ support (scoped enums, non-type template parameters, templated overloads, variadic templates, universal references, etc).
- Limited support for generic code beyond fused types
- Have to create stubs for anything outside standard library
- Great for wrapping a few functions, not so great for large codebases
- Debugging compiled Cython extensions is **pain**

Cython

```
def f(n: int):
    for i in range(n):  # <-------
        pass
```

↓

```
$ cythonize example.pyx
```

↓

```
 /\____/\
( o   o )
(  =^=  )
(        )
(         )
(          ))))))))))
```

Cython

```
/* "example.pyx":2
 * def f(n: int):
 *     for i in range(n):             # <<<<<<<<<<<<<<
 *         pass
 *
 */
  __pyx_t_1 = PyTuple_New(1); if (unlikely(!__pyx_t_1)) __PYX_ERR(0, 2, __pyx_L1_error)
  __Pyx_GOTREF(__pyx_t_1);
  __Pyx_INCREF(__pyx_v_n);
  __Pyx_GIVEREF(__pyx_v_n);
  PyTuple_SET_ITEM(__pyx_t_1, 0, __pyx_v_n);
  __pyx_t_2 = __Pyx_PyObject_Call(__pyx_builtin_range, __pyx_t_1, NULL); if (unlikely(!__pyx_t_2)) __PYX_ERR(0, 2, __pyx_L1_error)
  __Pyx_GOTREF(__pyx_t_2);
  __Pyx_DECREF(__pyx_t_1); __pyx_t_1 = 0;
  if (likely(PyList_CheckExact(__pyx_t_2)) || PyTuple_CheckExact(__pyx_t_2)) {
    __pyx_t_1 = __pyx_t_2; __Pyx_INCREF(__pyx_t_1); __pyx_t_3 = 0;
    __pyx_t_4 = NULL;
  } else {
    __pyx_t_3 = -1; __pyx_t_1 = PyObject_GetIter(__pyx_t_2); if (unlikely(!__pyx_t_1)) __PYX_ERR(0, 2, __pyx_L1_error)
    __Pyx_GOTREF(__pyx_t_1);
    __pyx_t_4 = Py_TYPE(__pyx_t_1)->tp_iternext; if (unlikely(!__pyx_t_4)) __PYX_ERR(0, 2, __pyx_L1_error)
  }
  __Pyx_DECREF(__pyx_t_2); __pyx_t_2 = 0;
  for (;;) {
    if (likely(!__pyx_t_4)) {
      if (likely(PyList_CheckExact(__pyx_t_1))) {
        if (__pyx_t_3 >= PyList_GET_SIZE(__pyx_t_1)) break;
        #if CYTHON_ASSUME_SAFE_MACROS && !CYTHON_AVOID_BORROWED_REFS
        __pyx_t_2 = PyList_GET_ITEM(__pyx_t_1, __pyx_t_3); __Pyx_INCREF(__pyx_t_2); __pyx_t_3++; if (unlikely(0 < 0)) __PYX_ERR(0, 2, __pyx_L1_error)
        #else
        __pyx_t_2 = PySequence_ITEM(__pyx_t_1, __pyx_t_3); __pyx_t_3++; if (unlikely(!__pyx_t_2)) __PYX_ERR(0, 2, __pyx_L1_error)
        __Pyx_GOTREF(__pyx_t_2);
        #endif
      } else {
        if (__pyx_t_3 >= PyTuple_GET_SIZE(__pyx_t_1)) break;
        #if CYTHON_ASSUME_SAFE_MACROS && !CYTHON_AVOID_BORROWED_REFS
        __pyx_t_2 = PyTuple_GET_ITEM(__pyx_t_1, __pyx_t_3); __Pyx_INCREF(__pyx_t_2); __pyx_t_3++; if (unlikely(0 < 0)) __PYX_ERR(0, 2, __pyx_L1_error)
        #else
        __pyx_t_2 = PySequence_ITEM(__pyx_t_1, __pyx_t_3); __pyx_t_3++; if (unlikely(!__pyx_t_2)) __PYX_ERR(0, 2, __pyx_L1_error)
        __Pyx_GOTREF(__pyx_t_2);
        #endif
      }
    } else {
      __pyx_t_2 = __pyx_t_4(__pyx_t_1);
      if (unlikely(!__pyx_t_2)) {
        PyObject* exc_type = PyErr_Occurred();
        if (exc_type) {
          if (likely(exc_type == PyExc_StopIteration || PyErr_GivenExceptionMatches(exc_type, PyExc_StopIteration))) PyErr_Clear();
          else __PYX_ERR(0, 2, __pyx_L1_error)
        }
        break;
      }
      __Pyx_GOTREF(__pyx_t_2);
    }
    __Pyx_XDECREF_SET(__pyx_v_i, __pyx_t_2);
    __pyx_t_2 = 0;
  }
  __Pyx_DECREF(__pyx_t_1); __pyx_t_1 = 0;
```

Cython

```
def f(n: int):
    cdef int i
    for i in range(n):
        pass
```

```
__pyx_t_1 = __Pyx_PyInt_As_long(__pyx_v_n);
if (unlikely((__pyx_t_1 == (long)-1) && PyErr_Occurred()))
  __PYX_ERR(0, 3, __pyx_L1_error)
for (__pyx_t_2 = 0; __pyx_t_2 < __pyx_t_1; __pyx_t_2+=1) {
  __pyx_v_i = __pyx_t_2;
}
```

Boost.Python

Why not?

- Requires building `boost_python` library
- . . . (!) which requires Boost (full library is 1.5M LOC headers)
- . . . and uses SCons for building (Python2-only build tool)
- Relies heavily on Boost.MPL due to being stuck in C++03
- . . . so large extension modules may take very long to compile
- . . . and resulting binaries may end up being very big
- $<$ 200 commits in the last 5 years

(still a great library if you're already using Boost; `pybind11`'s syntax and initial API design were heavily influenced by Boost.Python)

pybind11

`pybind11` is a lightweight header-only library that allows interacting with Python interpreter and writing Python extension modules in modern C++.

- Header-only; no dependencies; doesn't require specific build tools
- 5K LOC core codebase (8K entire library)
- Heavily optimized for binary size; fast compile time
- GCC, Clang, MSVS or ICC (Linux, macOS, Windows)
- CPython 2.7, CPython 3.x, PyPy
- Support for C++11, C++14 and C++17 language features
- Support for NumPy without having to include NumPy headers
- Support for embedding Python interpreter
- STL data types, overloaded functions, enumerations, callbacks, iterators and ranges, single and multiple inheritance, smart pointers, custom operators, automatic refcounting, capturing lambdas, function vectorization, arbitrary exception types, virtual class wrapping, etc ...

Link: `http://github.com/pybind/pybind11`

Hello, World!

First things first

Requirements:

- CPython 2.7.x, 3.x or PyPy $\geqslant$ 5.7, with headers
- `pybind11` package installed (`pip install pybind11`)
- Non-ancient compiler (Clang $\geqslant$ 3.3, GCC $\geqslant$ 4.8, MSVS $\geqslant$ 2015)

Boilerplate (will be omitted in most further examples):

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

namespace py = pybind11;

PYBIND11_MODULE(example, m) {
    ...
}
```

Let's write a module

Let's bind a C function that adds two integers:

```
int add(int a, int b) {
    return a + b;
}

PYBIND11_MODULE(myadd, m) {
    m.def("add", &add, "Add two integers.");
}
```

. . . or, C++11 style:

```
PYBIND11_MODULE(myadd, m) {
    m.def("add", [](int a, int b) { return a + b; },
          "Add two integers.");
}
```

Trying it out

After the code is compiled, it can be used like a normal Python module:

```
>>> from myadd import add

>>> help(add)
add(arg0: int, arg1: int) -> int

Add two integers.

>>> add(1, 2)
3

>>> add('foo', 'bar')
TypeError: add(): incompatible function arguments.
The following argument types are supported:
    1. (arg0: int, arg1: int) -> int

Invoked with: 'foo', 'bar'
```

Compiling a module

There's a few possible ways to build a `pybind11` module...

Compiling a module – manually

Linux (Python 3):

```
$ c++ -O3 -shared -std=c++11 -fPIC
    $(python -m pybind11 --includes) myadd.cpp
    -o myadd$(python3-config --extension-suffix)
```

If the build succeeds, it will create a binary module like this:

```
myadd.cpython-36m-x86_64-linux-gnu.so
```

macOS: same as above, plus `-undefined dynamic_lookup` flag.

Windows: possible but not fun.

Compiling a module – distutils

Integrating into `setup.py`:

```
from setuptools import setup, Extension
from setuptools.command import build_ext
from pybind11 import get_include

setup(
    ...,
    ext_modules=[
        Extension(
            'myadd', ['myadd.cpp'],
            include_dirs=[get_include()],
            language='c++',
            extra_compile_args=['-std=c++11']
        )
    ],
    cmdclass={'build_ext': build_ext.build_ext}
)
```

Compiling a module – ipybind

In IPython console or Jupyter notebook (requires installing `ipybind`):

```
%load_ext ipybind
```

```
%%pybind11

PYBIND11_MODULE(myadd, m) {
    m.def("add", [](int a, int b) { return a + b; },
          "Add two integers.");
}
```

After the module is built, its contents are imported automatically:

```
>>> add(1, 2)
3
```

Compiling a module – CMake

In a CMake project:

```
pybind11_add_module(myadd myadd.cpp)
```

Simple classes

C++ example class

Let's create Python bindings for a simple HTTP response class:

```
#include <string>

struct Response {
    int status;
    std::string reason;
    std::string text;

    Response(int status,
             std::string reason,
             std::string text = "")
        : status(status)
        , reason(std::move(reason))
        , text(std::move(text))
    {}

    Response() : Response(200, "OK")  {}
};
```

Binding the type

```
struct Response {
    ...
}
```

↓

```
PYBIND11_MODULE(response, m) {
    py::class_<Response>(m, "Response");
}
```

Constructors

```
struct Response {
    ...
    Response(int status,
             std::string reason,
             std::string text = "");
    Response();
};
```

↓

```
py::class_<Response>(m, "Response")
    .def(py::init<>())
    .def(py::init<int, std::string>())
    .def(py::init<int, std::string, std::string>());
```

Instance attributes

```
struct Response {
    ...
    int status;
    std::string reason;
    std::string text;
};
```

```
py::class_<Response>(m, "Response")
    ...
    .def_readonly("status", &Response::status)
    .def_readonly("reason", &Response::reason)
    .def_readonly("text", &Response::text);
```

Properties

```
struct Response {
    ...
    bool ok() const {
        return status >= 200 && status < 400;
    }
};
```

↓

```
py::class_<Response>(m, "Response")
    ...
    .def_property_readonly("ok", &Response::ok);
```

Operators

```cpp
bool operator==(const Response& r1,
                const Response& r2) {
    return r1.status == r2.status
        && r1.reason == r2.reason
        && r1.text == r2.text;
}
```

```cpp
py::class_<Response>(m, "Response")
    ...
    .def("__eq__", [](const Response& self,
                      const Response& other) {
        return self == other;
    });
```

Operators (`py::self`)

Wrapping operators is a very common thing to do:

```
...

.def("__eq__", [](const Response& self,
                  const Response& other) {
    return self == other;
}, py::is_operator())
```

. . . so there's a shortcut:

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

...

.def(py::self == py::self)
```

(also works with arithmetic operators, binary operators, etc.)

Methods

Define string representation via `__repr__()`:

```
py::class_<Response>(m, "Response")
    ...
    .def("__repr__", [](const Response& self) {
        return std::string()
            + "<" + std::to_string(self.status)
            + ": " + self.reason + ">";
    });
```

Full binding code

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

PYBIND11_MODULE(response, m) {
    py::class_<Response>(m, "Response")
        .def(py::init<>())
        .def(py::init<int, std::string>())
        .def(py::init<int, std::string, std::string>())
        .def_readonly("status", &Response::status)
        .def_readonly("reason", &Response::reason)
        .def_readonly("text", &Response::text)
        .def_property_readonly("ok", &Response::ok)
        .def("__repr__", [](const Response& self) {
            return std::string()
                + "<" + std::to_string(self.status)
                + ": " + self.reason + ">";
        })
        .def(py::self == py::self);
}
```

Trying it out

```
>>> from response import Response

>>> Response()
<200: OK>

>>> Response().ok
True

>>> r = Response(404, 'Not Found')

>>> r.reason
'Not Found'

>>> r.ok
False

>>> Response(200, 'OK') == Response()
True
```

Function signatures

Docstrings and argument names

Docstrings can be set by passing string literals to `def()`.

Arguments can be named via `py::arg("...")`.

```
m.def("greet", [](const std::string& name) {
        py::print("Hello, " + name + ".");
    },
    "Greet a person.",
    py::arg("name")
);
```

```
>>> greet('stranger')
Hello, stranger.

>>> greet?
greet(name: str) -> None

Greet a person.
```

Keyword arguments with default values

Default argument values can be set by assigning to `py::arg()`.

```
m.def("greet", [](const std::string& name, int times) {
        for (int i = 0; i < times; ++i)
            py::print("Hello, " + name + ".");
    },
    "Greet a person.",
    py::arg("name"), py::arg("times") = 1
);
```

```
>>> greet('Jeeves')
Hello, Jeeves.

>>> greet('Wooster', times=2)
Hello, Wooster.
Hello, Wooster.
```

Python objects as arguments

Functions can take arbitrary Python objects as arguments:

```
m.def("count_strings", [](py::list list) {
    int n = 0;
    for (auto item : list)
        if (py::isinstance<py::str>(item))
            n++;
    return n;
});
```

```
>>> count_strings(['foo', 'bar', 1, {}, 'baz'])
3
```

*args and **kwargs

Variadic positional and keyword arguments can be passed via `py::args` (subclass of `py::tuple`) and `py::kwargs` (subclass of `py::dict`):

```
m.def("count_args", [](py::args a, py::kwargs kw) {
    py::print(a.size(), "args,", kw.size(), "kwargs");
});
```

```
>>> count_args(10, 20, 30, x='a', y='b')
3 args, 2 kwargs
```

Function overloads

It is possible to bind multiple C++ overloads to a single Python name:

```
m.def("f", [](int x) { return "int"; });
m.def("f", [](float x) { return "float"; });
```

```
>>> f(42)
'int'

>>> f(3.14)
'float'

>>> f('cat')
TypeError: f(): incompatible function arguments.
The following argument types are supported:
    1. (arg0: int) -> str
    2. (arg0: float) -> str
```

Everything else

Type conversions

Three ways to communicate objects between C++ and Python:

1. native in C++, wrapper in Python:

```
py::class_<Foo>(m, "Foo");
m.def("f1", [](const Foo& foo) { ... });
```

2. wrapper in C++, native in Python:

```
m.def("f2", [](py::list list) { ... });
```

3. native in C++, native in Python (*type conversion*):

```
m.def("f3", [](int x) { ... });
m.def("f4", [](const std::string& s) { ... });
m.def("f5", [](const std::vector<int>& v) { ... });
```

(always requires a *copy*)

Built-in conversions

Some of the supported C++ types:

- Scalar: integer types, `float`, `double`, `bool`, `char`
- Strings: `std::string`, `const char *`
- Tuples: `std::pair<F, S>`, `std::tuple<...>`
- Sequences: `std::vector<T>`, `std::list<T>`, `std::array<T, n>`
- Maps: `std::map<K, V>`, `std::unordered_map<K, V>`
- Sets: `std::set<T>`, `std::unordered_set<T>`
- Polymorphic functions: `std::function<...>`
- Date/time: `std::chrono::duration`, `std::chrono::time_point`
- Optional: `std::optional<T>`, `std::experimental::optional<T>`

Classes

- Single and multiple inheritance.
- Overriding C++ virtual methods from Python.
- Custom constructors:

```
py::class<A>(m, "A")
    .def("__init__", [](A& self, int arg) {
        new (&self) A(arg);  // py::init<int>()
    });
```

- Implicit conversions:

```
py::class_<A>(m, "A");
py::class_<B>(m, "B")
    .def(py::init<A>());
py::implicitly_convertible<A, B>();
m.def("f", [](const B& arg) { ... });
```

- Operator overloading, `py::self` helper.
- Static methods, properties, attributes.

Python interface

- Objects with / without refcounting (`py::object` / `py::handle`)
- Built-in types (`py::int_`, `py::list`, `py::module`, `py::function`, ...)
- Casting: `py::cast(cpp_obj)`, `py_obj.cast<T>()`.
- Calling Python functions via `()`, `*args` and `**kwargs` unpacking:

```
using namespace pybind11::literals;
auto ship = py::make_tuple("USS Enterprise", 1701);
auto bridge = py::dict("Jim"_a=1, "Spock"_a=2);
auto others = py::dict("Scotty"_=4);
py::function engage = ...;
engage(*ship, **bridge, "McCoy"_a=3, **others);
```

- Import modules – `py::module::import()`.
- `print()` function – `py::print()`.
- `str.format()` method – `py::str::format()`.
- `py::len()`, `py::isinstance<>()`, etc.
- Run Python code – `py::eval()`, `py::eval_file()`.

Buffer protocol and NumPy

- Buffer protocol for a type: `.def_buffer()`.
- `py::buffer`, `py::memoryview`.
- NumPy: `py::array`, `py::array_t<T>`.
- Checked (default) or unchecked element access.
- Fast access to array properties via NumPy C API.
- Support for registering structured NumPy dtypes.
- Automatic function vectorization (`py::vectorize`).
- Also: Eigen support.

... and a few other things

- Return value policies (`copy`, `move`, `reference`, `reference_internal`, `automatic`, `automatic_reference`).
- Call policies: `py::keep_alive<Nurse, Patient>`.
- Automatic translation of built-in exceptions.
- Custom exception translators.
- Smart pointers and custom holder types.
- `pybind11` runtime: capsule, registered types map, registered instances map.

Functions and callbacks

Functions and callbacks

Type conversions for `std::function<...>` can be enabled by including an optional `pybind11/functional.h` header. Python to C++ callback:

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

m.def("for_even", [](int n, std::function<void(int)> f) {
    for (int i = 0; i < n; ++i)
        if (i % 2 == 0)
            f(i);
});
```

```
>>> def callback(x):
...     print('received:', x)

>>> for_even(3, callback)
received: 0
received: 2
```

Higher order functions...

```cpp
using int_fn = std::function<int(int)>;

int_fn apply_n(int_fn f, int n) {                    // f(f(..(x))
    return [f, n](int x) {
        for (int i = 0; i < n; ++i)
            x = f(x);
        return x;
    };
}

m.def("apply_n", apply_n);
```

```
>>> def f(x):
...     return x * 2

>>> g = apply_n(f, 8)
>>> g(10)
2560
```

... and higher...

```cpp
using int_fn = std::function<int(int)>;

int_fn apply_n(int_fn f, int n) {                    // f(f(..(x))
    return [f, n](int x) {
        for (int i = 0; i < n; ++i)
            x = f(x);
        return x;
    };
}

std::function<int_fn(int_fn)> apply_n(int n) { // decorator
    return [n](int_fn f) {
        return apply_n(f, n);
    }
}

m.def("apply_n", py::overload_cast<int_fn, int>(apply_n));
m.def("apply_n", py::overload_cast<int>(apply_n));
```

... decorators?

```
def f(x):
    return x * 2

>>> apply_n(f, 8)(10)
2560
```

```
@apply_n(8)
def g(x):
    return x * 2

>>> g(10)
2560
```

NumPy example

NumPy example: rolling stats

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

struct Stats { double mean; double std; };

auto rolling_stats(py::array_t<double> arr, size_t window) {
    if (arr.ndim() != 1)
        throw std::runtime_error("expected 1-D array");

    py::array_t<Stats> stats(arr.size());
    auto a = arr.unchecked<1>();
    auto s = stats.mutable_unchecked<1>();

    double sum = 0, sqr = 0;
    for (size_t i = 0; i < arr.size(); ++i) {
        if (i >= window) {
            auto x = a(i - window); sum -= x; sqr -= x * x;
        }
        auto x = a(i); sum += x; sqr += x * x;
        double n = i >= window ? window : (i + 1)
        double mean = sum / n;
        s(i) = { mean, std::sqrt((sqr - sum * mean) / (n - 1)) };
    }
    return stats;
}

PYBIND11_MODULE(example, m) {
    PYBIND11_NUMPY_DTYPE(Stats, mean, std);
    m.def("rolling_stats", rolling_stats);
}
```

NumPy example: rolling stats

```
>>> import pandas as pd
>>> pd.DataFrame(rolling_stats([1, 4 9, 16, 25], window=2))
```

	mean	std
0	1.0	NaN
1	2.5	2.121320
2	6.5	3.535534
3	12.5	4.949747
4	20.5	6.363961

Correctness check:

```
import numpy as np

a = np.random.random(25 * 1000 * 1000)
stats = rolling_stats(a, window=1000)
rolling = pd.Series(a).rolling(window=1000, min_periods=0)

assert np.allclose(stats['mean'], rolling.mean())
assert np.allclose(stats['std'], rolling.std(), equal_nan=1)
```

NumPy example: rolling stats

Performance check:

```
a = np.random.random(25 * 1000 * 1000)
stats = rolling_stats(a, window=1000)
rolling = pd.Series(a).rolling(window=1000, min_periods=0)
```

```
>>> %timeit rolling.mean()
1.1 s ± 24.9 ms per loop
```

```
>>> %timeit rolling.std()
1.18 s ± 16.3 ms per loop
```

```
>>> %timeit rolling_stats(a, 1000)
264 ms ± 4.36 ms per loop
```

Thanks

Thanks

Wenzel Jakob (`@wjakob`) – for creating this awesome project.

Jason Rhinelander (`@jagerman`) and Dean Moldovan (`@dean0x7d`) – for maintaining it and adding a metric ton of features and tests.

Jonas Adler, Sylvain Corlay, Trent Houliston, Axel Huebl, `@hulucc`, Sergey Lyskov, Johan Mabille, Tomasz Miąsko, Ben Pritchard, Boris Schäling, Pim Schellart, Patrick Stewart and myself (`@aldanor`) – for contributing significant features or improvements.

Dave Abrahams – for creating Boost.Python.

Thanks for listening!