Integrating C++ and Python

Lecture 21

Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4700/

Python and C++

- C++ and Python stand as giants
- Python, known for its user-friendly syntax, vast libraries, and strong community support
  - It is often the language of choice for scientists and researchers for data analysis, machine learning, visualization, and rapid prototyping
- C++ is a preferred option for performance-intensive computations
- Imagine performing complex data analyses with Python's ease and visualizing results with the raw power of C++
- Integrating C++ and Python isn't just a theoretical exercise; it's a practical need in software development

Python and C++: How to?

- SWIG (https://www.swig.org/): Great for multi-language projects, but can be less efficient and less "Pythonic"
- Cython (https://cython.org/): Known for allowing C extensions in Python-like syntax
- ctypes (https://docs.python.org/3/library/ctypes.html): A part of Python's standard library, suitable for basic tasks involving C functions
- Boost.Python (https://www.boost.org/): A well-documented and widely used library that offers seamless interoperability between C++ and Python.
- pybind11 (https://pybind11.readthedocs.io/en/stable/): Stands out for its modern approach, being lightweight and easy to use
  - Newer variation: Nanobind (https://nanobind.readthedocs.io/)

Extension Modules

- Python 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 Python extension modules in pure C, but…
  - Manual reference counting
  - Manual exception handling
  - A lot of 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 (not supported: scoped enums, non-type template parameters, templated overloads, variadic templates, universal references, etc.)
  - Limited support for generic code
  - 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):
    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;
}
```

PyBind11

- Pybind11 is a lightweight header-only library that allows interacting with Python interpreter and writing Python extension modules in modern C++ - 8K lines of code
  - CPython 2.7, CPython 3.x, PyPy
  - 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 reference counting, capturing lambdas, function vectorization, arbitrary exception types, virtual class wrapping, etc . . .
- Link: http://github.com/pybind/pybind11

PyBind11

- PyBind11 is a modern tool that is designed for simplicity, yet it doesn't compromise on power
- It helps creating Python bindings of existing C++ code
- Boilerplate (will be omitted in most further examples):

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


// Syntax: PYBIND11_MODULE(<module name>, <module object>)
PYBIND11_MODULE(example, m) {
    // …
}
```

Lets write a Python Module

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

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

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

- Or using a C++ Lambda:

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

Lets write a 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 Python Module

- In a CMake project:

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

- Need to install python-dev package (Python headers and libraries)

Python Code

- File: `response.py`

```
class Response:
    __slots__ = [ 'status', 'reason', 'text' ]

    def __init__(self, status=200, reason='Ok', text=''):
        self.status = status
        self.reason = reason
        self.text = text

    def __eq__(self, other):
        return self.status == other.status and \
            self.reason == other.reason and \
            self.text == other.text

    def __repr__(self):
        return '<%d: %s>' % (self.status, self.reason)

    def _get_ok(self):
        return self.status >= 200 and self.status < 400

    ok = property(fget=_get_ok)
```

Equivalent C++ Type

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

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

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

    Response() : response(200, "OK") {}
    bool ok() const { return status >= 200 && status < 400; }
};

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

Binding the Type

```cpp
struct Response
{
    // ...
};
```

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

Constructors

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

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

Instance Attributes (Data Members)

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

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

Properties

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

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

Operators

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

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

Operators (py::self)

- Wrapping operators is a very common thing to do, . . . so there's a shortcut:

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

py::class_<Response>(m, "Response")
    // ...
    .def(py::self == py::self);
```

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

Methods (Member Functions)

- Define string representation via `__repr__()`:

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

Full Binding Code for Response Type

```
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__",
            [](Response const& 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
```

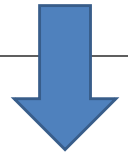
Binding 101

- The `PYBIND11_MODULE()` macro is the heart of pybind11's binding mechanism
  - It creates a function that Python calls upon importing the module
- The `py::module_::def()` method produces binding code
  - Effectively allowing the C++ functions to be accessible and usable within Python
- This type also exposes many other facilities, like `def_readonly`, or `def_property_readonly`

Docstrings and Argument Names

- Docstrings can be set by passing string literals to `def()`
- Arguments can be named via `py::arg("...")`.

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

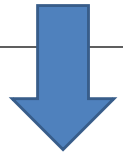


```
>>> greet('stranger')
Hello, stranger.

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

Greet a person.
```

Keyword Arguments with default Values

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

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

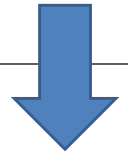


```
>>> greet('Jeeves')
Hello, Jeeves.

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

Python Objects as Arguments

- Functions can take arbitrary Python objects as arguments (here `py::list`):

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

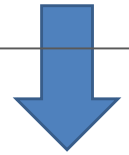


```
>>> count_strings(['foo', 'bar', 1, {}, 'baz'])
3
```

Python Objects as Arguments

- Functions can take arbitrary Python objects as arguments (here `py::dict`):

```cpp
m.def("print_dict",
    [](py::dict const& dict) {
        for (auto item : dict)
            std::cout << "key="
                      << std::string(py::str(item.first))
                      << ", value="
                      << std::string(py::str(item.second))
                      << "\n";
    });
```



```
>>> print_dict({"foo": 123, "bar": "hello"})
key=foo, value=123
key=bar, value=hello
```

*args and **kwargs Arguments

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

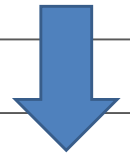
```
m.def("count_args",
    [](py::args a, py::kwargs kw) {
        py::print(a.size(), "args,", kw.size(), "kwargs");
    });
```

```
>>> 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:
(arg0: int) -> str
(arg0: float) -> str
```

PyBind11: Supported Data Types

- Offers comprehensive support for a wide range of data types
  - Facilitating seamless conversions between C++ and Python
- Supports all standard Python data types such as integers, floating-point numbers, strings, lists, tuples, dictionaries, and sets
- Those are automatically converted to `int`, `double`, `std::string`, `std::vector`, `std::tuple`, `std::map`, and `std::set`
- Fully compatible with Python's NumPy library (https://numpy.org/)
  - Are efficiently converted to and from `std::vector` and `std::array`
  - Interoperability extends to SciPy (https://scipy.org/), Matplotlib (https://matplotlib.org/), pandas (https://pandas.pydata.org/), etc.

Type Conversions

- Three ways to communicate objects between C++ and Python:
- native in C++, wrapper in Python
  - Use a native C++ type everywhere
  - The type must be wrapped using pybind11-generated bindings so that Python can interact with it
- wrapper in C++, native in Python
  - Use a native Python type everywhere
  - It will need to be wrapped so that C++ functions can interact with it
- native in C++, native in Python (always requires a copy)
  - Use a native C++ type on the C++ side and a native Python type on the Python side
  - PyBind11 refers to this as a type conversion

Type Conversions

- native in C ++, wrapper in Python
- Exposing a custom C++ type using `py::class_` (as described)
- The underlying data structure is always the original C++ class
  - The `py::class_` wrapper provides a Python interface
- When an object like this is sent from C++ to Python, pybind11 will just add the wrapper layer over the native C++ object
- Getting it back from Python is just a matter of peeling off the wrapper

Type Conversions

- wrapper in C ++, native in Python
- Type is native to Python, like a Python tuple or a Python list
- Interface is based on `py::object` family of wrappers
  - `py::str`, `py::bytes`, `py::tuple`, `py::list`, `py::dict`, `py::slice`, `py::none`, etc.

```
py::dict d(py::str("spam")=py::none(), py::str("eggs")=42);
py::tuple tup = py::make_tuple(42, py::none(), "spam");
```

- Casting back and forth:

```
MyClass *cls = ...;
py::object obj = py::cast(cls);
```

```
py::object obj = ...;
MyClass *cls = obj.cast<MyClass *>();
```

Type Conversions

- native in C ++, native in Python
  - Native types on both sides with automatic conversion between them

```cpp
m.def("print_vector",
    [](std::vector<int> const& v) {
        for (auto item : v)
            std::cout << item << "\n";
    });
```

```
>>> print_vector([1, 2, 3])
1 2 3
```

  - Construct a new `std::vector<int>` and copy each element from the Python list
  - Same applies to return values, just conversion in opposite direction

Type Conversions

- native in C ++, native in Python
- Supported:
  - All built-in types (integrals, floating point types, etc.)
  - All STL container types:
    - 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::variant<T>`
- Always involves creating a copy of the data!

PyBind11: More References

- PyBind11 documentation: https://pybind11.readthedocs.io/en/stable/
- Example PyBind11 module built with a CMake-based build system: https://github.com/pybind/cmake_example
- Example PyBind11 module built with a Python-based build system: https://github.com/pybind/python_example

More Python Interface

- Objects with / without reference counting (`py::object`/`py::handle`)
- Calling Python functions via `()`, `*args` and `**kwargs` unpacking:

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

- Import modules: `py::module::import()`

```
py::object scipy = py::module_::import("scipy");
return scipy.attr("__version__");
```

- Python print() function: `py::print()`
- Python str.format() method: `py::str::format()`

More Python Interface

- Run Python code: `py::eval()`, `py::eval_file()`

```
int main()
{
    py::scoped_interpreter guard;
    py::exec(R"(
        kwargs = {name: "World", number: 42}
        message = "Hello, {name}! The answer is {number}".format(**kwargs)
        print(message)
    )");
}
```

Functions and Callbacks

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

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

```
>>> 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)
{
    return [f, n](int x) {          // f(f(..(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)              # 10 * 2^8
2560
```

… and even higher Order …

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

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

// decorator
std::function<int_fn(int_fn)> apply_n(int n) {
    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));
```

… Python Decorators

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

>>> apply_n(f, 8)(10)   # 10 * 2^8
2560
```

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

>>> g(10)               # 10 * 2^8
2560
```

NumPy Example

```cpp
auto rolling_stats(py::array_t<double> arr, size_t window)
{
    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);
}
```

```cpp
struct Stats
{
    double mean;
    double std;
};
```

NumPy Example

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

	mean	std
1	1.0	NaN
2	2.5	2.121320
3	6.5	3.535534
4	12.5	4.949747
5	20.5	6.363961

NumPy Example

- 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

- Performance check:

```
a = np.random.random(25 * 1000 * 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, window=1000)
264 ms ± 4.36 ms per loop
```

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
  - Eigen is a very fast C++ Linear Algebra Library (https://eigen.tuxfamily.org/)

… 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, GIL management

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP