The Abdus Salam International Centre for Theoretical Physics

How to bind C++ with Python

Serafina Di Gioia
Postdoctoral researcher in ML @ICTP

Viable solutions to bind C++ code to Python

- ctypes
  (https://docs.python.org/3/library/ctypes.html)
- Cython
  (https://cython.org)
- PyBind11
  (https://pybind11.readthedocs.io/en/stable/advanced)
- CFFI
  (https://cffi.readthedocs.io/en/latest/overview.html)
- SWIG
  https://www.swig.org/

Intro to pybind11

It is a lightweight header-only library that exposes C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code.

GOAL: minimize boilerplate code in traditional extension modules by inferring type information using compile-time introspection

This library is very similar to Boost.Python with everything stripped away that isn't relevant for binding generation.

NOTE: The core header files only require ~4K lines of code and depend on Python (3.6+, or PyPy) and the C++ standard library. This compact implementation was possible thanks to some of the new C++11 language features (specifically: tuples, lambda functions and variadic templates).

What can be passed from/to C++ with pybind11?

- Functions accepting and returning custom data structures per value, reference, or pointer
- Instance methods and static methods
- Overloaded functions
- Instance attributes and static attributes
- Arbitrary exception types
- Enumerations
- Callbacks
- Iterators and ranges
- Custom operators
- Single and multiple inheritance
- STL data structures
- Smart pointers with reference counting like `std::shared_ptr`
- Internal references with correct reference counting
- C++ classes with virtual (and pure virtual) methods can be extended in Python

Pybind11 advantages

- bind C++11 lambda functions with captured variables. The lambda capture data is stored inside the resulting Python function object.
- **efficient transfer of custom data types**, using C++11 move constructors and move assignment operators whenever possible to.
- It’s easy to expose the internal storage of custom data types through Pythons’ buffer protocols. This is handy e.g. for fast conversion between C++ matrix classes like Eigen and NumPy without expensive copy operations.
- **automatic vectorization of functions** so that they are transparently applied to all entries of one or more NumPy array arguments.
- fast support for the Python’s slice-based access and assignment operations (just a few lines of code).
- **very light library** (in particular with respect to Boost.Python) . Everything is contained in just a few header files; there is no need to link against any additional libraries.

Installing pybind11

Using pip

```
pip install pybind11
```

Using conda, via conda-forge:

```
conda install -c conda-forge pybind11
```

When you are working on a project in Git, you can use the pybind11 repository as a submodule. From your git repository, use:

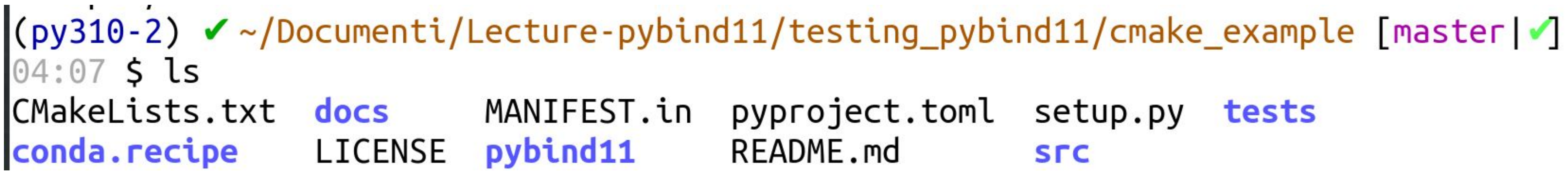
```
git submodule add -b stable ../../pybind/pybind11 extern/pybind11
git submodule update --init
```

Testing pybind installation

To test pybind11 we can download the cmake_example repo from Github

```
git clone --recursive https://github.com/pybind/cmake_example.git
```

This repo contains the following files/directories



```
(py310-2) ✔ ~/Documenti/Lecture-pybind11/testing_pybind11/cmake_example [master|✔]
04:07 $ ls
CMakeLists.txt  docs     MANIFEST.in  pyproject.toml  setup.py  tests
conda.recipe    LICENSE  pybind11     README.md       src
```

To install the package using the setup.py (that call Cmake_build)

```
pip install -e . -vvv
```

To launch the tests contained in ‘tests/’ we use pytest:

```
pytest tests
```

What do we need build Python packages containing C++ code with Pybind11 + cmake?

To build Python packages we need a setup.py and a CMakeLists.txt file.
The CMakeLists.txt file for the ‘cmake_example’ is shown below:

```
cmake_minimum_required(VERSION 3.4...3.18)
project(cmake_example)

add_subdirectory(pybind11)
pybind11_add_module(cmake_example src/main.cpp)

# EXAMPLE_VERSION_INFO is defined by setup.py and passed into the C++ code as a
# define (VERSION_INFO) here.
target_compile_definitions(cmake_example
                           PRIVATE VERSION_INFO=${EXAMPLE_VERSION_INFO})
```

while for the setup.py file I list here the import blocks:

- import of necessary libraries (os, setuptools)
- class CMakeExtension(Extension):
- class CMakeBuild(build_ext):
- call to the setup() function

```
setup(
    name="cmake_example",
    version="0.0.1",
    author="Dean Moldovan",
    author_email="dean0x7d@gmail.com",
    description="A test project using pybind11 and CMake",
    long_description="",
    ext_modules=[CMakeExtension("cmake_example")],
    cmdclass={"build_ext": CMakeBuild},
    zip_safe=False,
    extras_require={"test": ["pytest>=6.0"]},
    python_requires=">=3.7",
)
```

Simplest Pybind11 example

With pybind11 we need only a .cpp file to define and import in Python a C++ function:

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

int add(int i, int j) {
    return i + j;
}

PYBIND11_MODULE(example, m) {
    m.doc() = "pybind11 example plugin"; // optional module docstring

    m.def("add", &add, "A function that adds two numbers");
}
```

To compile this code we use the following command:

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

This produces a binary module file that can be imported to Python.

```
>>> import example
>>> example.add(1, 2)
3
```

Adding names and default values to argument of add function

To add names to the args of add():

```
// regular notation
m.def("add1", &add, py::arg("i"), py::arg("j"));
// shorthand
using namespace pybind11::literals;
m.def("add2", &add, "i"_a, "j"_a);
```

To add visible default values:

```
m.def("add", &add, "A function which adds two numbers",
      py::arg("i") = 1, py::arg("j") = 2);
```

let's have a look a the help() output on this function:

```
>>> help(example)

....

FUNCTIONS
    add(...)
        Signature : (i: int = 1, j: int = 2) -> int

        A function which adds two numbers
```

How to pass different type of objects passed from/to C++ with pybind11…

There are three fundamentally different ways to provide access to native Python types in C++ and vice versa:

1. Use a native C++ type everywhere. In this case, the type must be wrapped using pybind11-generated bindings so that Python can interact with it.
2. Use a native Python type everywhere. It will need to be wrapped so that C++ functions can interact with it.
3. 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*.

Hereafter I show two examples referring to case 2 and 3

case 2

```
void print_list(py::list my_list) {
    for (auto item : my_list)
        std::cout << item << " ";
}
```

case 3

```
void print_vector(const std::vector<int> &v) {
    for (auto item : v)
        std::cout << item << "\n";
}
```

Example case3: Wrapping C++ class with Pybind11

If we have a C++ class
defined a particular data type,
like 2D vector,
with associated operator
overloading

```cpp
class Vector2 {
public:
    Vector2(float x, float y) : x(x), y(y) { }

    Vector2 operator+(const Vector2 &v) const { return Vector2(x + v.x, y + v.y); }
    Vector2 operator*(float value) const { return Vector2(x * value, y * value); }
    Vector2& operator+=(const Vector2 &v) { x += v.x; y += v.y; return *this; }
    Vector2& operator*=(float v) { x *= v; y *= v; return *this; }

    friend Vector2 operator*(float f, const Vector2 &v) {
        return Vector2(f * v.x, f * v.y);
    }

    std::string toString() const {
        return "[" + std::to_string(x) + ", " + std::to_string(y) + "]";
    }
private:
    float x, y;
};
```

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

PYBIND11_MODULE(example, m) {
    py::class_<Vector2>(m, "Vector2")
        .def(py::init<float, float>())
        .def(py::self + py::self)
        .def(py::self += py::self)
        .def(py::self *= float())
        .def(float() * py::self)
        .def(py::self * float())
        .def(-py::self)
        .def("__repr__", &Vector2::toString);
}
```

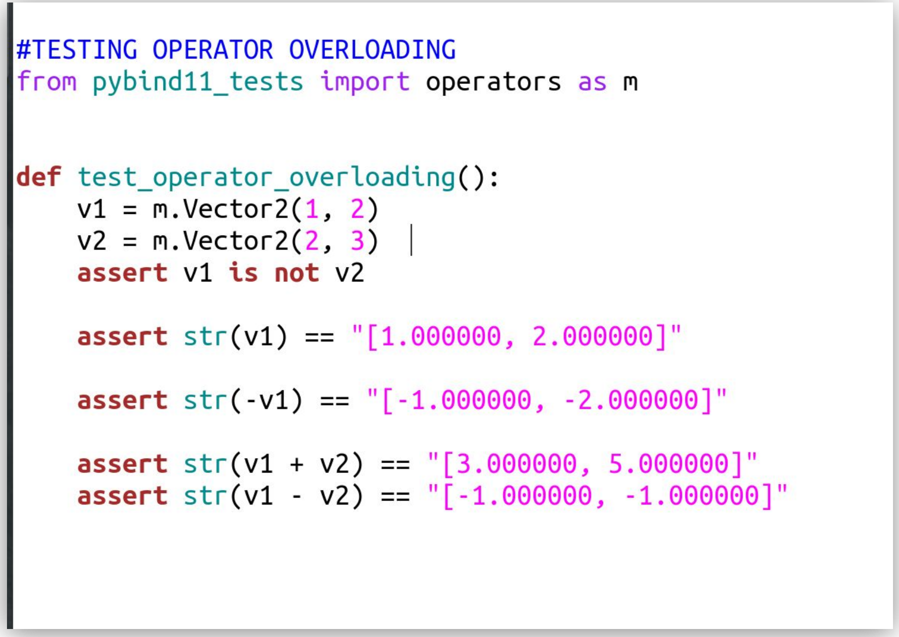
to wrap this class and its instance methods with pybind11 we need to write in the .cpp file the commands on the left

corresponds to
the long notation

```cpp
.def("__mul__", [](const Vector2 &a, float b) {
    return a * b;
}, py::is_operator())
```

Testing the Vector2D class

But how do we call the vector2D class in python? and how do we test it?



```python
#TESTING OPERATOR OVERLOADING
from pybind11_tests import operators as m


def test_operator_overloading():
    v1 = m.Vector2(1, 2)
    v2 = m.Vector2(2, 3)
    assert v1 is not v2

    assert str(v1) == "[1.000000, 2.000000]"

    assert str(-v1) == "[-1.000000, -2.000000]"

    assert str(v1 + v2) == "[3.000000, 5.000000]"
    assert str(v1 - v2) == "[-1.000000, -1.000000]"
```

How to pass different type of objects passed from/to C++ with pybind11…

There are three fundamentally different ways to provide access to native Python types in C++ and vice versa:

1. Use a native C++ type everywhere. In this case, the type must be wrapped using pybind11-generated bindings so that Python can interact with it.
2. Use a native Python type everywhere. It will need to be wrapped so that C++ functions can interact with it.
3. 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*.

most natural option

Lots of these conversions are supported out of the box, as shown in the table available at this website:
https://pybind11.readthedocs.io/en/stable/advanced/cast/overview.html#conversion-table

Vectorizing C++ functions with pybind11

Starting from the scalar function below

```cpp
double my_func(int x, float y, double z) {
    py::print("my_func(x:int={}, y:float={:.0f}, z:float={:.0f})"_s.format(x, y, z));
    return (float) x * y * z;
}
```

in Pybind11 to vectorize this function, in order to apply it to arbitrary NumPy array arguments (vectors, matrices, general N-D arrays) in addition to its normal arguments, we need only few steps:

1. import numpy header file

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

2. calling inside m.def() py:vectorize(func)

```cpp
// test_vectorize, test_docs, test_array_collapse
// Vectorize all arguments of a function (though non-vector arguments are also allowed)
m.def("vectorized_func", py::vectorize(my_func));

// Vectorize a lambda function with a capture object (e.g. to exclude some arguments from the
// vectorization)
m.def("vectorized_func2", [](py::array_t<int> x, py::array_t<float> y, float z) {
    return py::vectorize([z](int x, float y) { return my_func(x, y, z); })(std::move(x),
                                                                          std::move(y));
});
```

How to interface C++ code with Pandas using Pybind11

We can use the Apache Arrow (normally referred to as Arrow) library and the Pyrarrow lib ontaining the Python bindings.

Both the libraries and their dependencies can be installed with conda:

```
conda install arrow-cpp=13.0.* -c conda-forge
conda install pyarrow=13.0.* -c conda-forge
```

We can use apache arrow also to interface Python and R code, once installed r-arrow (this allow to use C R-API for the data transfer, being more efficient than using rpy2)

```
conda install r-arrow=13.0.* -c conda-forge
```

Arrow is not limited to CPU buffers (located in the computer’s main memory, also named “host memory”). It also has provisions for accessing buffers located on a CUDA-capable GPU device (in “device memory”).

Type metadata in Apache Arrow/Pyarrow

Apache Arrow defines language agnostic column-oriented data structures for array data. These include:

- **Fixed-length primitive types**: numbers, booleans, date and times, fixed size binary, decimals, and other values that fit into a given number
- **Variable-length primitive types**: binary, string
- **Nested types**: list, map, struct, and union
- **Dictionary type**: An encoded categorical type (more on this later)

These data structures are exposed in Python through a series of interrelated classes:

- **Type Metadata**: Instances of `pyarrow.DataType`, which describe a logical array type
- **Schemas**: Instances of `pyarrow.Schema`, which describe a named collection of types. These can be thought of as the column types in a table-like object.
- **Arrays**: Instances of `pyarrow.Array`, which are atomic, contiguous columnar data structures composed from Arrow Buffer objects
- **Record Batches**: Instances of `pyarrow.RecordBatch`, which are a collection of Array objects with a particular Schema
- **Tables**: Instances of `pyarrow.Table`, a logical table data structure in which each column consists of one or more `pyarrow.Array` objects of the same type.

Pandas integration with PyArrow

The equivalent to a pandas DataFrame in Arrow is a Table. Both consist of a set of named columns of equal length. While pandas only supports flat columns, the Table also provides nested columns, thus it can represent more data than a DataFrame, so a full conversion is not always possible.

Conversion from a Table to a DataFrame is done by calling `pyarrow.Table.to_pandas()`. The inverse is then achieved by using `pyarrow.Table.from_pandas()`.

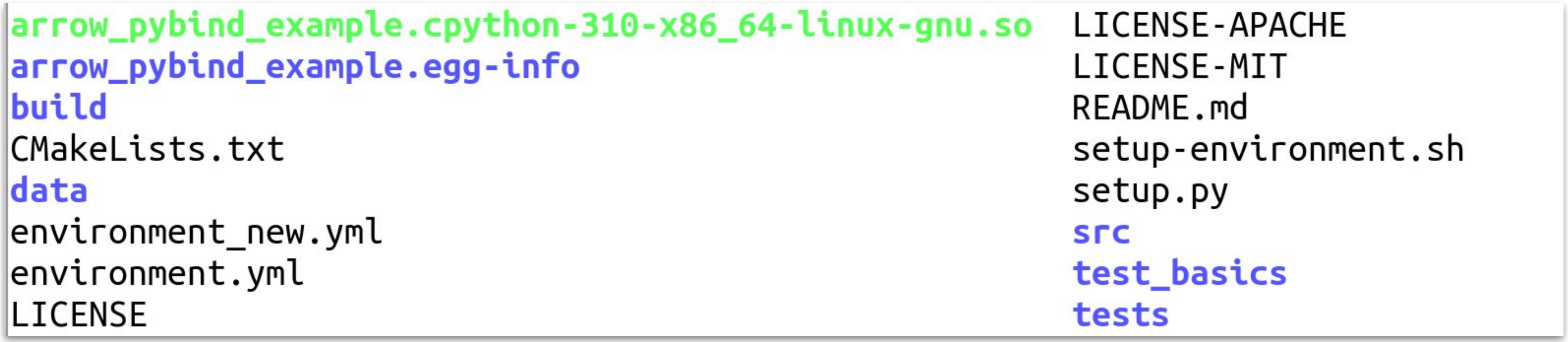
Methods like `pyarrow.Table.from_pandas()` have a `preserve_index` option which defines how to preserve (store) or not to preserve (to not store) the data in the `index` member of the corresponding pandas object. This data is tracked using schema-level metadata in the internal `arrow::Schema` object.

The default of `preserve_index` is `None`, which behaves as follows:

- `RangeIndex` is stored as metadata-only, not requiring any extra storage.
- Other index types are stored as one or more physical data columns in the resulting `Table`

An example of how to use Pybind11+Arrow to operate on Pandas objects from C++ (and viceversa)

Let’s have a look at the sub-folder example-binding-python-Cpp inside the lecture folder



```
arrow_pybind_example.cpython-310-x86_64-linux-gnu.so  LICENSE-APACHE
arrow_pybind_example.egg-info                         LICENSE-MIT
build                                                 README.md
CMakeLists.txt                                        setup-environment.sh
data                                                  setup.py
environment_new.yml                                   src
environment.yml                                       test_basics
LICENSE                                               tests
```

To use the code in this folder we need to have installed a conda env with the dependencies listed in the environment file.

SWIG vs Pybind11

MAIN DIFFERENCE: pybind11 uses C++ templates with compile time introspection to generate wrapper code, where Swig uses a custom C++ parser.

As with Boost Python, and unlike Swig, types need to be wrapped explicitly by specifying the classes and (member) functions. This has the downside of having to write and maintain the wrappers but has the advantage of being explicit and allowing wrappers to be made more Pythonic.

Moreover, unlike Swig the wrapper files are just C++11, rather than a custom language. This makes it easier for C++ developers to read, write and understand than Swig. With pybind11, developers only need proficiency in two languages (C++ and Python).

An additional advantage over Swig is that pybind11 more easily allows for build parallelization, since not all headers are wrapped into one module. Overall build speeds are comparable (for the packages wrapped so far), but partial rebuilds might take substantially less time.

References

PYBIND11 documentation:
https://pybind11.readthedocs.io/en/stable/basics.html#

PYARROW documentation:
https://arrow.apache.org/docs/python/

PYBIND_CUDA example repo:
https://github.com/pkestene/pybind11-cuda/tree/master

example to interface R and Python with PyArrow:
https://voltrondata.com/resources/data-transfer-between-python-and-r-with-rpy2-and-apache-arro
w