Lecture 12

Integrating C++ and Python codes.

Advanced Programming - SISSA, UniTS, 2025-2026

Pasquale Claudio Africa

09 Dec 2025

Outline

1. Integrating C++ and Python
2. pybind11
    - Installation
    - Basics
    - Binding object-oriented code
3. How to build and import pybind11 modules
4. Examples

Part of these lecture notes is re-adapted from the official pybind11 documentation (license).

Integrating C++ and Python

Why integrate C++ and Python code?

Complementary strengths:

- **C++**: Performance, memory control, system-level programming.
- **Python**: Rapid development, rich ecosystem, readability.

Use cases:

- **Performance-critical sections**: Implement bottlenecks in C++, interface in Python.
- **Legacy code integration**: Wrap existing C++ libraries for Python users.
- **Scientific computing**: Combine NumPy/SciPy with optimized C++ kernels.
- **Machine learning**: Write custom operators/layers in C++ for frameworks like PyTorch.

Example scenario: A data science pipeline where Python handles data loading and visualization, while C++ processes millions of records efficiently.

Common libraries for C++ and Python integration (1/5)

Several libraries are available for integrating C++ and Python codes, each with its own set of advantages and drawbacks. Here are some of the most commonly used libraries:

1. **Boost.Python**
    - Pros:
        - Well-documented, widely used.
        - Seamless interoperability between C++ and Python.
        - Exposes C++ classes to Python and vice versa.
    - Cons:
        - Complex setup for beginners.
        - Larger binary size.
        - Slower compile times.

Common libraries for C++ and Python integration (2/5)

2. **SWIG** (Simplified Wrapper and Interface Generator)
   - Pros:
     - Generates bindings for multiple languages.
     - Relatively easy for simple tasks.
     - Useful for multi-language projects.
   - Cons:
     - Less efficient, less *Pythonic* interface code.
     - Difficult to debug.
     - Complex for advanced use cases.

Common libraries for C++ and Python integration (3/5)

3. **Cython**
   - Pros:
     - C extensions in Python-like syntax.
     - Significant performance improvements.
     - Good integration with Python ecosystem.
   - Cons:
     - Requires learning new syntax.
     - Not for exposing existing C++ codebases.
     - Variable performance gains.

Common libraries for C++ and Python integration (4/5)

4. **ctypes**
   - Pros:
     - Part of Python standard library.
     - Simple for basic tasks.
     - Good for calling C functions from Python.
   - Cons:
     - Limited to C functions, not C++.
     - Manual type conversions.
     - Complex error handling.

Common libraries for C++ and Python integration (5/5)

5. **pybind11**
   - Pros:
     - Modern, lightweight, easy to use.
     - Header-only library.
     - More *Pythonic* bindings.
     - Good documentation, community support.
   - Cons:
     - Less advanced features than Boost.Python.
     - More manual work for complex bindings.

Quick comparison of C++/Python integration tools

Library	Ease of use	Performance	Best for
Boost.Python	★★★	★★★★	Complex C++ features
SWIG	★★★★	★★★	Multi-language bindings
Cython	★★★	★★★★★	New code from scratch
ctypes	★★★★	★★★	Simple C function calls
pybind11	★★★★★	★★★★	Modern C++ projects, new bindings

PyPy and Numba overview

- **PyPy**
  - Alternative Python implementation focusing on speed.
  - JIT compiler for runtime compilation.
  - Less memory usage, compatible with CPython.
  - Faster for long-running processes.
  - **Limitations**: Library support, JIT warm-up time.
- **Numba**
  - JIT compiler for Python and NumPy code.
  - Easy to use, significant performance improvements.
  - Integrates with Python scientific stack.
  - Supports CUDA GPU programming.
  - **Limitations**: Focused on numerical computing, learning curve for parallel programming, debugging challenges.

Why pybind11?

- Modern, relevant, and practical for industry demands.
- Header-only library, which simplifies the build process.
- Lightweight, and easy to use.
- Balances ease of use with powerful features.
- Generates more *Pythonic* bindings compared to alternatives.
- Suitable for a range of projects, enhancing problem-solving skills.

⚠️ **Note**: pybind11 may require more manual work for complex bindings.

pybind11: installation

pybind11 overview

pybind11 is a lightweight, header-only library that connects C++ types with Python. This tool is crucial for creating Python bindings of existing C++ code.

Key features:

- **Header-only**: No separate compilation needed for pybind11 itself.
- **Modern C++**: Leverages C++11/14/17 features for cleaner syntax.
- **Automatic conversions**: STL containers, smart pointers, exceptions.
- **NumPy integration**: Efficient zero-copy array passing (with `pybind11/numpy.h`).
- **Minimal boilerplate**: Compared to Boost.Python or SWIG.

Supported C++ standards: C++11, C++14, C++17.
Supported Python versions: 3.6+ (as of pybind11 2.11+).

Documentation link .

How to install (1/3)

There are multiple methods to acquire the pybind11 source, available on GitHub. The recommended approaches include via PyPI, through Conda, building from source, or importing it as a Git submodule.

Include with PyPI

pybind11 can be installed as a Python package from PyPI using pip:

```
pip install pybind11
```

Include with conda-forge

Conda users can obtain pybind11 via conda-forge:

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

How to install (2/3)

Global install with brew

For macOS and Linux users, the brew package manager offers pybind11. Install it using:

```
brew install pybind11
```

Build from source

If you prefer to build from source, use the following commands:

```
wget https://github.com/pybind/pybind11/archive/refs/tags/v3.0.1.tar.gz
tar xzvf v3.0.1.tar.gz

cd pybind11-3.0.1/
mkdir build && cd build

cmake .. -DCMAKE_INSTALL_PREFIX=/opt/pybind11
[sudo] make -j<N> install
```

How to install (3/3)

Include as a submodule

For Git-based projects, pybind11 can be incorporated as a submodule. In your Git repository, execute the following commands:

```
git submodule add -b stable https://github.com/pybind/pybind11 extern/pybind11
git submodule update --init
```

This method assumes dependency placement in `extern/`. Remember, some servers might require the `.git` extension.

After setup, include `extern/pybind11/include` in your project, or employ pybind11's integration tools.

pybind11: basics

Header and namespace conventions

For brevity, all code examples assume that the following two lines are present:

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

namespace py = pybind11;
```

Some features may require additional headers, but those will be specified as needed.

Creating bindings for a simple function

Let's start by creating Python bindings for an extremely simple function, which adds two numbers and returns their result. For simplicity, we'll put both this function and the binding code into a file named `example.cpp` with the following contents:

```
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");
}
```

It's good practice to keep implementation and binding code in separate files.

The `PYBIND11_MODULE` macro

The `PYBIND11_MODULE` macro creates a function that will be called when an `import` statement is issued from within Python.

The module name (`example`) is given as the first macro argument (it should not be in quotes).

The second argument (`m`) defines a variable of type `py::module_<module>` which is the main interface for creating bindings.

The method `module_::def` generates binding code that exposes the `add()` function to Python.

Note (1/2)

Notice how little code was needed to expose our function to Python: all details regarding the function's parameters and return value were automatically inferred using template metaprogramming. This overall approach and the used syntax are borrowed from Boost.Python, though the underlying implementation is very different.

pybind11 is a header-only library, hence it is not necessary to link against any special libraries and there are no intermediate (magic) translation steps. On Linux, the above example can be compiled using the following command:

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

If you included pybind11 as a Git submodule, then use `$(python3-config --includes) -Iextern/pybind11/include` instead of `$(python3 -m pybind11 --includes)` in the above compilation.

Note (2/2)

The `python3 -m pybind11 --includes` command fetches the include paths for both pybind11 and Python headers. This assumes that pybind11 has been installed using `pip` or `conda`. If it hasn't, you can also manually specify `-I <path-to-pybind11>/include` together with the Python includes path `python3-config --includes`.

On macOS: the build command is almost the same but it also requires passing the `-undefined dynamic_lookup` flag so as to ignore missing symbols when building the module.

How to import a C++ bound module

Building the above C++ code will produce a binary module file that can be imported to Python. Assuming that the compiled module is located in the current directory, the following interactive Python session shows how to load and execute the example:

```
import example
example.add(1, 2) # Output: 3
```

Keyword arguments

With a simple code modification, it is possible to inform Python about the names of the arguments (`i` and `j` in this case).

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

`arg` is one of several special tag classes which can be used to pass metadata into `module_::def`. With this modified binding code, we can now call the function using keyword arguments, which is a more readable alternative particularly for functions taking many parameters:

```
import example
example.add(i=1, j=2) # Output: 3L
```

Documentation

The keyword names also appear in the function signatures within the documentation.

```
help(example)
```

```
...
FUNCTIONS
    add(...)
        Signature : (i: int, j: int) -> int

        A function which adds two numbers
```

Shorthand for keyword arguments

A shorter notation for named arguments is also available:

```
// Regular notation.
m.def("add1", &add, py::arg("i"), py::arg("j"));

// Shorthand.
using namespace py::literals;
m.def("add2", &add, "i"_a, "j"_a);
```

The `_a` suffix forms a C++11 literal which is equivalent to `py::arg`. Note that the literal operator must first be made visible with the directive `using namespace py::literals`. This does not bring in anything else from the `pybind11` namespace except for literals.

Default arguments

Suppose now that the function to be bound has default arguments, e.g.:

```
int add(int i = 1, int j = 2) {
    return i + j;
}
```

Unfortunately, pybind11 cannot automatically extract these parameters, since they are not part of the function's type information. However, they are simple to specify using an extension of `arg`:

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

// Shorthand.
m.def("add2", &add, "i"_a = 1, "j"_a = 2);
```

The default values also appear within the documentation.

Exporting variables

To expose a value from C++, use the `attr` function to register it in a module as shown below. Built-in types and general objects (more on that later) are automatically converted when assigned as attributes, and can be explicitly converted using the function `py::cast<type>`.

```
PYBIND11_MODULE(example, m) {
    m.attr("the_answer") = 42;
    py::object world = py::cast("World");
    m.attr("what") = world;
}
```

These are then accessible from Python:

```
import example
example.the_answer # Output: 42
example.what # Output: 'World'
```

Supported data types

A large number of data types are supported out of the box and can be used seamlessly as functions arguments, return values or with `py::cast` in general.

For a full overview, see the official documentation.

pybind11: binding object-oriented code

Creating bindings for a custom type

Let's now look at a more complex example where we'll create bindings for a custom C++ data structure named `Pet`.

```
struct Pet {
    Pet(const std::string &name) : name(name) { }
    void set_name(const std::string &name_) { name = name_; }
    const std::string &get_name() const { return name; }

    std::string name;
};

PYBIND11_MODULE(example, m) {
    py::class_<Pet>(m, "Pet")
        .def(py::init<const std::string &>())
        .def("set_name", &Pet::set_name)
        .def("get_name", &Pet::get_name);
}
```

`class_`

`class_` creates bindings for a C++ *class* or *struct*-style data structure. `init` is a convenience function that takes the types of a **constructor**'s parameters as template arguments and wraps the corresponding constructor. An interactive Python session demonstrating this example is shown below:

```
import example
p = example.Pet("Molly")
print(p) # Output: <example.Pet object at 0x10cd98060>
p.get_name() # Output: 'Molly'
p.set_name("Charlie")
p.get_name() # Output: 'Charlie'
```

Note: It is possible to specify keyword and default arguments using the syntax discussed in the previous section.

Binding lambda functions (1/2)

Note how `print(p)` produced a rather useless summary of our data structure in the example above:

```
print(p) # Output: <example.Pet object at 0x10cd98060>
```

To address this, we could bind a utility function that returns a human-readable summary to the special method slot named `__repr__`. Unfortunately, there is no suitable functionality in the `Pet` data structure, and it would be nice if we did not have to change it.

Binding lambda functions (2/2)

This can easily be accomplished by binding a lambda function instead:

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def("set_name", &Pet::set_name)
    .def("get_name", &Pet::get_name)
    .def("__repr__",
         [](const Pet &a) {
             return "<example.Pet named '" + a.name + "'>";
         }
    );
```

With the above change, the same Python code now produces the following output:

```
print(p) # Output: <example.Pet named 'Molly'>
```

Instance and static fields

We can also directly expose the `name` field using the `class_::def_readwrite` method. A similar `class_::def_readonly` method also exists for `const` fields.

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def_readwrite("name", &Pet::name)
    // ...
```

This makes it possible to write

```
p = example.Pet("Molly")
p.name # Output: 'Molly'
p.name = "Charlie"
p.name # Output: 'Charlie'
```

Static member functions can be bound in the same way using `class_::def_static`.

Private fields (1/2)

Now suppose that `Pet::name` was a private internal variable that can only be accessed via setters and getters.

```
class Pet {
public:
    Pet(const std::string &name) : name(name) { }
    void set_name(const std::string &name_) { name = name_; }
    const std::string &get_name() const { return name; }

private:
    std::string name;
};
```

Private fields (2/2)

In this case, the method `class_::def_property` (`class_::def_property_readonly` for read-only data) can be used to provide a field-like interface within Python that will transparently call the setter and getter functions:

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def_property("name", &Pet::get_name, &Pet::set_name)
    // ...
```

Write only properties can be defined by passing `nullptr` as the input for the read function.

Note: Similar functions `class_::def_readwrite_static`, `class_::def_readonly_static` `class_::def_property_static`, and `class_::def_property_readonly_static` are provided for binding static variables and properties.

Dynamic attributes (1/3)

Native Python classes can pick up new attributes dynamically:

```
class Pet:
    name = "Molly"

p = Pet()
p.name = "Charlie" # Overwrite existing.
p.age = 2 # Dynamically add a new attribute.
```

Dynamic attributes (2/3)

By default, classes exported from C++ do not support this and the only writable attributes are the ones explicitly defined using `class_::def_readwrite` or `class_::def_property`.

```
py::class_<Pet>(m, "Pet")
    .def(py::init<>())
    .def_readwrite("name", &Pet::name);
```

Trying to set any other attribute results in an error:

```
p = example.Pet()
p.name = "Charlie" # Ok: attribute defined in C++.
p.age = 2
```

> AttributeError: 'Pet' object has no attribute 'age'

Dynamic attributes (3/3)

The `py::dynamic_attr` tag enables dynamic attributes for C++ classes:

```
py::class_<Pet>(m, "Pet", py::dynamic_attr())
    .def(py::init<>())
    .def_readwrite("name", &Pet::name);
```

Now everything works as expected:

```
p = example.Pet()
p.name = "Charlie" # Ok: overwrite value in C++.
p.age = 2 # Ok: dynamically add a new attribute.
p.__dict__  # Output: {'age': 2}
```

Note that there is a small runtime cost for a class with dynamic attributes. Not only because of the addition of a `__dict__`, but also because of more expensive garbage collection tracking which must be activated to resolve possible circular references. Native Python classes incur this same cost by default, anyway.

Inheritance, downcasting and polymorphism (1/6)

Suppose now that the example consists of two data structures with an inheritance relationship:

```
struct Pet {
    Pet(const std::string &name) : name(name) { }
    std::string name;
};

struct Dog : public Pet {
    Dog(const std::string &name) : Pet(name) { }
    std::string bark() const { return "woof!"; }
};
```

Inheritance, downcasting and polymorphism (2/6)

There are two different ways of indicating a hierarchical relationship to pybind11: the first specifies the C++ base class as an extra template parameter of the `class_`:

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def_readwrite("name", &Pet::name);

// Method 1: template parameter.
py::class_<Dog, Pet /* C++ parent type. */>(m, "Dog")
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);
```

Inheritance, downcasting and polymorphism (3/6)

Alternatively, we can also assign a name to the previously bound `Pet` `class_` object and reference it when binding the `Dog` class:

```
py::class_<Pet> pet(m, "Pet");
pet.def(py::init<const std::string &>())
   .def_readwrite("name", &Pet::name);

// Method 2: pass parent class_ object.
py::class_<Dog>(m, "Dog", pet /* <- Specify Python parent instance. */)
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);
```

Inheritance, downcasting and polymorphism (4/6)

Functionality-wise, both approaches are equivalent.

```
p = example.Dog("Molly")
p.name # Output: 'Molly'
p.bark() # Output: 'woof!'
```

The C++ classes defined above are regular non-polymorphic types with an inheritance relationship. This is reflected in Python:

```
// Return a base pointer to a derived instance.
m.def("pet_store", [](){ return std::unique_ptr<Pet>(new Dog("Molly")); });
```

```
p = example.pet_store()
type(p) # Output: Pet
# No pointer downcasting for regular non-polymorphic types.
p.bark() # Output: AttributeError: 'Pet' object has no attribute 'bark'
```

Inheritance, downcasting and polymorphism (5/6)

The function returned a `Dog` instance, but because it's a non-polymorphic type behind a base pointer, Python only sees a `Pet`. In C++, a type is only considered polymorphic if it has at least one virtual function and pybind11 will automatically recognize this:

```
struct PolymorphicPet {
    virtual ~PolymorphicPet() = default;
};

struct PolymorphicDog : public PolymorphicPet {
    std::string bark() const { return "woof!"; }
};

py::class_<PolymorphicPet>(m, "PolymorphicPet");
py::class_<PolymorphicDog, PolymorphicPet>(m, "PolymorphicDog")
    .def(py::init<>())
    .def("bark", &PolymorphicDog::bark);

// Again, return a base pointer to a derived instance.
m.def("pet_store2", []() { return std::unique_ptr<PolymorphicPet>(new PolymorphicDog); });
```

Inheritance, downcasting and polymorphism (6/6)

```
p = example.pet_store2()
type(p) # Output: PolymorphicDog
p.bark() # Output: 'woof!'
```

Given a pointer to a polymorphic base, pybind11 performs automatic downcasting to the actual derived type. Note that this goes beyond the usual situation in C++: we don't just get access to the virtual functions of the base, we get the concrete derived type including functions and attributes that the base type may not even be aware of.

Overloaded methods (1/2)

Sometimes there are several overloaded C++ methods with the same name taking different kinds of input arguments:

```
struct Pet {
    Pet(const std::string &name, int age) : name(name), age(age) { }

    void set(int age_) { age = age_; }
    void set(const std::string &name_) { name = name_; }

    std::string name;
    int age;
};
```

Attempting to bind `Pet::set` will cause an error since the compiler does not know which method the user intended to select.

Overloaded methods (2/2)

We can disambiguate by casting them to function pointers. Binding multiple functions to the same Python name automatically creates a chain of function overloads that will be tried in sequence.

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &, int>())
    .def("set", static_cast<void (Pet::*)(int)>(&Pet::set), "Set the pet's age")
    .def("set", static_cast<void (Pet::*)(const std::string &)>(&Pet::set), "Set the pet's name");
```

The overload signatures are also visible in the method's docstring. If you have a C++14 compatible compiler, you can use an alternative syntax to cast the overloaded function:

```
py::class_<Pet>(m, "Pet")
    .def("set", py::overload_cast<int>(&Pet::set), "Set the pet's age")
    .def("set", py::overload_cast<const std::string &>(&Pet::set), "Set the pet's name");
```

Here, `py::overload_cast` only requires the parameter types to be specified. The return type and class are deduced.

Overloaded `const` methods

If a function is overloaded based on constness, the `py::const_` tag should be used:

```
struct Widget {
    int foo(int x, float y);
    int foo(int x, float y) const;
};

py::class_<Widget>(m, "Widget")
    .def("foo_mutable", py::overload_cast<int, float>(&Widget::foo))
    .def("foo_const",   py::overload_cast<int, float>(&Widget::foo, py::const_));
```

Note: this approach also works for multiple overloaded constructors.

Enumerations and internal types (1/2)

Let's now suppose that the example class contains internal types like enumerations, e.g.:

```
struct Pet {
    enum Kind {
        Dog = 0,
        Cat
    };

    struct Attributes {
        float age = 0;
    };

    Pet(const std::string &name, Kind type) : name(name), type(type) { }

    std::string name;
    Kind type;
    Attributes attr;
};
```

Enumerations and internal types (2/2)

The binding code for this example looks as follows:

```
py::class_<Pet> pet(m, "Pet");

pet.def(py::init<const std::string &, Pet::Kind>())
    .def_readwrite("name", &Pet::name)
    .def_readwrite("type", &Pet::type)
    .def_readwrite("attr", &Pet::attr);

py::enum_<Pet::Kind>(pet, "Kind")
    .value("Dog", Pet::Kind::Dog)
    .value("Cat", Pet::Kind::Cat)
    .export_values();

py::class_<Pet::Attributes>(pet, "Attributes")
    .def(py::init<>())
    .def_readwrite("age", &Pet::Attributes::age);
```

Nesting and scoping

To ensure that the nested types `Kind` and `Attributes` are created within the scope of `Pet`, the `pet` `class_` instance must be supplied to the `enum_` and `class_` constructor. The `enum_::export_values` function exports the enum entries into the parent scope, which should be skipped for newer C++11-style enum classes.

```
p = Pet("Lucy", Pet.Cat)
p.type # Output: Kind.Cat
int(p.type) # Output: 1L
```

The entries defined by the enumeration type are exposed in the `__members__` property:

```
Pet.Kind.__members__ # Output: {'Dog': Kind.Dog, 'Cat': Kind.Cat}
```

The `__name__` property returns the name of the enum value as a string. Contrary to Python customs, enum values from the wrappers should not be compared using `is`, but with `==`.

How to build and import pybind11 modules

How to build and import pybind11 modules

The Python example and CMake example repositories good places to start to understand how to build and import pybind11 modules. There are three main ways to do it:

1. Manual compilation.
2. Compilation using `CMake`.
3. Compilation using Python packaging.

In order to be able to import the compiled module in Python, add the folder containing your dynamic library to the environment variable `PYTHONPATH` accordingly.

Manual compilation

To manually compile C++ code with pybind11, use one of the following commands:

```
# pybind11 installed via pip or conda.
g++ -O3 -Wall -shared -std=c++11 -fPIC \
    $(python3 -m pybind11 --includes) \
    example.cpp -o example$(python3-config --extension-suffix)

# pybind11 built from source.
g++ -std=c++11 -O3 -shared -fPIC \
    -I/path/to/pybind11/include $(python3-config --cflags --ldflags --libs) \
    example.cpp -o example$(python3-config --extension-suffix)

# pybind11 included as a Git submodule.
g++ -std=c++11 -O3 -shared -fPIC \
    -Iextern/pybind11/include $(python3-config --cflags --ldflags --libs) \
    example.cpp -o example$(python3-config --extension-suffix)
```

On macOS: add the `-undefined dynamic_lookup` flag so as to ignore missing symbols when building the module.

How to compile using CMake (1/2)

To compile and run your pybind11 code with CMake, create a `CMakeLists.txt` script as follows:

```
cmake_minimum_required(VERSION 3.15)
project(example)

set(PYBIND11_FINDPYTHON ON)
find_package(pybind11 CONFIG REQUIRED)

pybind11_add_module(example example.cpp)
```

or, if pybind11 is included as a subfolder:

```
cmake_minimum_required(VERSION 3.15)
project(example)

add_subdirectory(pybind11)
pybind11_add_module(example example.cpp)
```

How to compile using CMake (2/2)

Then:

```
cmake -S . -B build -Dpybind11_DIR=/path/to/pybind ..
cmake --build build -j<N>
```

Please refer to the CMake example repository for further details.

How to compile using CMake: recommended project structure

```
my_project/
├── CMakeLists.txt
├── src/
│   ├── example.{hpp,cpp} # C++ implementation.
│   └── bindings.cpp      # Python bindings.
├── python/
│   ├── init.py
│   └── wrapper.py        # Python wrapper/utilities.
└── tests/
    ├── test_cpp.cpp      # C++ tests.
    └── test_python.py    # Python tests.
```

Tip: Separate core logic (C++) from bindings for better maintainability.

How to build using modern Python packaging (1/4)

Traditional `setup.py` is being replaced by `pyproject.toml` (PEP 517/518). Using scikit-build-core is nowadays the recommended way. The `pyproject.toml` file looks like:

```toml
[build-system]
requires = ["scikit-build-core>=0.8", "pybind11"]
build-backend = "scikit_build_core.build"

[project]
name = "example_package"
version = "0.1.0"
requires-python = ">=3.8"
dependencies = []

[project.optional-dependencies]
test = ["pytest>=7.0"]

[tool.scikit-build]
cmake.minimum-version = "3.15"
cmake.build-type = "Release"
```

How to build using modern Python packaging (2/4)

`CMakeLists.txt` (compatible with scikit-build-core):

```
cmake_minimum_required(VERSION 3.15)
project(${SKBUILD_PROJECT_NAME} VERSION ${SKBUILD_PROJECT_VERSION})

set(PYBIND11_FINDPYTHON ON)
find_package(pybind11 CONFIG REQUIRED)

pybind11_add_module(_core MODULE src/bindings.cpp)

install(TARGETS _core DESTINATION ${SKBUILD_PROJECT_NAME})
```

Build and install:

```
pip install .             # Install package.
pip install -e .          # Install in editable mode (for development).
pip install -e ".[test]" # Install with test dependencies.
```

How to build using modern Python packaging (3/4)

Alternative: using setuptools with `pyproject.toml`:

```toml
[build-system]
requires = ["setuptools>=61", "wheel", "pybind11>=2.11"]
build-backend = "setuptools.build_meta"

[project]
name = "example_package"
version = "0.1.0"
dependencies = ["numpy>=1.21"]

[tool.setuptools]
packages = ["example_package"]

[tool.setuptools.package-data]
example_package = ["*.so", "*.pyd"]
```

How to build using modern Python packaging (4/4)

`setup.py` (simplified for setuptools):

```
from setuptools import setup, Extension
from pybind11.setup_helpers import Pybind11Extension, build_ext

ext_modules = [Pybind11Extension(
                    "example_package._core",
                    ["src/bindings.cpp"],
                    cxx_std=17,
                    extra_compile_args=["-O3"],
                )]

setup(cmdclass={"build_ext": build_ext},
      ext_modules=ext_modules)
```

Further reading:

- Python packaging guide
- scikit-build-core docs

Type conversions and performance tips

Automatic type conversions:

C++ type	Python type	Header required
`int`, `long`, `double`	`int`, `float`	`pybind11/pybind11.h`
`std::string`, `char*`	`str`	`pybind11/pybind11.h`
`std::vector<T>`	`list`	`pybind11/stl.h`
`std::map<K,V>`	`dict`	`pybind11/stl.h`
`std::array<T,N>`	`tuple`	`pybind11/stl.h`
Raw arrays	`numpy.ndarray`	`pybind11/numpy.h`
`Eigen::Matrix`	`numpy.ndarray`	`pybind11/eigen.h`

Two recommendations

Memory management

```
// ❌ Python won't manage this memory.
m.def("get_array", []() { return new int[100]; });

// ✅ Use smart pointers, instead!
m.def("get_array", []() { return std::make_shared<std::vector<int>>(100); });
```

Exception handling

```
// ❌ C++ exceptions don't propagate to Python.
void risky_function() { throw std::runtime_error("Error!"); }

// ✅ But you can wrap them!
py::register_exception<MyException>(m, "PyMyException");
```

Examples

C++ vs. native Python benchmark (1/2)

The example provided in the `c++_vs_py` folder showcases a performance comparison test between bound C++ code and native Python code on a simple linear algebra operation, such as a matrix-matrix product.

The code can be compiled with:

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

C++ vs. native Python benchmark (2/2)

Typically, the C++ implementation should be significantly faster than the pure Python implementation for several reasons:

- **Execution speed**: C++ is a compiled language and is generally faster than Python, an interpreted language, especially for computation-intensive tasks.
- **Optimization**: Compilers for C++ can optimize the code for performance, whereas Python's flexibility and dynamic typing can introduce overhead.
- **Handling of loops**: C++ is more efficient in handling loops and arithmetic operations compared to Python.

Notes

- The actual performance gain can vary depending on the system, the size of the matrices, and compiler optimizations.
- For matrix operations, libraries like NumPy in Python are highly optimized (uses BLAS/LAPACK underneath) and can offer performance close to C++, but in this comparison, we are using a pure Python implementation to illustrate the difference more clearly.
- For custom algorithms not available in NumPy, C++ bindings are invaluable.
- Remember that developing and maintaining C++ code requires more effort compared to Python, so the decision to use C++ should consider both performance benefits and development costs.

Other examples

The examples provided in the `examples/` folder are adapted and extended versions from this GitHub repository . They demonstrate the use of pybind11 in various scenarios.

Further readings

- pybind11 documentation
- pybind11 testsuite
- Using pybind11

Python's ecosystem for scientific computing.