Automatic Python/C++ bindings

Clair / c2py

Olivier Parcollet

CCQ

FLATIRON INSTITUTE

Goal

- Automatic interfacing between C++ and Python
    - Have the C++ compiler (LLVM/Clang) write the binding code for us.

- Why ?
    - Interface is large (many functions/classes), it evolves over time
    - Multiple languages issue: C++, Python, Julia, (Rust, Fortran, C, Matlab) …
    - Similar API between languages. Subset of C++.

How far can we automate this task ?

Status

- Partial rewrite of a tool used in our TRIQS project since 2014 (*TRIQS/cpp2py*)
  Collaborators *M. Ferrero (Paris), N. Wentzell (CCQ), TRIQS developers.*

- Beta version 0.1. Used in some TRIQS based projects.

- Comparable tools: *SWIG, cppyy (from CERN Root), binder (+pybind11),*

A compiler plugin and a library

- A plugin for the LLVM/clang C++ compiler : clair/c2py
  - Generates Python/C++ binding code.
  - A well established technology (*LLVM/clang LibTooling*) plugins/tools to generate, check, rewrite C++ code, e.g. *clang-tidy*.

    Clair : *CLANG based Introspection and Reflection tools*

- A small C++20 library for the bindings *c2py*. Similar role as *pybind11/nanobind*

https://github.com/flatironinstitute/clair

https://github.com/flatironinstitute/c2py

Outline

- A few examples/demo
- How far can we automate ? Customization.
- Calling back : calling Python from C++.

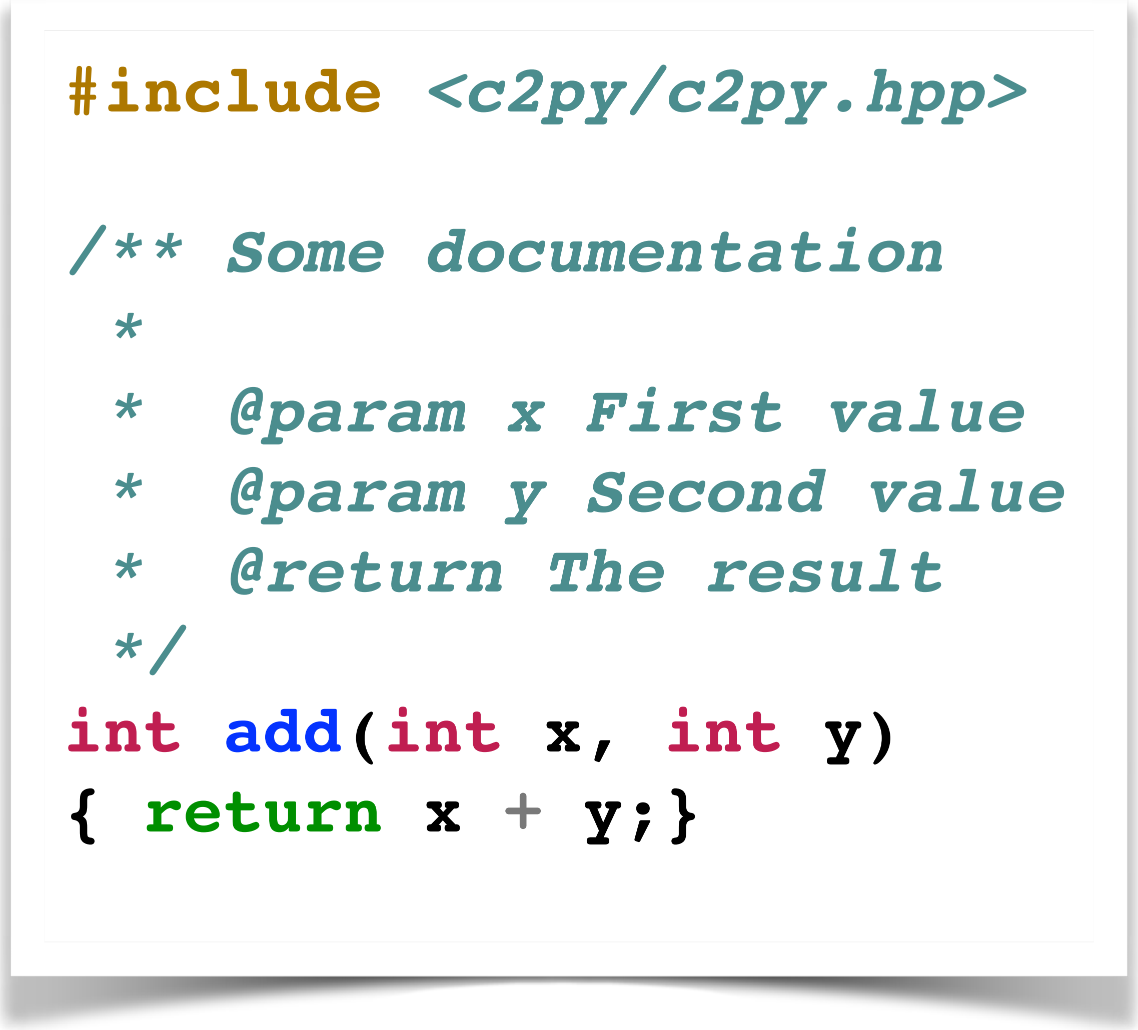
Demo

A few simple examples

Example 1: a simple function

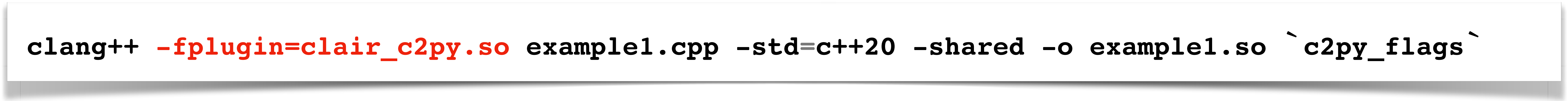
https://github.com/flatironinstitute/sciware/blob/main/30_CCQ/clair/example1.cpp

- C++ code



```
#include <c2py/c2py.hpp>

/** Some documentation
 *
 *   @param x First value
 *   @param y Second value
 *   @return The result
 */
int add(int x, int y)
{ return x + y;}
```

- Compile



```
clang++ -fplugin=clair_c2py.so example1.cpp -std=c++20 -shared -o example1.so `c2py_flags`
```

- Use

```
>>> import example1 as M
>>> M.add(1,2)
3
```

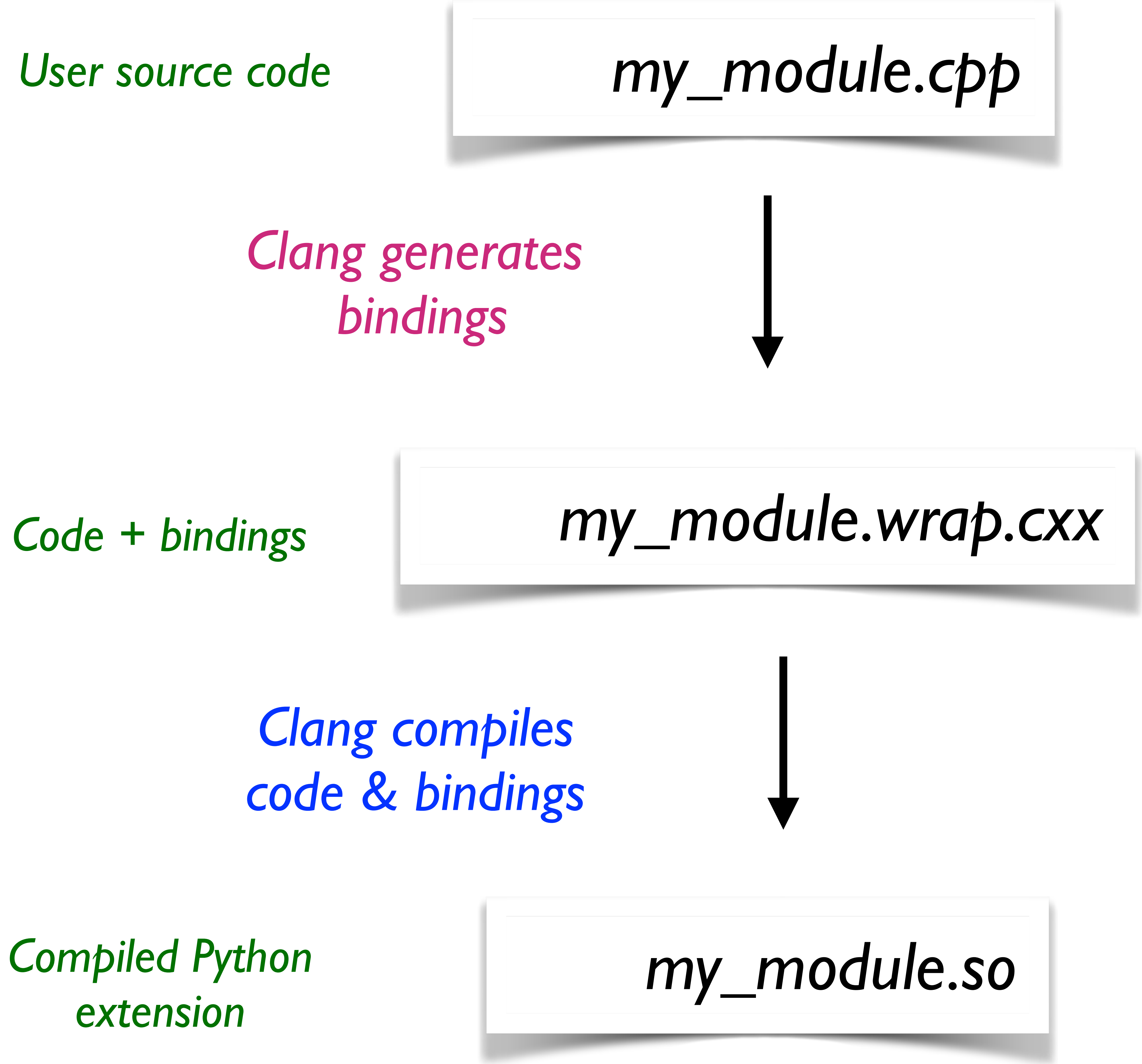
```
>>> help(M.add)
    add(...)
        Dispatched C++ function
        [1]  (x: int, y: int) -> int

           Some documentation

           Parameters
           ----------

           x:
              First value
           y:
              Second value

           Returns
           -------

           The result
```

What happened ?

```
clang++ -fplugin=clair_c2py.so my_module.cpp -std=c++20 -shared -o my_module.so `c2py_flags`
```

- The plugin modifies the compilation process

1. Parse the C++ code.
2. Generate Python bindings
3. Continue compiling code + bindings.



User source code

my_module.cpp

Clang generates bindings

Code + bindings

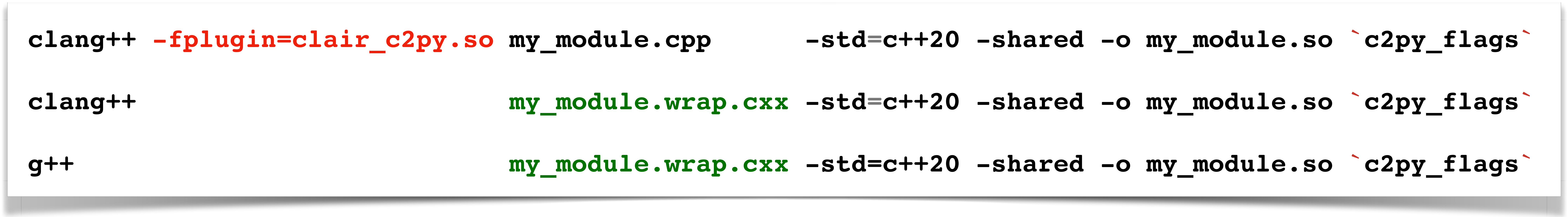
my_module.wrap.cxx

Clang compiles code & bindings

Compiled Python extension

my_module.so

Reuse bindings with other compilers/platforms

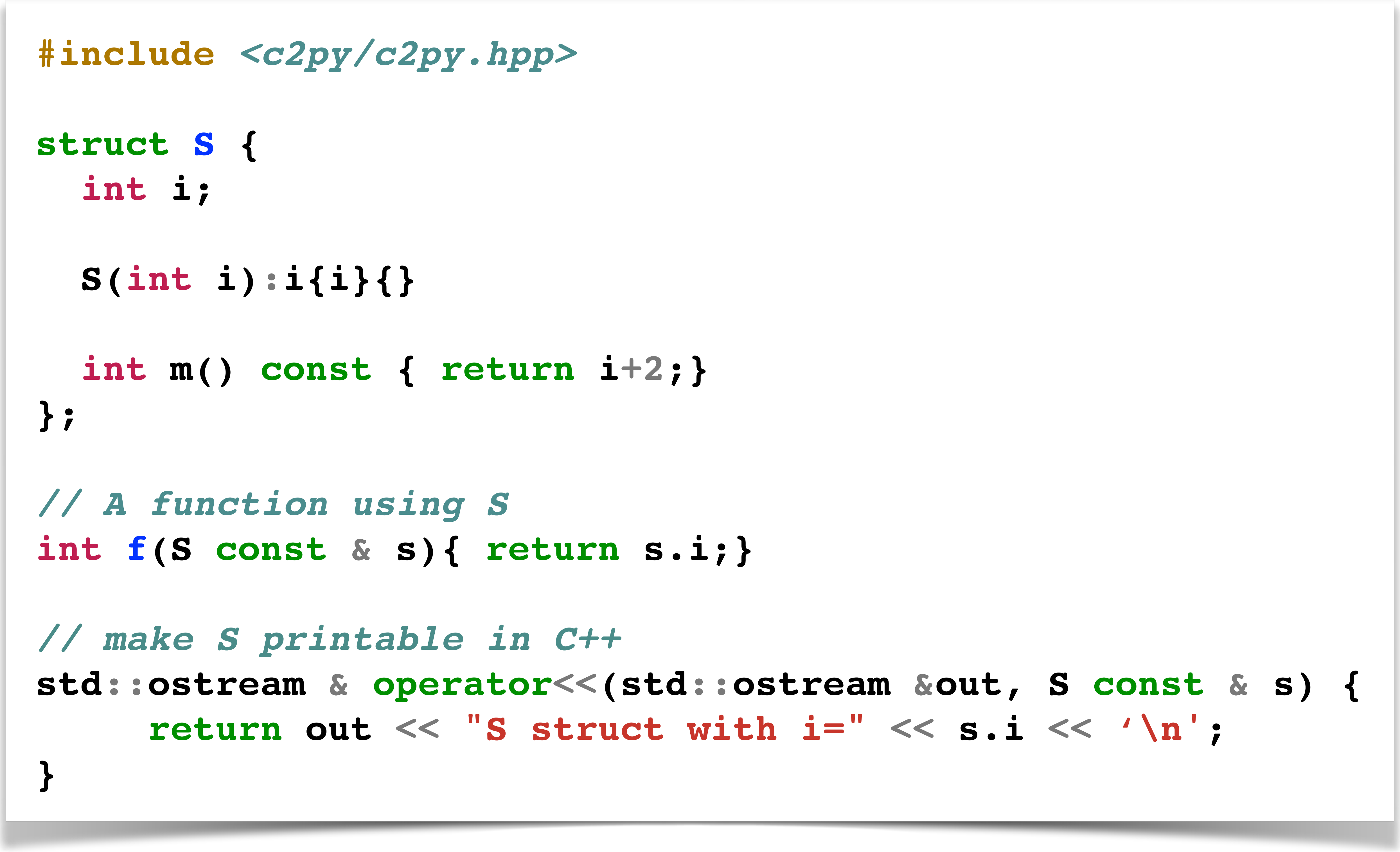


```
clang++ -fplugin=clair_c2py.so my_module.cpp      -std=c++20 -shared -o my_module.so `c2py_flags`

clang++                        my_module.wrap.cxx -std=c++20 -shared -o my_module.so `c2py_flags`

g++                            my_module.wrap.cxx -std=c++20 -shared -o my_module.so `c2py_flags`
```

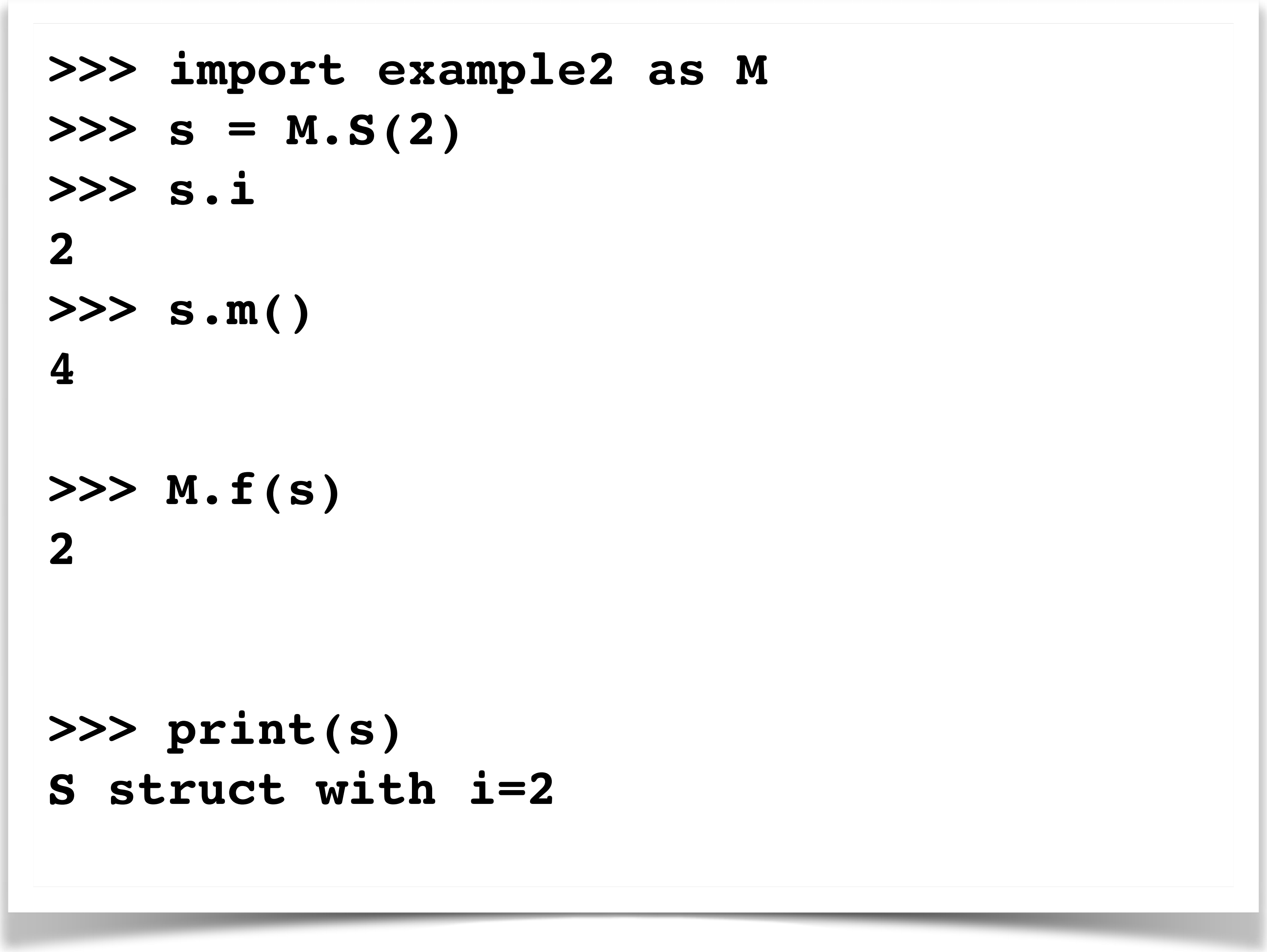
- Develop with Clang and Clair plugin.
- Use the bindings with any compiler
  - The binding code depends only on *Python C API*

Notebook demo

Example 2: a struct



```cpp
#include <c2py/c2py.hpp>

struct S {
  int i;

  S(int i):i{i}{}

  int m() const { return i+2;}
};

// A function using S
int f(S const & s){ return s.i;}

// make S printable in C++
std::ostream & operator<<(std::ostream &out, S const & s) {
     return out << "S struct with i=" << s.i << ‘\n';
}
```



```
>>> import example2 as M
>>> s = M.S(2)
>>> s.i
2
>>> s.m()
4

>>> M.f(s)
2


>>> print(s)
S struct with i=2
```

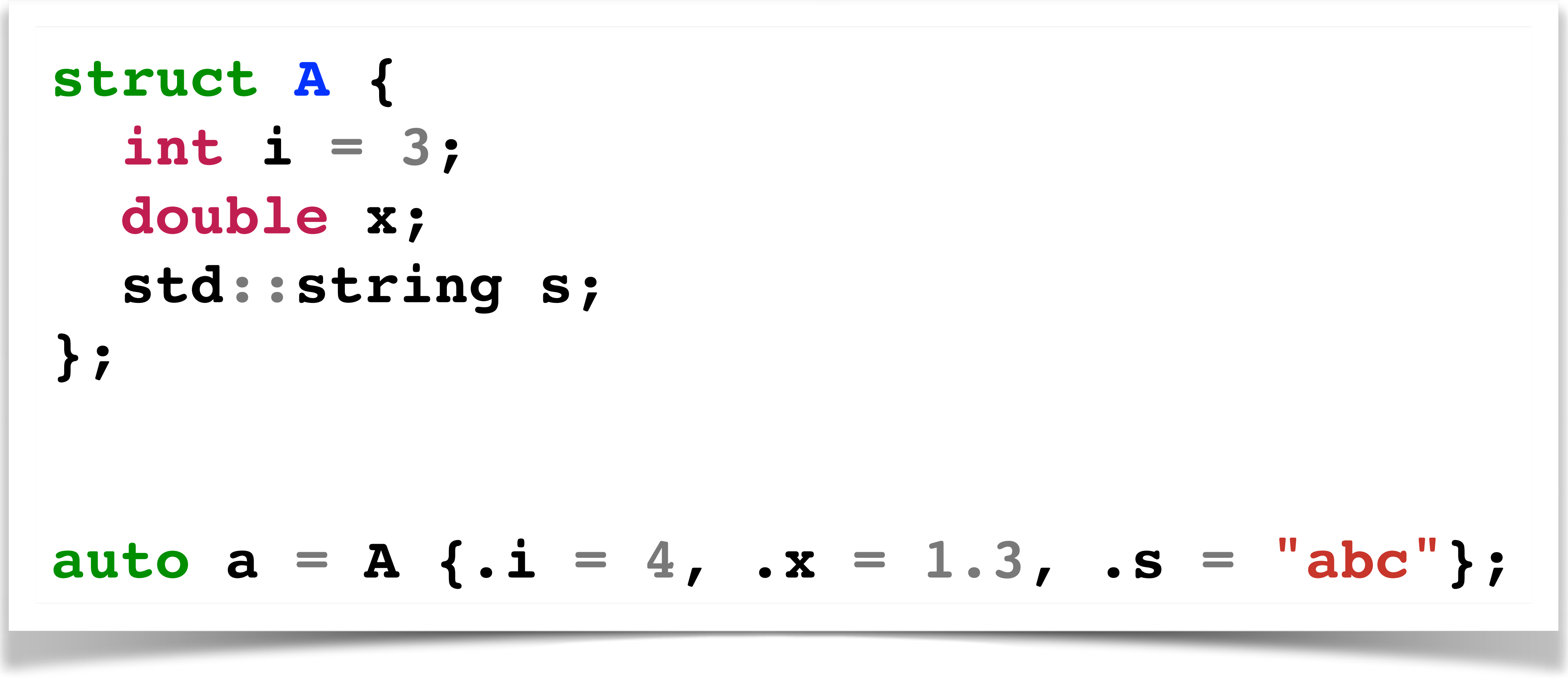
```
clang++ -fplugin=clair_c2py.so example2.cpp -std=c++20 -shared -o example2.so `c2py_flags`
```

https://github.com/flatironinstitute/sciware/blob/main/30_CCQ/clair/example2.cpp

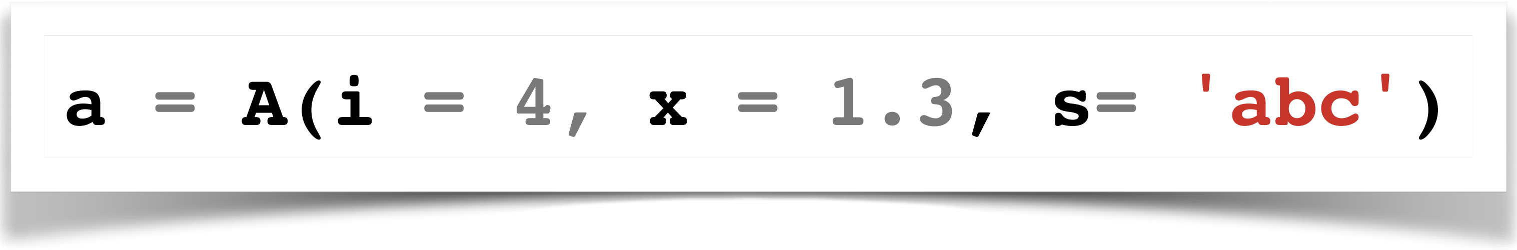
Example 3: simple struct with no constructor

C++

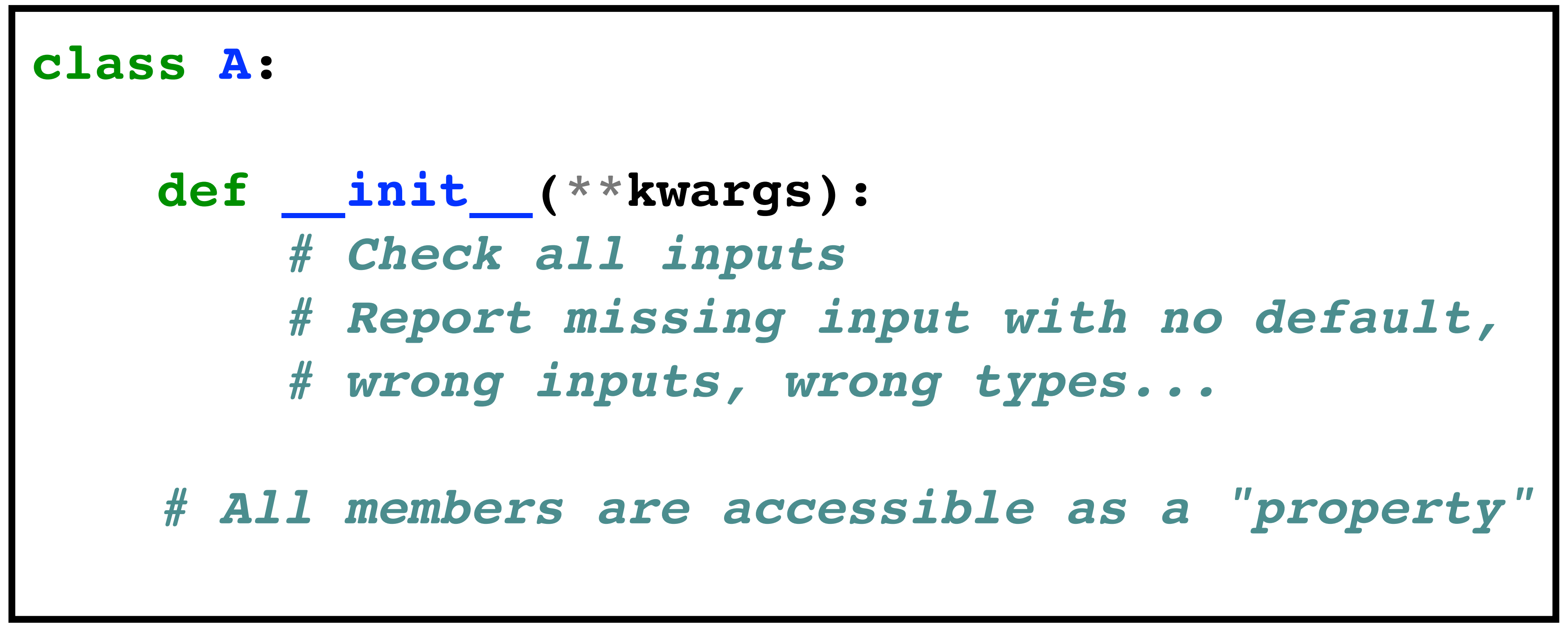
Python



```cpp
struct A {
  int i = 3;
  double x;
  std::string s;
};


auto a = A {.i = 4, .x = 1.3, .s = "abc"};
```



```python
a = A(i = 4, x = 1.3, s= 'abc')
```



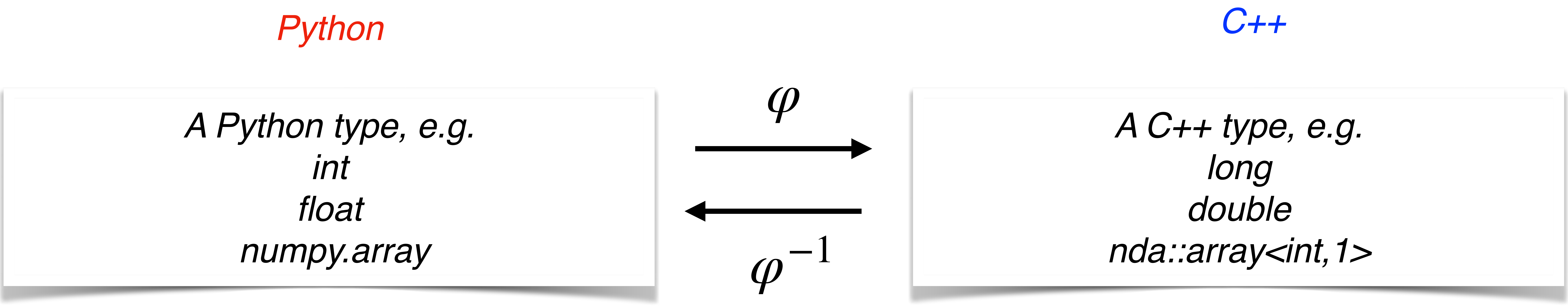
```python
class A:

    def __init__(**kwargs):
        # Check all inputs
        # Report missing input with no default,
        # wrong inputs, wrong types...

    # All members are accessible as a "property"
```

- Synthesize a Python constructor with keywords arguments (with checks).

https://github.com/flatironinstitute/sciware/blob/main/30_CCQ/clair/example3.cpp

How general is such tool ?

Type Conversion



Python

C++

A Python type, e.g.
int
float
numpy.array

φ

φ^{-1}

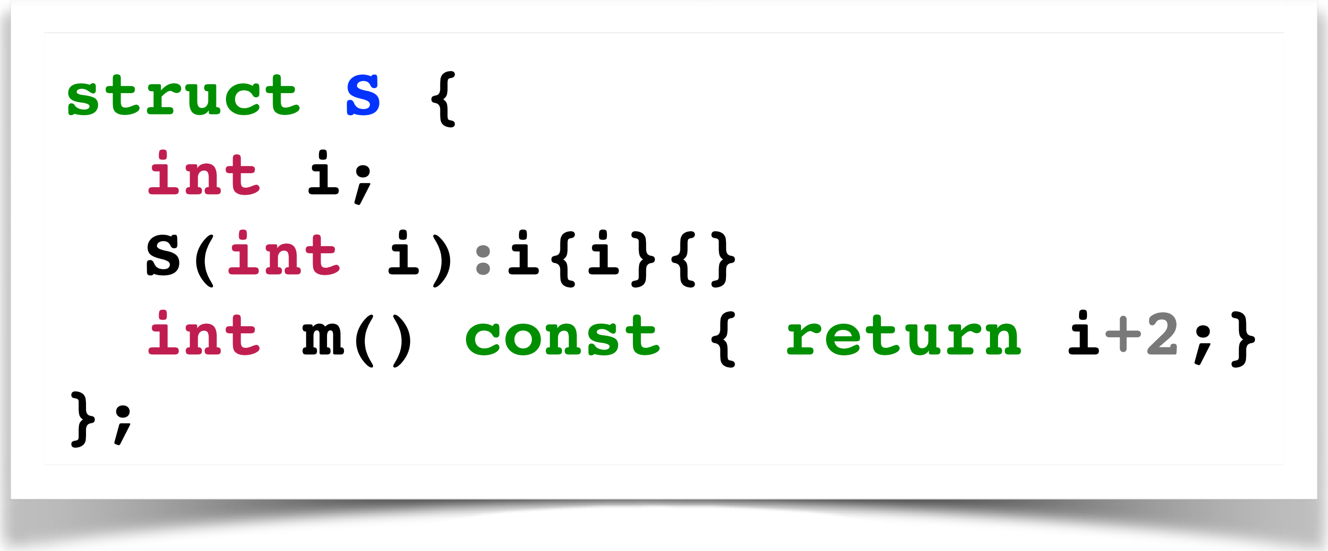
A C++ type, e.g.
long
double
nda::array<int,1>

- Calling a C++ function from Python
  - Convert arguments | Call | Convert result.

$$f_{Py}(x_1, \ldots, x_n) = \varphi^{-1}\left(f_{C++}\left(\varphi(x_1), \ldots, \varphi(x_n)\right)\right)$$

- The reverse is also possible (Python from C++, see later)

Wrapped type



```cpp
struct S {
  int i;
  S(int i):i{i}{}
  int m() const { return i+2;}
};
```

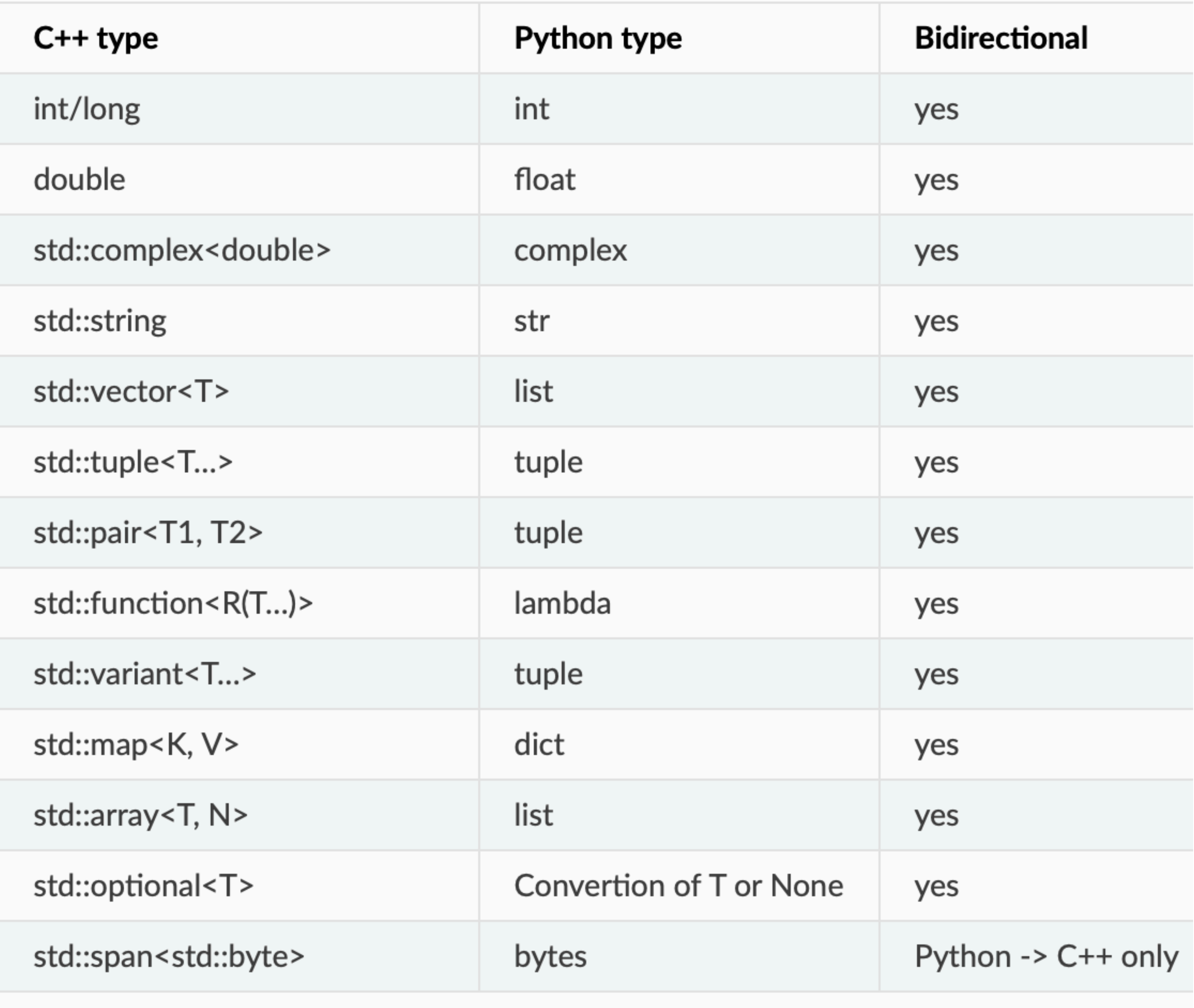
Python *C++*

$$\begin{tikzcd}
\textit{class S} \arrow[r, shift left, "\varphi"] & \textit{struct S;} \arrow[l, shift left, "\varphi^{-1}"]
\end{tikzcd}$$

- New Python type called ***S*** = a C++ ***S*** disguised as a Python object: same API, calling C++ methods behind the scene
- Lifetime managed by Python (reference counting).

What is convertible ?

- Basic types: `int, double, string, …`
- Wrapped types
- Any type with an explicit converter
  - Example: a third party library like
    `nda::array` or `Eigen` ↔ `numpy.array`
  - Standard library types
    - Composable e.g.
      `std::vector<std::tuple<int, int>>`
- Iterable types that yield convertible types



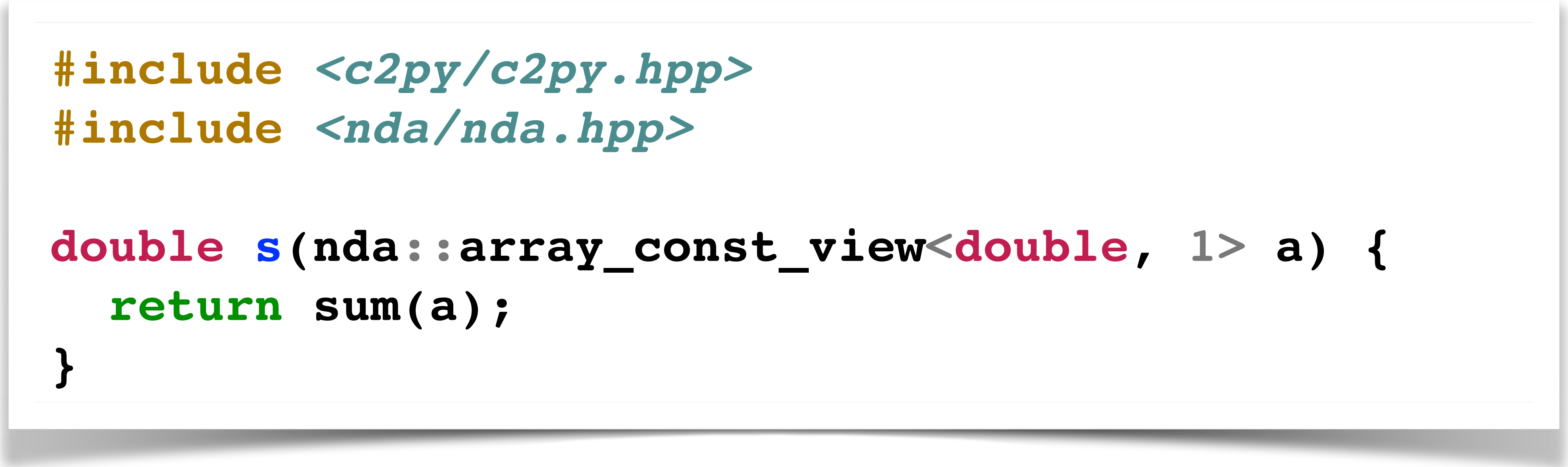
C++ type	Python type	Bidirectional
int/long	int	yes
double	float	yes
std::complex<double>	complex	yes
std::string	str	yes
std::vector<T>	list	yes
std::tuple<T...>	tuple	yes
std::pair<T1, T2>	tuple	yes
std::function<R(T...)>	lambda	yes
std::variant<T...>	tuple	yes
std::map<K, V>	dict	yes
std::array<T, N>	list	yes
std::optional<T>	Convertion of T or None	yes
std::span<std::byte>	bytes	Python -> C++ only

Third party libraries

https://github.com/TRIQS/nda

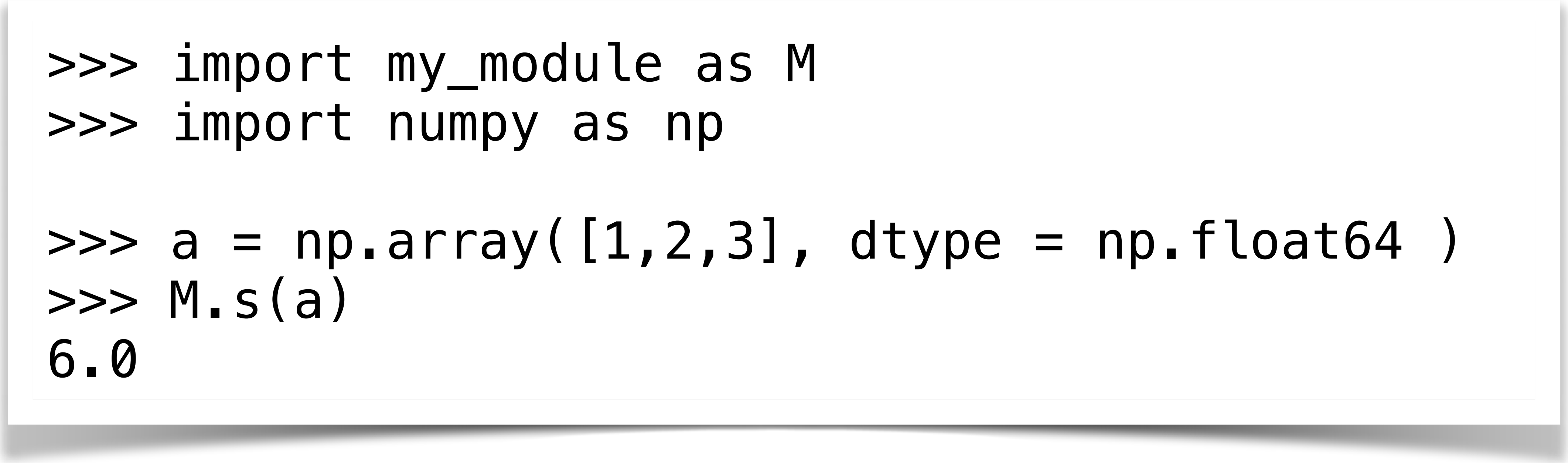
- Example: nda = CCQ C++20 N-dimensional array library
- Automatically include the proper converters (e.g. *nda::array* ⟷ *numpy*)

C++



```cpp
#include <c2py/c2py.hpp>
#include <nda/nda.hpp>

double s(nda::array_const_view<double, 1> a) {
  return sum(a);
}
```

Python

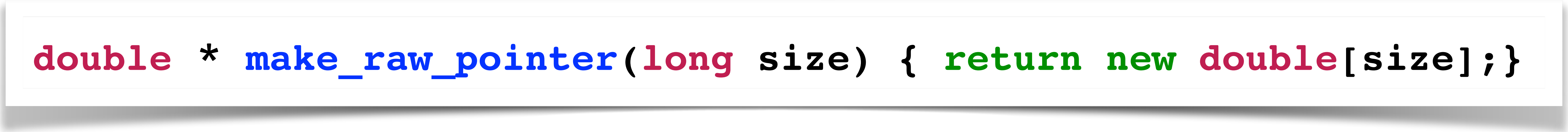


```
>>> import my_module as M
>>> import numpy as np

>>> a = np.array([1,2,3], dtype = np.float64 )
>>> M.s(a)
6.0
```

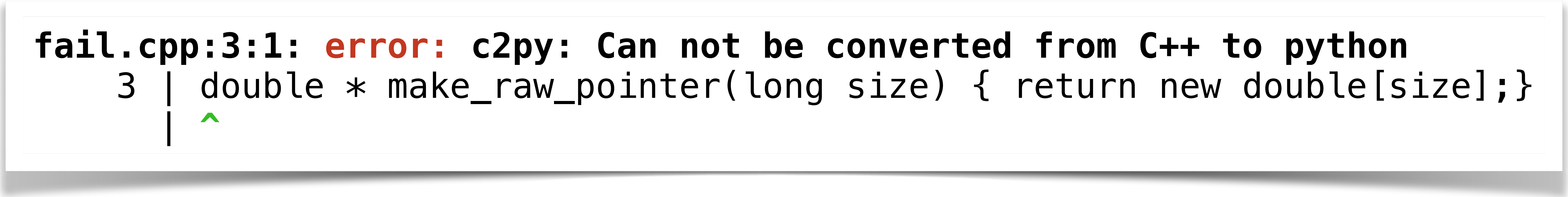
- Non intrusive. Adapt any C++ array library

What happens if not convertible ?

- The compiler will tell you
- Example: returning raw pointers
  - Issue : ownership of the data. Not recommended C++ API/practice anyway.



```
double * make_raw_pointer(long size) { return new double[size];}
```



```
fail.cpp:3:1: error: c2py: Can not be converted from C++ to python
    3 | double * make_raw_pointer(long size) { return new double[size];}
      | ^
```

- Non convertible type $\Longrightarrow$ compilation error
  - No surprise at runtime …

Dynamical dispatch

- Python function dynamically dispatches to all corresponding C++ overloads.

Python

g(x)

List of C++ functions

```
/// first overload
int g(int x) { return 1;}

/// second overload
int g(std::string const &x){ return 2;}
```

- At runtime: select which overloads to call, from the Python type of the arguments
- NB :Very similar to Julia. Except C++ is compiled *Ahead Of Time*, not *Just In Time*

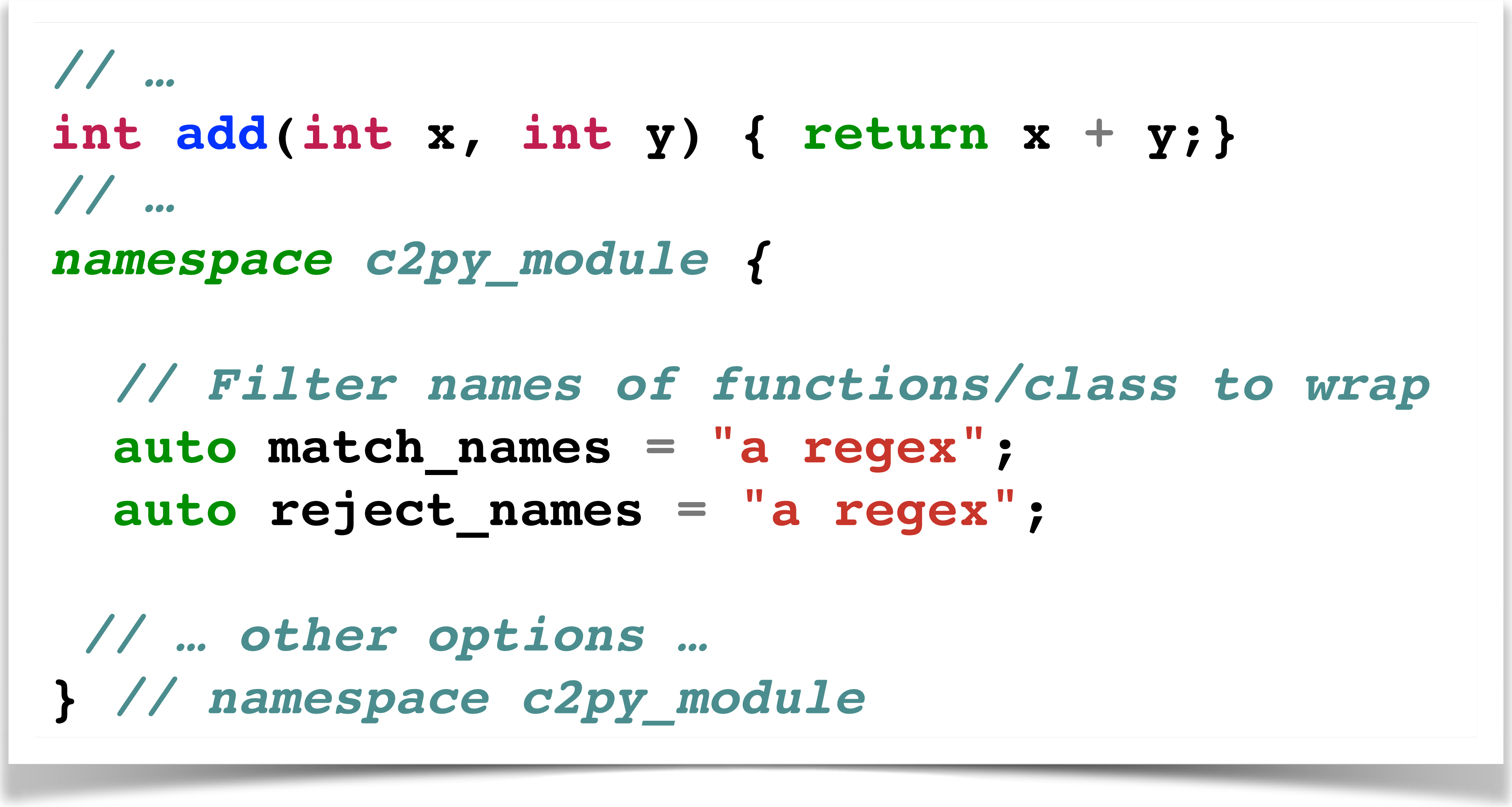
Customize

Customization

- Annotations in the code
- Options in reserved namespace *c2py_module*

- Filters
  - *match_names/ reject_names*
    - Regular expression to select only some function/classes.
    - Ignore standard library, files included with *-isystem (vs -I)*
- Other options, e.g.
  - Select template instantiations

my_module.cpp

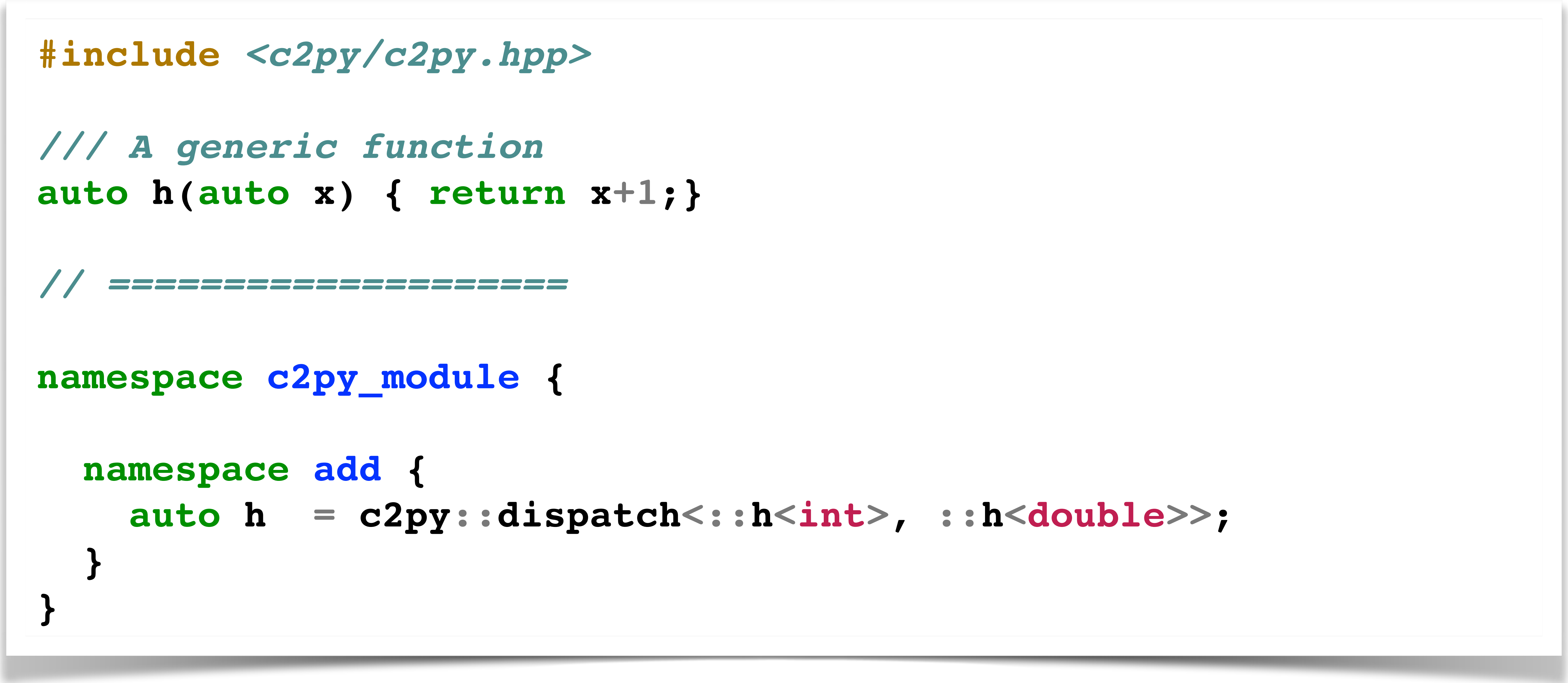


```
// …
int add(int x, int y) { return x + y;}
// …
namespace c2py_module {

  // Filter names of functions/class to wrap
  auto match_names = "a regex";
  auto reject_names = "a regex";

 // … other options …
} // namespace c2py_module
```

https://flatironinstitute.github.io/clair/latest/reference/customize.html

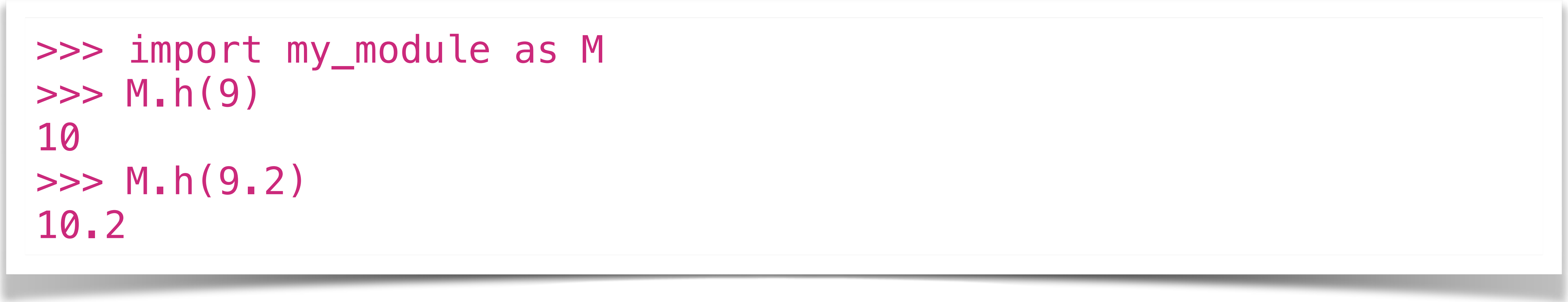
Generic (template) function

C++



```cpp
#include <c2py/c2py.hpp>

/// A generic function
auto h(auto x) { return x+1;}

// ====================

namespace c2py_module {

  namespace add {
    auto h  = c2py::dispatch<::h<int>, ::h<double>>;
  }
}
```

Python



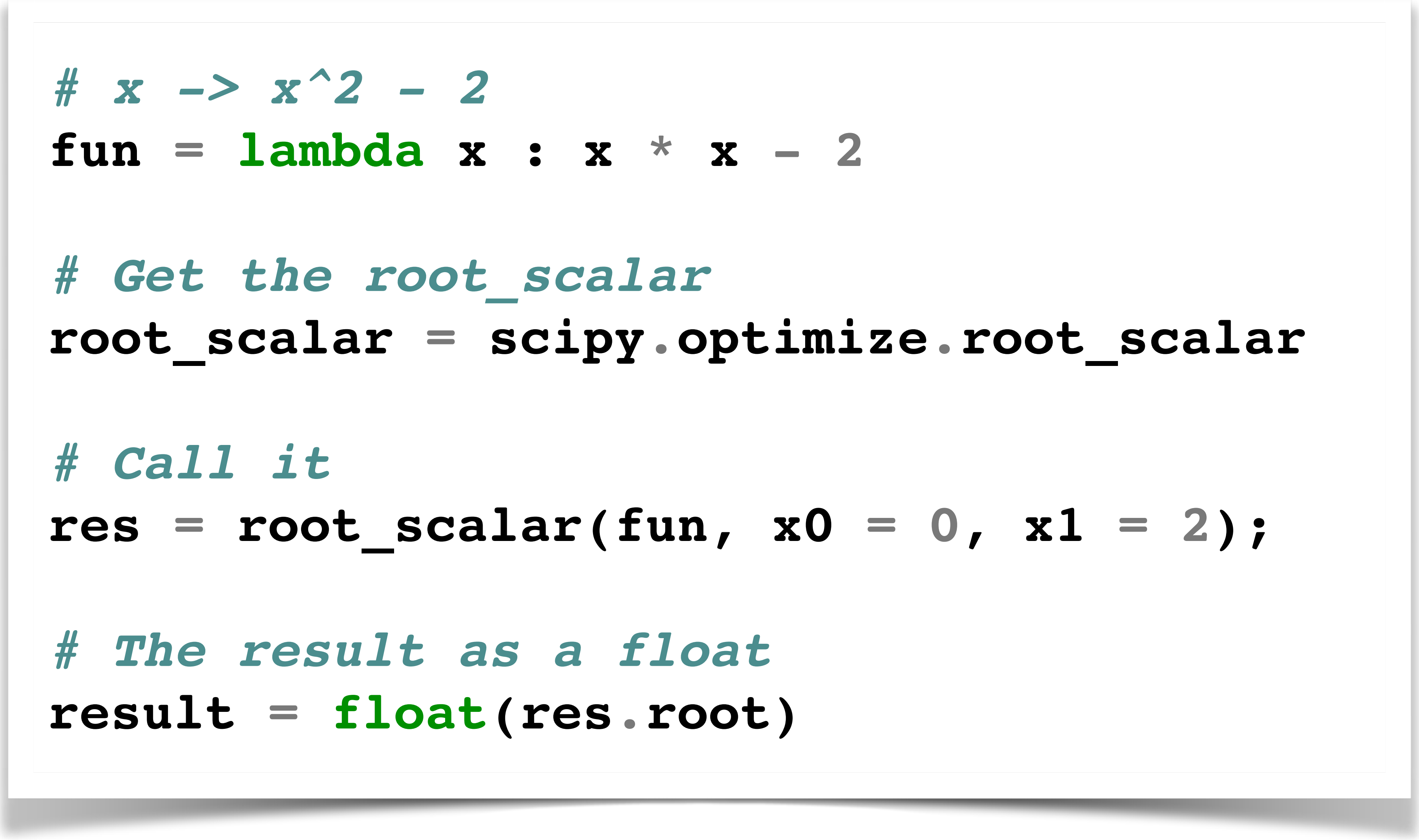
```
>>> import my_module as M
>>> M.h(9)
10
>>> M.h(9.2)
10.2
```

Call back

Calling back : Python from C++

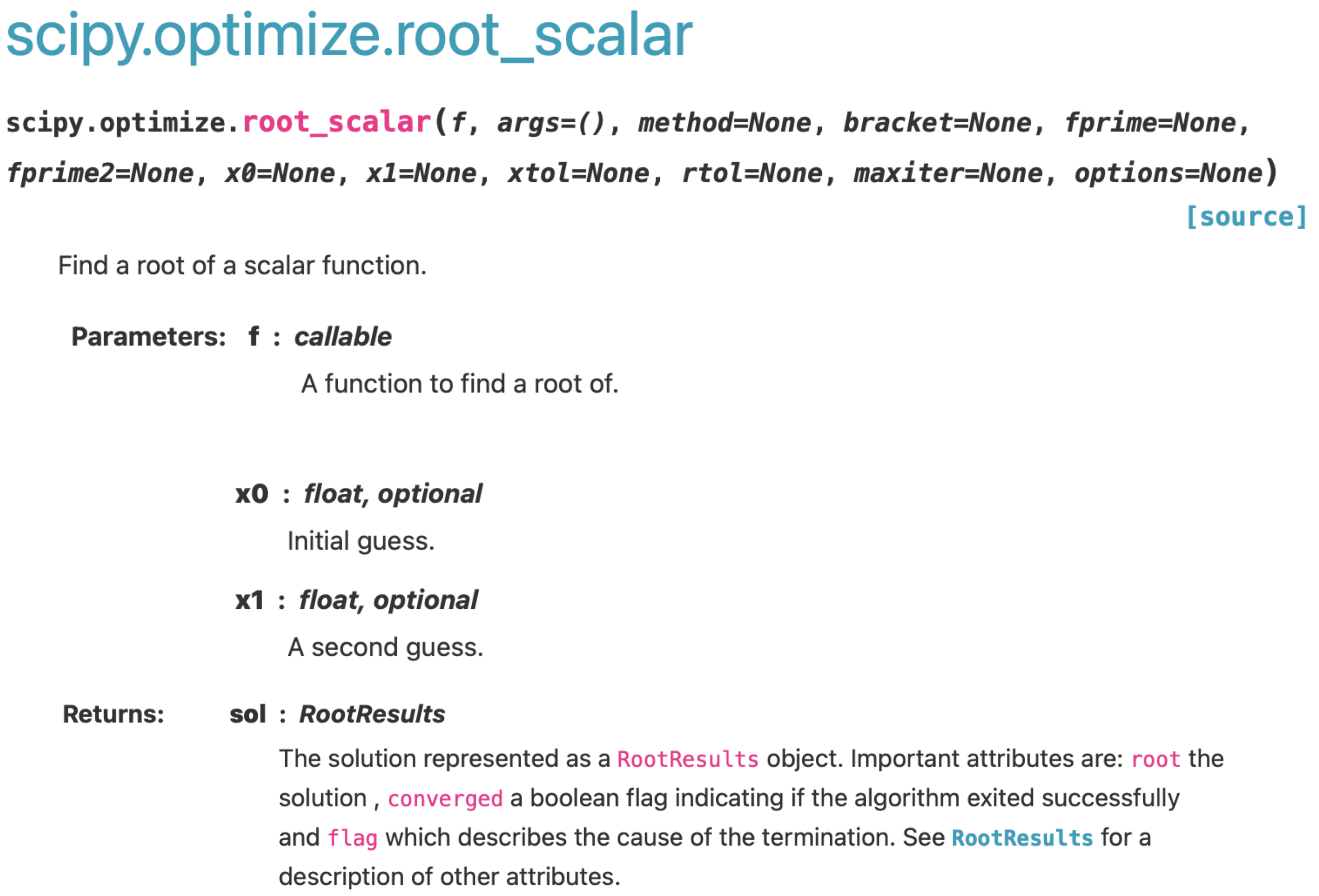
- Example : calling *scipy* root finder from C++

Python



```
# x -> x^2 - 2
fun = lambda x : x * x - 2

# Get the root_scalar
root_scalar = scipy.optimize.root_scalar

# Call it
res = root_scalar(fun, x0 = 0, x1 = 2);

# The result as a float
result = float(res.root)
```

Numpy documentation



scipy.optimize.root_scalar

`scipy.optimize.root_scalar(f, args=(), method=None, bracket=None, fprime=None, fprime2=None, x0=None, x1=None, xtol=None, rtol=None, maxiter=None, options=None)` [source]

Find a root of a scalar function.

Parameters: **f** : ***callable***

A function to find a root of.

x0 : ***float, optional***

Initial guess.

x1 : ***float, optional***

A second guess.

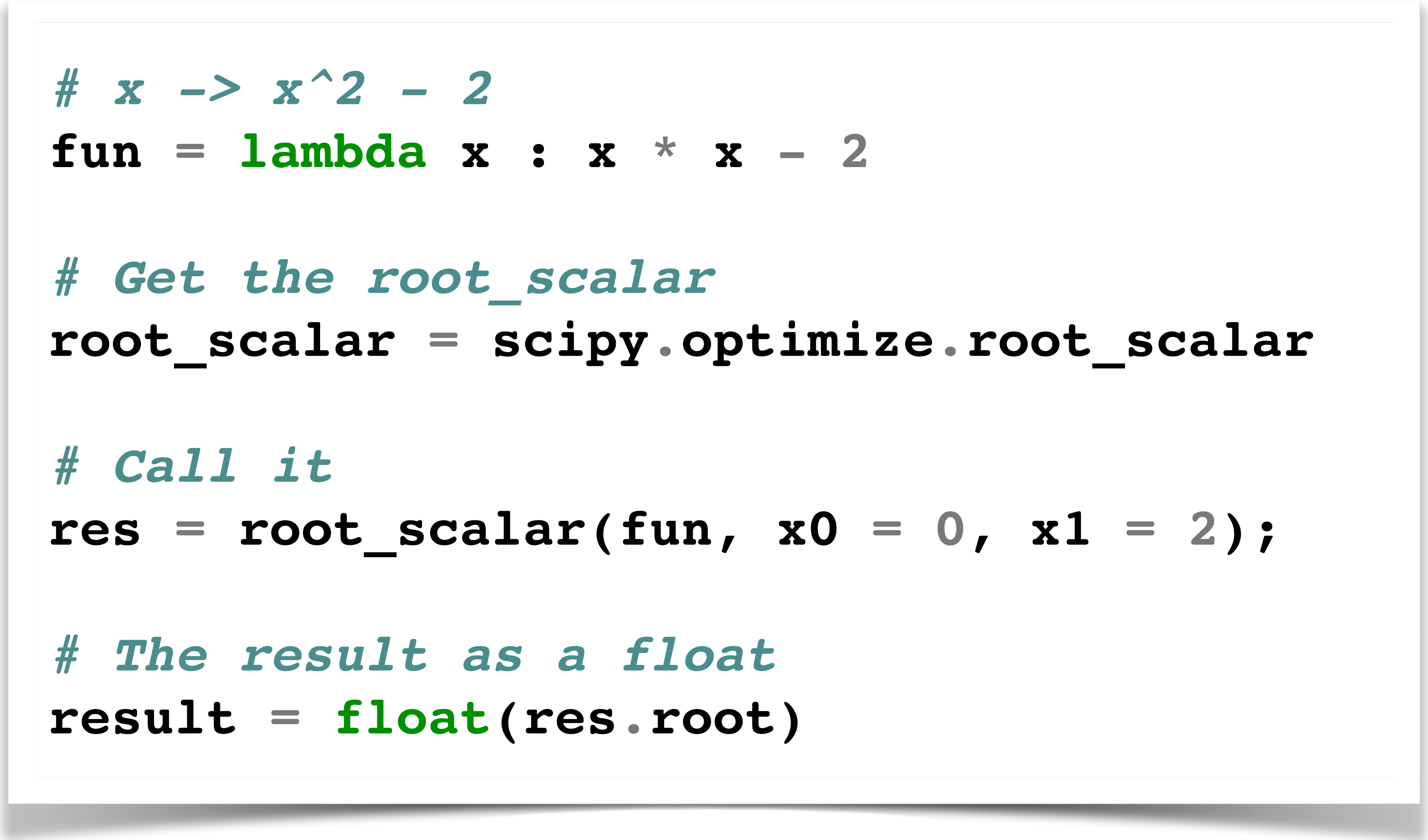
Returns: **sol** : ***RootResults***

The solution represented as a `RootResults` object. Important attributes are: `root` the solution , `converged` a boolean flag indicating if the algorithm exited successfully and `flag` which describes the cause of the termination. See `RootResults` for a description of other attributes.

Calling back : Python from C++

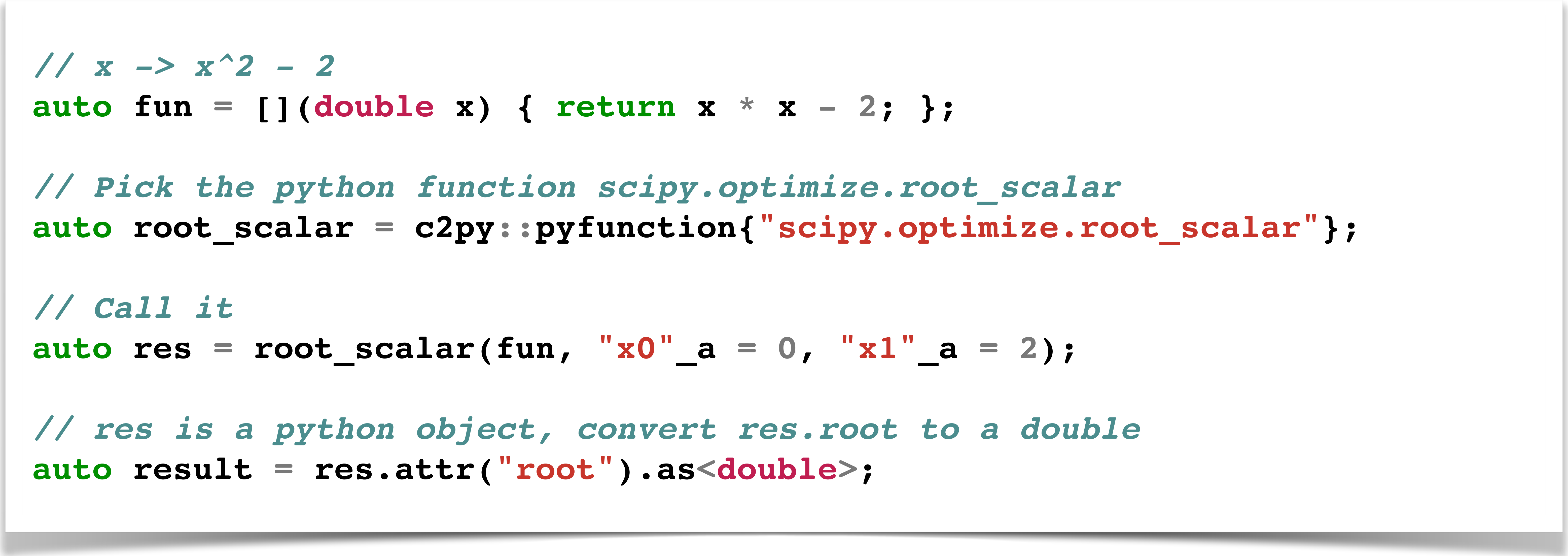
- Example : calling *scipy* root finder from C++

Python



```python
# x -> x^2 - 2
fun = lambda x : x * x - 2

# Get the root_scalar
root_scalar = scipy.optimize.root_scalar

# Call it
res = root_scalar(fun, x0 = 0, x1 = 2);

# The result as a float
result = float(res.root)
```

C++



```cpp
// x -> x^2 - 2
auto fun = [](double x) { return x * x - 2; };

// Pick the python function scipy.optimize.root_scalar
auto root_scalar = c2py::pyfunction{"scipy.optimize.root_scalar"};

// Call it
auto res = root_scalar(fun, "x0"_a = 0, "x1"_a = 2);

// res is a python object, convert res.root to a double
auto result = res.attr("root").as<double>;
```

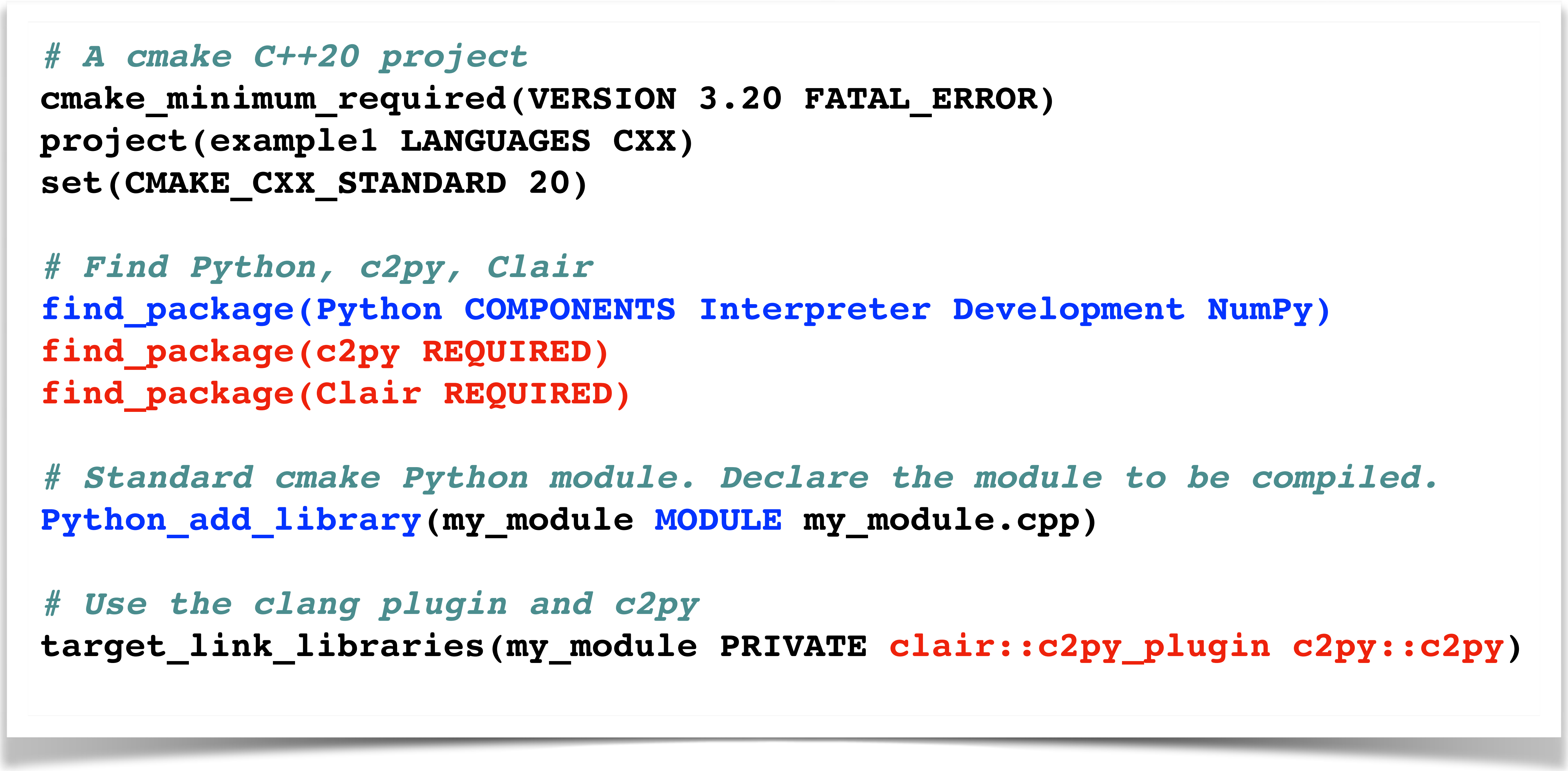
Documentation in progress

- Just *c2py*. No wrapping, no plugin.
- Similar code with pybind11. Python C API + some C++ helper functions.

CMake

CMake integration

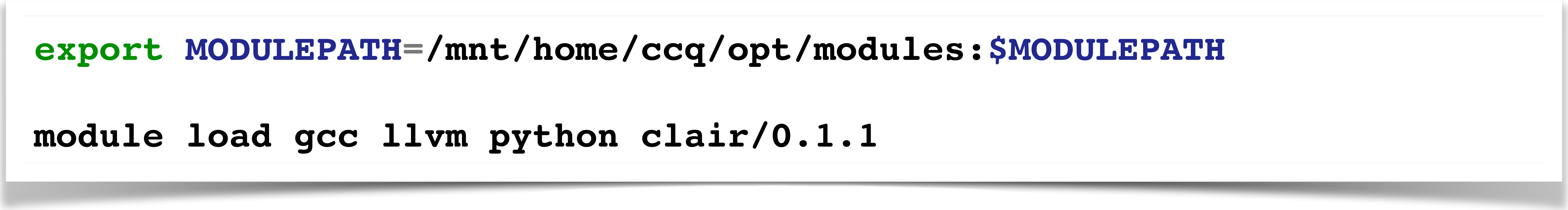
- Two "CMake packages" : *c2py* and *clair*
- Minimal addition to CMake standard procedure to compile a python module
- All other flags/libraries/targets are "as usual".



```
# A cmake C++20 project
cmake_minimum_required(VERSION 3.20 FATAL_ERROR)
project(example1 LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 20)

# Find Python, c2py, Clair
find_package(Python COMPONENTS Interpreter Development NumPy)
find_package(c2py REQUIRED)
find_package(Clair REQUIRED)

# Standard cmake Python module. Declare the module to be compiled.
Python_add_library(my_module MODULE my_module.cpp)

# Use the clang plugin and c2py
target_link_libraries(my_module PRIVATE clair::c2py_plugin c2py::c2py)
```

How to try Clair/c2py ?

- Experimental module on rusty



```
export MODULEPATH=/mnt/home/ccq/opt/modules:$MODULEPATH

module load gcc llvm python clair/0.1.1
```

- Experimental homebrew formula for OS X

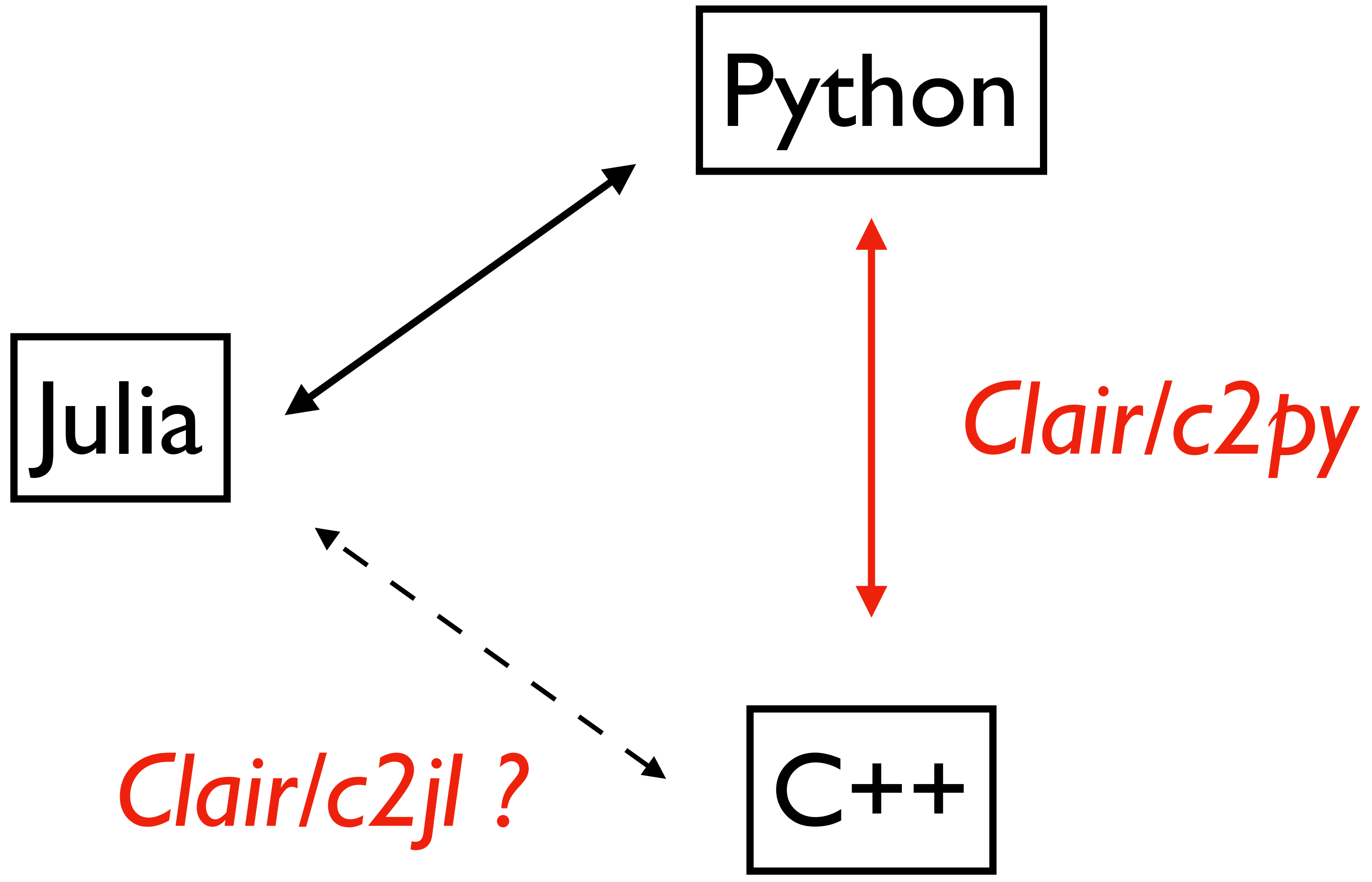
```
brew install parcollet/ccq/c2py parcollet/ccq/clair
```

- Documentation

https://flatironinstitute.github.io/clair/

Summary

- Automated tool for C++/Python binding.
- Clang Tools/plugins. Many other potential applications.
- Beta version. Feedback welcome. Documentation in progress, some features to be merged.
- Next step : C++/Julia ? Like Clang.jl but for C++.



Thank you for your attention!