pybind11

Introduction, practical guide, and limitations

Andrew James

May 23, 2022

Slides on github: amjames/pybind11-tech-share

Overview

Overview

Motivations and goals of the project

Overview

Motivations and goals of the project

Binding Functions

Overview

Motivations and goals of the project

Binding Functions

Binding C++ classes

Overview

Motivations and goals of the project

Binding Functions

Binding C++ classes

Overloads

Overview

Motivations and goals of the project

Binding Functions

Binding C++ classes

Overloads

NumPy/ Buffer Protocol

Overview

Motivations and goals of the project

Binding Functions

Binding C++ classes

Overloads

NumPy/ Buffer Protocol

Embedding the Python Interpreter

Overview

Motivations and goals of the project

Binding Functions

Binding C++ classes

Overloads

NumPy/ Buffer Protocol

Embedding the Python Interpreter

What is going on in the background

What is pybind11

> *pybind11 is a lightweight header-only library that exposes C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code.*

- Generate C++ wrappers for native python types.
- Python `PyObject` wrappers for C++ types.

Motivations - Alternatives

Motivations - Alternatives

ctypes

The most low-level option. Your library must have an `extern "C"` interface. It is not easy to export types from your library, and you must use C data types at the call site. It does not scale well.

Motivations - Alternatives

ctypes

The most low-level option. Your library must have an `extern "C"` interface. It is not easy to export types from your library, and you must use C data types at the call site. It does not scale well.

Cython

Generating bindings for a C++ library will require writing and maintaining a layer in the Cython language which is neither python, nor C. Shines when you want to accelerate parts of the codebase.

Motivations - Alternatives

ctypes

The most low-level option. Your library must have an `extern "C"` interface. It is not easy to export types from your library, and you must use C data types at the call site. It does not scale well.

Cython

Generating bindings for a C++ library will require writing and maintaining a layer in the Cython language which is neither python, nor C. Shines when you want to accelerate parts of the codebase.

Boost.Python

Very similar to pybind11 in features and syntax. You write binding code in C++. Mostly, you are selectively adding features to the python API. The main drawback is boost itself.

Motivations

> “*Think of this library as a tiny self-contained version of Boost.Python with everything stripped away that isn’t relevant for binding generation. This compact implementation was possible thanks to some of the new C++11 features...*

Leveraging C++11 features to write a compact library for python bindings, hence the name pybind11.

Writing a simple module

cppimport is a neat little tool for playing around with simple pybind11 extensions. I have used for self contained examples of some concepts.

My environment:

```
conda create -n pybind-examples pybind11 ipython pip
conda activate pybind-examples
pip install cppimport
```

Writing a simple module

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

double add(double a, double b) {
  return a+b;
}

PYBIND11_MODULE(example1, m) {
  m.def("add", &add);
}

//
//<%
//setup_pybind11(cfg)
//%>
//
```

A few notes:

- 1st line Indicates that the file should be importable.
- Entire example fits on a slide!
- I will be showing the important stuff moving forward, but all of the examples will be on github.

Writing a simple module

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

double add(double a, double b) {
  return a+b;
}

PYBIND11_MODULE(example1, m) {
  m.def("add", &add);
}
```

We can import this and run it directly:

```python
>>> import cppimport.import_hook
>>> import example1
>>> example1.add(2.5, 3.2)
5.7
```

It will take a moment to (re)-compile after editing

Docstrings

When generating python bindings it is important to consider that the user of that API will be expecting certain things from a python library. Docstrings area great example, there is no need to support `help()` in C++.

Docstrings

```
>>> help(example1.add)
...
add(...) method of builtins.PyCapsule instance
    add(arg0: float, arg1: float) -> float
```

Docstrings

```
>>> help(example1.add)
...
add(...) method of builtins.PyCapsule instance
    add(arg0: float, arg1: float) -> float
```

This is pretty good!

Docstrings

```
>>> help(example1.add)
...
add(...) method of builtins.PyCapsule instance
    add(arg0: float, arg1: float) -> float
```

This is pretty good!

We are missing argument names.

Docstrings - named arguments

```
namespace py = pybind11;
PYBIND11_MODULE(example1, m) {
  m.def("subtract", &subtract, "Computes a - b",
    py::arg("a"), py::arg("b"));
}
```

This alias is widely adopted convention

Similar to `import numpy as np`

Docstrings - named arguments

```
namespace py = pybind11;
PYBIND11_MODULE(example1, m) {
  m.def("subtract", &subtract, "Computes a - b",
      py::arg("a"), py::arg("b"));
}
```

We have now attached names to our arguments by annotating the function binding with some additional information.

Docstrings - More Metadata

```
namespace py = pybind11;
PYBIND11_MODULE(example1, m) {
  m.def("subtract", &subtract, "Computes a - b",
      py::arg("a") = 1, py::arg("b") = 2);
}
```

We can also specify the default values for the arguments.

The number of arguments is statically checked, however the types are not!

Docstrings - More Metadata

```
using namespace pybind11::literals;
PYBIND11_MODULE(example1, m) {
  m.def("subtract", &subtract, "Computes a - b",
      "a"_a = 1, "b"_a = 2);
}
```

We may also use the `_a` suffix (C++11 literals)

Docstrings

Now the docstring will look more complete

```
>>> help(example2.subtract)
...
subtract(...) method of builtins.PyCapsule instance
    subtract(a: float = 1, b: float = 2) -> float

    Computes a - b
```

Binding a C++ class

Using `py::class_`:

- Creates a binding for a C++ class or struct.
- The `py:class_` has methods available for binding additional information to the class
- Very similar to `py::module_` (Without additional boilerplate, so no macro is required)

Binding a C++ class

Using `py::class_`:

- Creates a binding for a C++ class or struct.
- The `py:class_` has methods available for binding additional information to the class
- Very similar to `py::module_` (Without additional boilerplate, so no macro is required)

```cpp
struct Pet {
    std::string name;
};
```

```cpp
namespace py = pybind11;
PYBIND11_MODULE(example3, m) {
  py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string&>());
}
```

Binding a C++ class

Using `py::class_`:

- Creates a binding for a C++ class or struct.
- The `py:class_` has methods available for binding additional information to the class
- Very similar to `py::module_` (Without additional boilerplate, so no macro is required)

```
struct Pet {
    std::string name;
};
```

```
namespace py = pybind11;
PYBIND11_MODULE(example3, m) {
  py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string&>());
}
```

- The `py::init` wrapper is used to bind constructors for a class
- The template parameters should correspond to a constructor signature
- Without template parameters a callable can be provided which returns the type by value or the appropriate "holder"

Binding a C++ class

We use the `py::class_` object to define the python visible interface for the class.

Anything we want to be available in python, must be explicitly declared in the binding

Binding a C++ class

We use the `py::class_` object to define the python visible interface for the class.

Anything we want to be available in python, must be explicitly declared in the binding

```
import example3
p = example3.Pet("Spot")
print(p.name)  # AttributeError!
```

Binding a C++ class

We use the `py::class_` object to define the python visible interface for the class.

Anything we want to be available in python, must be explicitly declared in the binding

```
import example3
p = example3.Pet("Spot")
print(p.name)  # AttributeError!
```

We haven not added an attribute `name` to the python interface for the `Pet` class.

```
namespace py = pybind11;
PYBIND11_MODULE(example3, m) {
  py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string&>());
}
```

Binding a C++ class - Attributes

```
namespace py = pybind11;
PYBIND11_MODULE(example3, m) {
  py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string&>());
}
```

Binding a C++ class - Attributes

```
namespace py = pybind11;
PYBIND11_MODULE(example3, m) {
  py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string&>());
}
```

We have a few different options:

Attributes:

```
    .def_readonly("name", &Pet::name)
    .def_readwrite("name", &Pet::name)
```

Binding a C++ class - Attributes

```cpp
namespace py = pybind11;
PYBIND11_MODULE(example3, m) {
  py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string&>());
}
```

We have a few different options:

Attributes:

```cpp
    .def_readonly("name", &Pet::name)
    .def_readwrite("name", &Pet::name)
```

Or property:

```cpp
    .def_property("name", /*getter*/, /*setter*/)
    .def_property_readonly("name", /*getter*/)
    .def_property("name", nullptr, /*setter*/) //Write-only property
```

Binding a C++ class - Attributes

If we didn't have a getter/setter already the attribute route is a good one.

We can also use a lambda.

```
    .def_property("name",
        /*getter*/[](const Pet& self) { return self.name; },
        /*setter*/[](Pet& self, std::string value) { self.name = value; })
```

Lambdas are particularly useful when you want to do something more pythonic, but don't want to introduce py::types into your library code

```
enum FeatureTypes{...};
std::tuple<bool, bool, bool,...> check_features(...);
```

...

Lambdas are particularly useful when you want to do something more pythonic, but don't want to introduce py::types into your library code

```
enum FeatureTypes{...};
std::tuple<bool, bool, bool,...> check_features(...);
```

...

```
//In binding code
using namespace pybind11::literals;
m.def("check_features", [](...) {
    auto feature_tuple = check_features(...);
    return py::dict("feature_a"_a = std::get<FeatureTypes::A>(feature_tuple),...);
    });
```

Lambdas are particularly useful when you want to do something more pythonic, but don't want to introduce py::types into your library code

```cpp
enum FeatureTypes{...};
std::tuple<bool, bool, bool,...> check_features(...);
```

...

```cpp
//In binding code
using namespace pybind11::literals;
m.def("check_features", [](...) {
    auto feature_tuple = check_features(...);
    return py::dict("feature_a"_a = std::get<FeatureTypes::A>(feature_tuple),...);
    });
```

Using lambdas at the binding layer to translate c++ patterns to pythonic variants is a popular practice.

Binding Overloads

If a function has multiple overloads we will have some trouble with the basic pattern for generating a binding

```
struct Reader {
  // reads everything
  size_t read()
  // reads from begin to the end
  size_t read(size_t begin)
  // reads from offset begin to offset end
  size_t read(size_t begin, size_t end)
};
```

Binding Overloads

If a function has multiple overloads we will have some trouble with the basic pattern for generating a binding

```
struct Reader {
  // reads everything
  size_t read()
  // reads from begin to the end
  size_t read(size_t begin)
  // reads from offset begin to offset end
  size_t read(size_t begin, size_t end)
};
```

```
py::class_<Reader>(m, "Reader")
    .def("read", &Reader::read)
```

Binding Overloads

If a function has multiple overloads we will have some trouble with the basic pattern for generating a binding

```
struct Reader {
  // reads everything
  size_t read()
  // reads from begin to the end
  size_t read(size_t begin)
  // reads from offset begin to offset end
  size_t read(size_t begin, size_t end)
};
```

```
py::class_<Reader>(m, "Reader")
    .def("read", &Reader::read)
```

The compiler is not able to read your mind!

Binding Overloads

We can disambiguate by casting to a function pointer

```
py::class_<Reader>(m, "Reader")
    .def("read", static_cast<size_t (Reader::*)(size_t)>(&Rader::read))
```

Binding Overloads

We can disambiguate by casting to a function pointer

```
py::class_<Reader>(m, "Reader")
    .def("read", static_cast<size_t (Reader::*)(size_t)>(&Rader::read))
```

Or there is a handy helper (C++14)

```
py::class_<Reader>(m, "Reader")
    .def("read", py::overload_cast<size_t>(&Reader::read))
```

Note: The `py::init` wrapper we use to bind constructors takes care of this for us

Binding Overloads

It would not be very useful if we could only bind a single overload of a function.

Binding Overloads

It would not be very useful if we could only bind a single overload of a function.

We could manually differentiate by binding to different names:

```
py::class_<Reader>(m, "Reader")
    .def("read_all", py::overload_cast<>(&Reader::read))
    .def("read_from", py::overload_cast<size_t>(&Reader::read))
    .def("read_between", py::overload_cast<size_t,size_t>(&Reader::read))
```

Binding Overloads

It would not be very useful if we could only bind a single overload of a function.

We could manually differentiate by binding to different names:

```
py::class_<Reader>(m, "Reader")
    .def("read_all", py::overload_cast<>(&Reader::read))
    .def("read_from", py::overload_cast<size_t>(&Reader::read))
    .def("read_between", py::overload_cast<size_t,size_t>(&Reader::read))
```

It is not desirable to have our python and C++ api diverge like this.

Binding Overloads

It would not be very useful if we could only bind a single overload of a function.

We could manually differentiate by binding to different names:

```
py::class_<Reader>(m, "Reader")
    .def("read_all", py::overload_cast<>(&Reader::read))
    .def("read_from", py::overload_cast<size_t>(&Reader::read))
    .def("read_between", py::overload_cast<size_t,size_t>(&Reader::read))
```

It is not desirable to have our python and C++ api diverge like this.

Thankfully pybind allows us to bind multiple methods to the same name!

```
py::class_<Reader>(m, "reader")
    .def("read", py::overload_cast<>(&Reader::read))
    .def("read", py::overload_cast<size_t>(&Reader::read))
    .def("read", py::overload_cast<size_t,size_t>(&Reader::read))
```

Binding Overloads

```
py::class_<Reader>(m, "Reader")
    .def("read", py::overload_cast<>(&Reader::read))
    .def("read", py::overload_cast<size_t>(&Reader::read))
    .def("read", py::overload_cast<size_t,size_t>(&Reader::read))
```

How does that work?

Binding Overloads

```
py::class_<Reader>(m, "Reader")
    .def("read", py::overload_cast<>(&Reader::read))
    .def("read", py::overload_cast<size_t>(&Reader::read))
    .def("read", py::overload_cast<size_t,size_t>(&Reader::read))
```

How does that work?

When we make the first call to `py::class_::def` pybind is going to create an entry in the class `__dict__` for the method read. It points to the wrapper built around the bound method.

Binding Overloads

```
py::class_<Reader>(m, "Reader")
    .def("read", py::overload_cast<>(&Reader::read))
    .def("read", py::overload_cast<size_t>(&Reader::read))
    .def("read", py::overload_cast<size_t,size_t>(&Reader::read))
```

How does that work?

When we make the first call to `py::class_::def` pybind is going to create an entry in the class `__dict__` for the method read. It points to the wrapper built around the bound method.

When we `.def` the second and third overloads, the existence of the first is detected. This triggers the new method to be added to the end of the linked list of methods in the slot.

Binding Overloads

```
py::class_<Reader>(m, "Reader")
    .def("read", py::overload_cast<>(&Reader::read))
    .def("read", py::overload_cast<size_t>(&Reader::read))
    .def("read", py::overload_cast<size_t,size_t>(&Reader::read))
```

How does that work?

When we make the first call to `py::class_::def` pybind is going to create an entry in the class `__dict__` for the method read. It points to the wrapper built around the bound method.

When we `.def` the second and third overloads, the existence of the first is detected. This triggers the new method to be added to the end of the linked list of methods in the slot.

There is some inspection of the metadata for the function to make sure the overload is compatible.

Binding Overloads

```
py::class_<Reader>(m, "Reader")
    .def("read", py::overload_cast<>(&Reader::read))
    .def("read", py::overload_cast<size_t>(&Reader::read))
    .def("read", py::overload_cast<size_t,size_t>(&Reader::read))
```

How does that work?

When we make the first call to `py::class_::def` pybind is going to create an entry in the class `__dict__` for the method read. It points to the wrapper built around the bound method.

When we `.def` the second and third overloads, the existence of the first is detected. This triggers the new method to be added to the end of the linked list of methods in the slot.

There is some inspection of the metadata for the function to make sure the overload is compatible.

In general every callable (instance bound, class bound, or module bound) is set up this way. Even if no overloads are defined.

Binding Overloads

If you want to influence overload resolution:

- `py::arg().noconvert()` to prevent casting of an argument even in the relaxed pass
- The order of `.def` statements will change the order of the list and therefore the search.
- You can add `py::prepend` to the tags section of the `def` to place it at the beginning of the chain.

The first matching overload is always selected to be called.

There is no priority for minimizing the number of casts required for example.

Casts/Conversions from Pybind11's Point of view

Consider:

```
py::float_ foo(py::list arr);                /* with       */   m.def("foo", &foo);
double bar(std::vector<double> arr);         /* bindings   */   m.def("bar", &bar);
void baz(CustomT& ct);                       /* somewhere */   m.def("baz", &baz);
```

Casts/Conversions from Pybind11's Point of view

Consider:

```
py::float_ foo(py::list arr);             /* with       */   m.def("foo", &foo);
double bar(std::vector<double> arr);      /* bindings   */   m.def("bar", &bar);
void baz(CustomT& ct);                    /* somewhere */   m.def("baz", &baz);
```

When calling `foo([1,2,3])` in python, the layer wrapping the function `foo` applies a thin `py::list` wrapper around some `PyListObject`.

Casts/Conversions from Pybind11's Point of view

Consider:

```
py::float_ foo(py::list arr);              /* with       */   m.def("foo", &foo);
double bar(std::vector<double> arr);       /* bindings   */   m.def("bar", &bar);
void baz(CustomT& ct);                     /* somewhere */   m.def("baz", &baz);
```

When calling `foo([1,2,3])` in python, the layer wrapping the function `foo` applies a thin `py::list` wrapper around some `PyListObject`.

The return involves creating a `py::float_` in c++, but it is also a thin wrapper around a native python type and removed on the way out.

Casts/Conversions from Pybind11's Point of view

Consider:

```
py::float_ foo(py::list arr);                 /* with       */   m.def("foo", &foo);
double bar(std::vector<double> arr);          /* bindings   */   m.def("bar", &bar);
void baz(CustomT& ct);                        /* somewhere */   m.def("baz", &baz);
```

When calling `foo([1,2,3])` in python, the layer wrapping the function `foo` applies a thin `py::list` wrapper around some `PyListObject`.

The return involves creating a `py::float_` in c++, but it is also a thin wrapper around a native python type and removed on the way out.

When the wrapping layer for `baz` sees the Python interface for a `py::class_<CustomT>` it does much the same, pulling a wrapper off to expose the native C++ object.

Casts/Conversions from Pybind11's Point of view

Consider:

```
py::float_ foo(py::list arr);               /* with       */   m.def("foo", &foo);
double bar(std::vector<double> arr);        /* bindings   */   m.def("bar", &bar);
void baz(CustomT& ct);                      /* somewhere */   m.def("baz", &baz);
```

When calling `foo([1,2,3])` in python, the layer wrapping the function `foo` applies a thin `py::list` wrapper around some `PyListObject`.

The return involves creating a `py::float_` in c++, but it is also a thin wrapper around a native python type and removed on the way out.

When the wrapping layer for `baz` sees the Python interface for a `py::class_<CustomT>` it does much the same, pulling a wrapper off to expose the native C++ object.

None of these operations are considered casts

Casts/Conversions from Pybind11's Point of view

Consider:

```
py::float_ foo(py::list arr);              /* with       */   m.def("foo", &foo);
double bar(std::vector<double> arr);       /* bindings   */   m.def("bar", &bar);
void baz(CustomT& ct);                     /* somewhere */   m.def("baz", &baz);
```

What happens with `bar`?

Casts/Conversions from Pybind11's Point of view

Consider:

```
py::float_ foo(py::list arr);              /* with       */   m.def("foo", &foo);
double bar(std::vector<double> arr);       /* bindings   */   m.def("bar", &bar);
void baz(CustomT& ct);                     /* somewhere */   m.def("baz", &baz);
```

What happens with `bar`?

The wrapping layer finds no appropriate overload in the strict search.

Casts/Conversions from Pybind11's Point of view

Consider:

```
py::float_ foo(py::list arr);                /* with       */   m.def("foo", &foo);
double bar(std::vector<double> arr);         /* bindings   */   m.def("bar", &bar);
void baz(CustomT& ct);                       /* somewhere */   m.def("baz", &baz);
```

What happens with `bar`?

The wrapping layer finds no appropriate overload in the strict search.

If the argument were a `list` or `tuple` and the contents were all numeric objects. It would be able to cast and make the call.

Casts/Conversions from Pybind11's Point of view

Consider:

```
py::float_ foo(py::list arr);               /* with       */   m.def("foo", &foo);
double bar(std::vector<double> arr);        /* bindings   */   m.def("bar", &bar);
void baz(CustomT& ct);                      /* somewhere */    m.def("baz", &baz);
```

What happens with `bar`?

The wrapping layer finds no appropriate overload in the strict search.

If the argument were a `list` or `tuple` and the contents were all numeric objects. It would be able to cast and make the call.

The key difference is there is no *simple* conversion involving the addition/removal of a pybind wrapper.

Casts/Conversions from Pybind11's Point of view

Consider:

```
py::float_ foo(py::list arr);               /* with       */   m.def("foo", &foo);
double bar(std::vector<double> arr);        /* bindings   */   m.def("bar", &bar);
void baz(CustomT& ct);                      /* somewhere */    m.def("baz", &baz);
```

What happens with `bar`?

The wrapping layer finds no appropriate overload in the strict search.

If the argument were a `list` or `tuple` and the contents were all numeric objects. It would be able to cast and make the call.

The key difference is there is no *simple* conversion involving the addition/removal of a pybind wrapper.

This is important to consider when defining overloads. If `bar` had an overload accepting `py::list` the `std::vector<double>` version would *never* match if passed a list on the python side.

Holder types

All pybind wrapped types `py::class_<T>` uses a special holder type to manage references to the object. The default is to use `std::unique_ptr<T>`.

This will mean that the pybind11 wrapper is going to reference count for us and delete the object when it no longer referenced anywhere.

Sometimes, a codebase may rely on `std::shared_ptr<T>` heavily, and it is possible to specify this as holder type for the object.

```
class AlwaysShared {};
py::class_<AlwaysShared, std::shared_ptr<AlwaysShared>>(m, "AlwaysShared");
```

Now pybind11 will allow the smart pointer to do the reference counting.

Holder types

All pybind wrapped types `py::class_<T>` uses a special holder type to manage references to the object. The default is to use `std::unique_ptr<T>`.

This will mean that the pybind11 wrapper is going to reference count for us and delete the object when it no longer referenced anywhere.

Sometimes, a codebase may rely on `std::shared_ptr<T>` heavily, and it is possible to specify this as holder type for the object.

```
class AlwaysShared {};
py::class_<AlwaysShared, std::shared_ptr<AlwaysShared>>(m, "AlwaysShared");
```

Now pybind11 will allow the smart pointer to do the reference counting.

Careful! Any bound function returning a raw pointer will improperly be captured in a *new* `shared_ptr`.

NumPy / Buffer Protocol

Built in support for the Python Buffer protocol

```cpp
class Matrix {
public:
    Matrix(size_t rows, size_t cols)
        : m_rows(rows)
        , m_cols(cols) {
        m_data = new float[rows*cols];
    }
    float *data() { return m_data; }
    size_t rows() const { return m_rows; }
    size_t cols() const { return m_cols; }
private:
    size_t m_rows, m_cols;
    float *m_data;
};
```

```cpp
py::class_<Matrix>(m, "Matrix",
  py::buffer_protocol()
).def_buffer([](Matrix &m){
  return py::buffer_info(
    /* Pointer to buffer */
    m.data(),
    /* Size of one scalar */
    sizeof(float),
    /* Python struct format descriptor */
    py::format_descriptor<float>::format(),
    /* Number of dimensions */
    2,
    /* Buffer dimensions */
    { m.rows(), m.cols() },
    /* Strides (in bytes) */
    { sizeof(float) * m.cols(),
      sizeof(float) }
  );
});
```

NumPy / Buffer Protocol

This makes it possible construct a NumPy array from our `Matrix` object without an expensive copy of the underlying data.

We could design a constructor for our `Maxtrix` object which makes it a view on some NumPy owned memory.

NumPy / Buffer Protocol

This makes it possible construct a NumPy array from our `Matrix` object without an expensive copy of the underlying data.

We could design a constructor for our `Maxtrix` object which makes it a view on some NumPy owned memory.

There is also direct support for NumPy arrays `py::array`.

NumPy / Buffer Protocol

This makes it possible construct a NumPy array from our `Matrix` object without an expensive copy of the underlying data.

We could design a constructor for our `Maxtrix` object which makes it a view on some NumPy owned memory.

There is also direct support for NumPy arrays `py::array`.

A `py::vectorize` wrapper for transforming scalar ops into vectorized point wise ops over NumPy arrays

```
double add_3(double a, double b, double c) { return a + b + c }
m.def("vectorized_add_3", py::vectorize(add_3))"
```

Embedding the interpreter

```python
# plots.py
def plot_x_squared(x):
    import matplotlib.pyplot as plt
    plt.plot(x, x**2)
    plt.show()
```

```cpp
#include <pybind11/embed.h>
namespace py = pybind11;
int main()
{
    using namespace py::literals;
    py::scoped_interpreter guard{};
    py::module_ np = py::module_::import("numpy")
    py::object x = np.attr("arange")(0, 100);
    py::module_ plots= py::module_::import("plots");
    plots.attr("plot_x_squared")(x);
    return 0;
}
```

Very easy to embed a scripting console in your application!

Thank you!

Andrew James

May 23, 2022

Slides on github: amjames/pybind11-tech-share