

Chapter 1

Section 1

Section 2

Chapter 2

Section 1

Section 2

Chapter 3

Section 1

Section 2

Chapter 4

Section 1

Section 2

Chapter 5

Section 1

Section 2

Chapter 6

Section 1

Section 2

Here is an inline example, $\pi(\theta)$,
an equation,

$$\nabla f(x) \in \mathbb{R}^n,$$

and a regular $\$$ symbol.

Define $f(x)$:

$$f(x) = x^2 \quad x \in \mathbb{R}$$

```
graph TD
  A[ Anyone ] -->|Can help | B( Go to github.com/yuzutech/kroki )
  B --> C[ How to contribute? ]
  C --> D[ Reporting bugs ]
  C --> E[ Sharing ideas ]
  C --> F[ Advocating ]
```

pybind11——C++Python

```
pybind11 C++ Python Python C++ C++
Python David Abrahams Boost.Python

```

```
Boost.Python Boost Boost Boost C++
BUG Boost C++11

```

```

pybind11Boost.Pythonpythonpybind11
4KPython2.73.5+PyPyC ++C++11(lambd
Boost.Pythonpybind11python

```

1.1 背景

pybind11 C++ Python

- 00000000000000000000000000000000
- 00000000000000
- 000000
- 00000000000000
- 000000000
- 0000
- 000000
- 000000ranges
- 00000000
- 00000000000
- STL000000
- 000000
- Internal references with correct reference counting
- 0000Python000000000000000000C++00

1.2 背景

```
pybind11
```

- Python 2.7, 3.5+, PyPy/PyPy3 7.3
- `lambda` `lambda` Python
- `pybind11` C++11 (pybind11 uses C++11 move constructors and move assignment operators whenever possible to efficiently transfer custom data types.)

- Python buffer objects and C++ Eigen NumPy
- pybind11 NumPy
- Python
- Boost.Python
- `constexpr`
- C++ Python pickle/unpickle

1.3

1. Clang/LLVM 3.3 (Apple Xcode's clang 5.0.0)
2. GCC 4.8
3. Microsoft Visual Studio 2015 Update 3
4. Intel classic C++ compiler 18 or newer (ICC 20.2 tested in CI)
5. Cygwin/GCC (previously tested on 2.5.1)
6. NVCC (CUDA 11.0 tested in CI)
7. NVIDIA PGI (20.9 tested in CI)

1.4

This project was created by Wenzel Jakob. Significant features and/or improvements to the code were contributed by Jonas Adler, Lori A. Burns, Sylvain Corlay, Eric Cousineau, Aaron Gokaslan, Ralf Grosse-Kunstleve, Trent Houlston, Axel Huebl, @hulucc, Yannick Jadoul, Sergey Lyskov, Johan Mabille, Tomasz Miąsko, Dean Moldovan, Ben Pritchard, Jason Rhinelander, Boris Schäling, Pim Schellart, Henry Schreiner, Ivan Smirnov, Boris Staletic, and Patrick Stewart.

We thank Google for a generous financial contribution to the continuous integration infrastructure used by this project.

1.5

See the [contributing guide](#) for information on building and contributing to pybind11.

1.6 License

pybind11 is provided under a BSD-style license that can be found in the [LICENSE](#) file. By using, distributing, or contributing to this project, you agree to the terms and conditions of this license.

1111

[illegible]



3. 安装

克隆 `pybind/pybind11` on GitHub 到 `pybind11` 目录中 `pybind11` 目录中

3.1 使用 Git

克隆 `Git` 到 `pybind11` 目录中 `git` 到 `pybind11` 目录中

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

克隆 `extern` 到 `Git` 到 `Git` 到 `https` `ssh` URL 到 `URL` `../../pybind/pybind11` 到 `.git` 到 `Git` 到

克隆 `include` `extern/pybind11/include` 到 `Build System` 到 `pybind11`

3.2 使用 PyPI

克隆 `pip` 到 `PyPI` 到 `Pybind11` `Python` 到 `CMake` 到

```
pip install pybind11
```

克隆 `pybind11` 到 `Python` 到 `root` 到 `pybind11` 到

```
pip install "pybind11[global]"
```

克隆 `Python` 到 `root` 到 `/usr/local/include/pybind11` 到 `/usr/local/share/cmake/pybind11` 到 `pyproject.toml` 到

3.3 使用 conda-forge

You can use pybind11 with conda packaging via [conda-forge](#):

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

3.4 使用 vcpkg

克隆 `Microsoft vcpkg` 到 `pybind11`

```
git clone https://github.com/Microsoft/vcpkg.git
cd vcpkg
./bootstrap-vcpkg.sh
./vcpkg integrate install
vcpkg install pybind11
```

3.5 brew

brew on macOS, or Linuxbrew on Linux

```
brew install pybind11
```

3.6

Other locations you can find pybind11 are [listed here](#); these are maintained by various packagers and the community.

4. 4.1 First steps

pybind11

4.1

Linux/macOS

Linux python-dev python3-dev cmake macOS python cmake

```
mkdir build
cd build
cmake ..
make check -j 4
```

Windows

Windows C++11 Visual Studio 15

Note Visual Studio 2017(MSVC 14.1) C++17 pybind11 `/permissive-` Visual Studio 2019

```
mkdir build
cd build
cmake ..
cmake --build . --config Release --target check
```

Visual Studio

Note Python i386 x86_64 x86_64
vs `cmake -A x64 ..`

4.2 编译选项

编译选项如下：

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

编译选项如下：

4.3 编译选项

编译选项如下：pybind11

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

编译选项如下：example.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 which adds two numbers");
}
```

PYBIND11_MODULE 编译选项 Python import 编译选项 example 编译选项
编译选项 py::module_ 编译选项 module_::def() 编译选项 add 编译选项 Python

Note 编译选项 Python 编译选项 Boost.Python

pybind11 head-only 编译选项 Linux

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

Note 编译选项 pybind11 编译选项 \$(python3-config --includes) -
Iextern/pybind11/include 编译选项 \$(python3 -m pybind11 --includes) 编译选项

编译选项 Linux MacOS

Pythonのimport文は、C++の#include文と異なり、import文はPythonのモジュールをインポートする際に使用される。Pythonのimport文は、C++の#include文と異なり、import文はPythonのモジュールをインポートする際に使用される。

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

4.4 Pythonのモジュール

Pythonのモジュールは、C++のモジュールと異なり、Pythonのモジュールは、Pythonのモジュールをインポートする際に使用される。

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

arg関数は、module::def()関数の引数として使用される。arg関数は、Pythonのモジュールをインポートする際に使用される。

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

Pythonのモジュールは、C++のモジュールと異なり、Pythonのモジュールは、Pythonのモジュールをインポートする際に使用される。

```
>>> help(example)

....

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

    A function which adds two numbers
```

Pythonのモジュールは、C++のモジュールと異なり、Pythonのモジュールは、Pythonのモジュールをインポートする際に使用される。

```
// 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);
```

Pythonのモジュールは、C++のモジュールと異なり、Pythonのモジュールは、Pythonのモジュールをインポートする際に使用される。

4.5 Pythonのモジュール

Pythonのモジュールは、C++のモジュールと異なり、Pythonのモジュールは、Pythonのモジュールをインポートする際に使用される。

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

pybind11 `arg`

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

```
>>> help(example)

....

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

    A function which adds two numbers
```

```
// regular notation
m.def("add1", &add, py::arg("i") = 1, py::arg("j") = 2);
// shorthand
m.def("add2", &add, "i"_a=1, "j"_a=2);
```

4.6

`attr` Python C++ `attriutes`
`py::cast`

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

```
Python
```python
>>> import example
>>> example.the_answer
42
>>> example.what
'World'
```

## 4.7 `py::cast`

`py::cast` (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 Type conversions section.)

## 5. 問題

## 5.1 背景背景背景背景

C++ Pet

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

 std::string name;
};
```

10/10

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

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

```
class_ C++ class struct init() Python
```

```
>>> import example
>>> p = example.Pet("Molly")
>>> print(p)
<example.Pet object at 0x10cd98060>
>>> p.getName()
u'Molly'
>>> p.setName("Charly")
>>> p.getName()
u'Charly'
```

**See also** `class_::def_static`

## 5.2 関数定義

0000040000000000000000000000000004000000

## 5.3 repr

例 `print(p)` 输出对象的地址

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

我们可以在 `__repr__` 方法中返回一个更友好的字符串

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

现在 Python 的输出是

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

pybind11 的 `lambda` 函数可以返回任意类型的值

## 5.4 属性

例 `class_::def_readwrite` 方法可以定义 `class_::def_readonly` 方法

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

Python 的输出是

```
>>> p = example.Pet("Molly")
>>> p.name
u'Molly'
>>> p.name = "Charly"
>>> p.name
u'Charly'
```

例 `Pet::name` 方法可以定义 setter 和 getters

```
class Pet {
public:
 Pet(const std::string &name) : name(name) { }
 void setName(const std::string &name_) { name = name_; }
 const std::string &getName() const { return name; }
private:
 std::string name;
};
```

class\_::def\_property() (class\_::def\_property\_readonly() ) setter  
getter

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

read nullptr

**see also:** class\_::def\_readwrite\_static(), class\_::def\_readonly\_static()  
class\_::def\_property\_static(), class\_::def\_property\_readonly\_static()

## 5.5 Python

Python

```
>>> class Pet:
... name = "Molly"
...
>>> p = Pet()
>>> p.name = "Charly" # overwrite existing
>>> p.age = 2 # dynamically add a new attribute
```

C++ class\_::def\_readwrite class\_::def\_property

```
>>> p = example.Pet()
>>> p.name = "Charly" # OK, attribute defined in C++
>>> p.age = 2 # fail
AttributeError: 'Pet' object has no attribute 'age'
```

C++ py::class\_ py::dynamic\_attr

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



```
>>> p = example.Pet()
>>> p.name = "Charly" # OK, overwrite value in C++
>>> p.age = 2 # OK, dynamically add a new attribute
>>> p.__dict__ # just like a native Python class
{'age': 2}
```

Pythonの\_\_dict\_\_は、C++のstd::mapと似ていますが、Pythonの\_\_dict\_\_は、C++のstd::mapよりも柔軟で、動的に追加された属性も保持します。

## 5.6 PythonとC++

PythonとC++の連携

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

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

pybind11は、C++のクラスをPythonのclass\_としてPythonにエクスポートするためのライブラリです。

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

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

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

PythonからC++のクラスを呼び出す

```
>>> p = example.Dog("Molly")
>>> p.name
u'Molly'
>>> p.bark()
u'woof!'
```

PythonからC++のクラスを呼び出す

```
// 测试非多态的Pet
m.def("pet_store", []() { return std::unique_ptr<Pet>(new Dog("Molly")); });
```

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

pet\_store 返回 Dog 实例，Python 的 Pet 基类 C++ 的 Dog 类，pybind11 的 Pet 基类

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

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

// Same binding code
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); });
```

```
>>> p = example.pet_store2()
>>> type(p)
PolymorphicDog # automatically downcast
>>> p.bark()
u'woof!'
```

pybind11 的 PolymorphicPet 基类 C++ 的 PolymorphicPet 基类，pybind11 的 PolymorphicDog 基类

## 5.7 多态

多态的 Pet 类

```
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;
};
```

```
Pet::set Python
Python
```

```
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");
```

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

```
>>> help(example.Pet)

class Pet(__builtin__.object)
 Methods defined here:

 __init__(...)
 Signature : (Pet, str, int) -> NoneType

 set(...)
 1. Signature : (Pet, int) -> NoneType

 Set the pet's age

 2. Signature : (Pet, str) -> NoneType

 Set the pet's name
```

# C++14

```
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");
```

[illegible]

```
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_));
```

```

c++11 py::overload_cast py::detail::overload_cast_impl

```

```
template <typename... Args>
using overload_cast_ = pybind11::detail::overload_cast_impl<Args...>;

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

Note: `py::init<...>()` is deprecated

## 5.8 Enums

Enums in C++

```
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;
};
```

Enums in Python

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

Enums in C++ are represented in Python as `enum_` objects. The `enum_::export_values()` method exports the enum values to Python.

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

enum 的 \_\_members\_\_ 属性

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

name 属性返回枚举成员名称的字符串，str(enum) 返回枚举成员名称的字典

```
>>> p = Pet("Lucy", Pet.Cat)
>>> pet_type = p.type
>>> pet_type
Pet.Cat
>>> str(pet_type)
'Pet.Cat'
>>> pet_type.name
'Cat'
```

Note: enum\_ 模块的 py::arithmetic() 函数在 pybind11 中定义，用于将枚举成员名称转换为整数

```
py::enum_<Pet::Kind>(pet, "Kind", py::arithmetic())
...
```

enum 的 \_\_members\_\_ 属性

## 6. 練習問題

練習問題

## 7.

45Python

### 7.1

PythonC++no-trivialPythonC++pybind11

```
model::def() { class_def() return_value_policy::automatic }
```

```
/* Function declaration */
Data *get_data() { return _data; /* (pointer to a static data structure) */
...

/* Binding code */
m.def("get_data", &get_data); // <-- KABOOM, will cause crash when called from Python
```

Pythonget\_data()C++Python

return\_value\_policy::automaticpybind11\_data

Python\_dataPythonpybind11C++operator delete()

return\_value\_policy::reference

```
m.def("get_data", &get_data, py::return_value_policy::reference);
```

pybind11

return_value_policy	
return_value_policy::take_ownership	Pythondelete
return_value_policy::copy	Python
return_value_policy::move	std::movePython
return_value_policy::reference	C++PythonC++

return_value_policy	説明
<code>return_value_policy::reference_internal</code>	<p>この関数は、この関数の内部で定義されたオブジェクトの参照を返す。この関数は、<code>this</code> が <code>self</code> の参照を返す。この関数は、<code>keep_alive&lt;0, 1&gt;</code> の参照を返す。Python の <code>def_property</code> は <code>def_readwrite</code> の参照を返す。</p>
<code>return_value_policy::automatic</code>	<p>この関数は、この関数の内部で定義されたオブジェクトの参照を返す。この関数は、<code>return_value_policy::take_ownership</code> の参照を返す。この関数は、<code>return_value_policy::copy</code> の参照を返す。Python の <code>py::class_</code> は <code>py::class_</code> の参照を返す。</p>
<code>return_value_policy::automatic_reference</code>	<p>この関数は、この関数の内部で定義されたオブジェクトの参照を返す。この関数は、<code>return_value_policy::reference</code> の参照を返す。C++ の <code>pybind11/stl.h</code> は <code>casters</code> の参照を返す。</p>

この関数は、この関数の内部で定義されたオブジェクトの参照を返す。

```
class_<MyClass>(m, "MyClass")
 .def_property("data", &MyClass::getData, &MyClass::setData,
 py::return_value_policy::copy);
```

この関数は、この関数の内部で定義されたオブジェクトの参照を返す。この関数は、`getter` と `setter` の参照を返す。この関数は、`cpp_function` の参照を返す。

```
class_<MyClass>(m, "MyClass")
 .def_property("data"
 py::cpp_function(&MyClass::getData, py::return_value_policy::copy),
 py::cpp_function(&MyClass::setData)
);
```

この関数は、この関数の内部で定義されたオブジェクトの参照を返す。この関数は、`free` の参照を返す。この関数は、`free` の参照を返す。

この関数は、この関数の内部で定義されたオブジェクトの参照を返す。

1. この関数は、この関数の内部で定義されたオブジェクトの参照を返す。この関数は、`pybind11` の参照を返す。この関数は、`pybind11` の参照を返す。
2. この関数は、この関数の内部で定義されたオブジェクトの参照を返す。この関数は、`Python` の参照を返す。
3. この関数は、この関数の内部で定義されたオブジェクトの参照を返す。この関数は、`C++` の `Python` の参照を返す。この関数は、`crash` の参照を返す。

## 7.2 関数の参照

この関数は、この関数の内部で定義されたオブジェクトの参照を返す。



## keep alive

C++  
 Patient  
 None

nurse  
 pybind11  
 nurse  
 pybind11

“Could not cativate keep\_alive!”  
 review  
 list append

```
py::class_<List>(m, "List").def("append", &List::append, py::keep_alive<1, 2>());
```

this  
 void

```
py::class_<Nurse>(m, "Nurse").def(py::init<Patient &>(), py::keep_alive<1, 2>());
```

Note: `keep_alive` `Boost.Python` `with_custodian_and_ward` `with_custodian_and_ward_postcall`

## Call guard

`call_guard<T>` `scope guard`

```
m.def("foo", foo, py::call_guard<T>());
```

```
m.def("foo", [](args...) {
 T scope_guard;
 return foo(args...); // forwarded arguments
});
```

`gil_scoped_release`

`call_guard` `call_guard<T1, T2, T3 ...>`

See also: `test/test_call_policies.cpp` `keep_alive` `call_guard`

## 7.3 Python dict

pybind11 can interact with C++ Python dict types

```
void print_dict(const py::dict& dict) {
 /* Easily interact with Python types */
 for (auto item : dict)
 std::cout << "key=" << std::string(py::str(item.first)) << ", "
 << "value=" << std::string(py::str(item.second)) << std::endl;
}

// it can be exported as follow:
m.def("print_dict", &print_dict);
```

Python dict

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

## 7.4 \*args and \*\*kwargs

Python generic function

```
def generic(*args, **kwargs):
 ... # do something with args and kwargs
```

pybind11 generic function

```
void generic(py::args args, const py::kwargs& kwargs) {
 /// .. do something with args
 if (kwargs)
 /// .. do something with kwargs
}

/// Binding code
m.def("generic", &generic);
```

py::args is py::tuple and py::kwargs is py::dict

test/test\_kwargs\_and\_defaults.cpp

## 7.5 Python class

Python class

```
py::class_<MyClass>("MyClass").def("myFunction", py::arg("arg") = SomeType(123));
```

```
SomeType binding py::class_
```

```
__repr__
```

#### FUNCTIONS

```
myFunction(...)
Signature : (MyClass, arg : SomeType = <SomeType object at 0x101b7b080>) -> NoneType
```

```
SomeType.__repr__ arg_v
```

```
py::class_<MyClass>("MyClass")
 .def("myFunction", py::arg_v("arg", SomeType(123), "SomeType(123)"));
```

```
~~~~~
```

```
``c++
```

```
py::class_<MyClass>("MyClass")
    .def("myFunction", py::arg("arg") = static_cast<SomeType *>(nullptr));
```

## 7.6 Keyword-only

Python3 keyword-only \*

```
def f(a, *, b): # a can be positional or via keyword; b must be via keyword
    pass
```

```
f(a=1, b=2) # good
f(b=2, a=1) # good
f(1, b=2) # good
f(1, 2) # TypeError: f() takes 1 positional argument but 2 were given
```

```
pybind11 py::kw_only
```

```
m.def("f", [](int a, int b) { /* ... */ },
    py::arg("a"), py::kw_only(), py::arg("b"));
```

```
py::args
```

## 7.7 Positional-only

python3.8 Positional-only pybind11 py::pos\_only()

```
m.def("f", [](int a, int b) { /* ... */ },
    py::arg("a"), py::pos_only(), py::arg("b"));
```

~~~~~ a ~~~~~keyword-only~~~~~

7.8 Non-converting

~~~~~

- `py::implicitly_convertible<A,B>()` ~~~~~
- ~~~~~
- ~~~~~`std::complex<float>` ~~~~~
- Calling a function taking an Eigen matrix reference with a numpy array of the wrong type or of an incompatible data layout.

~~~~~ `py::arg` ~~~~ `.noconvert()` ~~~~~

```
m.def("floats_only", [](double f) { return 0.5 * f; }, py::arg("f").noconvert());
m.def("floats_preferred", [](double f) { return 0.5 * f; }, py::arg("f"));
```

~~~~~ `TypeError` ~~~~

```
>>> floats_preferred(4)
2.0
>>> floats_only(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: floats_only(): incompatible function arguments. The following argument types are
supported:
    1. (f: float) -> float

Invoked with: 4
```

~~~~~ `_a` ~~~~~ `py::arg().noconvert()` ~

7.9 /~~~~~

~~~~~ `py::class_` ~~~~C++~~~~~shared holder(~~~~~)~~~~~pybind11~~~~~Python~~~~~None~~~~~  
C++~~~~~`nullptr`~~~~~

~~~~~ `py::arg` ~~~~ `.none` ~~~~~

```
py::class_<Dog>(m, "Dog").def(py::init<>());
py::class_<Cat>(m, "Cat").def(py::init<>());
m.def("bark", [](Dog *dog) -> std::string {
    if (dog) return "woof!"; /* Called with a Dog instance */
    else return "(no dog)"; /* Called with None, dog == nullptr */
}, py::arg("dog").none(true));
m.def("meow", [](Cat *cat) -> std::string {
    // Can't be called with None argument
    return "meow";
}, py::arg("cat").none(false));
```

Python `bark(None)` returns `"(no dog)"` `meow(None)` raises `TypeError`

```
>>> from animals import Dog, Cat, bark, meow
>>> bark(Dog())
'woof!'
>>> meow(Cat())
'meow'
>>> bark(None)
'(no dog)'
>>> meow(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: meow(): incompatible function arguments. The following argument types are
supported:
    1. (cat: animals.Cat) -> str

Invoked with: None
```

Python `None`

Note: Even when `.none(true)` is specified for an argument, `None` will be converted to a `nullptr` only for custom and opaque types. Pointers to built-in types (`double *`, `int *`, ...) and STL types (`std::vector<T> *`, ...; if `pybind11/stl.h` is included) are copied when converted to C++ (see [Overview](#)) and will not allow `None` as argument. To pass optional argument of these copied types consider using `std::optional<T>`

7.10 Python

pybind11 `py::arg().noconvert()` `py::prepend()`

`TypeError`

Note: pybind11 `py::prepend()`

8.

~~~~~

### 8.1 Python

~~~~~C++~~~~~Python~~~~~

```
class Animal {
public:
    virtual ~Animal() { }
    virtual std::string go(int n_times) = 0;
};

class Dog : public Animal {
public:
    std::string go(int n_times) override {
        std::string result;
        for (int i=0; i<n_times; ++i)
            result += "woof! ";
        return result;
    }
};
```

~~~~~Animal~~~~~go() ~~~~

```
std::string call_go(Animal *animal) {
    return animal->go(3);
}
```

pybind11~~~~~

```
PYBIND11_MODULE(example, m) {
    py::class_<Animal>(m, "Animal")
        .def("go", &Animal::go);

    py::class_<Dog, Animal>(m, "Dog")
        .def(py::init<>());

    m.def("call_go", &call_go);
}
```

~~~~~Animal~~~~~"No constructor defined!"~~~~~Animal~~~~~"  
(trampoline)"~~~~~Python

~~~~~Python~~~~~Animal~~~~~

```

class PyAnimal : public Animal {
public:
    /* Inherit the constructors */
    using Animal::Animal;

    /* Trampoline (need one for each virtual function) */
    std::string go(int n_times) override {
        PYBIND11_OVERRIDE_PURE(
            std::string, /* Return type */
            Animal,      /* Parent class */
            go,          /* Name of function in C++ (must match Python name) */
            n_times      /* Argument(s) */
        );
    }
};

```

PYBIND11\_OVERRIDE\_PURE PYBIND11\_OVERRIDE  
 PYBIND11\_OVERRIDE\_PURE\_NAME PYBIND11\_OVERRIDE\_NAME CPython  
 \_\_str\_\_

```

std::string toString() override {
    PYBIND11_OVERRIDE_NAME(
        std::string, // Return type (ret_type)
        Animal,     // Parent class (cname)
        "__str__",  // Name of method in Python (name)
        toString,   // Name of function in C++ (fn)
    );
}

```

Animal

```

PYBIND11_MODULE(example, m) {
    py::class_<Animal, PyAnimal /* <--- trampoline */>(m, "Animal")
        .def(py::init<>())
        .def("go", &Animal::go);

    py::class_<Dog, Animal>(m, "Dog")
        .def(py::init<>());

    m.def("call_go", &call_go);
}

```

pybind11 class\_ PyAnimal Python Animal

```

py::class_<Animal, PyAnimal /* <--- trampoline */>(m, "Animal");
    .def(py::init<>())
    .def("go", &PyAnimal::go); /* <--- THIS IS WRONG, use &Animal::go */

```

Python Animal Dog

Python Animal::go

```

from example import *
d = Dog()
call_go(d)      # u'woof! woof! woof! '
class Cat(Animal):
    def go(self, n_times):
        return "meow! " * n_times

c = Cat()
call_go(c)      # u'meow! meow! meow! '

```

Python C++ ( `__init__` ) `TypeError`

```

class Dachshund(Dog):
    def __init__(self, name):
        Dog.__init__(self) # Without this, a TypeError is raised.
        self.name = name

    def bark(self):
        return "yap!"

```

`__init__` `super()` `super()` Python C++ Python MRO C++

#### Note

pybind11 Python

- because in these cases there is no C++ variable to reference (the value is stored in the referenced Python variable), pybind11 provides one in the `PYBIND11_OVERRIDE` macros (when needed) with static storage duration. Note that this means that invoking the overridden method on *any* instance will change the referenced value stored in *all* instances of that type.
- Attempts to modify a non-const reference will not have the desired effect: it will change only the static cache variable, but this change will not propagate to underlying Python instance, and the change will be replaced the next time the override is invoked.

## 8.2

Python `Animal` `Dog`



```

class Animal {
public:
    virtual std::string go(int n_times) = 0;
    virtual std::string name() { return "unknown"; }
};
class Dog : public Animal {
public:
    std::string go(int n_times) override {
        std::string result;
        for (int i=0; i<n_times; ++i)
            result += bark() + " ";
        return result;
    }
    virtual std::string bark() { return "woof!"; }
};

```

~~~~~Animal~~~~~Python~~~~~ Dog ~~~~~ Dog ~~~~~ bark() ~~~~  
 Animal go() name() ~~~~~Dog~~~~~name~~~~~

```

class PyAnimal : public Animal {
public:
    using Animal::Animal; // Inherit constructors
    std::string go(int n_times) override { PYBIND11_OVERRIDE_PURE(std::string, Animal, go,
n_times); }
    std::string name() override { PYBIND11_OVERRIDE(std::string, Animal, name, ); }
};
class PyDog : public Dog {
public:
    using Dog::Dog; // Inherit constructors
    std::string go(int n_times) override { PYBIND11_OVERRIDE(std::string, Dog, go,
n_times); }
    std::string name() override { PYBIND11_OVERRIDE(std::string, Dog, name, ); }
    std::string bark() override { PYBIND11_OVERRIDE(std::string, Dog, bark, ); }
};

```

~~~ name() ~ bark() ~~~~~

~~~~~pybind11~~~~~

```

class Husky : public Dog {};
class PyHusky : public Husky {
public:
    using Husky::Husky; // Inherit constructors
    std::string go(int n_times) override { PYBIND11_OVERRIDE_PURE(std::string, Husky, go,
n_times); }
    std::string name() override { PYBIND11_OVERRIDE(std::string, Husky, name, ); }
    std::string bark() override { PYBIND11_OVERRIDE(std::string, Husky, bark, ); }
};

```

~~~~~

```
template <class AnimalBase = Animal> class PyAnimal : public AnimalBase {
public:
    using AnimalBase::AnimalBase; // Inherit constructors
    std::string go(int n_times) override { PYBIND11_OVERRIDE_PURE(std::string, AnimalBase,
go, n_times); }
    std::string name() override { PYBIND11_OVERRIDE(std::string, AnimalBase, name, ); }
};
template <class DogBase = Dog> class PyDog : public PyAnimal<DogBase> {
public:
    using PyAnimal<DogBase>::PyAnimal; // Inherit constructors
    // Override PyAnimal's pure virtual go() with a non-pure one:
    std::string go(int n_times) override { PYBIND11_OVERRIDE(std::string, DogBase, go,
n_times); }
    std::string bark() override { PYBIND11_OVERRIDE(std::string, DogBase, bark, ); }
};
```

pybind11::class\_<Animal, PyAnimal>> animal(m, "Animal");

pybind11::class\_<Dog, Animal, PyDog>> dog(m, "Dog");

```
py::class_<Animal, PyAnimal>> animal(m, "Animal");
py::class_<Dog, Animal, PyDog>> dog(m, "Dog");
py::class_<Husky, Dog, PyDog<Husky>> husky(m, "Husky");
// ... add animal, dog, husky definitions
```

pybind11::class\_<Husky, Dog, PyDog<Husky>> husky(m, "Husky");

Python::class\_<Husky, Dog, PyDog<Husky>> husky(m, "Husky");

```
class ShihTzu(Dog):
    def bark(self):
        return "yip!"
```

## 8.3 Python bindings

### 8.3.1 Python bindings

Python bindings are implemented by creating a `py::class_<T, C++Class>` object, where `T` is the Python type and `C++Class` is the C++ class. The `py::class_<T, C++Class>` object is created by calling `py::class_<T, C++Class>(m, "name")`, where `m` is the module object and `"name"` is the name of the class in Python.

The `py::class_<T, C++Class>` object is created by calling `py::class_<T, C++Class>(m, "name")`, where `m` is the module object and `"name"` is the name of the class in Python.

```
py::init_alias<Args, ...>() py::init<Args, ...>() py::init<Args, ...>()
py::init_alias<Args, ...>() py::init<Args, ...>() py::init<Args, ...>()
```

**See also** See the file `tests/test_virtual_functions.cpp` for complete examples showing both normal and forced trampoline instantiation.

Page 10

```
C++C++
```

`get_override()` Python Consider for example a C++ method which has the signature `bool myMethod(int32_t& value)`, where the return indicates whether something should be done with the `value`. This can be made convenient on the Python side by allowing the Python function to return `None` or an `int`:

```
bool MyClass::myMethod(int32_t& value)
{
    pybind11::gil_scoped_acquire gil; // Acquire the GIL while in this scope.
    // Try to look up the overridden method on the Python side.
    pybind11::function override = pybind11::get_override(this, "myMethod");
    if (override) { // method is found
        auto obj = override(value); // Call the Python function.
        if (py::isinstance<py::int_>(obj)) { // check if it returned a Python integer type
            value = obj.cast<int32_t>(); // Cast it and assign it to the value.
            return true; // Return true; value should be used.
        } else {
            return false; // Python returned none, return false.
        }
    }
    return false; // Alternatively return MyClass::myMethod(value);
}
```

## 8.4 練習問題

```
class Example {
private:
    Example(int); // private constructor
public:
    // Factory function:
    static Example create(int a) { return Example(a); }
};

py::class_<Example>(m, "Example")
    .def(py::init(&Example::create));
```

```

create Python .def(py::init(...))
create py::init()
std::unique_ptr

```

```

class Example {
private:
    Example(int); // private constructor
public:
    // Factory function - returned by value:
    static Example create(int a) { return Example(a); }

    // These constructors are publicly callable:
    Example(double);
    Example(int, int);
    Example(std::string);
};

py::class_<Example>(m, "Example")
    // Bind the factory function as a constructor:
    .def(py::init(&Example::create))
    // Bind a lambda function returning a pointer wrapped in a holder:
    .def(py::init([](std::string arg) {
        return std::unique_ptr<Example>(new Example(arg));
    })))
    // Return a raw pointer:
    .def(py::init([](int a, int b) { return new Example(a, b); })))
    // You can mix the above with regular C++ constructor bindings as well:
    .def(py::init<double>())
    ;

```

Python pybind11 C++ Python

py::init()

py::init\_alias<...>

```

#include <pybind11/factory.h>
class Example {
public:
    // ...
    virtual ~Example() = default;
};
class PyExample : public Example {
public:
    using Example::Example;
    PyExample(Example &&base) : Example(std::move(base)) {}
};
py::class_<Example, PyExample>(m, "Example")
    // Returns an Example pointer. If a PyExample is needed, the Example
    // instance will be moved via the extra constructor in PyExample, above.
    .def(py::init[]() { return new Example(); })
    // Two callbacks:
    .def(py::init[]() { return new Example(); } /* no alias needed */,
        []() { return new PyExample(); } /* alias needed */)
    // *Always* returns an alias instance (like py::init_alias<>())
    .def(py::init[]() { return new PyExample(); })
    ;

```

```
struct Aggregate {
    int a;
    std::string b;
};

py::class_(m, "Aggregate")
    .def(py::init<int, const std::string &>());
```

```
/* ... definition ... */

class MyClass {
private:
    ~MyClass() { }
};

/* ... binding code ... */

py::class_<MyClass, std::unique_ptr<MyClass, py::nodelete>>(m, "MyClass")
    .def(py::init<>())
```

## 37 / 150

```

class MyClass {
public:
    ~MyClass() {
        try {
            py::print("Even printing is dangerous in a destructor");
            py::exec("raise ValueError('This is an unraisable exception')");
        } catch (py::error_already_set &e) {
            // error_context should be information about where/why the occurred,
            // e.g. use __func__ to get the name of the current function
            e.discard_as_unraisable(__func__);
        }
    }
};

```

Note: pybind11 C++ `noexcept(false)`

## 8.7

A B A B

```

py::class_<A>(m, "A")
    /// ... members ...

py::class_<B>(m, "B")
    .def(py::init<A>())
    /// ... members ...

m.def("func",
    [](const B &) { /* .... */ }
);

```

func A Python func(B(a)) C++ func(a) A B

B A Python

```
py::implicitly_convertible<A, B>();
```

Note: A B pybind11 B

an implicit conversion invoked as part of another implicit conversion of the same type (i.e. from A to B) will fail.

## 8.8

getter setter self Python Python type C++ lambda getter self

```
py::class_<Foo>(m, "Foo")
    .def_property_readonly_static("foo", [](py::object /* self */) { return Foo(); });
```

## 8.9 向量的实现

向量的实现 `Vector2` 的实现如下：

```
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;
};
```

向量的实现

```
#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);
}
```

`.def(py::self * float())` 的实现如下：

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

## 8.10 pickle

Python `pickle` module Python `pybind11` `py::pickle()` `pickle` `unpickle` C++

```
class Pickleable {
public:
    Pickleable(const std::string &value) : m_value(value) { }
    const std::string &value() const { return m_value; }

    void setExtra(int extra) { m_extra = extra; }
    int extra() const { return m_extra; }
private:
    std::string m_value;
    int m_extra = 0;
};
```

Python `__setstate__` `__getstate__` `pciking` `pybind11` `py::pickle()`

```
py::class_<Pickleable>(m, "Pickleable")
    .def(py::init<std::string>())
    .def("value", &Pickleable::value)
    .def("extra", &Pickleable::extra)
    .def("setExtra", &Pickleable::setExtra)
    .def(py::pickle(
        [](const Pickleable &p) { // __getstate__
            /* Return a tuple that fully encodes the state of the object */
            return py::make_tuple(p.value(), p.extra());
        },
        [](py::tuple t) { // __setstate__
            if (t.size() != 2)
                throw std::runtime_error("Invalid state!");

            /* Create a new C++ instance */
            Pickleable p(t[0].cast<std::string>());

            /* Assign any additional state */
            p.setExtra(t[1].cast<int>());

            return p;
        }
    ));
```

`py::pickle()` `__setstate__` `py::init()` holder type

Python



```
try:
    import cPickle as pickle # Use cPickle on Python 2.7
except ImportError:
    import pickle

p = Pickleable("test_value")
p.setExtra(15)
data = pickle.dumps(p, 2)
```

Note: Note that only the cPickle module is supported on Python 2.7.

The second argument to `dumps` is also crucial: it selects the pickle protocol version 2, since the older version 1 is not supported. Newer versions are also fine—for instance, specify `-1` to always use the latest available version. Beware: failure to follow these instructions will cause important pybind11 memory allocation routines to be skipped during unpickling, which will likely lead to memory corruption and/or segmentation faults.

## 8.11 深拷贝

Python 的 `copy` 模块提供了深拷贝和浅拷贝的功能。

Python 3 使用 `pickle` 模块的 `__copy__` 和 `__deepcopy__` 方法来实现深拷贝。Python 2.7 使用 `pybind11` 的 `cPickle` 模块来实现深拷贝。

以下是一个使用 `__copy__` 和 `__deepcopy__` 方法的示例：

```
py::class_<Copyable>(m, "Copyable")
    .def("__copy__", [](const Copyable &self) {
        return Copyable(self);
    })
    .def("__deepcopy__", [](const Copyable &self, py::dict) {
        return Copyable(self);
    }, "memo"_a);
```

Note: 在 Python 2.7 中，使用 `pybind11` 的 `cPickle` 模块来实现深拷贝。

## 8.12 基类

pybind11 支持使用 `class_` 来定义基类。

```
py::class_<MyType, BaseType1, BaseType2, BaseType3>(m, "MyType")
    ...
```

以下是一个使用 `holder` 和 `pybind11` 来定义基类的示例：

Python C++ Python

pybind11 multiple\_inheritance

```
py::class_<MyType, BaseType2>(m, "MyType", py::multiple_inheritance());
```

## 8.13 Module-local

pybind11

```
// In the module1.cpp binding code for module1:
py::class_<Pet>(m, "Pet")
    .def(py::init<std::string>())
    .def_readonly("name", &Pet::name);

// In the module2.cpp binding code for module2:
m.def("create_pet", [](std::string name) { return new Pet(name); });
```

```
>>> from module1 import Pet
>>> from module2 import create_pet
>>> pet1 = Pet("Kitty")
>>> pet2 = create_pet("Doggy")
>>> pet2.name()
'Doggy'
```

Python

C++ Python

```
// dogs.cpp

// Binding for external library class:
py::class_<pets::Pet>(m, "Pet")
    .def("name", &pets::Pet::name);

// Binding for local extension class:
py::class_<Dog, pets::Pet>(m, "Dog")
    .def(py::init<std::string>());
```

```
// cats.cpp, in a completely separate project from the above dogs.cpp.

// Binding for external library class:
py::class<pets::Pet>(m, "Pet")
    .def("get_name", &pets::Pet::name);

// Binding for local extending class:
py::class<Cat, pets::Pet>(m, "Cat")
    .def(py::init<std::string>());
```

```
>>> import cats
>>> import dogs
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: generic_type: type "Pet" is already registered!
```

py::class\_ py::module\_local()

```
// Pet binding in dogs.cpp:
py::class<pets::Pet>(m, "Pet", py::module_local())
    .def("name", &pets::Pet::name);
```

```
// Pet binding in cats.cpp:
py::class<pets::Pet>(m, "Pet", py::module_local())
    .def("get_name", &pets::Pet::name);
```

Python dogs.Pet cats.Pet 1  
Pet Python2 Python

C++Python py::module\_local C++module-local  
cats dogs dogs.Pet cats.Pet

```
m.def("pet_name", [](const pets::Pet &pet) { return pet.name(); });
```

cats.cpp dogs.cpp frogs.cpp frogs.cpp Pets

```
>>> import cats, dogs, frogs # No error because of the added py::module_local()
>>> mycat, mydog = cats.Cat("Fluffy"), dogs.Dog("Rover")
>>> (cats.pet_name(mycat), dogs.pet_name(mydog))
('Fluffy', 'Rover')
>>> (cats.pet_name(mydog), dogs.pet_name(mycat), frogs.pet_name(mycat))
('Rover', 'Fluffy', 'Fluffy')
```

py::module\_local() module-local  
C++PythonPython

Note: STL bindings (as provided via the optional `pybind11/stl_bind.h` header) apply `py::module_local` by default when the bound type might conflict with other modules; see Binding STL containers for details.

The localization of the bound types is actually tied to the shared object or binary generated by the compiler/linker. For typical modules created with `PYBIND11_MODULE()`, this distinction is not significant. It is possible, however, when [Embedding the interpreter](#) to embed multiple modules in the same binary (see [Adding embedded modules](#)). In such a case, the localization will apply across all embedded modules within the same binary.

## 8.14 `protected`

Python `protected`

```
class A {
protected:
    int foo() const { return 42; }
};

py::class_<A>(m, "A")
    .def("foo", &A::foo); // error: 'foo' is a protected member of 'A'
```

Python `protected`

```
class A {
protected:
    int foo() const { return 42; }
};

class Publicist : public A { // helper type for exposing protected functions
public:
    using A::foo; // inherited with different access modifier
};

py::class_<A>(m, "A") // bind the primary class
    .def("foo", &Publicist::foo); // expose protected methods via the publicist
```

`&Publicist::foo` `&A::foo` `Publicist` `public`

Python `protected` `publicist` pattern `trampoline`

```

class A {
public:
    virtual ~A() = default;

protected:
    virtual int foo() const { return 42; }
};

class Trampoline : public A {
public:
    int foo() const override { PYBIND11_OVERRIDE(int, A, foo, ); }
};

class Publicist : public A {
public:
    using A::foo;
};

py::class_<A, Trampoline>(m, "A") // <-- `Trampoline` here
    .def("foo", &Publicist::foo); // <-- `Publicist` here, not `Trampoline`!

```

## 8.15 `final`

C++11 introduces `final` and `py::is_final` Python mimics C++ `final`

```

class IsFinal final {};

py::class_<IsFinal>(m, "IsFinal", py::is_final());

```

Python

```

class PyFinalChild(IsFinal):
    pass

```

```

TypeError: type 'IsFinal' is not an acceptable base type

```

## 8.16

“””pybind11C++Sometimes, you might want to provide this automatic downcasting behavior when creating bindings for a class hierarchy that does not use standard C++ polymorphism, such as LLVM. As long as there’s some way to determine at runtime whether a downcast is safe, you can proceed by specializing the `pybind11::polymorphic_type_hook` template:

```

enum class PetKind { Cat, Dog, Zebra };
struct Pet { // Not polymorphic: has no virtual methods
    const PetKind kind;
    int age = 0;
protected:
    Pet(PetKind _kind) : kind(_kind) {}
};
struct Dog : Pet {
    Dog() : Pet(PetKind::Dog) {}
    std::string sound = "woof!";
    std::string bark() const { return sound; }
};

namespace pybind11 {
    template<> struct polymorphic_type_hook<Pet> {
        static const void *get(const Pet *src, const std::type_info& type) {
            // note that src may be nullptr
            if (src && src->kind == PetKind::Dog) {
                type = &typeid(Dog);
                return static_cast<const Dog*>(src);
            }
            return src;
        }
    };
} // namespace pybind11

```

When pybind11 wants to convert a C++ pointer of type `Base*` to a Python object, it calls `polymorphic_type_hook<Base*>::get()` to determine if a downcast is possible. The `get()` function should use whatever runtime information is available to determine if its `src` parameter is in fact an instance of some class `Derived` that inherits from `Base`. If it finds such a `Derived`, it sets `type = &typeid(Derived)` and returns a pointer to the `Derived` object that contains `src`. Otherwise, it just returns `src`, leaving `type` at its default value of `nullptr`. If you set `type` to a type that pybind11 doesn't know about, no downcasting will occur, and the original `src` pointer will be used with its static type `Base*`.

It is critical that the returned pointer and `type` argument of `get()` agree with each other: if `type` is set to something non-null, the returned pointer must point to the start of an object whose type is `type`. If the hierarchy being exposed uses only single inheritance, a simple `return src;` will achieve this just fine, but in the general case, you must cast `src` to the appropriate derived-class pointer (e.g. using `static_cast<Derived*>(src)`) before allowing it to be returned as a `void*`.

## 8.17 Python C++

Python C++

```
py::type T_py = py::type::of<T>();
```

Python C++ Python Python `type(ob)`

## 9.

### 9.1 C++Python

Pythonpybind11C++pybind11C++PythonPythonPython

pybind11 `std::exception` PythonPythonPython Python C APIC++pybind11Python

Exception thrown by C++	Translated to Python exception type
<code>std::exception</code>	<code>RuntimeError</code>
<code>std::bad_alloc</code>	<code>MemoryError</code>
<code>std::domain_error</code>	<code>ValueError</code>
<code>std::invalid_argument</code>	<code>ValueError</code>
<code>std::length_error</code>	<code>ValueError</code>
<code>std::out_of_range</code>	<code>IndexError</code>
<code>std::range_error</code>	<code>ValueError</code>
<code>std::overflow_error</code>	<code>OverflowError</code>
<code>pybind11::stop_iteration</code>	<code>StopIteration</code> (used to implement custom iterators)
<code>pybind11::index_error</code>	<code>IndexError</code> (used to indicate out of bounds access in <code>__getitem__</code> , <code>__setitem__</code> , etc.)
<code>pybind11::key_error</code>	<code>KeyError</code> (used to indicate out of bounds access in <code>__getitem__</code> , <code>__setitem__</code> in dict-like objects, etc.)
<code>pybind11::value_error</code>	<code>ValueError</code> (used to indicate wrong value passed in <code>container.remove(...)</code> )
<code>pybind11::type_error</code>	<code>TypeError</code>
<code>pybind11::buffer_error</code>	<code>BufferError</code>
<code>pybind11::import_error</code>	<code>ImportError</code>
<code>pybind11::attribute_error</code>	<code>AttributeError</code>
Any other exception	<code>RuntimeError</code>

PythonPython `pybind11::error_already_set`

Python `handle::call()` `cast_error`

## 9.2 pybind11 exceptions

pybind11 class objects are global C++ objects. `what()` returns a C++ Python exception object.

```
py::register_exception<CppExp>(module, "PyExp");
```

PyExp Python exception CppExp PyExp

PyExp

```
py::register_local_exception<CppExp>(module, "PyExp");
```

handle

```
py::register_exception<CppExp>(module, "PyExp", PyExc_RuntimeError);
py::register_local_exception<CppExp>(module, "PyExp", PyExc_RuntimeError);
```

PyExp PyExp RuntimeError

Python Python Standard Exceptions PyExc\_Exception

```
py::register_exception_translator(translator)
py::register_local_exception_translator(translator)
void(std::exception_ptr)
```

C++

Inside the translator, `std::rethrow_exception` should be used within a try block to re-throw the exception. One or more catch clauses to catch the appropriate exceptions should then be used with each clause using `PyErr_SetString` to set a Python exception or `ex(string)` to set the python exception to a custom exception type (see below).

To declare a custom Python exception type, declare a `py::exception` variable and use this in the associated exception translator (note: it is often useful to make this a static declaration when using it inside a lambda expression without requiring capturing).

The following example demonstrates this for a hypothetical exception classes `MyCustomException` and `OtherException`: the first is translated to a custom python exception `MyCustomError`, while the second is translated to a standard python `RuntimeError`:



```
static py::exception<MyCustomException> exc(m, "MyCustomError");
py::register_exception_translator([](std::exception_ptr p) {
    try {
        if (p) std::rethrow_exception(p);
    } catch (const MyCustomException &e) {
        exc(e.what());
    } catch (const OtherException &e) {
        PyErr_SetString(PyExc_RuntimeError, e.what());
    }
});
```

Multiple exceptions can be handled by a single translator, as shown in the example above. If the exception is not caught by the current translator, the previously registered one gets a chance.

If none of the registered exception translators is able to handle the exception, it is handled by the default converter as described in the previous section.

### 9.3 Local vs Global Exception Translators

When a global exception translator is registered, it will be applied across all modules in the reverse order of registration. This can create behavior where the order of module import influences how exceptions are translated.

If module1 has the following translator:

```
py::register_exception_translator([](std::exception_ptr p) {
    try {
        if (p) std::rethrow_exception(p);
    } catch (const std::invalid_argument &e) {
        PyErr_SetString("module1 handled this")
    }
})
```

and module2 has the following similar translator:

```
py::register_exception_translator([](std::exception_ptr p) {
    try {
        if (p) std::rethrow_exception(p);
    } catch (const std::invalid_argument &e) {
        PyErr_SetString("module2 handled this")
    }
})
```

then which translator handles the `invalid_argument` will be determined by the order that module1 and module2 are imported. Since exception translators are applied in the reverse order of registration, which ever module was imported last will “win” and that translator will be applied.

If there are multiple pybind11 modules that share exception types (either standard built-in or custom) loaded into a single python instance and consistent error handling behavior is needed, then local translators should be used.

Changing the previous example to use `register_local_exception_translator` would mean that when `invalid_argument` is thrown in the module2 code, the module2 translator will always handle it, while in module1, the module1 translator will do the same.

## 9.4 C++Python

C++PythonPythonPythonPythonpybind11Python  
`pybind11::error_already_set` C++Python `error_already_set`  
 PythonPythonC++

Exception raised in Python	Thrown as C++ exception type
Any Python <code>Exception</code>	<code>pybind11::error_already_set</code>

```
try {
    // open("missing.txt", "r")
    auto file = py::module_::import("io").attr("open")("missing.txt", "r");
    auto text = file.attr("read")();
    file.attr("close")();
} catch (py::error_already_set &e) {
    if (e.matches(PyExc_FileNotFoundError)) {
        py::print("missing.txt not found");
    } else if (e.matches(PyExc_PermissionError)) {
        py::print("missing.txt found but not accessible");
    } else {
        throw;
    }
}
```

C++PythonPython `error_already_set`.

```

try {
    py::eval("raise ValueError('The Ring')");
} catch (py::value_error &boromir) {
    // Boromir never gets the ring
    assert(false);
} catch (py::error_already_set &frodo) {
    // Frodo gets the ring
    py::print("I will take the ring");
}

try {
    // py::value_error is a request for pybind11 to raise a Python exception
    throw py::value_error("The ball");
} catch (py::error_already_set &cat) {
    // cat won't catch the ball since
    // py::value_error is not a Python exception
    assert(false);
} catch (py::value_error &dog) {
    // dog will catch the ball
    py::print("Run Spot run");
    throw; // Throw it again (pybind11 will raise ValueError)
}

```

## 9.5 Python C API

pybind11 wrappers Python C API Python C API  
 pybind11

Python C API Python `throw py::error_already_set();` pybind11  
 Python `PyErr_SetString`

```

PyErr_SetString(PyExc_TypeError, "C API type error demo");
throw py::error_already_set();

// But it would be easier to simply...
throw py::type_error("pybind11 wrapper type error");

```

`PyErr_Clear`

Python Python/pybind11

## 9.6 raise from

Python 3.3

```

try:
    print(1 / 0)
except Exception as exc:
    raise RuntimeError("could not divide by zero") from exc

```

pybind11 2.8 `py::raise_from` Python `throw`  
`py::error_already_set()` Python 3 only

```
try {
    py::eval("print(1 / 0)");
} catch (py::error_already_set &e) {
    py::raise_from(e, PyExc_RuntimeError, "could not divide by zero");
    throw py::error_already_set();
}
```

## 9.7 `unraisable`

Python C++ `noexcept(true)` `std::terminate()`

`__del__` Python `unraisable` Python 3.8+ system hook `auditing event`

`noexcept` `try-catch` `error_already_set` pybind11 Python  
 Python pybind11 C++ `noexcept` Python  
`discard_as_unraisable`

```
void nonthrowing_func() noexcept(true) {
    try {
        // ...
    } catch (py::error_already_set &eas) {
        // Discard the Python error using Python APIs, using the C++ magic
        // variable __func__. Python already knows the type and value and of the
        // exception object.
        eas.discard_as_unraisable(__func__);
    } catch (const std::exception &e) {
        // Log and discard C++ exceptions.
        third_party::log(e);
    }
}
```

## 10. Python

### 10.1 `std::unique_ptr`

Python `Example` `std::unique_ptr`

```
std::unique_ptr<Example> create_example() { return std::unique_ptr<Example>(new Example()); }

m.def("create_example", &create_example);
```

Python `std::unique_ptr` `pybind11`

```
void do_something_with_example(std::unique_ptr<Example> ex) { ... }
```

Python

### 10.2 `std::shared_ptr`

`class_` `std::unique_ptr<Type>` Python `std::shared_ptr`

```
py::class_<Example, std::shared_ptr<Example> /* <- holder type */>(m, "Example");
```

```
class Child { };

class Parent {
public:
    Parent() : child(std::make_shared<Child>()) { }
    Child *get_child() { return child.get(); } /* Hint: ** DON'T DO THIS ** */
private:
    std::shared_ptr<Child> child;
};

PYBIND11_MODULE(example, m) {
    py::class_<Child, std::shared_ptr<Child>>(m, "Child");

    py::class_<Parent, std::shared_ptr<Parent>>(m, "Parent")
        .def(py::init<>())
        .def("get_child", &Parent::get_child);
}
```

Python

```
from example import Parent
print(Parent().get_child())
```

`Parent::get_child()` `Child` `std::shared_ptr<...>` `pybind11` `std::shared_ptr<...>` `free` `shared`

1. `get_child()`

```
std::shared_ptr<Child> get_child() { return child; }
```

2. `Child` `std::enable_shared_from_this<T>` `Child` `pybind11` `std::shared_ptr<...>`

```
class Child : public std::enable_shared_from_this<Child> { };
```

## 10.3

`pybind11` `std::unique_ptr` `std::shared_ptr` `transparent conversions`

```
PYBIND11_DECLARE_HOLDER_TYPE(T, SmartPtr<T>);
```

`bool` `false`

```
PYBIND11_DECLARE_HOLDER_TYPE(T, SmartPtr<T>, true);
```

`SmartPtr<T>` `T*` `SmartPtr<T>` `T*` `T` `true`

General notes regarding convenience macros

`pybind11` `.get()` `holder_helper`

```
// Always needed for custom holder types
PYBIND11_DECLARE_HOLDER_TYPE(T, SmartPtr<T>);

// Only needed if the type's `.get()` goes by another name
namespace pybind11 { namespace detail {
    template <typename T>
    struct holder_helper<SmartPtr<T>> { // <-- specialization
        static const T *get(const SmartPtr<T> &p) { return p.getPointer(); }
    };
}}
```

pybind11 SmartPtr .getPointer() .get()

see also: [tests/test\\_smart\\_ptr.cpp](#) holder

## 11. pybind11

pybind11 is a library that allows you to connect C++ and Python. It provides a simple and efficient way to interface the two languages.

1. pybind11 provides a simple and efficient way to interface C++ and Python.
2. pybind11 provides a simple and efficient way to interface C++ and Python.
3. C++ code can use Python objects and Python code can use C++ objects. pybind11 provides a simple and efficient way to interface C++ and Python.

pybind11 provides a simple and efficient way to interface C++ and Python.

### 11.1 Native type in C++, wrapper in Python

#### 1. Native type in C++, wrapper in Python

py::class\_ is a wrapper for C++ classes. It provides a simple and efficient way to interface C++ and Python. py::class\_ is a wrapper for C++ classes. It provides a simple and efficient way to interface C++ and Python.

#### 2. Wrapper in C++, native type in Python

py::object is a wrapper for Python objects. It provides a simple and efficient way to interface C++ and Python. py::object is a wrapper for Python objects. It provides a simple and efficient way to interface C++ and Python.

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

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

Python list is a wrapper for C++ py::list. It provides a simple and efficient way to interface C++ and Python. Python list is a wrapper for C++ py::list. It provides a simple and efficient way to interface C++ and Python.

#### 3. Converting between native C++ and Python types

py::vector is a wrapper for C++ std::vector. It provides a simple and efficient way to interface C++ and Python. py::vector is a wrapper for C++ std::vector. It provides a simple and efficient way to interface C++ and Python.

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



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

pybind11 `std::vector<int>` Python list `print_vector` `list` C++ `vector`

This requires some manual effort and more details are available in the [Making opaque types](#) section.

`include`

Data type	Description	Header file
<code>int8_t</code> , <code>uint8_t</code>	8-bit integers	<code>pybind11/pybind11.h</code>
<code>int16_t</code> , <code>uint16_t</code>	16-bit integers	<code>pybind11/pybind11.h</code>
<code>int32_t</code> , <code>uint32_t</code>	32-bit integers	<code>pybind11/pybind11.h</code>
<code>int64_t</code> , <code>uint64_t</code>	64-bit integers	<code>pybind11/pybind11.h</code>
<code>ssize_t</code> , <code>size_t</code>	Platform-dependent size	<code>pybind11/pybind11.h</code>
<code>float</code> , <code>double</code>	Floating point types	<code>pybind11/pybind11.h</code>
<code>bool</code>	Two-state Boolean type	<code>pybind11/pybind11.h</code>
<code>char</code>	Character literal	<code>pybind11/pybind11.h</code>
<code>char16_t</code>	UTF-16 character literal	<code>pybind11/pybind11.h</code>
<code>char32_t</code>	UTF-32 character literal	<code>pybind11/pybind11.h</code>
<code>wchar_t</code>	Wide character literal	<code>pybind11/pybind11.h</code>
<code>const char *</code>	UTF-8 string literal	<code>pybind11/pybind11.h</code>
<code>const char16_t *</code>	UTF-16 string literal	<code>pybind11/pybind11.h</code>
<code>const char32_t *</code>	UTF-32 string literal	<code>pybind11/pybind11.h</code>
<code>const wchar_t *</code>	Wide string literal	<code>pybind11/pybind11.h</code>
<code>std::string</code>	STL dynamic UTF-8 string	<code>pybind11/pybind11.h</code>
<code>std::u16string</code>	STL dynamic UTF-16 string	<code>pybind11/pybind11.h</code>
<code>std::u32string</code>	STL dynamic UTF-32 string	<code>pybind11/pybind11.h</code>
<code>std::wstring</code>	STL dynamic wide string	<code>pybind11/pybind11.h</code>

Data type	Description	Header file
<code>std::string_view</code> , <code>std::u16string_view</code> , etc.	STL C++17 string views	<code>pybind11/pybind11.h</code>
<code>std::pair&lt;T1, T2&gt;</code>	Pair of two custom types	<code>pybind11/pybind11.h</code>
<code>std::tuple&lt;...&gt;</code>	Arbitrary tuple of types	<code>pybind11/pybind11.h</code>
<code>std::reference_wrapper&lt;...&gt;</code>	Reference type wrapper	<code>pybind11/pybind11.h</code>
<code>std::complex&lt;T&gt;</code>	Complex numbers	<code>pybind11/complex.h</code>
<code>std::array&lt;T, Size&gt;</code>	STL static array	<code>pybind11/stl.h</code>
<code>std::vector&lt;T&gt;</code>	STL dynamic array	<code>pybind11/stl.h</code>
<code>std::deque&lt;T&gt;</code>	STL double-ended queue	<code>pybind11/stl.h</code>
<code>std::valarray&lt;T&gt;</code>	STL value array	<code>pybind11/stl.h</code>
<code>std::list&lt;T&gt;</code>	STL linked list	<code>pybind11/stl.h</code>
<code>std::map&lt;T1, T2&gt;</code>	STL ordered map	<code>pybind11/stl.h</code>
<code>std::unordered_map&lt;T1, T2&gt;</code>	STL unordered map	<code>pybind11/stl.h</code>
<code>std::set&lt;T&gt;</code>	STL ordered set	<code>pybind11/stl.h</code>
<code>std::unordered_set&lt;T&gt;</code>	STL unordered set	<code>pybind11/stl.h</code>
<code>std::optional&lt;T&gt;</code>	STL optional type (C++17)	<code>pybind11/stl.h</code>
<code>std::experimental::optional&lt;T&gt;</code>	STL optional type (exp.)	<code>pybind11/stl.h</code>
<code>std::variant&lt;...&gt;</code>	Type-safe union (C++17)	<code>pybind11/stl.h</code>
<code>std::filesystem::path&lt;T&gt;</code>	STL path (C++17) <sup>1</sup>	<code>pybind11/stl.h</code>
<code>std::function&lt;...&gt;</code>	STL polymorphic function	<code>pybind11/functional.h</code>
<code>std::chrono::duration&lt;...&gt;</code>	STL time duration	<code>pybind11/chrono.h</code>
<code>std::chrono::time_point&lt;...&gt;</code>	STL date/time	<code>pybind11/chrono.h</code>
<code>Eigen::Matrix&lt;...&gt;</code>	Eigen: dense matrix	<code>pybind11/eigen.h</code>
<code>Eigen::Map&lt;...&gt;</code>	Eigen: mapped memory	<code>pybind11/eigen.h</code>
<code>Eigen::SparseMatrix&lt;...&gt;</code>	Eigen: sparse matrix	<code>pybind11/eigen.h</code>

## 11.2 Strings, bytes and Unicode conversions

Note: `std::string` is Python3 strings, `python2.7` is `unicode` or `str` or `bytes` or Python2.7 `from __future__ import unicode_literals` or `str` or `unicode`

### 11.2.1 Python strings to C++

`std::string` is `char *` or Python `str` or `pybind11` Python `UTF-8` or Python `str` or `UTF-8`

C++ is encoding agnostic or `UTF-8`

```
m.def("utf8_test",
      [](const std::string &s) {
          cout << "utf-8 is icing on the cake.\n";
          cout << s;
      }
);
m.def("utf8_charptr",
      [](const char *s) {
          cout << "My favorite food is\n";
          cout << s;
      }
);
```

```
>>> utf8_test("🍰")
utf-8 is icing on the cake.
🍰

>>> utf8_charptr("🍕")
My favorite food is
🍕
```

Note: `UTF-8` emoji or

C++ `const`

C++ `bytes`

`std::string` is `char *` or C++ or Python bytes or Python3 or bytes or `py::bytes`

### 11.2.2 Python to C++

C++ `std::string` is `char*` or Python `pybind11` `UTF-8` or Python `str` or Python `bytes.decode('utf-8')` or `pybind11` `UnicodeDecodeError`

```
m.def("std_string_return",
      []() {
          return std::string("This string needs to be UTF-8 encoded");
      }
    );
```

```
>>> isinstance(example.std_string_return(), str)
True
```

UTF-8ASCIIASCIIPythonUTF-8

Warning: char \* null

C++UTF-8stringpy::str

```
// This uses the Python C API to convert Latin-1 to Unicode
m.def("str_output",
      []() {
          std::string s = "Send your r\xe9sum\xe9 to Alice in HR"; // Latin-1
          py::str py_s = PyUnicode_DecodeLatin1(s.data(), s.length());
          return py_s;
      }
    );
```

```
>>> str_output()
'Send your résumé to Alice in HR'
```

Python C APIlibiconvUTF-8

C++

C++ std::string bytesPythonpy::bytes

```
m.def("return_bytes",
      []() {
          std::string s("\xba\xd0\xba\xd0"); // Not valid UTF-8
          return py::bytes(s); // Return the data without transcoding
      }
    );
```

```
>>> example.return_bytes()
b'\xba\xd0\xba\xd0'
```

pybind11bytesstd::stringstd::stringbytes

```
m.def("asymmetry",
    [](std::string s) { // Accepts str or bytes from Python
        return s; // Looks harmless, but implicitly converts to str
    }
);
```

```
>>> isinstance(example.asymmetry(b"have some bytes"), str)
True
```

```
>>> example.asymmetry(b"\xba\xd0\xba\xd0") # invalid utf-8 as bytes
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xba in position 0: invalid start byte
```

### 11.2.3 練習問題

```

std::wstring  wchar_t*  std::u16string  std::u32string  C++ Python str  str
UTF-16 UTF-32 C++ C++ Python str UTF-16
UTF-32

```

```
#define UNICODE
#include <windows.h>

m.def("set_window_text",
      [](HWND hwnd, std::wstring s) {
          // Call SetWindowText with null-terminated UTF-16 string
          ::SetWindowText(hwnd, s.c_str());
      }
);

m.def("get_window_text",
      [](HWND hwnd) {
          const int buffer_size = ::GetWindowTextLength(hwnd) + 1;
          auto buffer = std::make_unique< wchar_t[] >(buffer_size);

          ::GetWindowText(hwnd, buffer.data(), buffer_size);

          std::wstring text(buffer.get());

          // wstring will be converted to Python str
          return text;
      }
);
```

```
--enable-unicode=ucs2 Python 2.7.3
```

Shift-JIS UTF-8/16/32 Python

### 11.2.4 总结

C++ string

```
m.def("pass_char", [](char c) { return c; });
m.def("pass_wchar", [](wchar_t w) { return w; });
```

```
example.pass_char("A")
'A'
```

C++ char c = 0x65 pybind11 Python chr() Python

```
>>> example.pass_char(0x65)
TypeError

>>> example.pass_char(chr(0x65))
'A'
```

8-bit int8\_t uint8\_t

### 11.2.5 Grapheme clusters

A single grapheme may be represented by two or more Unicode characters. For example ‘é’ is usually represented as U+00E9 but can also be expressed as the combining character sequence U+0065 U+0301 (that is, the letter ‘e’ followed by a combining acute accent). The combining character will be lost if the two-character sequence is passed as an argument, even though it renders as a single grapheme.

```
>>> example.pass_wchar("é")
'é'

>>> combining_e_acute = "e" + "\u0301"

>>> combining_e_acute
'e'

>>> combining_e_acute == "é"
False

>>> example.pass_wchar(combining_e_acute)
'e'
```

Normalizing combining characters before passing the character literal to C++ may resolve *some* of these issues:

```
>>> example.pass_wchar(unicodedata.normalize("NFC", combining_e_acute))
'é'
```

In some languages (Thai for example), there are graphemes that cannot be expressed as a single Unicode code point, so there is no way to capture them in a C++ character type.

### 11.2.6 c++17 string\_view

C++17 string views are automatically supported when compiling in C++17 mode. They follow the same rules for encoding and decoding as the corresponding STL string type (for example, a `std::u16string_view` argument will be passed UTF-16-encoded data, and a returned `std::string_view` will be decoded as UTF-8).

## 11.3 STL

### 11.3.1

```
pybind11/stl.h
std::vector<> / std::deque<> / std::list<> / std::array<> / std::valarray<> ,
std::set<> / std::unordered_set<> , std::map<> / std::unordered_map<> Python list, set dict
std::pair<> std::tuple<> pybind11/pybind11.h
```

Python C++

Note:

### 11.3.2 C++17

```
pybind11/stl.h C++17 std::optional<> std::variant<> C++14
std::experimental::optional<>
```

C++11 Boost pybind11 `type_caster`

```
// `boost::optional` as an example -- can be any `std::optional`-like container
namespace pybind11 { namespace detail {
    template <typename T>
    struct type_caster<boost::optional<T>> : optional_caster<boost::optional<T>> {};
}}
```

Similarly, a specialization can be provided for custom variant types:

```
// `boost::variant` as an example -- can be any `std::variant`-like container
namespace pybind11 { namespace detail {
    template <typename... Ts>
    struct type_caster<boost::variant<Ts...>> : variant_caster<boost::variant<Ts...>> {};

    // Specifies the function used to visit the variant -- `apply_visitor` instead of
    `visit`
    template <>
    struct visit_helper<boost::variant> {
        template <typename... Args>
        static auto call(Args &&...args) -> decltype(boost::apply_visitor(args...)) {
            return boost::apply_visitor(args...);
        }
    };
}} // namespace pybind11::detail
```

The `visit_helper` specialization is not required if your `name::variant` provides a `name::visit()` function. For any other function name, the specialization must be included to tell pybind11 how to visit the variant.

Warning: When converting a `variant` type, pybind11 follows the same rules as when determining which function overload to call (Overload resolution order), and so the same caveats hold. In particular, the order in which the `variant`'s alternatives are listed is important, since pybind11 will try conversions in this order. This means that, for example, when converting `variant<int, bool>`, the `bool` variant will never be selected, as any Python `bool` is already an `int` and is convertible to a C++ `int`. Changing the order of alternatives (and using `variant<bool, int>`, in this example) provides a solution.

### 11.3.3 `opaque`

pybind11 `STL` `vector` `lists of hash maps of pairs of elementary and custom types`

`Python` `C++` `pass-by-reference`

```
void append_1(std::vector<int> &v) {
    v.push_back(1);
}
```

`Python`

```
>>> v = [5, 6]
>>> append_1(v)
>>> print(v)
[5, 6]
```



STL Python `def_readwrite` `def_readonly` STL

```
/* ... definition ... */

class MyClass {
    std::vector<int> contents;
};

/* ... binding code ... */

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

append

```
>>> m = MyClass()
>>> m.contents = [5, 6]
>>> print(m.contents)
[5, 6]
>>> m.contents.append(7)
>>> print(m.contents)
[5, 6]
```

pybind11 `PYBIND11_MAKE_OPAQUE(T)` `std::vector<int>`

```
PYBIND11_MAKE_OPAQUE(std::vector<int>);
```

`type_caster` `std::vector<int>` `opaque` `class_` Python

```
py::class_<std::vector<int>>(m, "IntVector")
    .def(py::init<>())
    .def("clear", &std::vector<int>::clear)
    .def("pop_back", &std::vector<int>::pop_back)
    .def("__len__", [](const std::vector<int> &v) { return v.size(); })
    .def("__iter__", [](std::vector<int> &v) {
        return py::make_iterator(v.begin(), v.end());
    }, py::keep_alive<0, 1>()) /* Keep vector alive while iterator is used */
    // ....
```

### 11.3.4 STL

STL Python pybind11 `pybind11/stl_bind.h` Python

```
// Don't forget this
#include <pybind11/stl_bind.h>

PYBIND11_MAKE_OPAQUE(std::vector<int>);
PYBIND11_MAKE_OPAQUE(std::map<std::string, double>);

// ...

// later in binding code:
py::bind_vector<std::vector<int>>(m, "VectorInt");
py::bind_map<std::map<std::string, double>>(m, "MapStringDouble");
```

STL pybind11 Module-local class bindings

py::module\_local py::module\_local strings Eigen STL module-local

std::vector<int>

py::module\_local() py\_module\_local(false) STL

```
py::bind_vector<std::vector<int>>(m, "VectorInt", py::module_local(false));
```

std::vector<int>

## 11.4

pybind11/functional.h

C++11 lambda std::function<> lambda lambda

int -> int

```
int func_arg(const std::function<int(int)> &f) {
    return f(10);
}
```

lambda f

```
std::function<int(int)> func_ret(const std::function<int(int)> &f) {
    return [f](int i) {
        return f(i) + 1;
    };
}
```

C++ python py::cpp\_function

```
py::cpp_function func_cpp() {
    return py::cpp_function([](int i) { return i+1; },
        py::arg("number"));
}
```

```
pybind11/functional.h
```

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

PYBIND11_MODULE(example, m) {
    m.def("func_arg", &func_arg);
    m.def("func_ret", &func_ret);
    m.def("func_cpp", &func_cpp);
}
```

# Python

```
$ python
>>> import example
>>> def square(i):
...     return i * i
...
>>> example.func_arg(square)
100L
>>> square_plus_1 = example.func_ret(square)
>>> square_plus_1(4)
17L
>>> plus_1 = func_cpp()
>>> plus_1(number=43)
44L
```

## Warning

```

C++Python
PythonC++

```

Python C++ Pybind11  
C++ C++ -> Python -> C++

## 11.5 Chrono

```
pybind11/chrono C++11 chrono Python datetime python floats
time.monotonic() time.perf_counter() time.process_time() durations
```

### 11.5.1 C++11

```
C++11C++11Python
```

```

std::chrono::system_clock  系统时钟
std::chrono::steady_clock  稳态时钟

std::chrono::high_resolution_clock  高精度系统时钟 或 稳态时钟
Python 的 datetime 和 timedelta
Python 的 datetime 和 timedelta

```

## 11.5.2 时间

### C++ 与 Python

- `std::chrono::system_clock::time_point` → `datetime.datetime`
- `std::chrono::duration` → `datetime.timedelta`
- `std::chrono::[other_clocks]::time_point` → `datetime.timedelta`

### Python 与 C++

- `datetime.datetime` OR `datetime.date` OR `datetime.time` → `std::chrono::system_clock::time_point`
- `datetime.timedelta` → `std::chrono::duration`
- `datetime.timedelta` → `std::chrono::[other_clocks]::time_point`
- `float` → `std::chrono::duration`
- `float` → `std::chrono::[other_clocks]::time_point`

## 11.6 Eigen

Eigen 库

## 11.7 绑定

pybind11 是一个 Python C API 的绑定库，它允许 C++ 代码与 Python 代码进行交互。

The following snippets demonstrate how this works for a very simple `inty` type that should be convertible from Python types that provide a `__int__(self)` method.

```

struct inty { long long_value; };

void print(inty s) {
    std::cout << s.long_value << std::endl;
}

```

The following Python snippet demonstrates the intended usage from the Python side:

```

class A:
    def __int__(self):
        return 123

from example import print

print(A())

```

To register the necessary conversion routines, it is necessary to add an instantiation of the `pybind11::detail::type_caster<T>` template. Although this is an implementation detail, adding an instantiation of this type is explicitly allowed.

```

namespace pybind11 { namespace detail {
    template <T> struct type_caster<T> {
    public:
        /**
         * This macro establishes the name 'inty' in
         * function signatures and declares a local variable
         * 'value' of type inty
         */
        PYBIND11_TYPE_CASTER(inty, _("inty"));

        /**
         * Conversion part 1 (Python->C++): convert a PyObject into a inty
         * instance or return false upon failure. The second argument
         * indicates whether implicit conversions should be applied.
         */
        bool load(handle src, bool) {
            /* Extract PyObject from handle */
            PyObject *source = src.ptr();
            /* Try converting into a Python integer value */
            PyObject *tmp = PyNumber_Long(source);
            if (!tmp)
                return false;
            /* Now try to convert into a C++ int */
            value.long_value = PyLong_AsLong(tmp);
            Py_DECREF(tmp);
            /* Ensure return code was OK (to avoid out-of-range errors etc) */
            return !(value.long_value == -1 && !PyErr_Occurred());
        }

        /**
         * Conversion part 2 (C++ -> Python): convert an inty instance into
         * a Python object. The second and third arguments are used to
         * indicate the return value policy and parent object (for
         * ``return_value_policy::reference_internal``) and are generally
         * ignored by implicit casters.
         */
        static handle cast(inty src, return_value_policy /* policy */, handle /* parent */)
        {
            return PyLong_FromLong(src.long_value);
        }
    };
}} // namespace pybind11::detail

```

Note: A `type_caster<T>` defined with `PYBIND11_TYPE_CASTER(T, ...)` requires that `T` is default-constructible ( `value` is first default constructed and then `load()` assigns to it).

Warning: When using custom type casters, it's important to declare them consistently in every compilation unit of the Python extension module. Otherwise, undefined behavior can ensue.

## 12. Python C++

pybind11 Python C++ Python C API

### 12.1 Python

#### 12.1.1

Python C++ `handle`, `object`, `bool_`, `int_`, `float_`, `str`, `bytes`, `tuple`, `list`, `dict`, `slice`, `none`, `capsule`, `iterable`, `iterator`, `function`, `buffer`, `array`, `array_t`.

Warning: Be sure to review the [Gotchas](#) before using this heavily in your C++ API.

#### 12.1.2 C++ Python

`dict`

```
using namespace pybind11::literals; // to bring in the `_a` literal
py::dict d("spam"_a=py::none(), "eggs"_a=42);
```

`tuple` `py::make_tuple()`

```
py::tuple tup = py::make_tuple(42, py::none(), "spam");
```

Python

simple namespace

```
using namespace pybind11::literals; // to bring in the `_a` literal
py::object SimpleNamespace = py::module_::import("types").attr("SimpleNamespace");
py::object ns = SimpleNamespace("spam"_a=py::none(), "eggs"_a=42);
```

namespace `py::delattr()` `py::getattr()` `py::setattr()` Simple namespaces

#### 12.1.3

C++ Python `py::cast()`

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

□□□□□□□□□□□□

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

□□□□□□□□□□□□□□ `cast_error` □□□

### 12.1.4 □C++□□□Python□

□C++□□□□□Python□□□□Python□□□sys.path)□□□□□□□□□□□□□□

```
// Equivalent to "from decimal import Decimal"
py::object Decimal = py::module_::import("decimal").attr("Decimal");

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

### 12.1.5 □□Python□□

□□ `operator()` □□□□Python□□□□□□□□□□

```
// Construct a Python object of class Decimal
py::object pi = Decimal("3.14159");

// Use Python to make our directories
py::object os = py::module_::import("os");
py::object makedirs = os.attr("makedirs");
makedirs("/tmp/path/to/somewhere");
```

One can convert the result obtained from Python to a pure C++ version if a `py::class_` or type conversion is defined.

```
py::function f = <...>;
py::object result_py = f(1234, "hello", some_instance);
MyClass &result = result_py.cast<MyClass*>();
```

### 12.1.6 □□Python□□□□□

□□ `.attr` □□□□□□□Python□□□□

```
// Calculate e^π in decimal
py::object exp_pi = pi.attr("exp")();
py::print(py::str(exp_pi));
```



In the example above `pi.attr("exp")` is a *bound method*: it will always call the method for that same instance of the class. Alternately one can create an *unbound method* via the Python class (instead of instance) and pass the `self` object explicitly, followed by other arguments.

```
py::object decimal_exp = Decimal.attr("exp");

// Compute the e^n for n=0..4
for (int n = 0; n < 5; n++) {
    py::print(decimal_exp(Decimal(n)));
}
```

### 12.1.7 Python

Python

```
def f(number, say, to):
    ... # function code

f(1234, say="hello", to=some_instance) # keyword call in Python
```

C++

```
using namespace pybind11::literals; // to bring in the `_a` literal
f(1234, "say"_a="hello", "to"_a=some_instance); // keyword call in C++
```

### 12.1.8 Python

Python

```
// * unpacking
py::tuple args = py::make_tuple(1234, "hello", some_instance);
f(*args);

// ** unpacking
py::dict kwargs = py::dict("number"_a=1234, "say"_a="hello", "to"_a=some_instance);
f(**kwargs);

// mixed keywords, * and ** unpacking
py::tuple args = py::make_tuple(1234);
py::dict kwargs = py::dict("to"_a=some_instance);
f(*args, "say"_a="hello", **kwargs);
```

Generalized unpacking according to [PEP448](#) is also supported:

```
py::dict kwargs1 = py::dict("number"_a=1234);
py::dict kwargs2 = py::dict("to"_a=some_instance);
f(**kwargs1, "say"_a="hello", **kwargs2);
```

## 12.1.9 `PyObject`

Python C++ Python `PyObject` dict `operator[]` `obj.attr()` `PyObject` C++

```
#include <pybind11/numpy.h>
using namespace pybind11::literals;

py::module_ os = py::module_::import("os");
py::module_ path = py::module_::import("os.path"); // like 'import os.path as path'
py::module_ np = py::module_::import("numpy"); // like 'import numpy as np'

py::str curdir_abs = path.attr("abspath")(path.attr("curdir"));
py::print(py::str("Current directory: ") + curdir_abs);
py::dict environ = os.attr("environ");
py::print(environ["HOME"]);
py::array_t<float> arr = np.attr("ones")(3, "dtype"_a="float32");
py::print(py::repr(arr + py::int_(1)));
```

`PyObject` `obj.cast()`

### Note

If a trivial conversion via move constructor is not possible, both implicit and explicit casting (calling `obj.cast()`) will attempt a “rich” conversion. For instance, `py::list env = os.attr("environ");` will succeed and is equivalent to the Python code `env = list(os.environ)` that produces a list of the dict keys.

## 12.1.10 `PyObject`

Python `py::error_already_set` C++ Python

## 12.1.11 Gotchas

### Default-Constructed Wrappers

Python `py::none` `PyObject*` `static_cast<bool>` `(my_wrapper)`

### Assigning `py::none()` to wrappers

C++ `py::str` `py::dict` `py::none` `None` Python `py::str(py::none)` `None`

## 12.2 NumPy

### 12.2.1 实现buffer protocol

Python实现buffer protocol  
Matrix

```
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;
};
```

Matrix实现buffer protocol  
Matrices NumPy arrays python  
`np.array(matrix_instance, copy = False)`

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

py::class\_ 实现 py::buffer\_protocol() 实现 def\_buffer() 实现  
matrix 实现 py::buffer\_info 实现 py::buffer\_info 实现 Python 实现

```
struct buffer_info {
    void *ptr;
    py::ssize_t itemsize;
    std::string format;
    py::ssize_t ndim;
    std::vector<py::ssize_t> shape;
    std::vector<py::ssize_t> strides;
};
```

Python buffer C++ 实现 py::buffer 实现 buffer 实现  
Eigen 实现 buffer 实现 NumPy  
matrix

```

/* Bind MatrixXd (or some other Eigen type) to Python */
typedef Eigen::MatrixXd Matrix;

typedef Matrix::Scalar Scalar;
constexpr bool rowMajor = Matrix::Flags & Eigen::RowMajorBit;

py::class_<Matrix>(m, "Matrix", py::buffer_protocol())
    .def(py::init([](py::buffer b) {
        typedef Eigen::Stride<Eigen::Dynamic, Eigen::Dynamic> Strides;

        /* Request a buffer descriptor from Python */
        py::buffer_info info = b.request();

        /* Some sanity checks ... */
        if (info.format != py::format_descriptor<Scalar>::format())
            throw std::runtime_error("Incompatible format: expected a double array!");

        if (info.ndim != 2)
            throw std::runtime_error("Incompatible buffer dimension!");

        auto strides = Strides(
            info.strides[rowMajor ? 0 : 1] / (py::ssize_t)sizeof(Scalar),
            info.strides[rowMajor ? 1 : 0] / (py::ssize_t)sizeof(Scalar));

        auto map = Eigen::Map<Matrix, 0, Strides>(
            static_cast<Scalar *>(info.ptr), info.shape[0], info.shape[1], strides);

        return Matrix(map);
    }));

```

~~~~~Eigen~~~~~ def\_buffer() ~~~~~

```

.def_buffer([](Matrix &m) -> py::buffer_info {
    return py::buffer_info(
        m.data(), /* Pointer to buffer */
        sizeof(Scalar), /* Size of one scalar */
        py::format_descriptor<Scalar>::format(), /* Python struct-style format descriptor */
        2, /* Number of dimensions */
        { m.rows(), m.cols() }, /* Buffer dimensions */
        { sizeof(Scalar) * (rowMajor ? m.cols() : 1),
          sizeof(Scalar) * (rowMajor ? 1 : m.rows()) }, /* Strides (in bytes) for each index */
    );
});

```

~~~~~Eigen~~~~~(~~~~~)~~~~~Eigen~~~~~

### 12.2.2 Arrays

~~~~~ py::buffer ~~~~ py::array ~~~~~NumPy array~~~~~Python~~~~~

~~~~~NumPy array~~~~~ py::array\_t<T> ~~~~~  
NumPy array

```
void f(py::array_t<double> array);
```

NumPy array `pybind11/numpy.h` NumPy 1.7.0

NumPy array C Fortran `py::array::c_style` `py::array::f_style`

```
void f(py::array_t<double, py::array::c_style | py::array::forcecast> array);
```

`py::array::forcecast`

arrays NumPy API

- `.dtype()`
- `.strides()` strides
- `.squeeze()`
- `.view(dtype)` dtype
- `.reshape({i, j, ...})` shape `.resize({})`
- `.index_at(i, j, ...)`

(0000)

### 12.2.3

`py::array_t` `PYBIND11_NUMPY_DTYPE`

```
struct A {
    int x;
    double y;
};

struct B {
    int z;
    A a;
};

// ...
PYBIND11_MODULE(test, m) {
    // ...

    PYBIND11_NUMPY_DTYPE(A, x, y);
    PYBIND11_NUMPY_DTYPE(B, z, a);
    /* now both A and B can be used as template arguments to py::array_t */
}
```

`std::complex` arrays C++ `std::array` plain

## 12.2.4 POD

NumPy arrays are C-contiguous, which means they can be passed to C++ code via `pybind11`.

```
double my_func(int x, float y, double z);
```

Then, in `pybind11/numpy.h`:

```
m.def("vectorized_func", py::vectorize(my_func));
```

The `numpy.vectorize()` function creates a new function that takes NumPy arrays as input and returns a NumPy array. The output array has the same dtype as the input arrays.

```
x = np.array([[1, 3], [5, 7]])
y = np.array([[2, 4], [6, 8]])
z = 3
result = vectorized_func(x, y, z)
```

The `z` argument is a scalar, so it is converted to a NumPy array with dtype `numpy.dtype.int64`. The `x` and `y` arguments are arrays with dtype `numpy.dtype.float32`.

### Note

NumPy arrays are C-contiguous, which means they can be passed to C++ code via `pybind11`.

The code is somewhat contrived, since it could have been done more simply using `vectorize`.

```

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

namespace py = pybind11;

py::array_t<double> add_arrays(py::array_t<double> input1, py::array_t<double> input2) {
    py::buffer_info buf1 = input1.request(), buf2 = input2.request();

    if (buf1.ndim != 1 || buf2.ndim != 1)
        throw std::runtime_error("Number of dimensions must be one");

    if (buf1.size != buf2.size)
        throw std::runtime_error("Input shapes must match");

    /* No pointer is passed, so NumPy will allocate the buffer */
    auto result = py::array_t<double>(buf1.size);

    py::buffer_info buf3 = result.request();

    double *ptr1 = static_cast<double *>(buf1.ptr);
    double *ptr2 = static_cast<double *>(buf2.ptr);
    double *ptr3 = static_cast<double *>(buf3.ptr);

    for (size_t idx = 0; idx < buf1.shape[0]; idx++)
        ptr3[idx] = ptr1[idx] + ptr2[idx];

    return result;
}

PYBIND11_MODULE(test, m) {
    m.def("add_arrays", &add_arrays, "Add two NumPy arrays");
}

```

### 12.2.5 内存管理

NumPy 数组的内存管理分为两种：不可变（immutable）和可变（mutable）。不可变数组的内存地址是固定的，而可变数组的内存地址可以改变。在 C++ 中，我们使用 `array_t<T>` 来定义数组，其中 `T` 是数据类型。对于不可变数组，我们使用 `unchecked<N>`，对于可变数组，我们使用 `mutable_unchecked<N>`，其中 `N` 是数组的维度。

```

m.def("sum_3d", [](py::array_t<double> x) {
    auto r = x.unchecked<3>(); // x must have ndim = 3; can be non-writeable
    double sum = 0;
    for (py::ssize_t i = 0; i < r.shape(0); i++)
        for (py::ssize_t j = 0; j < r.shape(1); j++)
            for (py::ssize_t k = 0; k < r.shape(2); k++)
                sum += r(i, j, k);
    return sum;
});
m.def("increment_3d", [](py::array_t<double> x) {
    auto r = x.mutable_unchecked<3>(); // Will throw if ndim != 3 or flags.writeable is false
    for (py::ssize_t i = 0; i < r.shape(0); i++)
        for (py::ssize_t j = 0; j < r.shape(1); j++)
            for (py::ssize_t k = 0; k < r.shape(2); k++)
                r(i, j, k) += 1.0;
}, py::arg().noconvert());

```

```

array auto r = myarray.mutable_unchecked<float, 2>()

arr_t.unchecked() arr.unchecked<T>()

```

array shape, strides, writeable flag reshape, typically by limiting the scope of the returned instance.

The returned proxy object supports some of the same methods as `py::array` so that it can be used as a drop-in replacement for some existing, index-checked uses of `py::array`:

- `.ndim()` returns the number of dimensions
- `.data(1, 2, ...)` and `r.mutable_data(1, 2, ...)` returns a pointer to the `const T` or `T` data, respectively, at the given indices. The latter is only available to proxies obtained via `a.mutable_unchecked()`.
- `.itemsize()` returns the size of an item in bytes, i.e. `sizeof(T)`.
- `.ndim()` returns the number of dimensions.
- `.shape(n)` returns the size of dimension `n`
- `.size()` returns the total number of elements (i.e. the product of the shapes).
- `.nbytes()` returns the number of bytes used by the referenced elements (i.e. `itemsize()` times `size()`).

## 12.2.6

Python 3 provides a convenient `...` ellipsis notation that is often used to slice multidimensional arrays. For instance, the following snippet extracts the middle dimensions of a tensor with the first and last index set to zero. In Python 2, the syntactic sugar `...` is not available, but the singleton `Ellipsis` (of type `ellipsis`) can still be used directly.

```

a = ... # a NumPy array
b = a[0, ..., 0]

```

The function `py::ellipsis()` function can be used to perform the same operation on the C++ side:

```

py::array a = /* A NumPy array */;
py::array b = a[py::make_tuple(0, py::ellipsis(), 0)];

```

## 12.2.7

C/C++ buffer memoryview 2\*4 uint8\_t memoryview



```

const uint8_t buffer[] = {
    0, 1, 2, 3,
    4, 5, 6, 7
};
m.def("get_memoryview2d", []() {
    return py::memoryview::from_buffer(
        buffer,                                // buffer pointer
        { 2, 4 },                             // shape (rows, cols)
        { sizeof(uint8_t) * 4, sizeof(uint8_t) } // strides in bytes
    );
})

```

memoryview Python buffer C++ buffer memoryview

memoryview::from\_memory

```

m.def("get_memoryview1d", []() {
    return py::memoryview::from_memory(
        buffer,            // buffer pointer
        sizeof(uint8_t) * 8 // buffer size
    );
})

```

Note: `memoryview::from_memory` is not available in Python 2.

## 12.3

### 12.3.1 C++ Python print

C++ `std::out` Python `print` `pybind11` `py::print` Python `sys.stdout`

Python `print` `sep`, `end`, `file`, `flush`

```

py::print(1, 2.0, "three"); // 1 2.0 three
py::print(1, 2.0, "three", "sep"_a="-"); // 1-2.0-three

auto args = py::make_tuple("unpacked", true);
py::print("->", *args, "end"_a="<-"); // -> unpacked True <-

```

### 12.3.2 ostream

C++ `std::cout` `std::cerr` Python `sys.stdout` `sys.stderr` `py::print` Python streams

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

...

// Add a scoped redirect for your noisy code
m.def("noisy_func", []() {
    py::scoped_ostream_redirect stream(
        std::cout, // std::ostream&
        py::module_::import("sys").attr("stdout") // Python output
    );
    call_noisy_func();
});
```

## Warning

`pybind11/iostream.h` `ostream` ( ) `ostream`

`scoped_ostream_redirect` Jupyter notebook C++ Python `py::scoped_estream_redirect` `<scoped_estream_redirect>` `py::call_guard`

```
// Alternative: Call single function using call guard
m.def("noisy_func", &call_noisy_function,
    py::call_guard<py::scoped_ostream_redirect,
        py::scoped_estream_redirect>());
```

The redirection can also be done in Python with the addition of a context manager, using the `py::add_ostream_redirect()` `<add_ostream_redirect>` function:

```
py::add_ostream_redirect(m, "ostream_redirect");
```

The name in Python defaults to `ostream_redirect` if no name is passed. This creates the following context manager in Python:

```
with ostream_redirect(stdout=True, stderr=True):
    noisy_function()
```

It defaults to redirecting both streams, though you can use the keyword arguments to disable one of the streams if needed.

## 12.3.3 Python

pybind11 provides the `eval`, `exec` and `eval_file` functions to evaluate Python expressions and statements. The following example illustrates how they can be used.

```
// At beginning of file
#include <pybind11/eval.h>

...

// Evaluate in scope of main module
py::object scope = py::module_::import("__main__").attr("__dict__");

// Evaluate an isolated expression
int result = py::eval("my_variable + 10", scope).cast<int>();

// Evaluate a sequence of statements
py::exec(
    "print('Hello')\n"
    "print('world!');",
    scope);

// Evaluate the statements in an separate Python file on disk
py::eval_file("script.py", scope);
```

C++11 raw string literals are also supported and quite handy for this purpose. The only requirement is that the first statement must be on a new line following the raw string delimiter `R"` (`,` ensuring all lines have common leading indent:

```
py::exec(R"(
    x = get_answer()
    if x == 42:
        print('Hello World!')
    else:
        print('Bye!')
)", scope);
```

## 13. Python

pybind11 C++ Python Python C++ pybind11

### 13.1

Cmake `pybind11::embed`

```
cmake_minimum_required(VERSION 3.4)
project(example)

find_package(pybind11 REQUIRED) # or `add_subdirectory(pybind11)`

add_executable(example main.cpp)
target_link_libraries(example PRIVATE pybind11::embed)
```

`main.cpp`

```
#include <pybind11/embed.h> // everything needed for embedding
namespace py = pybind11;

int main() {
    py::scoped_interpreter guard{}; // start the interpreter and keep it alive

    py::print("Hello, World!"); // use the Python API
}
```

Python API pybind11 Python guard `scoped_interpreter` guard Python

### 13.2 Python

12.3.3 `eval` `exec` `eval_file` Python Python

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

int main() {
    py::scoped_interpreter guard{};

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

pybind11 API

```
#include <pybind11/embed.h>
namespace py = pybind11;
using namespace py::literals;

int main() {
    py::scoped_interpreter guard{};

    auto kwargs = py::dict("name"_a="World", "number"_a=42);
    auto message = "Hello, {name}! The answer is {number}"_s.format(**kwargs);
    py::print(message);
}
```

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

namespace py = pybind11;
using namespace py::literals;

int main() {
    py::scoped_interpreter guard{};

    auto locals = py::dict("name"_a="World", "number"_a=42);
    py::exec(R"(
        message = "Hello, {name}! The answer is {number}".format(**locals())
    )", py::globals(), locals);

    auto message = locals["message"].cast<std::string>();
    std::cout << message;
}
```

## 12.3

`module_::import()` Python

```
py::module_ sys = py::module_::import("sys");
py::print(sys.attr("path"));
```

sys.path Python

```
"""calc.py located in the working directory"""
```

```
def add(i, j):
    return i + j
```

```
py::module_ calc = py::module_::import("calc");
py::object result = calc.attr("add")(1, 2);
int n = result.cast<int>();
assert(n == 3);
```

module\_::reload() Python

## 12.4

PYBIND11\_EMBEDDED\_MODULE Python

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

PYBIND11_EMBEDDED_MODULE(fast_calc, m) {
    // `m` is a `py::module_` which is used to bind functions and classes
    m.def("add", [](int i, int j) {
        return i + j;
    });
}

int main() {
    py::scoped_interpreter guard{};

    auto fast_calc = py::module_::import("fast_calc");
    auto result = fast_calc.attr("add")(1, 2).cast<int>();
    assert(result == 3);
}
```

Unlike extension modules where only a single binary module can be created, on the embedded side an unlimited number of modules can be added using multiple `PYBIND11_EMBEDDED_MODULE` definitions (as long as they have unique names).

These modules are added to Python's list of builtins, so they can also be imported in pure Python files loaded by the interpreter. Everything interacts naturally:

```

"""py_module.py located in the working directory"""
import cpp_module

a = cpp_module.a
b = a + 1
#include <pybind11/embed.h>
namespace py = pybind11;

PYBIND11_EMBEDDED_MODULE(cpp_module, m) {
    m.attr("a") = 1;
}

int main() {
    py::scoped_interpreter guard{};

    auto py_module = py::module_::import("py_module");

    auto locals = py::dict("fmt"_a="{} + {} = {}", **py_module.attr("__dict__"));
    assert(locals["a"].cast<int>() == 1);
    assert(locals["b"].cast<int>() == 2);

    py::exec(R"(
        c = a + b
        message = fmt.format(a, b, c)
    )", py::globals(), locals);

    assert(locals["c"].cast<int>() == 3);
    assert(locals["message"].cast<std::string>() == "1 + 2 = 3");
}

```

## 12.5 初始化

`scoped_interpreter` 初始化 Python 解释器。调用 `initialize_interpreter` / `finalize_interpreter` 来管理解释器的生命周期。

pybind11 提供了 `scoped_interpreter` 来管理 Python 解释器的生命周期。它封装了 CPython 的 `Py_Initialize` 和 `Py_Finalize` 函数。

### Warning

Creating two concurrent `scoped_interpreter` guards is a fatal error. So is calling `initialize_interpreter` for a second time after the interpreter has already been initialized.

Do not use the raw CPython API functions `Py_Initialize` and `Py_Finalize` as these do not properly handle the lifetime of pybind11's internal data.

14. ☐ ☐

## 14.1 練習問題

```
pybind11::PYBIND11_DECLARE_HOLDER_TYPE(, PYBIND11_OVERRIDE_*  
(  
    ))
```

```
PYBIND11_OVERRIDE(MyReturnType<T1, T2>, Class<T3, T4>, func)
```

```
00000000050000000000000300000000000000000000 PYBIND11_TYPE 0000
```

```
// Version 1: using a type alias
using ReturnType = MyReturnType<T1, T2>;
using ClassType = Class<T3, T4>;
PYBIND11_OVERRIDE(ReturnType, ClassType, func);

// Version 2: using the PYBIND11_TYPE macro:
PYBIND11_OVERRIDE(PYBIND11_TYPE(MyReturnType<T1, T2>),
                  PYBIND11_TYPE(Class<T3, T4>), func)
```

PYBIND11\_MAKE\_OPAQUE ██████████

## 14.2 PythonのGIL

```
Python C++ GIL gil_scoped_release gil_scoped_acquire GIL
C++ Python
```



```

class PyAnimal : public Animal {
public:
    /* Inherit the constructors */
    using Animal::Animal;

    /* Trampoline (need one for each virtual function) */
    std::string go(int n_times) {
        /* Acquire GIL before calling Python code */
        py::gil_scoped_acquire acquire;

        PYBIND11_OVERRIDE_PURE(
            std::string, /* Return type */
            Animal,      /* Parent class */
            go,           /* Name of function */
            n_times       /* Argument(s) */
        );
    }
};

PYBIND11_MODULE(example, m) {
    py::class_<Animal, PyAnimal> animal(m, "Animal");
    animal
        .def(py::init<>())
        .def("go", &Animal::go);

    py::class_<Dog>(m, "Dog", animal)
        .def(py::init<>());

    m.def("call_go", [](Animal *animal) -> std::string {
        /* Release GIL before calling into (potentially long-running) C++ code */
        py::gil_scoped_release release;
        return call_go(animal);
    });
}

```

~~~~~ call\_guard ~~~~~ call\_go ~~~~~

```
m.def("call_go", &call_go, py::call_guard<py::gil_scoped_release>());
```

14.3 ~~~~~

~~~~~

```

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

py::class_<Dog>(m, "Dog", pet /* <- specify parent */)
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);

```

~ Pet ~~~~~ basic ~~~~~ Dog ~~~~~ class\_<Dog> ~~~~~ Pet ~~~~~ Pet ~~~~~  
 ~~~~~ Pet ~~~~~ Dog ~~~~~

```

py::object pet = (py::object) py::module_::import("basic").attr("Pet");

py::class_<Dog>(m, "Dog", pet)
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);

```

class_ pybind11 Python import basic

```

py::module_::import("basic");

py::class_<Dog, Pet>(m, "Dog")
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);

```

pybind11 GCC/Clang -fvisibility=hidden

```

class PYBIND11_EXPORT Dog : public Animal {
    ...
};

```

C++ capsule pybind11

```

auto data = reinterpret_cast<MyData *>(py::get_shared_data("mydata"));
if (!data)
    data = static_cast<MyData *>(py::set_shared_data("mydata", new MyData(42)));

```

MyData

14.4

pybind11 Python capsules

```

auto cleanup_callback = []() {
    // perform cleanup here -- this function is called with the GIL held
};

m.add_object("_cleanup", py::capsule(cleanup_callback));

```

cleanup

capsule

```

auto cleanup_callback = []() { /* ... */ };
m.attr("BaseClass").attr("_cleanup") = py::capsule(cleanup_callback);

```

Python `_cleanup` API Python

`cleanup`

```
// Register a callback function that is invoked when the BaseClass object is collected
py::cpp_function cleanup_callback(
    [](py::handle weakref) {
        // perform cleanup here -- this function is called with the GIL held

        weakref.dec_ref(); // release weak reference
    }
);

// Create a weak reference with a cleanup callback and initially leak it
(void) py::weakref(m.attr("BaseClass"), cleanup_callback).release();
```

15.

15.1 “ImportError: dynamic module does not define init function”

1. `PYBIND11_MODULE`
- 2.

15.2 “Symbol not found: __Py_ZeroStruct __PyInstanceMethod_Type”

15.1

15.3 “SystemError: dynamic module not initialized properly”

15.1

15.4 Python

15.1

15.5

`C++`

```
void increment(int &i)
{
    i++;
}
void increment_ptr(int *i)
{
    (*i)++;
}
```

`Python`

```
def increment(i):
    i += 1 # nope..
```

`pybind11` `increment` `increment_ptr` Python `lambda` `lambda`

```
int foo(int &i)
{
    i++;
    return 123;
}
```

~~~~:

```
m.def("foo",
      [](int i) {
          int rv = foo(i);
          return std::make_tuple(rv, i);
      });
```

## 15.6 ~~~~?

~~~~ `example.cpp`

```
void init_ex1(py::module_ &);
void init_ex2(py::module_ &);
/* ... */
PYBIND11_MODULE(example, m) {
    init_ex1(m);
    init_ex2(m);
    /* ... */
}
```

`ex1.cpp`:

```
void init_ex1(py::module_ &m) {
    m.def("add",
          [](int a, int b) {
              return a + b;
          });
}
```

`ex2.cpp`:

```
void init_ex2(py::module_ &m) {
    m.def("sub",
          [](int a, int b) {
              return a - b;
          });
}
```

python ~~~

```
import example
```

```
example.add(1, 2) # 3
```

```
example.sub(1, 1) # 0
```

```
init_ex
```

```
1.
```

```
2.
```

```
3.
```

15.7 “recursive template instantiation exceeded maximum depth of 256”

```
GCC/Clang -ftemplate-depth=1024 C++14
```

15.8 “‘SomeClass’ declared with greater visibility than the type of its field ‘SomeClass::member’ [-Wattributes]”

```
-fvisibility pybind11 py::object py::list pybind -fvisibility=hidden -fvisibility=hidden pybind pybind Python POSIX RTLD_LOCAL dlopen Python -fvisibility=hidden
```

15.9

```
pybind11
```

```
__ZN8pybind112cpp_functionC1Iv8Example2JRNSt3__16vectorINS3_12basic_stringIwNS3_11char_traitsIwEENS3_9allocatorIwEEEEENS8_ISA_EEEEEJNS_4nameENS_7siblingENS_9is_methodEA28_cEEEMT0_FT_DpT1_EDpRKT2_
```

```
pybind11::cpp_function::cpp_function<void, Example2, std::__1::vector<std::__1::basic_string<wchar_t, std::__1::char_traits<wchar_t>, std::__1::allocator<wchar_t> >, std::__1::allocator<std::__1::basic_string<wchar_t, std::__1::char_traits<wchar_t>, std::__1::allocator<wchar_t> > >>>, pybind11::name, pybind11::sibling, pybind11::is_method, char [28]>(void (Example2::*)(std::__1::vector<std::__1::basic_string<wchar_t, std::__1::char_traits<wchar_t>, std::__1::allocator<wchar_t> >, std::__1::allocator<std::__1::basic_string<wchar_t, std::__1::char_traits<wchar_t>, std::__1::allocator<wchar_t> > >>>), pybind11::name const&, pybind11::sibling const&, pybind11::is_method const&, char const (&) [28])
```

196 111 ———
 -fvisibility=hidden
 Visual Studio -
 fvisibility=hidden pybind
 GCC Clang

15.10 Windows Visual Studio 2008

Python Windows C++11 Visual Studio
 Visual Studio 2015 Visual Studio 2008 Python
 DLL C malloc() free() ABI
 pybind11

15.11 Ctrl-C

Ctrl-C Python GIL
 PyErr_CheckSignals() Python
 py::error_already_set KeyboardInterrupt

```
PYBIND11_MODULE(example, m){
    m.def("long_running_func",
        [](){
            for (;;)
            {
                if (PyErr_CheckSignals() != 0)
                    throw py::error_already_set();
                // Long running iteration
            }
        });
}
```

15.12 CMake Python

CMake Python
 CMakeCache.txt -DPYTHON_EXECUTABLE=\$(which python) CMake \$(which python) python -DPYBIND11_FINDPYTHON=ON CMake FindPython

```
pybind11 CMake 3.12+3.15+ 3.18.2+ pybind11 CMakeLists.txt
```

15.13 CMake pybind11 Python

CMake Python find_package(PythonInterp) find_package(PythonLibs) pybind11 pybind11 pybind11 Python CMake Python CMake Python pybind11 Python2.7 3.x CMake

```
find_package(PythonInterp)
find_package(PythonLibs)
find_package(pybind11)
```

Python2.7 pybind11

```
find_package(pybind11)
find_package(PythonInterp)
find_package(PythonLibs)
```

pybind11 Python3.x find_package(PythonLibs)

1. CMake find_package(PythonInterp) find_package(PythonLibs) pybind11 Python pybind11 CMake
2. PYBIND11_FINDPYTHON True CMake find_package(Python COMPONENTS Interpreter Development) 3.12+3.15+3.18.2+ Pybind11 CMake FindPython Python
3. PYBIND11_NOPYTHON TRUE Pybind11 Python pybind11_add_module scikit-build Python

15.14 BibTeX

BibTeX pybind11

```
@misc{pybind11,
author = {Wenzel Jakob and Jason Rhinelander and Dean Moldovan},
year = {2017},
note = {https://github.com/pybind/pybind11},
title = {pybind11 -- Seamless operability between C++11 and Python} }
```


16. []

16.1 c/c++ [] [] [] []

- c [] [] []

```
char ca;  
unsigned char uca;
```

- c++ [] [] [] [] []

```
vector  
stl
```

pybind11 入门

1. 环境

1.1 安装

在python3-dev包中安装pybind11，并包含pybind11头文件，python3库，并编译cmake

```
set(PYTHON_TARGET_VER 3.6)
find_package(PythonInterp ${PYTHON_TARGET_VER} EXACT)
find_package(PythonLibs ${PYTHON_TARGET_VER} EXACT REQUIRED)

include_directories(pybind11_include_path)
include_directories(${PYTHON_INCLUDE_DIRS})
```

1.2 使用

```
#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 which adds two numbers");
}
```

在PYBIND11_MODULE中，Python中import module，并调用module::def()函数

1.2.1 参数

在py::arg中，Python中调用module::def()函数

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

在py::arg中

```
// 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);
```

Python

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

1.2.2

pybind11 `arg`

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

```
// regular notation
m.def("add1", &add, py::arg("i") = 1, py::arg("j") = 2);
// shorthand
m.def("add2", &add, "i"_a=1, "j"_a=2);
```

1.2.3

Python Python

```
m.def("add", static_cast<int*>(int, int)>(&add), "A function which adds two int numbers");
m.def("add", static_cast<double*>(double, double)>(&add), "A function which adds two
double numbers");
```

C++14

```
m.def("add", py::overload_cast<int, int>(&add), "A function which adds two int numbers");
m.def("add", py::overload_cast<double, double>(&add), "A function which adds two double
numbers");
```

`py::overload_cast` `(void (Pet::*))` `const`
`py::const`

1.3 Python

Python `attr` Python C++ `attributes` `py::cast`

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

```
Python
```python
>>> import example
>>> example.the_answer
42
>>> example.what
'World'
```

## 1.4 C++

C++ `Pet`

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

 std::string name;
};
```

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

PYBIND11_MODULE(example, m) {
 py::class_<Pet>(m, "Pet")
 .def(py::init<const std::string &>())
 .def("setName", &Pet::setName)
 .def("getName", &Pet::getName)
 .def("__repr__",
 [](const Pet &a) {
 return "<example.Pet named '" + a.name + "'>";
 });
}
```

`class_` C++ class `struct` `init()`

```

print(p) # __repr__ Pet

```

Python

```

>>> import example
>>> p = example.Pet("Molly")
>>> print(p)
<example.Pet named 'Molly'>
>>> p.getName()
u'Molly'
>>> p.setName("Charly")
>>> p.getName()
u'Charly'

```

```

class::def_static

```

### 1.4.1

```

class::def_readwrite class::def_readonly

```

```

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

```

Python

```

>>> p = example.Pet("Molly")
>>> p.name
u'Molly'
>>> p.name = "Charly"
>>> p.name
u'Charly'

```

```

Pet::name setter/getters

```

```

class Pet {
public:
 Pet(const std::string &name) : name(name) { }
 void setName(const std::string &name_) { name = name_; }
 const std::string &getName() const { return name; }
private:
 std::string name;
};

```

```

class::def_property() (class::def_property_readonly()) setter
getter

```

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

read nullptr

```
class_::def_readwrite_static(), class_::def_readonly_static()
class_::def_property_static(), class_::def_property_readonly_static()
```

## 1.4.2

Python

```
>>> class Pet:
... name = "Molly"
...
>>> p = Pet()
>>> p.name = "Charly" # overwrite existing
>>> p.age = 2 # dynamically add a new attribute
```

C++ class\_::def\_readwrite class\_::def\_property

```
>>> p = example.Pet()
>>> p.name = "Charly" # OK, attribute defined in C++
>>> p.age = 2 # fail
AttributeError: 'Pet' object has no attribute 'age'
```

C++ py::class\_ py::dynamic\_attr

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

```
>>> p = example.Pet()
>>> p.name = "Charly" # OK, overwrite value in C++
>>> p.age = 2 # OK, dynamically add a new attribute
>>> p.__dict__ # just like a native Python class
{'age': 2}
```

\_\_dict\_\_ Python pybind11 Python

### 1.4.3 结构体

结构体与C++中的结构体类似

```
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;
};

// method 1
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");

// method 2
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");
```

### 1.5 枚举

枚举与C++中的枚举类似

```
enum Flags {
 Read = 4,
 Write = 2,
 Execute = 1
};

py::enum_<Flags>(m, "Flags", py::arithmetic())
 .value("Read", Flags::Read)
 .value("Write", Flags::Write)
 .value("Execute", Flags::Execute)
 .export_values();
```

`enum_::export_values()` 与C++11中的`enum`类似

枚举成员`__members__` 是一个`name` 与`unicode` 的字典，`str(enum)` 返回枚举成员

### 1.6 \*args 与 \*\*kwargs

Python 中的`*args` 与`**kwargs`

```
def generic(*args, **kwargs):
 ... # do something with args and kwargs
```

pybind11

```
void generic(py::args args, const py::kwargs& kwargs) {
 /// .. do something with args
 if (kwargs)
 /// .. do something with kwargs
}

/// Binding code
m.def("generic", &generic);
```

`py::args` `py::tuple` `py::kwargs` `py::dict`

## 2.

### 2.1

PythonC++no-trivialPythonC++pybind11

`model::def()` `class_def()` `return_value_policy::automatic`

```
/* Function declaration */
Data *get_data() { return _data; /* (pointer to a static data structure) */ }
...

/* Binding code */
m.def("get_data", &get_data); // <-- KABOOM, will cause crash when called from Python
```

Python`get_data()`C++Python`return_value_policy::automatic`pybind11`_data`

Python`_data`Pythonpybind11C++operator delete()

`return_value_policy::reference`

```
m.def("get_data", &get_data, py::return_value_policy::reference);
```

pybind11



| return_value_policy                                   | 説明                                                                                                                         |
|-------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <code>return_value_policy::take_ownership</code>      | Pythonでdeleteされたオブジェクトは、C++でdeleteされたオブジェクトと同じように扱われる。                                                                     |
| <code>return_value_policy::copy</code>                | Pythonでdeleteされたオブジェクトは、C++でdeleteされたオブジェクトと同じように扱われる。                                                                     |
| <code>return_value_policy::move</code>                | std::moveで移動されたオブジェクトは、Pythonでdeleteされたオブジェクトと同じように扱われる。                                                                   |
| <code>return_value_policy::reference</code>           | C++でdeleteされたオブジェクトは、Pythonでdeleteされたオブジェクトと同じように扱われる。                                                                     |
| <code>return_value_policy::reference_internal</code>  | thisがselfを指しているreferenceのオブジェクトは、keep_alive<0, 1>でPythonでdeleteされたオブジェクトと同じように扱われる。def_propertyでdef_readwriteでgetterを定義する。 |
| <code>return_value_policy::automatic</code>           | return_value_policy::take_ownership、return_value_policy::copy、py::class_で定義されたオブジェクトは、自動的にdeleteされる。                       |
| <code>return_value_policy::automatic_reference</code> | return_value_policy::reference、pybind11/stl.hのcastersで定義されたオブジェクトは、自動的にdeleteされる。                                          |

以下は、MyClassの定義例。

```
class_<MyClass>(m, "MyClass")
 .def_property("data", &MyClass::getData, &MyClass::setData,
 py::return_value_policy::copy);
```

以下は、getter/setterの定義例。

```
class_<MyClass>(m, "MyClass")
 .def_property("data",
 py::cpp_function(&MyClass::getData, py::return_value_policy::copy),
 py::cpp_function(&MyClass::setData)
);
```

以下は、free関数の定義例。

以下

1. pybind11 Python
2. Python
3. C++ Python

## 2.2

### keep alive

C++ keep\_alive<Nurse, Patient> Nurse Patient 0 1 1 this 2 Nurse None

nurse pybind11 nurse pybind11

“Could not cativate keep\_alive!” review

list append

```
py::class_<List>(m, "List").def("append", &List::append, py::keep_alive<1, 2>());
```

1 this 0 void

```
py::class_<Nurse>(m, "Nurse").def(py::init<Patient &>(), py::keep_alive<1, 2>());
```

Note: keep\_alive Boost.Python with\_custodian\_and\_ward with\_custodian\_and\_ward\_postcall

### Call guard

call\_guard<T> scope guard

```
m.def("foo", foo, py::call_guard<T>());
```

```
m.def("foo", [](args...) {
 T scope_guard;
 return foo(args...); // forwarded arguments
});
```

~~~~~T~~~~~ `gil_scoped_release` ~~~~~

`call_guard` ~~~~~ `call_guard<T1, T2, T3 ...>` ~~~~~

See also: `test/test_call_policies.cpp` ~~~~~ `keep_alive` ~ `call_guard` ~~~~

## 2.3 Keyword-only

Python3 ~~~~keyword-only~~~~~ \* ~~~~~

```
def f(a, *, b): # a can be positional or via keyword; b must be via keyword
 pass

f(a=1, b=2) # good
f(b=2, a=1) # good
f(1, b=2) # good
f(1, 2) # TypeError: f() takes 1 positional argument but 2 were given
```

pybind11 ~~~~ `py::kw_only` ~~~~~

```
m.def("f", [](int a, int b) { /* ... */ },
 py::arg("a"), py::kw_only(), py::arg("b"));
```

~~~~~ `py::args` ~~~~~

## 2.4 Positional-only

python3.8 ~~~~Positional-only~~~~~pybind11 ~~~~ `py::pos_only()` ~~~~~

```
m.def("f", [](int a, int b) { /* ... */ },
 py::arg("a"), py::pos_only(), py::arg("b"));
```

~~~~~ `a` ~~~~~keyword-only~~~~~

## 2.5 Non-converting

~~~~~

- ~~~~ `py::implicitly_convertible<A,B>()` ~~~~~

- `std::complex<float>` `std::complex<float>`
- Calling a function taking an Eigen matrix reference with a numpy array of the wrong type or of an incompatible data layout.

`py::arg` `.noconvert()`

```
m.def("floats_only", [](double f) { return 0.5 * f; }, py::arg("f").noconvert());
m.def("floats_preferred", [](double f) { return 0.5 * f; }, py::arg("f"));
```

`TypeError`

```
>>> floats_preferred(4)
2.0
>>> floats_only(4)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: floats_only(): incompatible function arguments. The following argument types are
supported:
 1. (f: float) -> float

Invoked with: 4
```

`_a` `py::arg().noconvert()`

### 3. `std::string`

#### 3.1 `std::string`

`std::string`

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

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

`pybind11` `class_` `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 /* <- specify C++ parent type */>(m, "Dog")
 .def(py::init<const std::string &>())
 .def("bark", &Dog::bark);

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

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

```
>>> p = example.Dog("Molly")
>>> p.name
u'Molly'
>>> p.bark()
u'woof!'
```

# Python

```
// 宠物商店
m.def("pet_store", []() { return std::unique_ptr<Pet>(new Dog("Molly")); });
```

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

```
pet_store Dog Python Pet C++
pybind11
```

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

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

// Same binding code
py::class_(m, "PolymorphicPet");
py::class_(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); });
```

```
>>> p = example.pet_store2()
>>> type(p)
PolymorphicDog # automatically downcast
>>> p.bark()
u'woof!'
```

pybind11 C++ Python

## 3.2 Python C++

Python C++

```
class Animal {
public:
 virtual ~Animal() { }
 virtual std::string go(int n_times) = 0;
};

class Dog : public Animal {
public:
 std::string go(int n_times) override {
 std::string result;
 for (int i=0; i<n_times; ++i)
 result += "woof! ";
 return result;
 }
};

std::string call_go(Animal *animal) {
 return animal->go(3);
}

PYBIND11_MODULE(example, m) {
 py::class_<Animal>(m, "Animal")
 .def("go", &Animal::go);

 py::class_<Dog, Animal>(m, "Dog")
 .def(py::init<>());

 m.def("call_go", &call_go);
}
```

Python Animal "No constructor defined!"

Animal

```

class PyAnimal : public Animal {
public:
 /* Inherit the constructors */
 using Animal::Animal;

 /* Trampoline (need one for each virtual function) */
 std::string go(int n_times) override {
 PYBIND11_OVERRIDE_PURE(
 std::string, /* Return type */
 Animal, /* Parent class */
 go, /* Name of function in C++ (must match Python name) */
 n_times /* Argument(s) */
);
 }
};

```

PYBIND11\_OVERRIDE\_PURE PYBIND11\_OVERRIDE  
 PYBIND11\_OVERRIDE\_PURE\_NAME PYBIND11\_OVERRIDE\_NAME CPython  
 \_\_str\_\_

```

std::string toString() override {
 PYBIND11_OVERRIDE_NAME(
 std::string, // Return type (ret_type)
 Animal, // Parent class (cname)
 "__str__", // Name of method in Python (name)
 toString, // Name of function in C++ (fn)
);
}

```

Animal

```

PYBIND11_MODULE(example, m) {
 py::class_<Animal, PyAnimal /* <--- trampoline */>(m, "Animal")
 .def(py::init<>())
 .def("go", &Animal::go);

 py::class_<Dog, Animal>(m, "Dog")
 .def(py::init<>());

 m.def("call_go", &call_go);
}

```

pybind11 class\_ PyAnimal Python Animal

```

py::class_<Animal, PyAnimal /* <--- trampoline */>(m, "Animal");
 .def(py::init<>())
 .def("go", &PyAnimal::go); /* <--- THIS IS WRONG, use &Animal::go */

```

Python Animal::go

```

from example import *
d = Dog()
call_go(d) # u'woof! woof! woof! '
class Cat(Animal):
 def go(self, n_times):
 return "meow! " * n_times

c = Cat()
call_go(c) # u'meow! meow! meow! '

```

Pythonの\_\_init\_\_メソッドはC++の\_\_init\_\_メソッドと異なり、pybind11 2.6.0ではTypeErrorを返す。

```

class Dachshund(Dog):
 def __init__(self, name):
 Dog.__init__(self) # Without this, a TypeError is raised.
 self.name = name

 def bark(self):
 return "yap!"

```

Pythonの\_\_init\_\_メソッドはC++の\_\_init\_\_メソッドと異なり、pybind11 2.6.0ではTypeErrorを返す。Python MRO C++の\_\_init\_\_メソッド

### 3.3 Python

Pythonの\_\_init\_\_メソッドはC++の\_\_init\_\_メソッドと異なり、pybind11 2.6.0ではTypeErrorを返す。

```

class Animal {
public:
 virtual std::string go(int n_times) = 0;
 virtual std::string name() { return "unknown"; }
};
class Dog : public Animal {
public:
 std::string go(int n_times) override {
 std::string result;
 for (int i=0; i<n_times; ++i)
 result += bark() + " ";
 return result;
 }
 virtual std::string bark() { return "woof!"; }
};

```

Pythonの\_\_init\_\_メソッドはC++の\_\_init\_\_メソッドと異なり、pybind11 2.6.0ではTypeErrorを返す。Python MRO C++の\_\_init\_\_メソッド



```

class PyAnimal : public Animal {
public:
 using Animal::Animal; // Inherit constructors
 std::string go(int n_times) override { PYBIND11_OVERRIDE_PURE(std::string, Animal, go,
n_times); }
 std::string name() override { PYBIND11_OVERRIDE(std::string, Animal, name,); }
};
class PyDog : public Dog {
public:
 using Dog::Dog; // Inherit constructors
 std::string go(int n_times) override { PYBIND11_OVERRIDE(std::string, Dog, go,
n_times); }
 std::string name() override { PYBIND11_OVERRIDE(std::string, Dog, name,); }
 std::string bark() override { PYBIND11_OVERRIDE(std::string, Dog, bark,); }
};

```

```

 name() bark()

```

```

pybind11

```

```

class Husky : public Dog {};
class PyHusky : public Husky {
public:
 using Husky::Husky; // Inherit constructors
 std::string go(int n_times) override { PYBIND11_OVERRIDE_PURE(std::string, Husky, go,
n_times); }
 std::string name() override { PYBIND11_OVERRIDE(std::string, Husky, name,); }
 std::string bark() override { PYBIND11_OVERRIDE(std::string, Husky, bark,); }
};

```

```


```

```

template <class AnimalBase = Animal> class PyAnimal : public AnimalBase {
public:
 using AnimalBase::AnimalBase; // Inherit constructors
 std::string go(int n_times) override { PYBIND11_OVERRIDE_PURE(std::string, AnimalBase,
go, n_times); }
 std::string name() override { PYBIND11_OVERRIDE(std::string, AnimalBase, name,); }
};
template <class DogBase = Dog> class PyDog : public PyAnimal<DogBase> {
public:
 using PyAnimal<DogBase>::PyAnimal; // Inherit constructors
 // Override PyAnimal's pure virtual go() with a non-pure one:
 std::string go(int n_times) override { PYBIND11_OVERRIDE(std::string, DogBase, go,
n_times); }
 std::string bark() override { PYBIND11_OVERRIDE(std::string, DogBase, bark,); }
};

```

```


```

```

pybind11

```

```
py::class_<Animal, PyAnimal<>> animal(m, "Animal");
py::class_<Dog, Animal, PyDog<>> dog(m, "Dog");
py::class_<Husky, Dog, PyDog<Husky>> husky(m, "Husky");
// ... add animal, dog, husky definitions
```

Husky

# Python

```
class ShihTzu(Dog):
 def bark(self):
 return "yip!"
```

### 3.4 实验结果

```

pybind11::unique_ptr<Pybind11::nodelete> C++

```

```
/* ... definition ... */

class MyClass {
private:
 ~MyClass() { }
};

/* ... binding code ... */

py::class_<MyClass, std::unique_ptr<MyClass, py::nodelete>>(m, "MyClass")
 .def(py::init<>())
```

### 3.5 实验

□□□□□A□B□□□□□A□□□□□□□B□

```
py::class_<A>(m, "A")
 /// ... members ...

py::class_(m, "B")
 .def(py::init<A>())
 /// ... members ...

m.def("func",
 [](const B &) { /* ... */ }
);
```

func(A)aPython func(B(a)) C++ func(a) ABB

BBBBBBAAAAAAAAAAAAAAAAAAAAAA PythonAAAAAAAAAAAA

```
py::implicitly_convertible<A, B>());
```

### 3.6 3.6

Vector2

```
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;
};
```

```
#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);
}
```

```
.def(py::self * float())
```

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

### 3.7 深拷贝

Python 的深拷贝和浅拷贝，在 `copy` 模块中实现。

Python3 的 `pickle` 模块中的 `__copy__` 和 `__deepcopy__` 方法，在 Python2.7 中，  
在 `pybind11` 的 `cPickle` 模块中实现。

以下是一个简单的例子：

```
py::class_<Copyable>(m, "Copyable")
 .def("__copy__", [](const Copyable &self) {
 return Copyable(self);
 })
 .def("__deepcopy__", [](const Copyable &self, py::dict) {
 return Copyable(self);
 }, "memo"_a);
```

Note: 在 Python 中，深拷贝和浅拷贝的区别在于，深拷贝会递归地复制对象中的子对象，而浅拷贝只会复制对象本身，不会复制子对象。

### 3.8 多态

pybind11 支持 C++ 的多态，在 `class_` 中实现。

```
py::class_<MyType, BaseType1, BaseType2, BaseType3>(m, "MyType")
 ...
```

在 C++ 中，多态是通过虚函数实现的，而在 Python 中，多态是通过鸭子类型实现的。

Python 的多态与 C++ 的多态不同，Python 的多态是基于鸭子类型的。

pybind11 支持 C++ 的多态，在 `pybind11` 模块中实现。在 `multiple_inheritance` 模块中，

```
py::class_<MyType, BaseType2>(m, "MyType", py::multiple_inheritance());
```

以下是一个简单的例子：

### 3.9 受保护的成员

Python 的 `protected` 成员，在 `pybind11` 模块中实现。

```

class A {
protected:
 int foo() const { return 42; }
};

py::class_<A>(m, "A")
 .def("foo", &A::foo); // error: 'foo' is a protected member of 'A'

```

Python protected member access

```

class A {
protected:
 int foo() const { return 42; }
};

class Publicist : public A { // helper type for exposing protected functions
public:
 using A::foo; // inherited with different access modifier
};

py::class_<A>(m, "A") // bind the primary class
 .def("foo", &Publicist::foo); // expose protected methods via the publicist

```

Publicist::foo A::foo Publicist public

Python protected publicist pattern trampoline

```

class A {
public:
 virtual ~A() = default;

protected:
 virtual int foo() const { return 42; }
};

class Trampoline : public A {
public:
 int foo() const override { PYBIND11_OVERRIDE(int, A, foo,); }
};

class Publicist : public A {
public:
 using A::foo;
};

py::class_<A, Trampoline>(m, "A") // <-- `Trampoline` here
 .def("foo", &Publicist::foo); // <-- `Publicist` here, not `Trampoline`!

```

### 3.10 final

C++11 findal py::is\_final Python C++ final

```
class IsFinal final {};
```

```
py::class_<IsFinal>(m, "IsFinal", py::is_final());
```

Python

```
class PyFinalChild(IsFinal):
 pass
```

```
TypeError: type 'IsFinal' is not an acceptable base type
```

## 4.

### 4.1 C++ Python

Python pybind11 C++ pybind11 C++ Python Python Python

pybind11 `std::exception` Python Python Python Python C API C++ pybind11 Python

Exception thrown by C++	Translated to Python exception type
<code>std::exception</code>	<code>RuntimeError</code>
<code>std::bad_alloc</code>	<code>MemoryError</code>
<code>std::domain_error</code>	<code>ValueError</code>
<code>std::invalid_argument</code>	<code>ValueError</code>
<code>std::length_error</code>	<code>ValueError</code>
<code>std::out_of_range</code>	<code>IndexError</code>
<code>std::range_error</code>	<code>ValueError</code>
<code>std::overflow_error</code>	<code>OverflowError</code>
<code>pybind11::stop_iteration</code>	<code>StopIteration</code> (used to implement custom iterators)
<code>pybind11::index_error</code>	<code>IndexError</code> (used to indicate out of bounds access in <code>__getitem__</code> , <code>__setitem__</code> , etc.)
<code>pybind11::key_error</code>	<code>KeyError</code> (used to indicate out of bounds access in <code>__getitem__</code> , <code>__setitem__</code> in dict-like objects, etc.)
<code>pybind11::value_error</code>	<code>ValueError</code> (used to indicate wrong value passed in <code>container.remove(...)</code> )
<code>pybind11::type_error</code>	<code>TypeError</code>
<code>pybind11::buffer_error</code>	<code>BufferError</code>
<code>pybind11::import_error</code>	<code>ImportError</code>

Exception thrown by C++	Translated to Python exception type
<code>pybind11::attribute_error</code>	<code>AttributeError</code>
Any other exception	<code>RuntimeError</code>

Python Python `pybind11::error_already_set`

Python `handle::call()` `cast_error`

## 4.2

pybind11 pybind11 class global  
C++ `what()` C++ Python

```
py::register_exception<CppExp>(module, "PyExp");
```

PyExp Python CppExp PyExp

```
py::register_local_exception<CppExp>(module, "PyExp");
```

handle

```
py::register_exception<CppExp>(module, "PyExp", PyExc_RuntimeError);
py::register_local_exception<CppExp>(module, "PyExp", PyExc_RuntimeError);
```

PyExp PyExp RuntimeError

Python Python Standard Exceptions `PyExc_Exception`

```
py::register_exception_translator(translator)
py::register_local_exception_translator(translator)
void(std::exception_ptr)
```

## 9.3 C++ Python

C++ Python Python Python pybind11 Python  
`pybind11::error_already_set` C++ Python `error_already_set`  
Python Python C++

Exception raised in Python	Thrown as C++ exception type
Any Python <code>Exception</code>	<code>pybind11::error_already_set</code>

```
try {
 // open("missing.txt", "r")
 auto file = py::module_::import("io").attr("open")("missing.txt", "r");
 auto text = file.attr("read")();
 file.attr("close")();
} catch (py::error_already_set &e) {
 if (e.matches(PyExc_FileNotFoundError)) {
 py::print("missing.txt not found");
 } else if (e.matches(PyExc_PermissionError)) {
 py::print("missing.txt found but not accessible");
 } else {
 throw;
 }
}
```

```

C++PythonPythonerror_already_set.

```

```
try {
 py::eval("raise ValueError('The Ring')");
} catch (py::value_error &boromir) {
 // Boromir never gets the ring
 assert(false);
} catch (py::error_already_set &frodo) {
 // Frodo gets the ring
 py::print("I will take the ring");
}

try {
 // py::value_error is a request for pybind11 to raise a Python exception
 throw py::value_error("The ball");
} catch (py::error_already_set &cat) {
 // cat won't catch the ball since
 // py::value_error is not a Python exception
 assert(false);
} catch (py::value_error &dog) {
 // dog will catch the ball
 py::print("Run Spot run");
 throw; // Throw it again (pybind11 will raise ValueError)
}
```

## 9.4 Python C API

```

pybind11 wrappers Python C API Python C API
pybind11

```

```
Python C API Python throw py::error_already_set(); pybind11
Python PyErr_SetString
```

```
PyErr_SetString(PyExc_TypeError, "C API type error demo");
throw py::error_already_set();

// But it would be easier to simply...
throw py::type_error("pybind11 wrapper type error");
```



PyErr\_Clear

PythonPython/pybind11

## 9.5 unraisable

PythonC++ noexcept(true) c++std::terminate()C++Python error\_already\_set error\_already\_set::discard\_as\_unraisable() Python

\_\_del\_\_ PythonPython unraisable Python 3.8+ system hook auditing event

noexcepttry-catch error\_already\_set pybind11Python Python pybind11C++noexcept Python discard\_as\_unraisable

```
void nonthrowing_func() noexcept(true) {
 try {
 // ...
 } catch (py::error_already_set &eas) {
 // Discard the Python error using Python APIs, using the C++ magic
 // variable __func__. Python already knows the type and value and of the
 // exception object.
 eas.discard_as_unraisable(__func__);
 } catch (const std::exception &e) {
 // Log and discard C++ exceptions.
 third_party::log(e);
 }
}
```

## 5.

## 6. python C++

## 7.

### 7.1

pybind11 PYBIND11\_DECLARE\_HOLDER\_TYPE() PYBIND11\_OVERRIDE\_\*

```
PYBIND11_OVERRIDE(MyReturnType<T1, T2>, Class<T3, T4>, func)
```

```
5PYBIND11_TYPE
```

```
// Version 1: using a type alias
using ReturnType = MyReturnType<T1, T2>;
using ClassType = Class<T3, T4>;
PYBIND11_OVERRIDE(ReturnType, ClassType, func);

// Version 2: using the PYBIND11_TYPE macro:
PYBIND11_OVERRIDE(PYBIND11_TYPE(MyReturnType<T1, T2>),
 PYBIND11_TYPE(Class<T3, T4>), func)
```

```
PYBIND11_MAKE_OPAQUE
```

## 7.2 GIL

Python C++ GIL `gil_scoped_release` `gil_scoped_acquire` GIL  
C++ Python

```

class PyAnimal : public Animal {
public:
 /* Inherit the constructors */
 using Animal::Animal;

 /* Trampoline (need one for each virtual function) */
 std::string go(int n_times) {
 /* Acquire GIL before calling Python code */
 py::gil_scoped_acquire acquire;

 PYBIND11_OVERRIDE_PURE(
 std::string, /* Return type */
 Animal, /* Parent class */
 go, /* Name of function */
 n_times /* Argument(s) */
);
 }
};

PYBIND11_MODULE(example, m) {
 py::class_<Animal, PyAnimal> animal(m, "Animal");
 animal
 .def(py::init<>())
 .def("go", &Animal::go);

 py::class_<Dog>(m, "Dog", animal)
 .def(py::init<>());

 m.def("call_go", [](Animal *animal) -> std::string {
 /* Release GIL before calling into (potentially long-running) C++ code */
 py::gil_scoped_release release;
 return call_go(animal);
 });
}

```

~~~~~ call\_guard ~~~~~ call\_go ~~~~~

```

m.def("call_go", &call_go, py::call_guard<py::gil_scoped_release>());

```

# 🐙 Mermaid 🐙

🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙 **Mermaid** 🐙🐙,🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙,🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙,🐙🐙🐙🐙🐙🐙,🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙,🐙🐙 **markdown** 🐙🐙🐙.

🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙:

- 🐙🐙🐙🐙🐙🐙🐙🐙 **Mermaid** 🐙🐙;
- 🐙🐙🐙🐙🐙🐙🐙🐙 **Mermaid** 🐙🐙;
- 🐙🐙 **Gitbook** 🐙🐙🐙🐙🐙🐙🐙🐙.

## 🐙🐙 Mermaid 🐙🐙

🐙🐙

```
- 🐙🐙🐙🐙
- 🐙🐙🐙🐙
- 🐙🐙🐙🐙
```

🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙.

**Mermaid** 🐙🐙🐙🐙 **Javascript** 🐙🐙🐙🐙🐙🐙🐙🐙 **markdown** 🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙🐙.

🐙🐙

```
graph TD
 A[Christmas] -->|Get money| B(Go shopping)
 B --> C{Let me think}
 C -->|One| D[Laptop]
 C -->|Two| E[iPhone]
 C -->|Three| F[fa:fa-car Car]
```

🐙🐙

```
graph TD
 A[Christmas] -->|Get money| B(Go shopping)
 B --> C{Let me think}
 C -->|One| D[Laptop]
 C -->|Two| E[iPhone]
 C -->|Three| F[fa:fa-car Car]
```

- 🐙🐙🐙: <https://github.com/mermaid-js/mermaid>
- 🐙🐙🐙: <https://mermaidjs.github.io/mermaid-live-editor/>
- 🐙🐙🐙: <https://mermaid-js.github.io/mermaid/#/flowchart>



```
graph BT
 Start --> Stop
```

□□

```
graph BT
 Start --> Stop
```

- LR

□□□□: from **L**eft to **R**ight

□□

```
graph LR
 Start --> Stop
```

□□

```
graph LR
 Start --> Stop
```

- RL

□□□□: from **R**ight to **L**eft

□□

```
graph RL
 Start --> Stop
```

□□

```
graph RL
 Start --> Stop
```

□□

□□□

```

- []
+ []
 - [[]]
 - [()]
 - [{ }]
 - [/\/]
 - [\\\\]
 - [/\/]
 - [\//]
+ ()
 - (())
 - ([])
 - ({ })
+ { }
 - {{ }}
 - {[]}
 - {()}
+ >[]

```

```

- []
+ []
 - [[]]
 - [()]
 - [{ }]
 - [/\/]
 - [\\\\]
 - [/\/]
 - [\//]
+ ()
 - (())
 - ([])
 - ({ })
+ { }
 - {{ }}
 - {[]}
 - {()}
+ >[]

```

00000000,0000000000000000,000000000000,0000000000,00 [] 0000, ( ) 0000, {} 0000(000 <> 00  
 0)00.

00: 0000000000,00000000.

000000

000000,0000000000,0000000000 id 00,0000000000000000.

id 000000000000000000000000000000.

00

```
graph TD
 id
```

00

```
graph TD
 id
```

5/5

```
000000,000000,00000000 id 0000 <node shape> 00,000000000000000000.
```

```
id 0000000000,0000000000 <node shape> 0000,00 id 0000000000.
```

- □ □

```
[node description] , [] [node description]
```

00

```
graph LR
 id1[This is the text in the box]
```

00

```
graph LR
 id1[This is the text in the box]
```

- □ □ □ □

```

0000: (node description) , () 0000000000000000, node description 00000000.

```

11

```
graph LR
 id1(This is the text in the box)
```

11

```
graph LR
 id1(This is the text in the box)
```

- ☐ ☐ ☐



```
graph LR
 node1["node description"] --> node2["node description"]
```

graph LR

```
graph LR
 id1([This is the text in the box])
```

graph LR

```
graph LR
 id1([This is the text in the box])
```

- graph LR

```
graph LR
 node1["node description"] --> node2["node description"]
```

graph LR

```
graph LR
 id1[(Database)]
```

graph LR

```
graph LR
 id1[(Database)]
```

- graph LR

```
graph LR
 node1["node description"] --> node2["node description"]
```

graph LR

```
graph LR
 id1([This is the text in the circle])
```

graph LR

```
graph LR
 id1([This is the text in the circle])
```

- graph LR

```
node: >node description] ,node > ,node] node,node description
.
```

graph LR

```
graph LR
 id1>This is the text in the box]
```

graph LR

```
graph LR
 id1>This is the text in the box]
```

- node

```
node: {node description} ,{} node,node description
.
```

graph LR

```
graph LR
 id1{This is the text in the box}
```

graph LR

```
graph LR
 id1{This is the text in the box}
```

- node

```
node: {{ node description }} ,{} node {} node,node description
.
```

graph LR

```
graph LR
 id1\{\{This is the text in the box\}\}
```

Gitbook node {} node,node,node \ node.

graph LR

```
graph LR
 id1\{\{This is the text in the box\}\}
```

- 閉じタグ

```
閉じタグ: [/node description/] , [] 閉じタグ // 閉じタグ閉じタグ閉じタグ, node description 閉じタグ閉じタグ.
```

```
閉じ
```

```
graph TD
 id1[/This is the text in the box/]
```

```
閉じ
```

```
graph TD
 id1[/This is the text in the box/]
```

- 閉じタグ

```
閉じタグ: [\node description\] , [] 閉じタグ \ \ 閉じタグ閉じタグ閉じタグ, node description 閉じタグ閉じタグ.
```

```
閉じ
```

```
graph TD
 id1[\This is the text in the box\]
```

```
閉じ
```

```
graph TD
 id1[\This is the text in the box\]
```

- 閉じ

```
閉じタグ: [/node description\] , [] 閉じタグ /\ 閉じタグ閉じタグ閉じタグ, node description 閉じタグ閉じタグ.
```

```
閉じ
```

```
graph TD
 A[/Christmas\]
```

```
閉じ
```

```
graph TD
 A[/Christmas\]
```

- 閉じタグ

```

node: [\node description/] , [] \ / ,node description .

```

```


```

```

graph TD
 B[Go shopping/]

```

```


```

```

graph TD
 B[Go shopping/]

```

```


```

```


```

```

+ 00/00
- --
- -.
+ 000/000
- >
- -
+ 000/000
- 00
+ --0000
+ | 0000 |
- 00
+ -.0000
+ | 0000 |
+ 00
- ==
+ 0000
- -->
- ---
- -.->
- -. -
- 00000000
+ --0000-->
+ --> | 0000 |
- 00000000
+ --0000---
+ --- | 0000 |
- 00000000
+ -.0000-.->
+ -.-> | 0000 |
- 00000000
+ -.0000-.-
+ -. - | 0000 |
- ==>
- ===
- 0000000000(2)
+ ==0000==>
+ ==> | 0000 |
- 0000000000(2)
+ ==0000===
+ === | 0000 |

```

```

+ 00/00
- --
- -.
+ 000/000
- >
- -
+ 000/000
- 00
+ --0000
+ | 0000 |
- 00
+ -.0000
+ | 0000 |
+ 00
- ==
+ 0000
- -->
- ---
- -.->
- -. -
- 00000000
+ --0000-->
+ --> | 0000 |
- 00000000
+ --0000---
+ --- | 0000 |
- 00000000
+ -.0000-.->
+ -.-> | 0000 |
- 00000000
+ -.0000-.-
+ -. - | 0000 |
- ==>
- ===
- 0000000000(2)
+ ==0000==>
+ ==> | 0000 |
- 0000000000(2)
+ ==0000===
+ === | 0000 |

```

00000000,00000000000000000000,0000000000000000,00 -- 0000,00000000 -. - 0000,0000  
00000 > ,0000000000 -.

00: 000000,00000000,00000000000000000000,00000000000000.

- 0000000000

0000: --> ,00 -- 0000, > 00000.

00

```
graph LR
 A-->B
```

□□

```
graph LR
 A-->B
```

- □□□□□

□□□□: --- ,□□ -- □□□□, - □□□□□.

□□

```
graph LR
 A --- B
```

□□

```
graph LR
 A --- B
```

- □□□□□□□□□□

□□□□: --connection line description--> ,□□□□□ -- □□□□□□□□□□,□□□ --> □□□□□□□□□.

□□

```
graph LR
 A-- text -->B
```

□□

```
graph LR
 A-- text -->B
```

□□□□: |connection line description| ,□□ || □□□□□□□□□□.

□□

```
graph LR
 A-->|text|B
```

❏

```
graph LR
 A-->|text|B
```

- ❶❷❸❹❺❻❼❽❾❿

❶❷❸❹: `--connection line description` ,❶❷❸❹ `--` ❶❷❸❹❶❷❸❹❶❷❸❹❶❷❸❹ `---` ❶❷❸❹❶❷❸❹❶❷❸❹.

❏

```
graph LR
 A-- This is the text ---B
```

❏

```
graph LR
 A-- This is the text ---B
```

❶❷❸❹: `|connection line description|` ,❶❷❸❹ `||` ❶❷❸❹❶❷❸❹❶❷❸❹❶❷❸❹.

❏

```
graph LR
 A---|This is the text|B
```

❏

```
graph LR
 A---|This is the text|B
```

- ❶❷❸❹❺❻❼❽❾❿

❶❷❸❹: `-.connection line description.->` ,❶❷❸❹❶❷❸❹ `-. ❶❷❸❹❶❷❸❹❶❷❸❹❶❷❸❹` `.->` ❶❷❸❹❶❷❸❹❶❷❸❹.

❏

```
graph LR
 A-. text .-> B
```

❏



```
graph LR
 A-. text .-> B
```

- 图例

图例: `==>` ,图例.

图

```
graph LR
 A ==> B
```

图

```
graph LR
 A ==> B
```

- 图例

图例: `==connection line description` ,图例 == 图例,图例 ==> 图例.

图

```
graph LR
 A == text ==> B
```

图

```
graph LR
 A == text ==> B
```

- 图例

图例: `|connection line description|` ,图例 || 图例.

图

```
graph LR
 A ==>|text| B
```

图

```
graph LR
 A ==>|text| B
```

□□□□

□□□

```
+ -->-->
+ &
+ ""
+ %%
+ subgraph
```

```
+ -->-->
+ &
+ ""
+ %%
+ subgraph
```

- □□□□□□□□

□□

□□□□□□□□, A-->B-->C □□□ A-->B □ B-->C □□.

```
graph LR
 A -- text --> B -- text2 --> C
```

□□

```
graph LR
 A -- text --> B -- text2 --> C
```

- □□□□□□□□

□□□□□□□□, A-->B & C □□□ A-->B □ A-->C □□.

□□

```
graph LR
 a --> b & c--> d
```

□□

```
graph LR
 a --> b & c--> d
```

- □□□□□□□□

graph LR; A & B --> C & D; A-->C; A-->D; B-->C; B-->D

```
graph TB
 A & B--> C & D
```

```
graph TB
 A & B--> C & D
```

- graph LR

graph LR; id1["This is the (text) in the box"]

```
graph LR
 id1["This is the (text) in the box"]
```

```
graph LR
 id1["This is the (text) in the box"]
```

- graph LR

graph LR; A["A double quote:""] --> B["A dec char:#9829;"]

```
graph LR
 A["A double quote:""] --> B["A dec char:#9829;"]
```

```
graph LR
 A["A double quote:""] --> B["A dec char:#9829;"]
```

- graph LR

```
subgraph title
 graph definition
end
```

```
graph TB
 c1-->a2
 subgraph one
 a1-->a2
 end
 subgraph two
 b1-->b2
 end
 subgraph three
 c1-->c2
 end
```

```
graph TB
 c1-->a2
 subgraph one
 a1-->a2
 end
 subgraph two
 b1-->b2
 end
 subgraph three
 c1-->c2
 end
```


- 

%%

```
graph LR
%% this is a comment A -- text --> B{node}
A -- text --> B -- text2 --> C
```

```
graph LR
%% this is a comment A -- text --> B{node}
A -- text --> B -- text2 --> C
```

- 
- 
- 

mermaid-flow-chart-summary-simplemindmap.png

**Mermaid** , **Markdown** ,

图的基本概念

图是由顶点和边组成的集合。顶点是图的基本元素，边是连接顶点的线。图可以表示现实世界中的许多事物，如网络、地图、数据库等。

| 图  | 图        | 图           |
|----|----------|-------------|
| 图  | graph    | graph 图     |
| 子图 | subgraph | subgraph 子图 |
| 图  | top      | TB 图 BT ,图  |
| 图  | bottom   | BT 图 TB ,图  |
| 图  | left     | LR 图 RL ,图  |
| 图  | right    | RL 图 LR ,图  |

图的基本概念

图是由顶点和边组成的集合。顶点是图的基本元素，边是连接顶点的线。图可以表示现实世界中的许多事物，如网络、地图、数据库等。

- 图的基本概念

| 图   | 图 | 图 | 图 |
|-----|---|---|---|
| [ ] | 图 | 图 | 图 |
| ( ) | 图 | 图 | 图 |
| { } | 图 | 图 | 图 |
| < > | 图 | 图 | 图 |
| --  | 图 | 图 | 图 |
| -.  | 图 | 图 | 图 |
| ==  | 图 | 图 | 图 |
| =:  | 图 | 图 | 图 |
| >   | 图 | 图 | 图 |
| -   | 图 | 图 | 图 |
| xxx | 图 | 图 | 图 |
| --  | 图 | 图 | 图 |
| -.  | 图 | 图 | 图 |
| ==  | 图 | 图 | 图 |
| =:  | 图 | 图 | 图 |

- 图的基本概念

| □□□                                              | □□           | □□      | □□  |
|--------------------------------------------------|--------------|---------|-----|
| <code>[[]]</code>                                | □□□          | □□□□    | □□□ |
| <code>[()]</code>                                | □□□          | □□□□    | □□  |
| <code>[{}]</code>                                | □□□          | □□□□    | □□□ |
| <code>(())</code>                                | □□           | □□□□    | □□  |
| <code>([])</code>                                | □□□          | □□□□    | □□  |
| <code>({})</code>                                | □□           | □□□□    | □□□ |
| <code>⌈⌋⌈⌋</code>                                | □□□          | □□□□    | □□  |
| <code>{[]}</code>                                | □□□□         | □□□□    | □□□ |
| <code>{()}</code>                                | □□           | □□□□    | □□□ |
| <code>--&gt;</code>                              | □□□□□        | □□□□□□  | □□  |
| <code>---</code>                                 | □□□□□        | □□□□□□  | □□  |
| <code>-.&gt;</code>                              | □□□□□        | □□□□□□  | □□□ |
| <code>-.-&gt;</code>                             | □□□□□        | □□□□□□  | □□  |
| <code>.-&gt;</code>                              | □□□□□        | □□□□□□  | □□  |
| <code>.-&gt;</code>                              | □□□□□        | □□□□□□  | □□  |
| <code>.-&gt;</code>                              | □□□□□        | □□□□□□  | □□  |
| <code>==&gt;</code>                              | □□□□□□□      | □□□□□□  | □□  |
| <code>===</code>                                 | □□□□□□□      | □□□□□□  | □□  |
| <code>==&gt;</code>                              | □□□□□□□      | □□□□□□  | □□□ |
| <code>==&gt;</code>                              | □□□□□□□      | □□□□□□  | □□□ |
| <code>==&gt;</code>                              | □□□□□□□      | □□□□□□  | □□□ |
| <code>:=</code>                                  | □□□□□□□      | □□□□□□  | □□□ |
| <code>⌈⌋⌈⌋</code>                                | □□□□□□□□     | □□□□□□□ | □□  |
| <code>--connection line description--&gt;</code> | □□□□□□□□□□□□ | □□□□□□□ | □□  |
| <code>-.connection line description-.&gt;</code> | □□□□□□□□□□□□ | □□□□□□□ | □□  |
| <code>--connection line description---</code>    | □□□□□□□□□□□□ | □□□□□□□ | □□  |
| <code>-.connection line description.-</code>     | □□□□□□□□□□□□ | □□□□□□□ | □□  |
| <code>==connection line description==&gt;</code> | □□□□□□□□□□□□ | □□□□□□□ | □□  |
| <code>:=connection line description:=&gt;</code> | □□□□□□□□□□□□ | □□□□□□□ | □□□ |
| <code>==connection line description===</code>    | □□□□□□□□□□□□ | □□□□□□□ | □□  |
| <code>:=connection line description:=</code>     | □□□□□□□□□□□□ | □□□□□□□ | □□□ |

□□□□

□□□□□□□□□□□□,□□□□□□□□,□□□□□□□□□□□□,□□□□□□□□□□□□□□,□□□□□□□□□.

📖📖📖📖📖📖📖📖📖📖📖📖,📖📖📖 JS 📖 CSS 📖📖📖📖📖📖📖📖📖📖📖📖📖📖.

📖📖📖: <https://mermaid-js.github.io/mermaid/#/flowchart?id=styling-and-classes>

- 📖📖📖 Interaction : <https://mermaid-js.github.io/mermaid/#/flowchart?id=interaction>
- 📖📖📖 Styling and classes : <https://mermaid-js.github.io/mermaid/#/flowchart?id=interaction>
- 📖📖📖 Basic support for fontawesome: <https://mermaid-js.github.io/mermaid/#/flowchart?id=basic-support-for-fontawesome>
- 📖📖📖 <https://mermaid-js.github.io/mermaid/#/flowchart?id=graph-declarations-with-spaces-between-vertices-and-link-and-without-semicolon>

# Reference

The following admonishments are implemented by the `mdbook-admonish` plugin and are automatically themed to match Catppuccin.

## Directives

All supported directives are listed below.

### note

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

### abstract, summary, tldr

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

### info, todo

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

### tip, hint, important

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

### success, check, done

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

### question, help, faq

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

### warning, caution, attention

⚠ Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.



failure, fail, missing

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

danger, error

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

bug

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

example

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

quote, cite

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

# Bienvenue sur notre site de développement 3D !

Bienvenue sur notre site dédié au développement 3D. Ici, vous trouverez des ressources, des tutoriels et des informations utiles pour vous lancer dans le monde passionnant de la 3D.

A beautifully styled message.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla et euismod nulla. Curabitur feugiat, tortor non consequat finibus, justo purus auctor massa, nec semper lorem quam in massa.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla et euismod nulla. Curabitur feugiat, tortor non consequat finibus, justo purus auctor massa, nec semper lorem quam in massa.

✖ Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla et euismod nulla. Curabitur feugiat, tortor non consequat finibus, justo purus auctor massa, nec semper lorem quam in massa.

```

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla et euismod
nulla. Curabitur feugiat, tortor non consequat finibus, justo purus auctor
massa, nec semper lorem quam in massa.

```

```

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla et euismod
nulla. Curabitur feugiat, tortor non consequat finibus, justo purus auctor
massa, nec semper lorem quam in massa.

```

```

The opposite of *referencing* by using `&` is *dereferencing*, which is
accomplished with the dereference operator, `*`.

```

```

This syntax won't work in Python 3:
```python
print "Hello, world!"
```

```

## À propos de nous

Nous sommes une équipe passionnée par la 3D et nous avons pour mission de partager nos connaissances avec la communauté. Vous trouverez ici des articles, des exemples de code et des démonstrations pour vous aider à démarrer votre voyage dans le développement 3D.

```

graph LR
 A[Start] --> B{Error?};
 B -->|Yes| C[Hmm...];
 C --> D[Debug];
 D --> B;
 B ---->|No| E[Yay!];

```

## Pour commencer

Si vous êtes nouveau dans le domaine de la 3D, ne vous inquiétez pas ! Notre page [“Getting Started”](#) vous guidera à travers les étapes essentielles pour démarrer rapidement.

## Restons en contact

N'hésitez pas à nous suivre sur les réseaux sociaux pour rester à jour avec nos dernières publications et annonces. Si vous avez des questions ou des commentaires, n'hésitez pas à nous contacter !

Mizux



# Chapter 1

## Section 1

## Section 2

# Chapter 2

## Section 1

## Section 2

# Chapter 3

## Section 1

## Section 2

# Chapter 4

## Section 1

## Section 2

# Chapter 5

## Section 1

## Section 2

# Chapter 6

## Section 1

## Section 2

Here is an inline example,  $\pi(\theta)$ ,  
an equation,

$$\nabla f(x) \in \mathbb{R}^n,$$

and a regular \$ symbol.

Define  $f(x)$ :

$$f(x) = x^2 \quad x \in \mathbb{R}$$

```
graph TD
 A[Anyone] -->|Can help | B(Go to github.com/yuzutech/kroki)
 B --> C[How to contribute?]
 C --> D[Reporting bugs]
 C --> E[Sharing ideas]
 C --> F[Advocating]
```