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,

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

and a regular \$ symbol.

4/8/25. 6:59 AM

mdbook-demo

kuanghl

Define f(x):

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

```
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—— IIII III III Python

1.1 ПППП

pybind110000C++000000Python0

- ullet
- пппппппппппп
- ППППП
- 0000000
- 00000
- DDDDrangesD
- חחחחחחח
- חחחחחחחחח
- STL0000
- ПППППП
- Internal references with correct reference counting []

• 000Python000000000000C++00

1.2 ППППП

- 00Python2.7, 3.5+, PyPy/PyPy3 7.300000000
- 00000000000lambda000lambda00000000Python000000

- 00000000000Python000000000000
- 00Boost.Python0000000000000000
- DDDDDC++DDDDPython pickleDunpickleDDD

1.3 000000

- 1. Clang/LLVM 3.300 (Apple Xcode's clang005.0.00000)
- 2. GCC 4.800
- 3. Microsoft Visual Studio 2015 Update 300
- 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)

4/8/25, 6:59 AM

mdbook-demo

kuanghl

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 Houliston, 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.

0000				

mdbook-demo

kuanghl

4/8/25, 6:59 AM	mdbook-demo	kuanghl

4/8/25, 6:59 AM

mdbook-demo

kuanghl

3. ПППП

3.1 000000000

000000Git0000000pybind1100000000000000git0000000000pybind110

3.2 DPyPIDD

DDDDDpipDDDPyPIDDDPybind11DPythonDDDDDDDDDDDCMakeDDDDDD

pip install pybind11

System[][][][pybind11[]

DDpybind110000Python00000000000000000000pybind11000000

3.3 □□conda-forge□□

You can use pybind11 with conda packaging via conda-forge:

conda install -c conda-forge pybind11

3.4 □ □ vcpkg □ □

00000Microsoft vcpkg000000000000pybind110

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

3.5 □□**brew**□□□□□

brew install pybind11

3.6

Other locations you can find pybind11 are listed here; these are maintained by various packagers and the community.

4/8/25, 6:59 AM

mdbook-deme

kuanghl

4. 0000First steps0

00000pybind11000000000000000000pybind1100000000

4.1 חחחחח

Linux/macOS

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

Windows

□Windows□□□□□C++11□Visual Studio□□□15□□□□□□

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

חחחחחחחחחח Studio

4.2 0000000000

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

4.3 00000000

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

```
#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");
       0000m000000000000000
□add□□□Python□□□□□
 ПП
c++ -03 -Wall -shared -std=c++11 -fPIC $(python3 -m pybind11 --includes)
example.cpp -o example$(python3-config --extension-suffix)
 Note00000000000000pybind1100000000
                               ППП
```

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

4.4

0000000C++00000000000Python00000000000"i"0"j"00

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

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

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

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

      A function which adds two numbers
```

```
// 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);
```

4.5 ПППП


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

П

000000000000000

```
>>> 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 ПППП

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

Python図図図図図
```pyhton
>>> import example
>>> example.the_answer
42
>>> example.what
'World'
```

# **4.7** 0000000

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.** 000000

#### **5.1** 00000000000

00000000000000000C++0000000<del>Pet</del>000000

```
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);
}
```

000000000Python000000

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

#### 5.2 000000000

#### **5.3** חחחחח

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

0000000Python000000

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

pybind11000000000lambda0000lambda0000

#### **5.4** ПППП

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

## 

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

```
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;
};
```

```
4/8/25, 6:59 AM
 (0000001
 haaaaaaa
DDDDDDDDDsetterDgeterDDD
 py::class_<Pet>(m, "Pet")
 .def(py::init<const std::string &>())
 .def_property("name", &Pet::getName, &Pet::setName)
DDDDDDreadDDDDnullptrDDDD
 see also: 🗆 🗆 🗅 🗅
 0000000000000
5.5 ПППП
>>> class Pet:
 name = "Molly"
```

```
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}
```

#### **5.6** ПППППППП

#### 

```
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!"; }
};
```

```
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!'
```

#### 

```
// MAMMMAMMAMM
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'
```

```
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!'
```

# **5.7 0000**

```
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;
};
```

```
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");
```

#### 

4/8/25, 6:59 AM **kuanghl** 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"); пппі )0000000const00000000 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\_)); 0000000c++11000000 10000000 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"):

#### **5.8** חחחחחחח

```
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;
};
```

```
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);
```

4/8/25, 6:59 AM mdbook-demo kuanghl >>> p = Pet("Lucy", Pet.Cat) >>> p.type Kind.Cat >>> int(p.type) 1L >>> Pet.Kind.\_\_members\_\_ {'Dog': Kind.Dog, 'Cat': Kind.Cat} ПП >>> p = Pet("Lucy", Pet.Cat) >>> pet\_type = p.type >>> pet\_type Pet.Cat >>> str(pet\_type) 'Pet.Cat' >>> pet\_type.name 'Cat'

**6.** 0000

# **7.** ПП

#### 7.1 00000

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

00000	00
return_value_policy::take_ownership	00000000000000000000000000000000000000
<pre>return_value_policy::copy</pre>	00000000Python000000000000000000000000000000000000
return_value_policy::move	00 <b>66686666</b> 0000000000000000000000000000
<pre>return_value_policy::reference</pre>	00000000000000C++0000000 00000000000000
<pre>return_value_policy::reference_internal</pre>	00000000000000000000000000000000000000
return_value_policy::automatic	00000000000000000000000000000000000000
	<del></del>
return_value_policy::automatic_reference	DC++000000Python00000       DC++00000000000000000000000000000000000

#### **7.2** חחחחחחח

# □□□keep alive□

 Onurse
 Onurse<

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

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

```
Note: Reep alive Boost.Python Downth custodian and ward Dwith custodian and ward postcall DDD
```

### **Call guard**

call\_guard(12)00000T000Scope guard000000000000000

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

<del>AAAAAAAAA</del>

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

000000T0000000 git\_scoped\_release 0000000000



# **7.3 Python 0 0 0 0**

□Python

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

# 7.4 DD\*argsD\*\*kwatgsDD

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

0000000pybind110000000

```
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);

DUILD Rest/test_kwargs_and_defualts.cpp
```

# **7.5** 000000

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

DDDDDDDDDSomeTypeDDDDDbindingDDDDpy::class\_DDDDDDDDDDD

```
FUNCTIONS

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

0000000 **arg V**00000000000000000

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

pybind11000 pysskw\_only

# 7.7 Positional-only

python3.8000Positional-only00000pybind1100pythos\_only() 0000000000

# 7.8 Non-converting □ □

- DD py::implicitly\_convertible<A,B>() DDDDDD
- DDDDDDDfloatDDDDDD<mark>std::complex:float></mark>DDDDD
- Calling a function taking an Eigen matrix reference with a numpy array of the wrong type or of an incompatible data layout.

```
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"));
```

0000000000 **TypeError**000

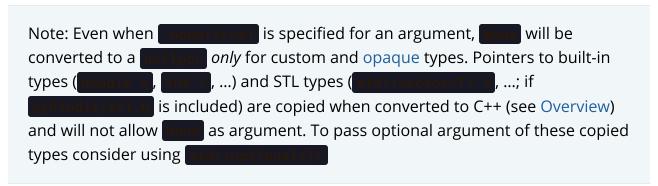
```
>>> floats_preferred(4)
2.0
>>> floats_only(4)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: floats_only(): incompatible function arguments. The following argument types are supported:
 1. (f: float) -> float
Invoked with: 4
```

000000000\_a

#### **7.9** 00/00000

```
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));
```

DDDPythonDD bark(None) DDD "(no.dog)" DDD meow(None) DDDDD TypeError D



## **7.10 DDDDD**

# **8.** 🗆

# **8.1 Python 0 0 0 0**

00000000C++0000000Python000000

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

pybind11000000

```
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);
}
```

00000Python000000Animal0000000

```
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)
);
}
```

```
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);
}
```

pybind11000 Delaware 00000000 PyAnimal000000 Python000 Animal00

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

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

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

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

 def bark(self):
 return "yap!"
```

#### Note□

- 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

```
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!"; }
};</pre>
```

```
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,); }
};
```

#### 

```
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,); }
};
```

00000pybind11000000

```
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
```

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

#### 8.3 חחחחחחחח

#### 8.3.1

**See also** See the file **Result of the Complete** examples showing both normal and forced trampoline instantiation.

```
which has the signature whether something should be done with the with the Python side by allowing the Python function to return with the or an arm:
```

## 8.4 000000

```
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));
```

```
class Example {
private:
 Example(int); // private constructor
 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")
 .def(py::init(&Example::create))
 .def(pv::init([](std::string arg) {
 return std::unique_ptr<Example>(new Example(arg));
 }))
 .def(py::init([](int a, int b) { return new Example(a, b); }))
 .def(py::init<double>())
```

Python00000000000pybind11000000000C++00000Python0000

```
#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")
 // instance will be moved via the extra constructor in PvExample.
 .def(py::init([]() { return new Example(); }))
 .def(py::init([]() { return new Example(); } /* no alias needed */,
 []() { return new PyExample(); } /* alias needed */))
 .def(py::init([]() { return new PyExample(); }))
```

 $\overline{\Pi}$ 

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

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

#### 8.5

```
/* ... definition ... */
class MyClass {
 private:
 ~MyClass() { }
};

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

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

# **8.6 000000 Python**

## **8.7 0000**

00000B0000A000000000000000000000Python0000000000

# py::implicitly\_convertible<A, B>();

Note: A0B000000000pybind11000B00000000

In Interior implicit conversion invoked as part of another implicit conversion of the same type (i.e. from to ) will fail.

## 8.8

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

#### 8.9

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

# 8.10 Dickle

```
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
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0</t

```
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 py::make_tuple(p.value(), p.extra());
 [](py::tuple t) { // __setstate__
 if (t.size() != 2)
 throw std::runtime_error("Invalid state!");
 Pickleable p(t[0].cast<std::string>());
 p.setExtra(t[1].cast<int>());
 return p;
 }
));
```

holder type 🛚

```
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 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 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

```
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);
```

## **8.12 0000**

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

00Python000000C++0000000C++00Python00

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

## 8.13 □□Module-local□

```
// 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'
```

```
py::class<pets::Pet>(m, "Pet")
 .def("name", &pets::Pet::name);
py::class<Dog, pets::Pet>(m, "Dog")
 .def(py::init<std::string>());
py::class<pets::Pet>(m, "Pet")
 .def("get_name", &pets::Pet::name);
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!
0000000000000000
py::class<pets::Pet>(m, "Pet", py::module_local())
 .def("name", &pets::Pet::name);
py::class<pets::Pet>(m, "Pet", py::module_local())
 .def("get_name", &pets::Pet::name);
□□□Python□□
 ППП
```

```
000000000000C++0Python00000000
 □□□C++□□□□module-local□□□
ПППППП
 m.def("pet_name", [](const pets::Pet &pet) { return pet.name(); });
000000000000000000
 >>> import cats, dogs, frogs # No error because of the added
 >>> 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')
000000C++00000000Python000000000000000000Python000
 Note: STL bindings (as provided via the optional
 header)
 by default when the bound type might conflict with
 apply
 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
 , 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 Oprotected Oprotected**

□□□□□□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'
```

```
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
```



```
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</pre>
 .def("foo", &Publicist::foo); // <-- `Publicist` here, not</pre>
```

# **8.15** □ □ final □

```
class PyFinalChild(IsFinal):
 pass

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

#### **8.16** ППППППППП

```
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) {
 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 to a Python object, it calls to determine if a downcast is possible.

The function should use whatever runtime information is available to determine if its parameter is in fact an instance of some class that inherits from the converted and and and and and are converted to the converted to the

returns a pointer to the object that contains Otherwise, it just returns leaving at its default value of nullptr. If you set to a type that pybind11 doesn't know about, no downcasting will occur, and the original pointer will be used with its static type other.

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

#### **8.17** חחחחחח

0000000C++000000000

<del>-4/8/25, 6:59 AM</del>

mdbook-demo

kuanghl

# 9. □□

# **9.1 C++**0000**Python**0000

Exception thrown by C++	Translated to Python exception type
<pre>std::exception</pre>	RuntimeError
<pre>std::bad_alloc</pre>	MemoryError
<pre>std::domain_error</pre>	ValueError
<pre>std::invalid_argument</pre>	ValueError
<pre>std::length_error</pre>	ValueError
<pre>std::out_of_range</pre>	IndexError
<pre>std::range_error</pre>	ValueError
<pre>std::overflow_error</pre>	OverflowError
<pre>pybind11::stop_iteration</pre>	iterators) (used to implement custom
<pre>pybind11::index_error</pre>	(used to indicate out of bounds access in <b>Constitution</b> , <b>Constitution</b> , etc.)
pybind11::key_error	(used to indicate out of bounds access in <b>Carolites</b> , <b>Carolites</b> in dict-like objects, etc.)

py::register\_exception<CppExp>(module, "PyExp", PyExc\_RuntimeError);
py::register\_local\_exception<CppExp>(module, "PyExp", PyExc\_RuntimeError);

DDDPyExpDDDDDDDPyExpDRuntimeErrorD

Inside the translator, 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 Python exception or to set the python exception to a custom exception type (see below). To declare a custom Python exception type, declare a 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 the first is translated to a custom python and , while the second is translated to a standard python exception 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) {

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.

PyErr\_SetString(PyExc\_RuntimeError, e.what());

});

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 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++DPython**

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

```
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;
}
```

```
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 D Python C API D D**

```
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");
```

OOOOO PyErr\_Clear OOOOOO

00Python0000000000000Python/pybind110000000

#### 9.6 □□□□raise from□

## 9.7 ∏∏unraiseable∏∏

```
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.** ПППП

# 10.1 std::unique\_ptr

```
std::unique_ptr<Example> create_example() { return
std::unique_ptr<Example>(new Example()); }
m.def("create_example", &create_example);
```

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

# 10.2 std::shared\_ptr

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

## 

#### 

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

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

#### 10.3

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

ПППППППППППППboolПППППППППfalseП

```
// 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();
}
 };
};
}
```

4/8/25, 6:59 AM	mdbook-demo	kuanghl	
000000pybind110000	martPtr 🛛 🗘 .getPoi	inter() 00 .get() 000	
see also: 🗆 🚾 🕶 💮	st_smart_ptr.cpp	00000000000000000000000000000000000000	

# **11.** 0000

- 1. 0000000C++0000000000000pybind11000000Python000000
- 2. 0000000Python0000000C++00000000

### **11.1** □□

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

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

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

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

4/6/25, 6:35 AM	masook-aemo	Kuangni	
Python list color characteristics and color characteristics.	ov::list	0000000000000000Pyt	hon0000000
	100000000000Pytl	hon0000000000	

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

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

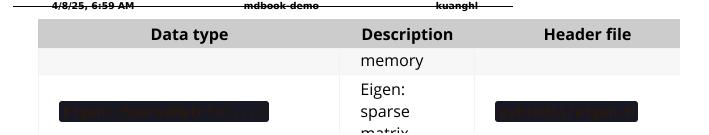
```
>>> print_vector([1, 2, 3])
1 2 3
```

Data type	Description	Header file
int8 t, wint8 t	8-bit integers	<pre>oybind11/pybind11.h</pre>
int16_t, wint16_t	16-bit integers	pybindl1/pybindl1.h
int32_t, uint32_t	32-bit integers	<pre>pybind11/pybind11.h</pre>
int64_t, uint64_t	64-bit integers	<pre>pybind11/pybind11.h</pre>

Data type	Description	Header file
ssize t, size t	Platform- dependent size	<pre>pybind11/pybind11.h</pre>
float, double	Floating point types	pybind11/pybind11.h
boal	Two-state Boolean type	pybind11/pybind11.h
Char	Character literal	<pre>pybind11/pybind11.h</pre>
char 16_t	UTF-16 character literal	<pre>pybind11/pybind11.h</pre>
char 32 1	UTF-32 character literal	<pre>pybind11/pybind11.h</pre>
wchar t	Wide character literal	<pre>pybind11/pybind11.h</pre>
const char *	UTF-8 string literal	<pre>pybind11/pybind11.h</pre>
const char16_t *	UTF-16 string literal	<pre>pybind11/pybind11.h</pre>
const char32_t *	UTF-32 string literal	<pre>pybind11/pybind11.h</pre>
<pre>const wchar_t *</pre>	Wide string literal	<pre>pybind11/pybind11.h</pre>
std::string	STL dynamic UTF-8 string	pybind11/pybind11.h
std::ul6string	STL dynamic UTF-16	<pre>pybind11/pybind11.h</pre>

4/0/25/ 0/35 AM Mabook demo	Kuungn	
Data type	Description	Header file
	string	
std::u32string	STL dynamic UTF-32 string	pybind11/pybind11.h
std::wstring	STL dynamic wide string	pybind11/pybind11.h
<pre>std::string_view, std::u16string_view, etc.</pre>	STL C++17 string views	pybindl1/pybindl1.h
std::pair <t1, t2=""></t1,>	Pair of two custom types	pybind11/pybind11.h
std::tuple<>	Arbitrary tuple of types	pybind11/pybind11.h
std::reference_wrapper<>	Reference type wrapper	<pre>pybind11/pybind11.h</pre>
std::complex <t></t>	Complex numbers	pybindl1/complex.h
std::array <t, size=""></t,>	STL static array	pybind11/stl.h
std::vector <t></t>	STL dynamic array	pybind11/stl.h
	STL double-	
std::deque <t></t>	ended queue	oybind11/stl.h
std::valarray <t></t>	STL value array	pybind11/stl.h
std::list <t></t>	STL linked list	pybind11/stl.h

Data type	Description	Header file
std::map <t1, t2=""></t1,>	STL ordered map	pybind11/stl.h
std::unordered_map <t1, t2=""></t1,>	STL unordered map	pybind11/stl.h
std::set <t></t>	STL ordered set	pybind11/stl.h
std::unordered_set <t></t>	STL unordered set	pybind11/stl.h
std::optional <t></t>	STL optional type (C++17)	pybind11/stl.h
<pre>std::experimental::optional<t></t></pre>	STL optional type (exp.)	pybind11/stl.h
std::variant<>	Type-safe union (C++17)	pybind11/stl.h
<pre>std::filesystem::path<t></t></pre>	STL path (C++17) 1	pybind11/stl.h
std::function<>	STL polymorphic function	pybind11/functional.h
<pre>std::chrono::duration&lt;&gt;</pre>	STL time duration	pybind11/chrono.h
<pre>std::chrono::time_point&lt;&gt;</pre>	STL date/time	pybind11/chrono.h
Eigen::Matrix<>	Eigen: dense matrix	pybind11/eigen.h
Eigen::Map<>	Eigen: mapped	pybind11/eigen.h



# 11.2 Strings, bytes and Unicode conversions



# 11.2.1 | Python strings | C++

```
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;
 }
);</pre>
```

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

**

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

00C++000000000000000const00000000

□C++□□bytes□□

## **11.2.2 Python C++**

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

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

DDUTF-8000ASCII000000ASCII0000Python0000000000000000000000TF-80

```
// 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-

 py::str py_s = PyUnicode_DecodeLatin1(s.data(), s.length());
 return py_s;
 }
);
```

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

## 

00000000**C++**000

```
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'
```

```
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

0000 enable unicode-ucs2 00000Python 2.703.300000000000000

DDDDDDDDDDDShift-JISDDDDDDUTF-8/16/32DDDDDDPythonD

#### **11.2.4** ПППП

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

### 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
'é'
>>> 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 argument will be passed UTF-16-encoded data, and a returned will be decoded as UTF-8).

### **11.3 STL**□□

### 11.3.1



#### 11.3.2 C++17 🗆 🗆 🗆

```
// `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>> {};
}}
```

under types:

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

Warning: When converting a 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 distribution is alternatives are listed is important, since pybind11 will try conversions in this order. This means that, for example, when converting distributions, the variant will never be selected, as any Python distributions is already and and is convertible to a C++ distribution. Changing the order of alternatives (and using in this example) provides a solution.

# **11.3.3** □□opaque□□

0000000000000000Python0C++00000000000000pass-by-reference0000

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

□Python□□□□□

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

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

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

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

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

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

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

<del>AAAAAAAAAAAAAAA</del>

```
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 **DOSTLOO**

ODDOOD OV::module local() Opy module local(false) ODDOODOODSTLOODOODOO

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

00000 **11.4** ПППП int func\_arg(const std::function<int(int)> &f) { return f(10); } std::function<int(int)> func\_ret(const std::function<int(int)> &f) { return [f](int i) { return f(i) + 1; **}**; } 0C++0000000python000000000 000000000000 py::cpp\_function func\_cpp() { return py::cpp\_function([](int i) { return i+1; }, py::arg("number")); }

```
#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
>>> 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

# 11.5 Chrono

ПППП

### 11.5.1 C++11 | | | | | |

### 11.5.2

# C++ Python

- std::chrono::system\_clock::time\_point → datetime.datetime
- std::chrono::duration → datetime.timedelta
- sta::chrono::[other\_ctocks]::time\_point → datetime.timedet

# Python C++



- 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

0000Eigen000000

### **11.7** חחחחחחח

The following snippets demonstrate how this works for a very simple type that that should be convertible from Python types that provide a method.

```
struct inty { long long_value; };

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

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

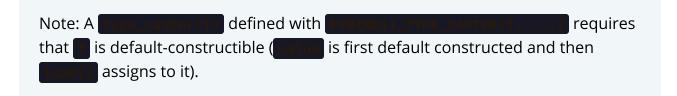
```
class A:
 def __int__(self):
 return 123

from example import print
print(A())
```

To register the necessary conversion routines, it is necessary to add an instantiation of the template. Although this is an implementation detail, adding an instantiation of this type is explicitly allowed.

```
namespace pybind11 { namespace detail {
 template <> struct type_caster<inty> {
 public:
 PYBIND11_TYPE_CASTER(inty, _("inty"));
 * instance or return false upon failure. The second argument
 bool load(handle src, bool) {
 PyObject *source = src.ptr();
 PyObject *tmp = PyNumber_Long(source);
 if (!tmp)
 return false;
 value.long_value = PyLong_AsLong(tmp);
 Py_DECREF(tmp);
 return !(value.long_value == -1 && !PyErr_Occurred());
 }
 static handle cast(inty src, return_value_policy /* policy */,
handle /* parent */) {
 return PyLong_FromLong(src.long_value);
 }
```

```
};
}} // namespace pybind11::detail
```



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++**□□

# **12.1 Python** □ □

#### **12.1.1 DDDDD**



Warning: Be sure to review the Gotchas before using this heavily in your C++ API.

## **12.1.2 C++DDDDDDPythonD**

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

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

000000000000Python000

simple namespace

#### 12.1.3

0000000000C++00000Python000000py::cast() 0000

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

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

# **12.1.4 C++DPython**

```
// 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 DPython**

00**0perator()**0000Python00000000

```
// 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 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 DPython DD**

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

In the example above 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 object explicitly, followed by other arguments.

4/8/25, 6:59 AM

<del>ndbook-dem</del>e

- kuanghl

```
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));
}</pre>
```

### **12.1.7 DDDDD**

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

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

```
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** ПППП

0000 **xargs** 0 **xxkwargs** 0000000000

```
// * 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** ПППП

```
#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)));
```

0<mark>0bject</mark>00000000000000000000<mark>0bj.cast()</mark>0

#### Note

If a trivial conversion via move constructor is not possible, both implicit and explicit casting (calling ) will attempt a "rich" conversion. For instance, will succeed and is equivalent to the Python code that produces a list of the dict keys.

### 12.1.10

\_PythonППППППП<mark>руссения already seet</mark>ПППППППППППП"ПC++ПППPythonПП"П

#### 12.1.11 Gotchas

## **Default-Constructed Wrappers**

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

# **12.2 NumPy**

### 

```
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;
};
```

```
py::class_<Matrix>(m, "Matrix", py::buffer_protocol())
 .def_buffer([](Matrix &m) -> py::buffer_info {
 return py::buffer_info(
 m.data(),
 sizeof(float),
 py::format_descriptor<float>::format(), /* Python struct-style
 format descriptor */
 2,
 dimensions */
 { m.rows(), m.cols() },
 { sizeof(float) * m.cols(),
 sizeof(float) }
);
 });
0000000
 000000000000matrix0000
 1000001
 0000Python00000000
 struct buffer_info {
```

```
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;
};
```

```
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;
 py::buffer_info info = b.request();
 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);
 }));
```

0000Eigen00000000(0000000)0000Eigen000

## **12.2.2 Arrays**

4/8/25, 6:59 AM mdbook-demo kuanghl

[Pyt Carray : Force as 1]

[DODG]

arrays0000NumPy API0000

- **.dtype()**
- strides() 0000strides0000

- reshape((i, j, ...)) 0000shape000000 restze({})) 0000
- index\_at(i, j, . .)

### **12.2.3** ППППП

```
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

*/
}
```

#### 12.2.4

```
double my_func(int x, float y, double z);
 0000000
m.def("vectorized_func", py::vectorize(my_func));
x = np.array([[1, 3], [5, 7]])
y = np.array([[2, 4], [6, 8]])
result = vectorized_func(x, y, z)
 Note
```

contrived, since it could have been done more simply using **[1887]**.

```
#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");
 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++)</pre>
 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

```
m.def("sum_3d", [](py::array_t<double> x) {
 auto r = x.unchecked<3>(); // x must have ndim = 3; can be non-
 double sum = 0;
 for (py::ssize_t i = 0; i < r.shape(0); i++)</pre>
 for (py::ssize_t j = 0; j < r.shape(1); j++)
 for (py::ssize_t k = 0; k < r.shape(2); k++)</pre>
 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++)</pre>
 for (py::ssize_t j = 0; j < r.shape(1); j++)</pre>
 for (py::ssize_t k = 0; k < r.shape(2); k++)
 r(i, j, k) += 1.0;
}, py::arg().noconvert());
```

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

- **and in ()** returns the number of dimensions
- Idea and and an action of the data, respectively, at the given indices. The latter is only available to proxies obtained via
- **Stemsize()** returns the size of an item in bytes, i.e. **Street(T)**.
- **and in ()** returns the number of dimensions.
- **Schange (m)** returns the size of dimension **m**

- **Report** returns the total number of elements (i.e. the product of the shapes).
- Inhoras (1) returns the number of bytes used by the referenced elements (i.e. times (1)).

### **12.2.6 000**

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 (of type can still be used directly.

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

The function 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 DDDD**

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

### **12.3 0000**

# **12.3.1 C++DDPython print**

```
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 <-</pre>
```

#### **12.3.2 □ostream □ □ □ □**

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

// Add a scoped redirect for your noisy code

m.def("noisy_func", []() {
 py::scoped_ostream_redirect stream(
 std::cout,
 py::module_::import("sys").attr("stdout") // Python output
);
 call_noisy_func();
});
```

## Warning

4/8/25, 6:59 AM mdbook-demo kuanghl

// Alternative: Call single function using call guard

m.def("noisy func", &call noisy function.

The redirection can also be done in Python with the addition of a context manager, using the function:

py::add\_ostream\_redirect(m, "ostream\_redirect");

The name in Python defaults to **passed** 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 00000000Python000**

pybind11 provides the and and and 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 , ensuring all lines have common leading indent:

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

4/8/25, 6:59 AM mdbook-demo

# **13. 00000**

kuanghl

### 13.1 0000

```
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)
```

0000000

```
#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
}
```

# **13.2** □ □ Python □ □

00000pybind11 API0000000000012000

```
#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);
}
```

### **12.3** ПППП

Domodule\_::import()

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

```
"""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);
```

### **12.4** ППППППП

```
#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 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

4/8/25, 6:59 AM

mdbook-demo

kuanghl

# Warning

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

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

14. □□

### **14.1** חחחחחחח

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

00000000000

## **14.2** ППППППП**GIL**П

```
class PyAnimal : public Animal {
public:
 using Animal::Animal;
 std::string go(int n_times) {
 py::gil_scoped_acquire acquire;
 PYBIND11 OVERRIDE PURE(
 std::string, /* Return type */
 Animal, /* Parent class */
 go,
 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 {
 py::gil_scoped_release release;
 return call_go(animal);
 });
```

DDDDDD call guard DDDDD call go DDDD

m.def("call\_go", &call\_go, py::call\_guard<py::gil\_scoped\_release>());

### **14.3 ППППППППППП**

```
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 */)</pre>
 .def(py::init<const std::string &>())
 .def("bark", &Dog::bark);
 'et 0000000
 00000000000
 000000000000 Pet
 000000000000
 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);
 000000000000000000
 000000000000
 py::module_::import("basic");
py::class_<Dog, Pet>(m, "Dog")
 .def(py::init<const std::string &>())
 .def("bark", &Dog::bark);
```

### 

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

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

### **14.4** ПППП

```
auto cleanup_callback = []() {
 // perform cleanup here -- this function is called with the GIL held
};
m.add_object("_cleanup", py::capsule(cleanup_callback));
```

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

# **15. DDDD**

# 15.1 "ImportError: dynamic module does not define init function"

- 1. DD **BYSTND::: MODULE** DDDDDDDDDDDDDDDDDDDDDDDDDDDDD.**so**DD

# 15.2 "Symbol not found: \_\_Py\_ZeroStruct \_PyInstanceMethod\_Type"

0015.1

# 15.3 "SystemError: dynamic module not initialized properly"

ПП15.1

# **15.4** 00000 **Python** 000000

0015.1

## **15.5**000000000

```
void increment(int &i)
 i++;
void increment_ptr(int *i)
{
 (*i)++;
}
 def increment(i):
 i += 1 # nope..
 0000000000000000000000000
 int foo(int &i)
 j++;
 return 123;
}
```

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

mdbook-demo 4/8/25, 6:59 AM kuanghl

## 15.6 00000000?

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

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

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

```
import example
example.add(1, 2) # 3
example.sub(1, 1) # 0
```

4/8/25, 6:59 AM	mdbook-demo	kuanghl	
0000000 <mark>init ex</mark> 00	000000000000000000000000000000000000000	100000000000000000000000000000000000000	][]
1. 000000000000000000000000000000000000			
3. 000000000000	000000000000000000000000000000000000000	J00000	
15.7 "recursive to 256"	emplate instantiatio	n exceeded maximum depth o	)f
00000000000000000000000000000000000000		ang DD ftemplate depth-1024 DDDD	][]
	declared with great :member' [-Wattribu	er visibility than the type of it	:s
00000000000000000000000000000000000000		nd1100000000000000000000000000000000000	
		300 <b>                                     </b>	
Python DOD DODDOOD		<b>,</b> ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,	
45 0 000000000000			
15.9	шш <b>?</b>		
00000000000000000000000000000000000000		00000000000000000000000000000000000000	

```
__ZN8pybind1112cpp_functionC1Iv8Example2JRNSt3__16vectorINS3_12basic_strin gIwNS3_
11char_traitsIwEENS3_9allocatorIwEEEENS8_ISA_EEEEEJNS_4nameENS_7siblingENS
9is
methodEA28_cEEEMT0_FT_DpT1_EDpRKT2_
```

### 

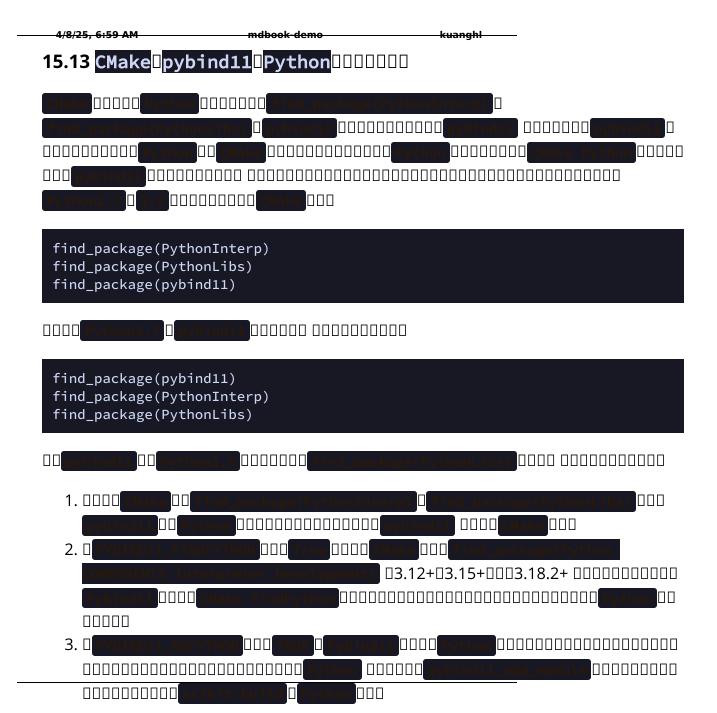
```
pybind11::cpp_function::cpp_function<void, Example2,</pre>
std::__1::vector<std::__1::basic_</pre>
string<wchar_t, std::__1::char_traits<wchar_t>,
std::_1::allocator<wchar_t> >,
std::__1::allocator<std::__1::basic string<wchar t,</pre>
std:: 1::char traits<wchar t>,
std::__1::allocator<wchar_t> > > >&, pybind11::name, pybind11::sibling,
pybind11::is_method, char [28]>(void (Example2::*)
(std:: 1::vector<std:: 1::basic</pre>
string<wchar_t, std::_1::char_traits<wchar_t>,
std::_1::allocator<wchar_t> >,
std:: 1::allocator<std:: 1::basic_string<wchar_t,</pre>
std::__1::char_traits<wchar_t>,
std::_1::allocator<wchar_t> > >&), pybind11::name const&,
pybind11::sibling
const&, pybind11::is_method const&, char const (&) [28])
```



# 15.11 000000000000000Ctrl-C

# 15.12 CMake DDDDDDDDPython DD





## **15.14 ППППППППП**

0000000 BibTeX 00000000 pybind110

```
@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} }
```

4/8/25, 6:59 AM

mdbook-demo

kuanghl

**16.** 🗆 🗆

# **16.1 c/c++**00000

• c 🗆 🗆 🗆

char ca;
unsigned char uca;

• C++000000

vector stl

# pybind11000

**1.** ПППП

## **1.1** חחחחח

 ППстаке
 ПППП

 ППстаке
 ПППП

```
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** ПППП

```
#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");
}
```

### 1.2.1

00**6///::----**00000000000Python00000000000000000000000000000

```
// 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 DDDD

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

### 1.2.2

```
// 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

#### 

```
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");
```

### 

```
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");
```

### 1.3

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

PythonMMMMMM

'``pyhton
>>> import example
>>> example.the_answer
42
>>> example.what
'World'
```

## **1.4 ППППППП**

0000000C++0000000**Pat**000000

```
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;
};
```

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

## 1.4.1

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

## 

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

# 

```
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;
};
```

## DDDDDDDDsetterDgeterDDD

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

# 0000000read00000nullptr0000

### 1.4.2

## 

```
>>> class Pet:
 name = "Molly"
 >>> p = Pet()
 >>> p.name = "Charly" # overwrite existing
 >>> p.age = 2 # dynamically add a new attribute
lannananananananana
 >>> p = example.Pet()
>>> p.name = "Charly" # OK, attribute defined in C++
>>> p.age = 2 # fail
 AttributeError: 'Pet' object has no attribute 'age'
00C++0000000000000
 0000000
 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}
```

### 1.4.3

### 

```
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** חחחחחח

```
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();
```

# 1.6 □□\*args□\*\*kwargs□□

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

0000000pybind110000000

### **2.** ПППППП

#### 2.1 00000

```
/* 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</pre>
```

 DPython
 DD
 <t

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



ППП

#### 2.2

## □□□keep alive□

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

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

```
Note: Reep alive Boost.Python Downth custodian and ward Dwith custodian and ward postcall DDD
```

#### Call guard

call\_guard(12)00000T000Scope guard000000000000000

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

<del>AAAAAAAAA</del>

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

000000T0000000 git\_scoped\_release 0000000000



# 2.3 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, b=2) # good
 f(1, 2) # TypeError: f() takes 1 positional argument but 2 were given
pybind11000 py::kw.only
```

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

## 2.4 Positional-only□□

python3.8000Positional-only00000pybind1100

# 2.5 Non-converting

- DD py::implicitly\_convertible<A,B>() DDDDDD
- DDDDDDDfloatDDDDD<mark>std::complex<float></mark>DDDDD
- Calling a function taking an Eigen matrix reference with a numpy array of the wrong type or of an incompatible data layout.

```
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"));
```

DDDDDDDDDD TypeError DDD

```
>>> 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
```

### **3.** 00000

#### **3.1** ППППП

```
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!"; }
};
```

```
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!'
```

#### 

```
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!'
```

# **3.2** 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)</pre>
 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);
```

00000000000000Animal0000000

```
class PyAnimal : public Animal {
 public:
 using Animal::Animal;
 std::string go(int n_times) override {
 PYBIND11_OVERRIDE_PURE(
 std::string, /* Return type */
 Animal,
 go,
 n_times
);
 };
|0000000000000000
 0000000000C0000
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,
```

);

```
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);
}
```

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

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

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

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

def bark(self):
 return "yap!"
```

#### 3.3 000000

```
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!"; }
};</pre>
```

```
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,); }
};
```

```
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
 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,); }
};
```

00000pybind11000000

```
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
```

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

#### 3.4

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

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

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

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

#### 3.5

ППППППАПВПППППАППППППППВП

 ППВППП

00000B0000A000000000000000000000Python000000000

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

#### **3.6** ППППП

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

#### **3.7 DDDDD**

 Description
 Description

```
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);
```

#### 3.8 ПППП

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

DDPythonDDDDDDC++DDDDDDDC++DDPythonDD

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

# **3.9 protected 1**

□□□□□□Python□□protected □□□□□

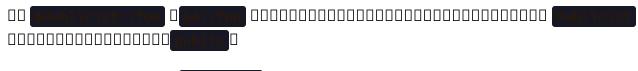
```
class A {
protected:
 int foo() const { return 42; }
};

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

```
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
```



```
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</pre>
 .def("foo", &Publicist::foo); // <-- `Publicist` here, not</pre>
```

# **3.10** □ □ final □

```
class PyFinalChild(IsFinal):
 pass

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

4/8/25, 6:59 AM

mdbook-dem

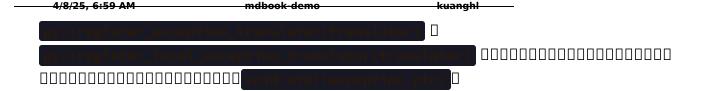
kuanghl

## 4. 0000

# **4.1 C++**00000**Python**0000

Exception thrown by C++	Translated to Python exception type
<pre>std::exception</pre>	RuntimeError
<pre>std::bad_alloc</pre>	MemoryError
<pre>std::domain_error</pre>	ValueError
<pre>std::invalid_argument</pre>	ValueError
<pre>std::length_error</pre>	ValueError
std::out_of_range	IndexError
std::range_error	ValueError
<pre>std::overflow_error</pre>	<pre>OverflowError</pre>
<pre>pybindl1::stop_iteration</pre>	(used to implement custom iterators)
<pre>pybind11::index_error</pre>	(used to indicate out of bounds access in <b>Exercises</b> ), <b>Exercises</b> , etc.)
pybindl1::key_error	(used to indicate out of bounds access in in dict-like objects, etc.)
pybind11::value error	passed in <b>Container</b> (used to indicate wrong value
<pre>pybind11::type_error</pre>	TypeEnron

4/ <del>8/25, 6:59 AM</del>	mdbook-demo	<u>kuanghl</u>		
Exception thrown b	y C++	Translated to Python exception type		
pybind11::buffer_er	or B	ufferError		
<pre>pybind11::import_er</pre>	or I	mportError		
pybind11::attribute	error	ttributeError		
Any other exception		untimeError		
00000000000000000000000000000000000000				
<pre>pybind11::error_alread</pre>	${\sf y\_set}$			
00000000000000000000000000000	rthon[][][][han-	ndle::call()		
<b>4.2</b> 00000000				
	nd11ПППППППП	0000000000000pybind11 class00000000000		
		1C++0Python0000000000000000		
py::register_exception	n <cppexp>(mod</cppexp>	dule, "PyExp");		
00000000000000000000PyE	xp[]Python[][[	00000CppExp00000000PyExp000		
000000000000000000000000000000000000000				
py::register_local_ex	ception <cppex< th=""><th>(module, "PyExp");</th></cppex<>	(module, "PyExp");		
00000000handle000000	3000			
		. <u> </u>		
		<pre>dule, "PyExp", PyExc_RuntimeError); xp&gt;(module, "PyExp", PyExc_RuntimeError);</pre>		
	D			
	kuntimeError⊔	J		
Python DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD				



# **9.3 C++DPython**

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

```
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;
 }
}
```

```
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 DPython C API**

```
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");
```

OOOOO PyErr\_Clear OOOOOO

#### 9.5 ∏∏unraiseable∏∏

```
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.** ПППП

6. python C++□□

**7.** ПП

#### **7.1** 00000000

#### **7.2** חחחחחחח**GIL**ח

```
class PyAnimal : public Animal {
public:
 using Animal::Animal;
 std::string go(int n_times) {
 py::gil_scoped_acquire acquire;
 PYBIND11_OVERRIDE_PURE(
 std::string, /* Return type */
 Animal, /* Parent class */
 go,
 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 {
 py::gil_scoped_release release;
 return call_go(animal);
 });
}
```

000000 call\_guard 00000 call\_go 0000

m.def("call\_go", &call\_go, py::call\_guard<py::gil\_scoped\_release>());

# □□□**□Mermaid**□□□

#### 

- 000000000 Mermaid 000;
- 000000000 Mermaid 000;
- 00 **61 thook** 0000000000.

## $\Pi\Pi\Pi$ **Mermaid** $\Pi\Pi\Pi$

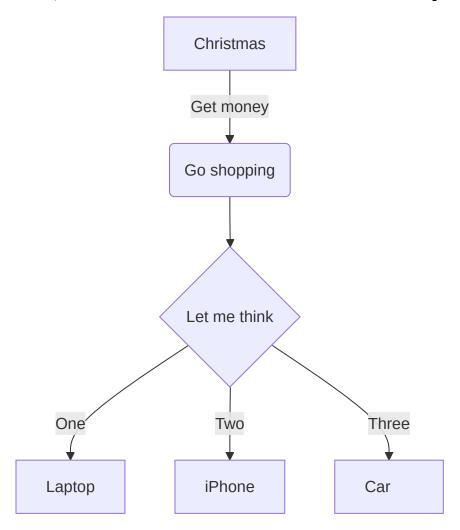
```
- XXXX
- XXXX
- XXXX
```



ПП

```
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]
```

ПП

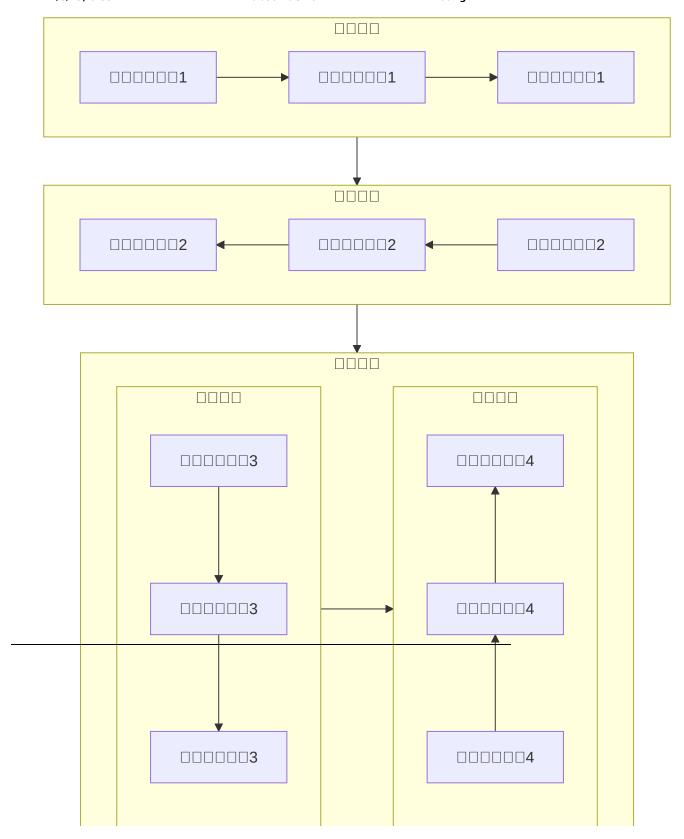


- DDDD: https://github.com/mermaid-js/mermaid
- DDDD: https://mermaidjs.github.io/mermaid-live-editor/
- DDDD: https://mermaid-js.github.io/mermaid/#/flowchart
- DDDD: https://mermaid.nodejs.cn/syntax/flowchart.html

1/9/25 6:50 AM	mdhaak-dama	kuanahl
7/0/23: 0:33 AM	mabook acmo	Kuunum

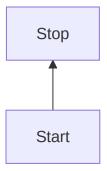
# **Mermaid**

- + TB
- + BT
- + LR
- + RL





graph BT Start --> Stop



• LR

 $\square\square\square\square$ : from **L**eft to **R**ight

graph LR Start --> Stop



• RL

□□□□: from **R**ight to **L**eft

```
graph RL
Start --> Stop
```

ПП



```
+ [🛛 🖺
 - [[XXXX]]
 - [(⊠⊠)]
 - [/\lambda \lambda \lambda
 - [/⊠⊠\]
 - [\⊠⊠/]
 + (🗆 🗆 🗎
 - ((⊠⊠))
 - ([⊠⊠⊠])
 + {□□}
 - {{\text{\ti}\}\\ \text{\tin}\tint{\text{\ticr{\tiin}\tint{\ti}\ti}\\\tict{\text{\text{\text{\text{\text{\ti
 - {[XXXX]}
 - {(\(\text{\overline{\to}\end{\overline{\to}\end{\overline{\to}\end{\overline{\to}\end{\overline{\to}\end{\overline{\to}\end{\overline{\to}\overline{\to}\end{\overline{\overline{\overline{\to}\end{\overline{\o
 + > 🗆 🗆 🗆
```



mdbook-demo

kuanghl -

00: 00000000,000000.

graph TD id

id

• 🗆

DDD: [node description] ,[] DDDDDDDDDDD, node description DDDDDDD.

mdbook-demo

kuanghl

graph LR
 id1[This is the text in the box]

This is the text in the box

• 0000

0000: (node description) ,() 00000000000, node description 0000000

graph LR
 id1(This is the text in the box)

This is the text in the box



mdbook-demo

kuanghl

```
graph LR
 id1([This is the text in the box])
```

This is the text in the box

• 🛛

```
0000: ((node description)) , () 00000 () 0000000000, node
```

```
graph LR
id1[(Database)]
```



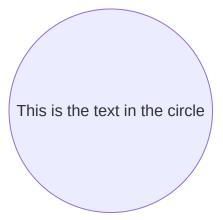
• 🛛

```
0000: ((node description)), () 00000 () 000000000, node description
```

mdbook-demo

kuanghl

graph LR
 id1((This is the text in the circle))

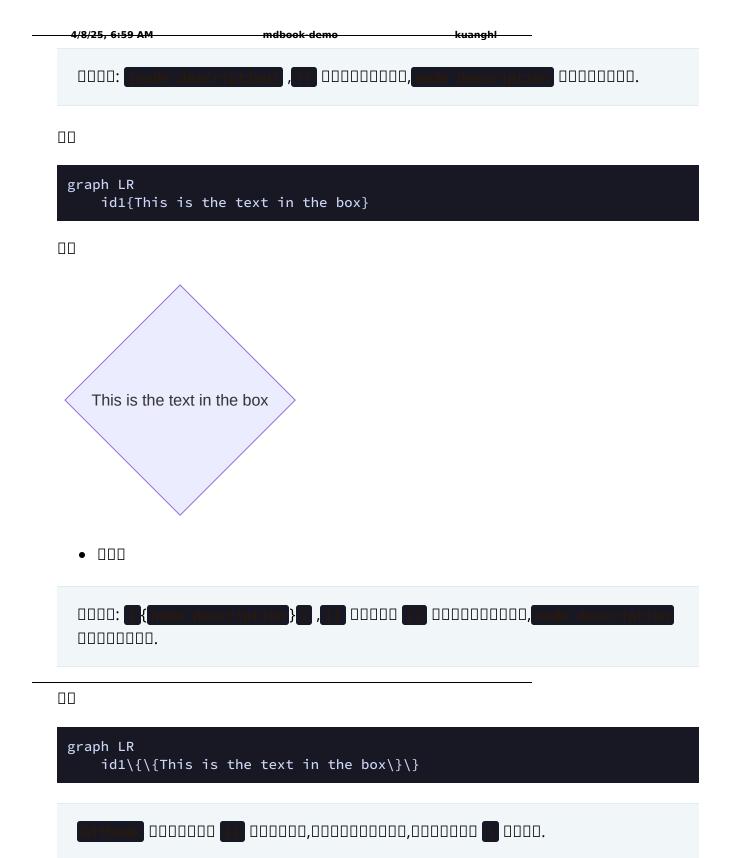


• 00000

graph LR
 id1>This is the text in the box]

This is the text in the box

• 🛛 🗎



4/8/25, 6:59 AM mdbook-demo kuanghl This is the text in the box • 00000 0000:1 00000 加 0000000000000, 00000000. graph TD id1[/This is the text in the box/] This is the text in the box • 00000

graph TD id1[\This is the text in the box\]

This is the text in the box

• 🛮 🗎

```
graph TD
A[/Christmas\]
```

Christmas

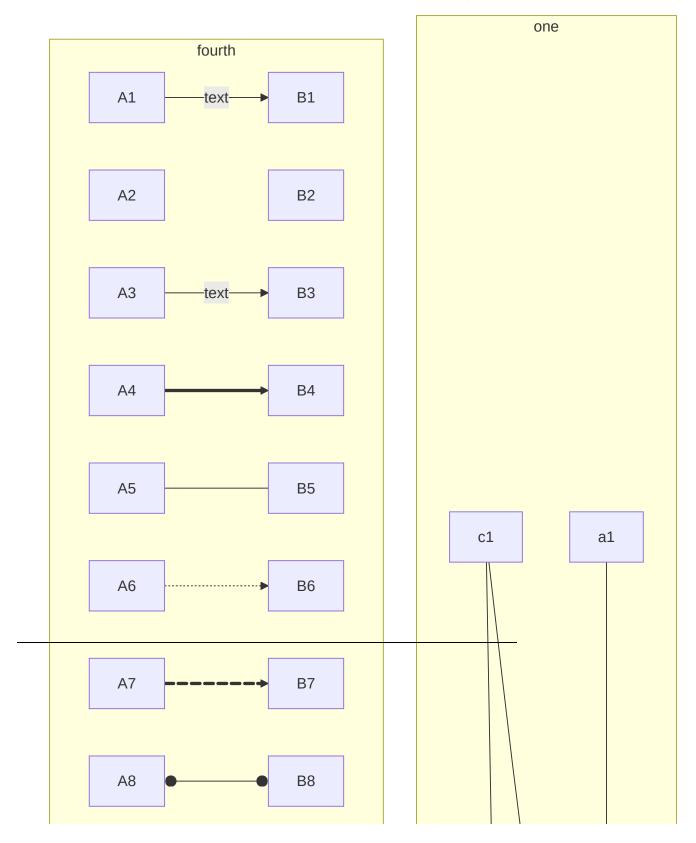
• 00000

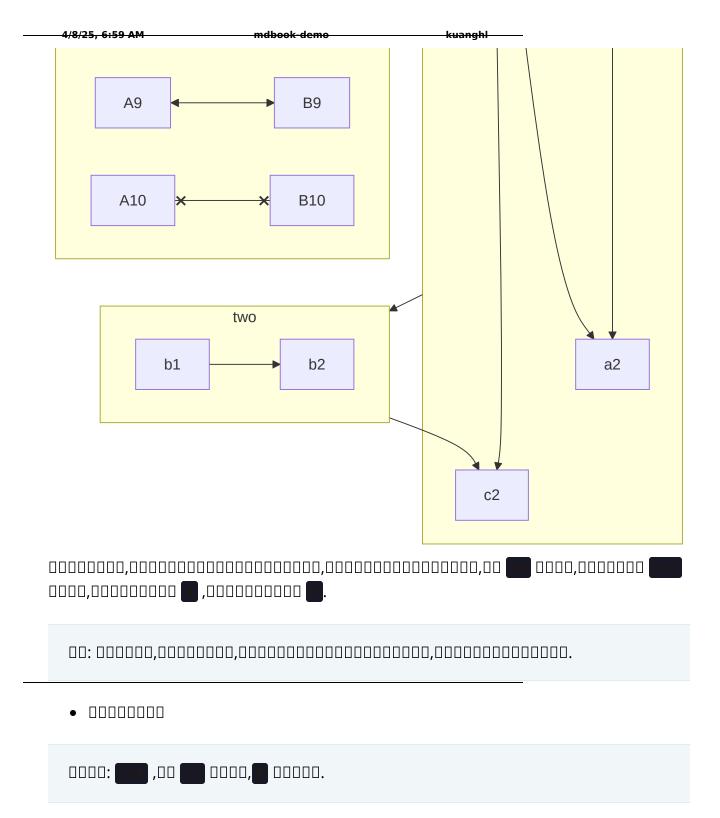
```
graph TD
B[\Go shopping/]
```

Go shopping

4/8/25, 6:59 AM	mdbook-demo	kuanghl
100		
100		

```
+ 🛛 🖂 / 🖺 🖾
+ 🛛 🖂 🗸 🗸 🖂 🖂
+ 🗆 🗆 🗆 🗎
 – ⋈⋈
 + --888
 + | 🛛 🖂 🖂 🗎
 — ☒☒
 + -.000
 + | 🛮 🗆 🖂 🗎
+ 🛛
+ 🛛 🖺 🖂
 - -->
 - -.->
 - -.-
 + --\\\\\
 + -->|\|\|\|\|\|
 + ---|\\\\\
 + -.\\\\\-.->
 + -.->|\\\\\|
 + -.\\\\\-.-
 + -.-|\\\\
 + ==> | 🛛 🖂 🖂 🗎
 - \square
 + ==\\\\\\
 + ===|\\\\\\\\\\
```





graph LR A-->B



• 00000

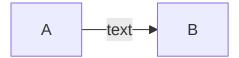


graph LR A --- B

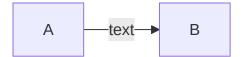


• 00000000

graph LR A-- text -->B 4/8/25, 6:59 AM mdbook-demo kuanghl



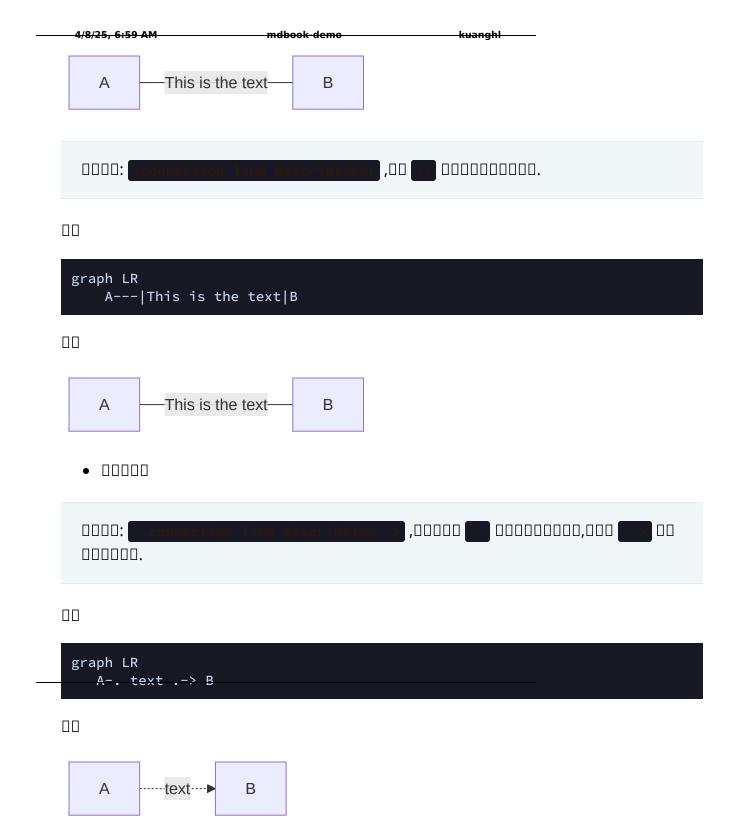
graph LR A-->|text|B



• 00000000

0000: --connection line description ,00000 -- 00000000,000 -- 00000

graph LR A-- This is the text ---B



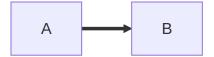
mdbook-demo

kuanghl

• 000000

0000: ,000000.

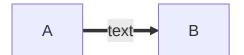
graph LR A ==> B



• 000000000



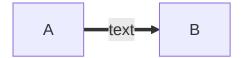
graph LR
A == text ==> B



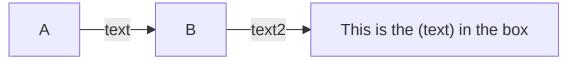
• 0000000000

4/8/25, 6:59 AM mdbook-demo kuanghl

graph LR
A ==>|text| B





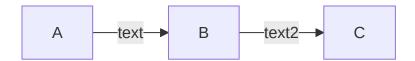


• 000000

mdbook-demo

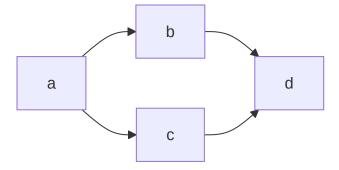
kuanghl

graph LR
A -- text --> B -- text2 --> C



• 000000





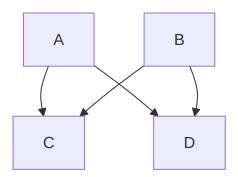
<del>• 000000</del>



graph TB A & B--> C & D

mdbook-demo

kuanghl



• 00000000

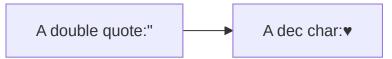
```
graph LR
 id1["This is the (text) in the box"]
```

This is the (text) in the box

• 00000000

```
graph LR
A["A double quote:#quot;"] -->B["A dec char:#9829;"]
```

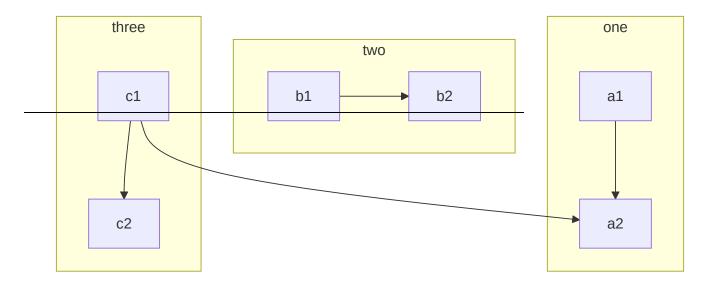
4/8/25, 6:59 AM mdbook-demo kuanghl



• 000000

```
subgraph title
graph definition
end
```

```
graph TB
 c1-->a2
 subgraph one
 a1-->a2
 end
 subgraph two
 b1-->b2
 end
 subgraph three
 c1-->c2
 end
```



mdbook-demo

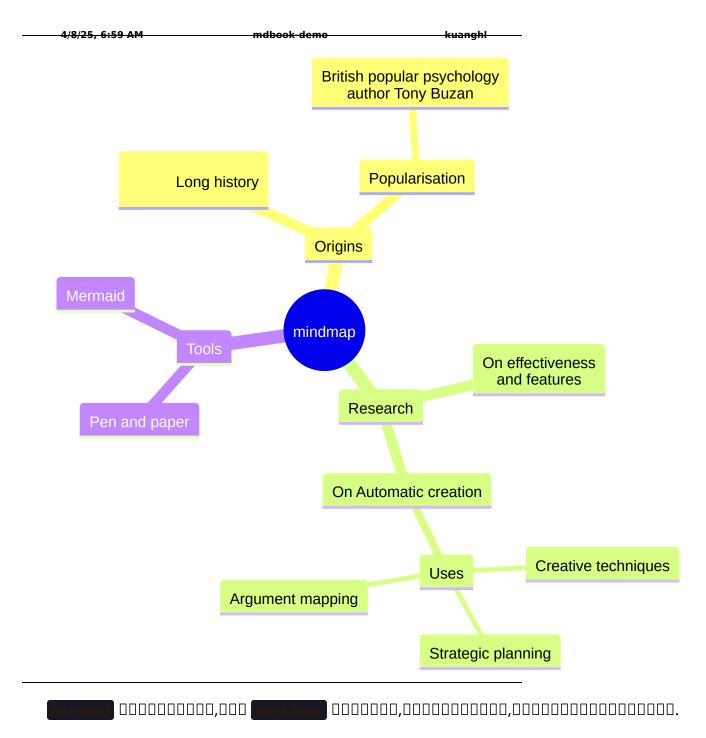
kuanghl

• 0000

000 988 0000000.







<del>- 4/8/25, 6:59 AM</del>

mdbook-dem

kuanghl

00	00
graph	graph 000000
subgraph	subgraph 0000000
top	000000000000000000000000000000000000000
bottom	<b>87</b> 0 <b>88</b> ,0000000000000
left	
right	<b></b> 0 <b></b> ,0000000000000

• 0000

	00	00	
	0000		
<b>{}</b>	00	0000	_ 00
<>			
		00000	
		00000	
	0000	00000	
= :	0000	00000	

4/8/25, 6:59 AM mdbook-demo kuanghl

000	00	00	
>	000	00000	
		00000	
	00000000	000000	
	0000000000	000000	
-	0000000000	000000	
	00000000000000	000000	
=:	0000000000000	000000	

### • 0000

000	00		
	000	0000	
	000		
	000	0000	
	00		
	00	0000	
	000		
	0000	0000	
	00	0000	
——»	00000	00000	
	00000	00000	
	00000	00000	00

4/8/25, 6:59 AM	mdbook-demo	kuanghl		
000		00	00	
-, ->		00000	00000	
>		00000	00000	
		00000	00000	
		00000	00000	
==>		000000		
		000000	00000	
=;>		000000	00000	
	1	000000	00000	
		000000	00000	
		000000	00000	
www.		00000000	000000	
connection line	e description-	00000000000000000000000000000000000000	000000	
connecti	on line	00000000000000000000000000000000000000	000000	
connection line	description	00000000000000000000000000000000000000	000000	
connecti	on line	000000000000	000000	
connecti	on line	000000000000	000000	
-:connecti	on line	000000000000	000000	00
==connecti descripti	on line on===	000000000000	00000	

4/8/25, 6:59 AM mdbook-demo kuanghl

000	00	00	
=:connection line	0000000000000	000000	
description=:=	000		

### 

□□□□: https://mermaid-js.github.io/mermaid/#/flowchart?id=styling-and-classes

- DDDD Interaction: https://mermaid-js.github.io/mermaid/#/flowchart? id=interaction
- DDDD Styling and classes: https://mermaid-js.github.io/mermaid/#/flowchart?id=interaction
- DDDD Basic support for fontawesome: https://mermaidjs.github.io/mermaid/#/flowchart?id=basic-support-for-fontawesome
- DDDD 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.

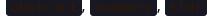
### **Directives**

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

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.



### Info

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.



### 

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.



### **Success**

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.



### Question

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.





### Warning

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.







Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.



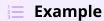
### **Danger**

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.



Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.



### 77 Quote

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.



### Info

A beautifully styled message.

### Un example

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.

### Une note

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.

### Un warning

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.

### Collapsing note

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.

### Le javascript c'est yolo préférez Typescript

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.

### Referencing and dereferencing

The opposite of *referencing* by using is *dereferencing*, which is accomplished with the dereference operator,

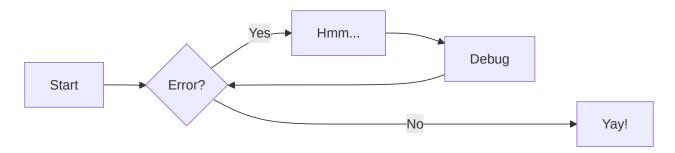


This syntax won't work in Python 3:

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.



### 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.

4/8/25, 6:59 AM mdbook-demo kuanghi

### 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**

HTML:

$$e^{i\theta} = \cos \theta + i \sin \theta$$

$$\Rightarrow x + iy = re^{i heta}$$

Markdown (requires mahook kates

$$\oint_C f(x,y) \, \mathrm{d}A$$

Inspect element and use tab (under tab (under on Firefox) to check that all CSS and fonts are properly loaded from GitHub pages instead of external CDN.

▼ Proof that 
$$e^{ix} = \cos x + i \sin x$$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \implies e^{ix} = \sum_{n=0}^{\infty} \frac{(ix)^n}{n!}$$

$$\cos x = \sum_{m=0}^{\infty} \frac{(-1)^m x^{2m}}{(2m)!} = \sum_{m=0}^{\infty} \frac{(ix)^{2m}}{(2m)!}$$

$$\sin x = \sum_{s=0}^{\infty} \frac{(-1)^s x^{2s+1}}{(2s+1)!} = \sum_{s=0}^{\infty} \frac{(ix)^{2s+1}}{i(2s+1)!}$$

$$\cos x + i \sin x = \sum_{l=0}^{\infty} \frac{(ix)^{2l}}{(2l)!} + \sum_{s=0}^{\infty} \frac{(ix)^{2s+1}}{(2s+1)!} = \sum_{n=0}^{\infty} \frac{(ix)^n}{n!}$$

$$= e^{ix}$$

Fourier Transform:

$$f(t) = \int_{-\infty}^{\infty} F(\omega) i^{4t\omega} d\omega \ F(\omega) = \int_{-\infty}^{\infty} f(t) i^{-4t\omega} dt$$

Pauli Matrices:

$$egin{aligned} \sigma_1 &= egin{pmatrix} 0 & 1 \ 1 & 0 \end{pmatrix} \ \sigma_2 &= egin{pmatrix} 0 & -i \ i & 0 \end{pmatrix} \ \sigma_3 &= egin{pmatrix} 1 & 0 \ 0 & -1 \end{pmatrix} \end{aligned}$$

### kroki

4/8/25, 6:59 AM

```
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]
```