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.

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

- пппппппппппп
- ППППП
- 0000000
- 000
- 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

- 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 3 🛮 🗘
- 4. Intel classic C++ compiler 18 or newer (ICC 20.2 tested in CI)
- 5. Cygwin/GCC (previously tested on 2.5.1)
- 6. NVCC (CUDA 11.0 tested in CI)
- 7. NVIDIA PGI (20.9 tested in CI)

1.4 □□

This project was created by Wenzel Jakob. Significant features and/or improvements to the code were contributed by Jonas Adler, Lori A. Burns, Sylvain Corlay, Eric Cousineau, Aaron Gokaslan, Ralf Grosse-Kunstleve, Trent 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.



3. 0000

3.1 000000000

000000Git000000pybind110000000000000git0000000000pybind110

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

3.2 DDPyPIDD

DDDDDpipDDDPyPIDDDPybind11DPythonDDDDDDDDDDDCMakeDDDDDD

```
pip install pybind11
```

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

pyproject.toml [][][]

3.3 □□conda-forge□□

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

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

3.4 □ □ vcpkg □ □

DDDDDMicrosoft vcpkgDDDDDDDDDDDDDpybind11D

```
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. 0000 First steps

00000pybind11000000000000000000pybind1100000000

4.1 חחחחח

Linux/macOS

0000000000000000000

```
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

```
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");
}
```

```
c++ -03 -Wall -shared -std=c++11 -fPIC $(python3 -m pybind11 --includes)
example.cpp -o example$(python3-config --extension-suffix)
```

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

4.4 00000

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

```
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;
}
```



```
>>> 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

```
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** חחחחחחחחח

```
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);
}
```

```
>>> 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** חחחחחחחחח

000040000000000000000000004000000

#### **5.3** חחחחחח

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

0000000Python000000

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

pybind1100000000lambda0000lambda0000 [] 00000000

#### **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;
};
```

0000 class\_::def\_property() (000000 class\_::def\_property\_readonly() )0000000

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

0000000read00000nullptr0000

#### 5.5

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

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

```
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** ППППППП

#### 

```
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!'
```

#### 

```
// MMMMMMMMMMMMM
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");
```

#### 

00000000C++1400000000000000000000

```
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");
```

```
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_));
```

py::detail::overload cast impl [][]

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

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

## **5.8 000000**

```
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);
```

0000000 Kind 0 Attributes 0 Pet 00000000000 enum\_ 0 class\_ 0000000 Pet class\_ 000 enum\_::export\_values() 0000000000C++1100000000000

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

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

```
>>> 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

```
/* 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
 \_data
 DPython
 DD
 DD

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

00000	00
return_value_policy::take_ownership	00000000000000000000000000000000000000
return_value_policy::copy	00000000Python000000000000000000000000000000000000
return_value_policy::move	00 std::move 000000000000000000000000000000000000
return_value_policy::reference	00000000000000C++0000000 00000000000000
return_value_policy::reference_internal	<pre>000000000000000000000000000000000000</pre>
return_value_policy::automatic	00000000000000000000000000000000000000
return_value_policy::automatic_reference	<pre>000000000000000000000000000000000000</pre>

ППП

#### **7.2** חחחחחחח

## □□□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: keep_alive \(\text{Boost.Python} \(\text{D} \) with_custodian_and_ward \(\text{D} \) with_custodian_and_ward_postcall \(\text{D} \) \(\text{D} \)
```

### **Call guard**

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

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

000000T000000 gil\_scoped\_release 000000000000

```
See also: test/test_call_policies.cpp DDDDDDDDDD keep_alive D call_guard DDDD
```

## **7.3 Python 0 0 0 0**

□Python

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

## **7.4 \( \Bargs \Bargs**

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

0000000pybind1100000000

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

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

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

```
000000 test/test_kwargs_and_defualts.cpp 0
```

#### **7.5 ПППППП**

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

000000000SomeType00000binding0000py::class\_000000000

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

## 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 py::kw\_only 0000000000

000000000 py::args 00000

## 7.7 Positional-only

python3.8000Positional-only00000pybind1100 py::pos\_only() 000000000

000000000000 a 00000000keyword-only000000

## 7.8 Non-converting□□

- DD py::implicitly\_convertible<A,B>() DDDDDD
- ппппппппппппппппппп
- DDDDDDDfloatDDDDD std::complex<float> 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"));
```

00000000000 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
```

#### **7.9** חח/חחחחח

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

□□□Python□□ bark(None) □□□ "(no dog)" □□□ meow(None) □□□□□ TypeError □

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

Invoked with: None
```

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

### **7.10 00000**

00000000000000000 TypeError 0

# **8.** $\Box$

# **8.1 Python 0 0 0 0 0**

000000000C++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

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

```
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 class\_ 00000000PyAnimal000000Python000Animal00

```
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! '
```

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

\_\_\_\_pybind11\_\_\_\_\_

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

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

#### 8.3 חחחחחחחח

#### 8.3.1

**See also** See the file tests/test\_virtual\_functions.cpp for complete examples showing both normal and forced trampoline instantiation.

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

### 8.4 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
 // Factory function - returned by value:
 static Example create(int a) { return Example(a); }
 // These constructors are publicly callable:
 Example(double);
 Example(int, int);
 Example(std::string);
};
py::class_<Example>(m, "Example")
 // Bind the factory function as a constructor:
 .def(py::init(&Example::create))
 // Bind a lambda function returning a pointer wrapped in a holder:
 .def(py::init([](std::string arg) {
 return std::unique_ptr<Example>(new Example(arg));
 }))
 // Return a raw pointer:
 .def(py::init([](int a, int b) { return new Example(a, b); }))
 // You can mix the above with regular C++ constructor bindings as
well:
 .def(py::init<double>())
```

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

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

py::class_<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**

Note: pybind11000C++000000 noexcept(false) 0

### **8.7 0000**

#### 000A0B0000A0000000B0

00000B0000A000000000000000000000Python0000000000

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

INDEPENDENCE OF THE SAME TYPE (i.e. from A to B) 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(py::self \* float()) 0000000000

```
.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;
};
```

```
py::class_<Pickleable>(m, "Pickleable")
 .def(py::init<std::string>())
 .def("value", &Pickleable::value)
 .def("extra", &Pickleable::extra)
 .def("setExtra", &Pickleable::setExtra)
 .def(py::pickle(
 [](const Pickleable &p) { // __getstate__
 /* Return a tuple that fully encodes the state of the object
*/
 return py::make_tuple(p.value(), p.extra());
 },
 [](py::tuple t) { // __setstate__
 if (t.size() != 2)
 throw std::runtime_error("Invalid state!");
 /* Create a new C++ instance */
 Pickleable p(t[0].cast<std::string>());
 /* Assign any additional state */
 p.setExtra(t[1].cast<int>());
 return p;
 }
));
```

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

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

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

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

### 8.11

```
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")
...
```

DDPythonDDDDDDDC++DDDDDDDC++DDPythonDD

```
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'
```

```
// dogs.cpp

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

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

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

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

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

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

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

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

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

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

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

The localization of the bound types is actually tied to the shared object or binary generated by the compiler/linker. For typical modules created with PYBIND11\_MODULE(), this distinction is not significant. It is possible, however, when Embedding the interpreter to embed multiple modules in the same binary (see Adding embedded modules). In such a case, the localization will apply across all embedded modules within the same binary.

## **8. 14** □□protected□□□□

□□□□□□Python□□protected □□□□□

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

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

```
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>
`Trampoline`!
```

# **8.15** □ □ final □

```
class IsFinal final {};
py::class_<IsFinal>(m, "IsFinal", py::is_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) {
 // note that src may be nullptr
 if (src && src->kind == PetKind::Dog) {
 type = &typeid(Dog);
 return static_cast<const Dog*>(src);
 }
 return src;
 }
 };
} // namespace pybind11
```

When pybind11 wants to convert a C++ pointer of type Base\* to a Python object, it calls polymorphic\_type\_hook<Base>::get() to determine if a downcast is possible. The get() function should use whatever runtime information is available to determine if its src parameter is in fact an instance of some class Derived that inherits from Base. If it finds such a Derived, it sets type = &typeid(Derived) and

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

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

### **8.17** חחחחחח

0000000C++000000000

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

# 9. □□

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

 DPython
 DD
 <t

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

Exception thrown by C++	Translated to Python exception type
pybind11::value_error	ValueError (used to indicate wrong value passed in container.remove())
<pre>pybind11::type_error</pre>	TypeError
<pre>pybind11::buffer_error</pre>	BufferError
<pre>pybind11::import_error</pre>	ImportError
<pre>pybind11::attribute_error</pre>	AttributeError
Any other exception	RuntimeError

pybind11::error\_already\_set []

### 9.2

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

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

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

DDDPyExpDDDDDDDPyExpDRuntimeErrorD

#### 

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

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

The following example demonstrates this for a hypothetical exception classes

MyCustomException and OtherException: the first is translated to a custom python

exception MyCustomError, while the second is translated to a standard python

RuntimeError:

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

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

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

## 9.3 Local vs Global Exception Translators

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

If module1 has the following translator:

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

and module2 has the following similar translator:

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

then which translator handles the invalid\_argument will be determined by the order that module1 and module2 are imported. Since exception translators are applied in the reverse order of registration, which ever module was imported last will "win" and that translator will be applied.

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

Changing the previous example to use register\_local\_exception\_translator would mean that when invalid\_argument is thrown in the module2 code, the module2 translator will always handle it, while in module1, the module1 translator will do the same.

## **9.4 C++DPython**

<b>Exception raised in Python</b>	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");
```

00000 PyErr\_Clear 00000

00Python0000000000000Python/pybind1100000000

### 9.6 □□□□raise from□

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

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

### 9.7 ∏∏unraiseable∏∏

000000 \_\_del\_\_ 00000Python0000000Python00unraisable000000Python 3.8+00000system hook0000auditing event000

```
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);
}
```

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

#### 

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

2. | Child | Child | Std::enable\_shared\_from\_this<T> | Child |

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

### 10.3

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

ПППППППППППППboolПППППППППfalseП

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

DDDDDDDDDD General notes regarding convenience macrosD

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

000000pybind110000 SmartPtr 00 .getPointer() 00 .get() 000

DD 74 / 215 BSN

# **11.** 0000

- 1. 0000000C++000000000000pybind11000000Python000000
- 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
```

## 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, uint8_t	8-bit integers	pybind11/pybind11.h
int16_t, uint16_t	16-bit integers	pybind11/pybind11.h
int32_t, uint32_t	32-bit integers	pybind11/pybind11.h
int64_t, uint64_t	64-bit integers	pybind11/pybind11.h

Data type	Description	Header file
ssize_t, size_t	Platform- dependent size	pybind11/pybind11.h
float, double	Floating point types	pybind11/pybind11.h
bool	Two-state Boolean type	pybind11/pybind11.h
char	Character literal	pybind11/pybind11.h
char16_t	UTF-16 character literal	pybind11/pybind11.h
char32_t	UTF-32 character literal	pybind11/pybind11.h
wchar_t	Wide character literal	pybind11/pybind11.h
const char *	UTF-8 string literal	pybind11/pybind11.h
const char16_t *	UTF-16 string literal	pybind11/pybind11.h
const char32_t *	UTF-32 string literal	pybind11/pybind11.h
<pre>const wchar_t *</pre>	Wide string literal	pybind11/pybind11.h
std::string	STL dynamic UTF-8 string	pybind11/pybind11.h
std::u16string	STL dynamic UTF-16	pybind11/pybind11.h

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	pybind11/pybind11.h
std::pair <t1, t2=""></t1,>	Pair of two custom types	pybind11/pybind11.h
std::tuple<>	Arbitrary tuple of types	pybind11/pybind11.h
<pre>std::reference_wrapper&lt;&gt;</pre>	Reference type wrapper	pybind11/pybind11.h
std::complex <t></t>	Complex numbers	pybind11/complex.h
<pre>std::array<t, size=""></t,></pre>	STL static array	pybind11/stl.h
std::vector <t></t>	STL dynamic array	pybind11/stl.h
std::deque <t></t>	STL double- ended queue	pybind11/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
<pre>std::unordered_map<t1, t2=""></t1,></pre>	STL unordered map	pybind11/stl.h
std::set <t></t>	STL ordered set	pybind11/stl.h
<pre>std::unordered_set<t></t></pre>	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
std::chrono::time_point<>	STL date/time	pybind11/chrono.h
Eigen::Matrix<>	Eigen: dense matrix	pybind11/eigen.h
Eigen::Map<>	Eigen: mapped	pybind11/eigen.h

Data type	Description	Header file
	memory	
	Eigen:	
<pre>Eigen::SparseMatrix&lt;&gt;</pre>	sparse	pybind11/eigen.h

# 11.2 Strings, bytes and Unicode conversions

```
Note: <code>00000string0000Python3 strings000python2.7000 unicode 00 str 0 str 0 bytes 0Python2.7000000 from __future__ import unicode_literals 000000 str 00 unicode 0</code>
```

## 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++000000000000000const0000000

## □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
```

00UTF-8000ASCII000000ASCII0000Python00000000000000000000UTF-80

ПППП

```
// 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'
```

DDDpybind11000bytes0000000 std::string DD0000000000 std::string Dbytes0

```
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

```
#define UNICODE
#include <windows.h>
m.def("set_window_text",
 [](HWND hwnd, std::wstring s) {
 // Call SetWindowText with null-terminated UTF-16 string
 ::SetWindowText(hwnd, s.c_str());
 }
);
m.def("get_window_text",
 [](HWND hwnd) {
 const int buffer_size = ::GetWindowTextLength(hwnd) + 1;
 auto buffer = std::make_unique< wchar_t[] >(buffer_size);
 ::GetWindowText(hwnd, buffer.data(), buffer_size);
 std::wstring text(buffer.get());
 // wstring will be converted to Python str
 return text;
 }
);
```

000000000Shift-JIS00000UTF-8/16/3200000Python0

#### **11.2.4** ПППП

Unicode

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

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

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

0000008-bit000000 int8\_t 0 uint8\_t 0000000

## 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 std::u16string\_view argument will be passed UTF-16-encoded data, and a returned std::string\_view will be decoded as UTF-8).

### **11.3 STL**

### 11.3.1

### 11.3.2 C++17 0 0 0

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

```
// `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:

```
// `boost::variant` as an example -- can be any `std::variant`-like
container
namespace pybind11 { namespace detail {
 template <typename... Ts>
 struct type_caster<boost::variant<Ts...>> :
variant_caster<boost::variant<Ts...>> {};
 // Specifies the function used to visit the variant -- `apply_visitor`
instead of `visit`
 template <>
 struct visit_helper<boost::variant> {
 template <typename... Args>
 static auto call(Args &&...args) ->
decltype(boost::apply_visitor(args...)) {
 return boost::apply_visitor(args...);
 }
 };
}} // namespace pybind11::detail
```

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

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

def\_readonly 0000STL000000

```
/* ... 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>);
```

```
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 **DISTI DI**

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

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

// ...

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

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

## **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;
 };
}
```

```
#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

#### 

### 11.5.2

# C++ Python

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

# Python C++

- ullet datetime.datetime or datetime.date or datetime.time ullet
  - std::chrono::system\_clock::time\_point
- datetime.timedelta → std::chrono::duration
- datetime.timedelta → std::chrono::[other\_clocks]::time\_point
- float → std::chrono::duration
- float → std::chrono::[other\_clocks]::time\_point

# 11.6 Eigen

0000**E**igen000000

## **11.7** חחחחחחח

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

```
struct inty { long long_value; };

void print(inty s) {
 std::cout << s.long_value << std::endl;
}</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 pybindl1::detail::type\_caster<T> template. Although this is an implementation detail, adding an instantiation of this type is explicitly allowed.

```
namespace pybind11 { namespace detail {
 template <> struct type_caster<inty> {
 public:
 /**
 * This macro establishes the name 'inty' in
 * function signatures and declares a local variable
 * 'value' of type inty
 */
 PYBIND11_TYPE_CASTER(inty, _("inty"));
 /**
 * Conversion part 1 (Python->C++): convert a PyObject into a inty
 * instance or return false upon failure. The second argument
 * indicates whether implicit conversions should be applied.
 */
 bool load(handle src, bool) {
 /* Extract PyObject from handle */
 PyObject *source = src.ptr();
 /* Try converting into a Python integer value */
 PyObject *tmp = PyNumber_Long(source);
 if (!tmp)
 return false;
 /* Now try to convert into a C++ int */
 value.long_value = PyLong_AsLong(tmp);
 Py_DECREF(tmp);
 /* Ensure return code was OK (to avoid out-of-range errors
etc) */
 return !(value.long_value == -1 && !PyErr_Occurred());
 }
 /**
 * Conversion part 2 (C++ -> Python): convert an inty instance
into
 * a Python object. The second and third arguments are used to
 * indicate the return value policy and parent object (for
 * ``return_value_policy::reference_internal``) and are generally
 * ignored by implicit casters.
 static handle cast(inty src, return_value_policy /* policy */,
handle /* parent */) {
 return PyLong_FromLong(src.long_value);
 }
```

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

Note: A type\_caster<T> defined with PYBIND11\_TYPE\_CASTER(T, ...) requires that T is default-constructible (value is first default constructed and then load() assigns to it).

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

\_\_\_ 96 / 215 BSN

# **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++00000Python0**

0000000 dict 00000000

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

tuple[][][][] py::make\_tuple() [][][]

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

0000000000000Python000

simple namespace

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

### 12.1.3

00000000000C++00000Python000000py::cast() 0000

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

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

0000000000000000000 cast\_error 000

# **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**

## Onerator() On Operator() On Operator()

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

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

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

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

## **12.1.6 DPython DD**

## 00 .attr 000000Python000

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

In the example above <code>pi.attr("exp")</code> 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 <code>self</code> object explicitly, followed by other arguments.

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

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

### **12.1.7** DDDDD

0000000Python000000

```
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

0000 \*args 0 \*\*kwargs 00000000000

```
// * 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)));
```

#### Note

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

### 12.1.10

#### 12.1.11 Gotchas

# **Default-Constructed Wrappers**

static\_cast<bool>(my\_wrapper) [][][]

## Assigning py::none() to wrappers

# **12.2 NumPy**

# 12.2.1 DDDDbuffer protocol

```
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(),
 /* Pointer to buffer
*/
 /* Size of one scalar
 sizeof(float),
*/
 py::format_descriptor<float>::format(), /* Python struct-style
format descriptor */
 /* Number of
dimensions */
 /* Buffer dimensions
 { m.rows(), m.cols() },
*/
 { sizeof(float) * m.cols(),
 /* Strides (in bytes)
for each index */
 sizeof(float) }
);
 });
```

```
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;
};
```

```
/* Bind MatrixXd (or some other Eigen type) to Python */
typedef Eigen::MatrixXd Matrix;
typedef Matrix::Scalar Scalar;
constexpr bool rowMajor = Matrix::Flags & Eigen::RowMajorBit;
py::class_<Matrix>(m, "Matrix", py::buffer_protocol())
 .def(py::init([](py::buffer b) {
 typedef Eigen::Stride<Eigen::Dynamic, Eigen::Dynamic> Strides;
 /* Request a buffer descriptor from Python */
 py::buffer_info info = b.request();
 /* Some sanity checks ... */
 if (info.format != py::format_descriptor<Scalar>::format())
 throw std::runtime_error("Incompatible format: expected a
double array!");
 if (info.ndim != 2)
 throw std::runtime_error("Incompatible buffer dimension!");
 auto strides = Strides(
 info.strides[rowMajor ? 0 : 1] / (py::ssize_t)sizeof(Scalar),
 info.strides[rowMajor ? 1 : 0] / (py::ssize_t)sizeof(Scalar));
 auto map = Eigen::Map<Matrix, 0, Strides>(
 static_cast<Scalar *>(info.ptr), info.shape[0], info.shape[1],
strides);
 return Matrix(map);
 }));
```

00000Eigen00000 def\_buffer() 000000000

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

0000Eigen00000000(0000000)0000Eigen000

## **12.2.2 Arrays**

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

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

arrays 🛮 🗘 🗘 🗘 🗘 🗘 O NumPy API 🗘 🗘 O

- .dtype() 000000000
- .strides() [][][]strides[][][]
- .view(dtype) DDDDdtypeDDDDDDD
- .reshape({i, j, ...}) 0000shape000000 .resize({}) 0000

### **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** ППППП

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

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

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

#### Note

contrived, since it could have been done more simply using vectorize).

```
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>
namespace py = pybind11;
py::array_t<double> add_arrays(py::array_t<double> input1,
py::array_t<double> input2) {
 py::buffer_info buf1 = input1.request(), buf2 = input2.request();
 if (buf1.ndim != 1 || buf2.ndim != 1)
 throw std::runtime_error("Number of dimensions must be one");
 if (buf1.size != buf2.size)
 throw std::runtime_error("Input shapes must match");
 /* No pointer is passed, so NumPy will allocate the buffer */
 auto result = py::array_t<double>(buf1.size);
 py::buffer_info buf3 = result.request();
 double *ptr1 = static_cast<double *>(buf1.ptr);
 double *ptr2 = static_cast<double *>(buf2.ptr);
 double *ptr3 = static_cast<double *>(buf3.ptr);
 for (size_t idx = 0; idx < buf1.shape[0]; idx++)</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-
writeable
 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++)</pre>
 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++)</pre>
 r(i, j, k) += 1.0;
}, py::arg().noconvert());
```

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

- .ndim() returns the number of dimensions
- .data(1, 2, ...) and r.mutable\_data(1, 2, ...) returns a pointer to the const T or T data, respectively, at the given indices. The latter is only available to proxies obtained via a.mutable\_unchecked().
- .itemsize() returns the size of an item in bytes, i.e. sizeof(T).
- .ndim() returns the number of dimensions.
- .shape(n) returns the size of dimension n

- .size() returns the total number of elements (i.e. the product of the shapes).
- .nbytes() returns the number of bytes used by the referenced elements (i.e. itemsize() times size()).

#### 12.2.6 □□□

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

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

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

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

#### **12.2.7 DDDD**

Note: memoryview::from\_memory is not available in Python 2.

# 12.3 0000

# **12.3.1 C++DDPython printDD**

DDDDDPython print DDD sep, end, file, flush DDDD

```
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 □ □ □ □**

# Warning

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

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

The name in Python defaults to ostream\_redirect if no name is passed. This creates the following context manager in Python:

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

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

### **12.3.3 000000000Python000**

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

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

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

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

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

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

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

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

# **13. 00000**

#### 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)
```

main.cpp 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 DPython**

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

□□ module\_::import() □□□□Python□□□

```
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 PYBIND11\_EMBEDDED\_MODULE definitions (as long as they have unique names).

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

```
"""py_module.py located in the working directory"""
import cpp_module
a = cpp_module.a
b = a + 1
#include <pybind11/embed.h>
namespace py = pybind11;
PYBIND11_EMBEDDED_MODULE(cpp_module, m) {
 m.attr("a") = 1;
}
int main() {
 py::scoped_interpreter guard{};
 auto py_module = py::module_::import("py_module");
 auto locals = py::dict("fmt"_a="{} + {} = {}",
**py_module.attr("__dict__"));
 assert(locals["a"].cast<int>() == 1);
 assert(locals["b"].cast<int>() == 2);
 py::exec(R"(
 c = a + b
 message = fmt.format(a, b, c)
)", py::globals(), locals);
 assert(locals["c"].cast<int>() == 3);
 assert(locals["message"].cast<std::string>() == "1 + 2 = 3");
}
```

#### 12.5

# Warning

Creating two concurrent scoped\_interpreter guards is a fatal error. So is calling initialize\_interpreter for a second time after the interpreter has already been initialized.

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

# **14.** □□

#### **14.1** חחחחחחח

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

#### 14.2 0000000GIL0

```
class PyAnimal : public Animal {
public:
 /* Inherit the constructors */
 using Animal::Animal;
 /* Trampoline (need one for each virtual function) */
 std::string go(int n_times) {
 /* Acquire GIL before calling Python code */
 py::gil_scoped_acquire acquire;
 PYBIND11_OVERRIDE_PURE(
 std::string, /* Return type */
 /* Parent class */
 Animal,
 /* Name of function */
 go,
 \mathsf{n}_{\mathsf{-}}\mathsf{times}
 /* Argument(s) */
);
 }
};
PYBIND11_MODULE(example, m) {
 py::class_<Animal, PyAnimal> animal(m, "Animal");
 animal
 .def(py::init<>())
 .def("go", &Animal::go);
 py::class_<Dog>(m, "Dog", animal)
 .def(py::init<>());
 m.def("call_go", [](Animal *animal) -> std::string {
 /* Release GIL before calling into (potentially long-running) C++
code */
 py::gil_scoped_release release;
 return call_go(animal);
 });
}
```

000000 call\_guard 00000 call\_go 0000

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

#### **14.3** חחחחחחחחחח

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

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

```
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);
```

```
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);
```

		_	_	_
1 E	- 1 1	11		ı
13.	- 1 1	11	ш	H

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

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

□□15.1

# 15.3 "SystemError: dynamic module not initialized properly"

□□15.1

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

0015.1

## 15.5000000000

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

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

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

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

### 15.6 00000000?

#### 

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

ex1.cpp:

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

ex2.cpp:

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

python [][]

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

- 2. 0000000000000000

# 15.7 "recursive template instantiation exceeded maximum depth of 256"

# 15.8 "SomeClass' declared with greater visibility than the type of its field 'SomeClass::member' [-Wattributes]"

#### 15.9 חחחחחחחחחחחח?

```
__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_
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_
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-C0

# 

□□□ 132 / 215 BSN

# 15.13 CMake pybind 11 Python 0 0 0 0 0 0 0 0

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

0000 Python2.7 0 pybind11 000000 000000000

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

#### **15.14** 000000000

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} }
```

DD 134 / 215 BSN

**16.** 🗆 🗆

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

• c 🗆 🗆 🗆

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

• C++000000

vector
stl

# pybind11000

**1.** ПППП

#### **1.1** חחחחח

```
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 0000**

```
#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

ПППППППП

```
// 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");
```

#### 00000000C++1400000000000000000000

```
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 ПППП**

#### **1.4** חחחחחחח

0000000C++000000 Pet 00000

```
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'
```

0000000000 class\_::def\_static 0000

#### 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'
```

#### DD Pet::name DDDDDDDDDDDDDDDsetterDgettersDDD

```
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;
};
```

0000 class\_::def\_property() (000000 class\_::def\_property\_readonly() )0000000

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

# DDDDDDreadDDDDDnullptrDDDD

#### 1.4.2

### 

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

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

```
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** חחחחח

```
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 | | | | | \*\* kwargs | | |

```
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);
```

py::args □□□ py::tuple □ py::kwargs □□□ py::dict □

## **2.** ПППППП

### 2.1

```
/* 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>
```

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

	00000	00
re	eturn_value_policy::take_ownership	00000000000000000000000000000000000000
re	eturn_value_policy::copy	00000000Python000000000000000000000000000000000000
re	eturn_value_policy::move	00 std::move 000000000000000000000000000000000000
re	eturn_value_policy::reference	00000000000000C++0000000 00000000000000
re	eturn_value_policy::reference_internal	this D self DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
re	eturn_value_policy::automatic	00000000000000000000000000000000000000
re	eturn_value_policy::automatic_reference	<pre>000000000000000000000000000000000000</pre>

### 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: keep_alive \(\text{Boost.Python} \(\text{D} \) with_custodian_and_ward \(\text{D} \) with_custodian_and_ward_postcall \(\text{D} \) \(\text{D} \)
```

## **Call guard**

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

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

```
See also: test/test_call_policies.cpp 0000000000 keep_alive 0 call_guard 0000
```

## 2.3 Keyword-only

Python3000keyword-only000000000 \* 00000000

```
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 0000000000

000000000 py::args 00000

# 2.4 Positional-only

python3.8000Positional-only00000pybind1100 py::pos\_only() 000000000

# 2.5 Non-converting

#### 

- DD py::implicitly\_convertible<A,B>() DDDDDD
- DDDDDDDfloatDDDDD std::complex<float> 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
```

## **3.** ППППП

#### 3.1

#### 

```
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!'
```

```
// MMMMMMMMMMMMMMMMMMm.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!'
```

# **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);
}
```

□□□□□□□□□Python□□□□□Animal□□□□□"No constructor defined!"□

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

```
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 class\_ 00000000PyAnimal00000Python000Animal00

```
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! "
```

```
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
 // 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!"
```

#### 3.4 0000000

#### 3.5

ППППППАПВПППППАППППППППВП

00000B0000A000000000000000000000Python000000000

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

#### 3.6

#### 

```
class Vector2 {
public:
 Vector2(float x, float y) : x(x), y(y) { }
 Vector2 operator+(const Vector2 &v) const { return Vector2(x + v.x, y
+ v.y); }
 Vector2 operator*(float value) const { return Vector2(x * value, y *
value); }
 Vector2& operator+=(const Vector2 &v) { x += v.x; y += v.y; return
*this; }
 Vector2& operator*=(float v) { x *= v; y *= v; return *this; }
 friend Vector2 operator*(float f, const Vector2 &v) {
 return Vector2(f * v.x, f * v.y);
 }
 std::string toString() const {
 return "[" + std::to_string(x) + ", " + std::to_string(y) + "]";
 }
private:
 float x, y;
};
```

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

PYBIND11_MODULE(example, m) {
 py::class_<Vector2>(m, "Vector2")
 .def(py::init<float, float>())
 .def(py::self + py::self)
 .def(py::self += py::self)
 .def(py::self *= float())
 .def(float() * py::self)
 .def(py::self * float())
 .def(-py::self)
 .def("__repr__", &Vector2::toString);
}
```

.def(py::self \* float()) 0000000000

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

### 3.7

```
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")
...
```

00Python0000000C++0000000C++00Python00

```
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>
`Trampoline`!
```

# **3.10** □ □ final □

```
class IsFinal final {};
py::class_<IsFinal>(m, "IsFinal", py::is_final());
```

```
class PyFinalChild(IsFinal):
 pass

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

## **4.** 0000

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

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

Exception thrown by C++	Translated to Python exception type
<pre>pybind11::buffer_error</pre>	BufferError
<pre>pybind11::import_error</pre>	ImportError
<pre>pybind11::attribute_error</pre>	AttributeError
Any other exception	RuntimeError

pybind11::error\_already\_set []

#### 4.2

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

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

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

## **9.3 C++DPython**

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

#### ПППППП

```
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;
 }
}
```

0000000C++0Python0000Python00000000 error\_already\_set .

```
try {
 py::eval("raise ValueError('The Ring')");
} catch (py::value_error &boromir) {
 // Boromir never gets the ring
 assert(false);
} catch (py::error_already_set &frodo) {
 // Frodo gets the ring
 py::print("I will take the ring");
}
try {
 // py::value_error is a request for pybind11 to raise a Python
exception
 throw py::value_error("The ball");
} catch (py::error_already_set &cat) {
 // cat won't catch the ball since
 // py::value_error is not a Python exception
 assert(false);
} catch (py::value_error &dog) {
 // dog will catch the ball
 py::print("Run Spot run");
 throw; // Throw it again (pybind11 will raise ValueError)
}
```

# **9.4 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");
```

00000 PyErr\_Clear 00000

### 9.5 □□unraiseable□□

```
error_already_set::discard_as_unraisable() [[][Python[][]
```

000000 \_\_del\_\_ 00000Python0000000Python00unraisable000000Python 3.8+00000system hook00000auditing event000

```
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

pybind110000000 PYBIND11\_DECLARE\_HOLDER\_TYPE() Declare\_type() Declare\_type()

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

#### **7.2** 0000000**GIL**0

```
class PyAnimal : public Animal {
public:
 /* Inherit the constructors */
 using Animal::Animal;
 /* Trampoline (need one for each virtual function) */
 std::string go(int n_times) {
 /* Acquire GIL before calling Python code */
 py::gil_scoped_acquire acquire;
 PYBIND11_OVERRIDE_PURE(
 std::string, /* Return type */
 Animal,
 /* Parent class */
 /* Name of function */
 go,
 \mathsf{n}_{\mathsf{-}}\mathsf{times}
 /* Argument(s) */
);
 }
};
PYBIND11_MODULE(example, m) {
 py::class_<Animal, PyAnimal> animal(m, "Animal");
 animal
 .def(py::init<>())
 .def("go", &Animal::go);
 py::class_<Dog>(m, "Dog", animal)
 .def(py::init<>());
 m.def("call_go", [](Animal *animal) -> std::string {
 /* Release GIL before calling into (potentially long-running) C++
code */
 py::gil_scoped_release release;
 return call_go(animal);
 });
}
```

ODDOOD call\_guard ODDOD call\_go ODDO

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

# □□□**□Mermaid**□□□

#### 

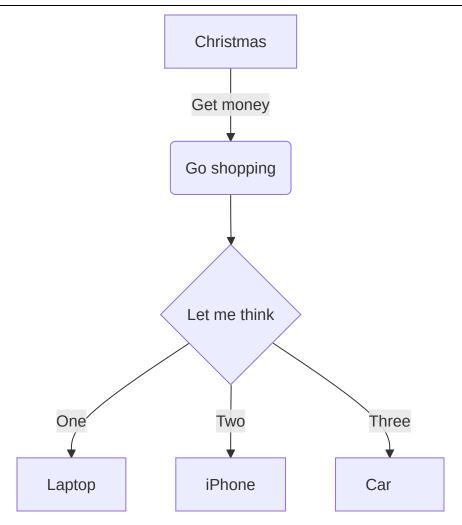
- 000000000 Mermaid 000;
- 000000000 Mermaid 000;
- 00 Gitbook 0000000000.

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

#### 

```
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]
```

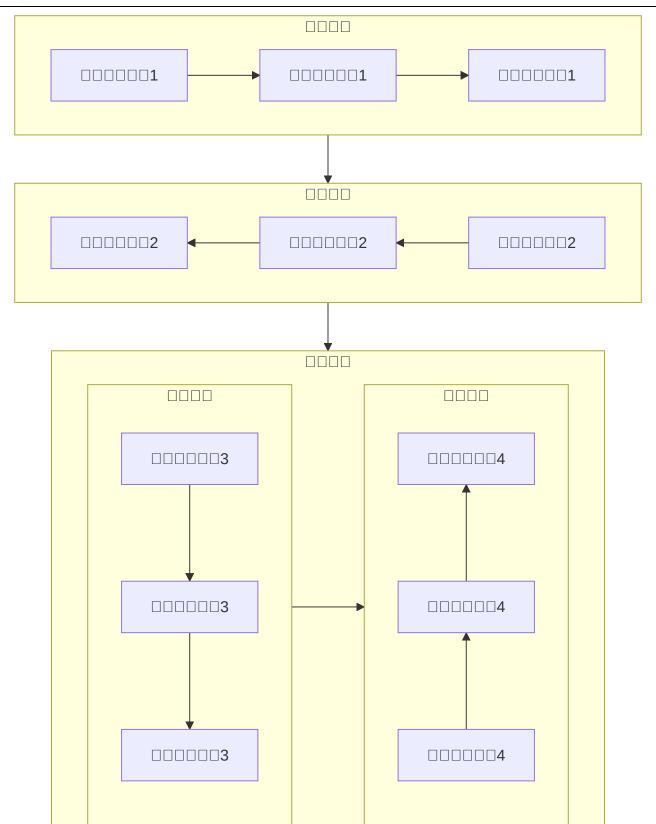
ПП



- 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

# **Mermaid**

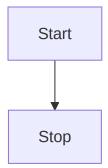
- + TB
- + BT
- + LR
- + RL



TB

□□□□: from **T**op to **B**ottom

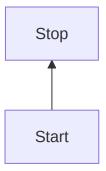
```
graph TB
 Start --> Stop
```



BT

□□□□: from **B**ottom to **T**op

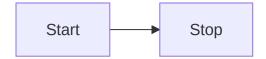
```
graph BT
 Start --> Stop
```



• LR

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

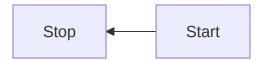
```
graph LR
Start --> Stop
```



• RL

□□□□: from **R**ight to **L**eft

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



```
- \square
 + [🛛 🖺
 - [[XXXX]]
 - [(⊠⊠)]
 - [{\text{\tin}\text{\tetx{\text{\tetx}\\ \text{\text{\text{\text{\texi}\\\ \text{\text{\texit{\text{\texi}\text{\text{\texi}}\\\ \ti}\\\ \titttt{\text{\text{\text{\
 - [/⊠⊠\]
 - [\⊠⊠/]
 + (\\\\\)
 - ((⊠⊠))
 - ([⊠⊠□])
 + {⊠⊠}
 - {[\(\omega \omega \omega \omega \omega \)]}
 - {(\(\omega \omega \omega \omega \omega)\)}
```



00: 00000000,000000.

graph TD id

id

id 0000000,0000000000 <node shape> 0000,00 id 000000000.

ПП

 $\square\square\square\square$ : [node description] , []  $\square\square\square\square\square\square\square\square\square\square\square\square\square\square$ , node description  $\square\square\square\square\square\square\square\square$ .

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

This is the text in the box

• 0000

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

This is the text in the box

• 000

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

This is the text in the box

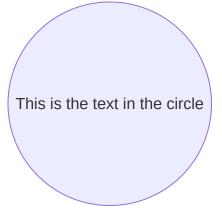
• 🔲

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



• 🛮 🗎

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



• 00000

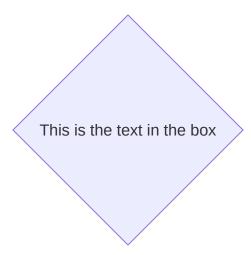
```
graph LR
 id1>This is the text in the box]
```

This is the text in the box

• 🔲

 $\square\square\square\square$ : {node description} , {}  $\square\square\square\square\square\square\square\square\square\square\square$ , node description  $\square\square\square\square\square\square\square\square$ .

```
graph LR
 id1{This is the text in the box}
```



• 000

DDDD: { $\{node\ description\}\}$ , { $\}$  DDDDDD { $\}$  DDDDDDDD, node description DDDDDDDDD.

```
graph LR
 id1\{\{This is the text in the box\}\}
```

Gitbook 000000 {} 000000,00000000,0000000 \ 0000.

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

• 00000

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

This is the text in the box

• 🛮 🗎

DDD: [/node description\] , [] DDDDD /\ DDDDDDDDDDDDDDDD, node description DDDDDDDD.

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

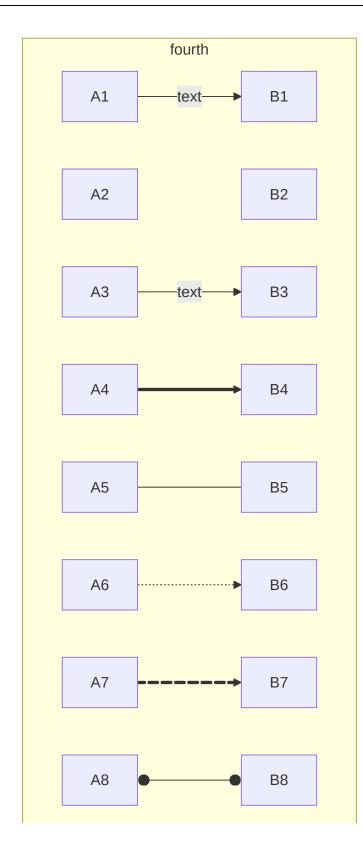
Christmas

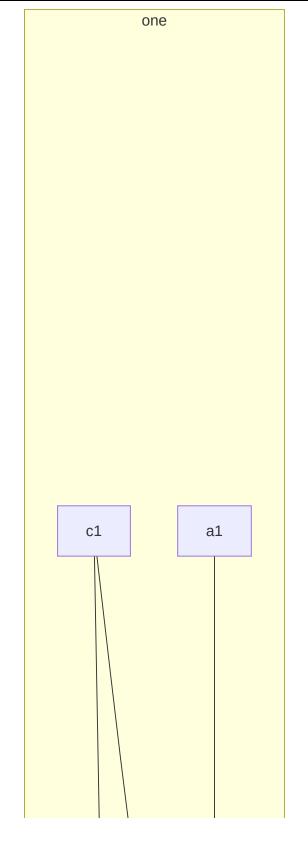
• 00000

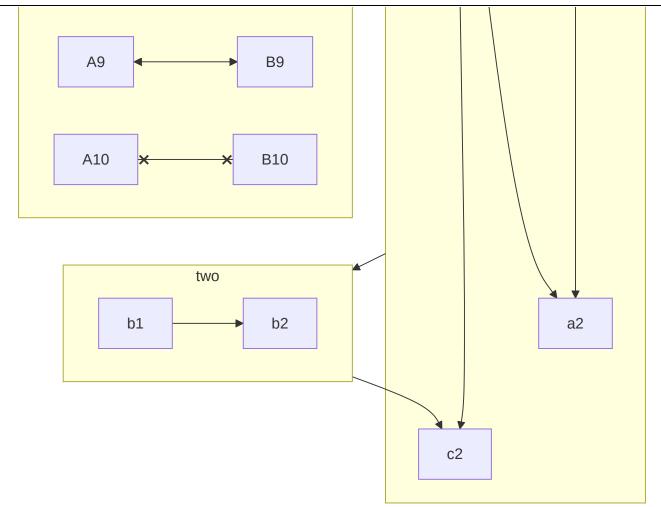
```
graph TD
 B[\Go shopping/]
```

Go shopping

```
+ 🛛 🗎 / 🖺 🗎
 - --
 - -.
+ 🛛 🖂 🗸 🗸 🖂 🖂
 - >
+ 🗆 🗆 🗸 🗸 🗆 🗆
 -
 + -->
 + | 🛛 🖂 🖂 🗎
 -
 + -.0000
 + | 🛮 🗷 🖂 🗎
+ 🛛
 - ==
+ 🛛 🖺 🛈
 - -->
 - -.->
 - \square
 + --\\\\\\-->
 + -->|\|\|\|\|\|
 - \square
 + -->
 - \square
 + -.\\\\\-.->
 + -.->|\|\|\|\|\|
 + -.0000-.-
 + -.-| \
 - ==>
 - ===
 - \square
 + ==> | 🛛 🗎 🖂 🗎
 - \square
 + ===|\|\|\|\|\|\|
```

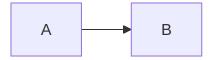






• 0000000

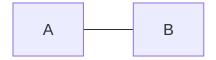
graph LR A-->B



• 00000

0000: --- ,00 -- 0000, - 00000.

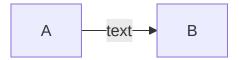
graph LR A --- B



• 00000000

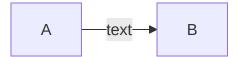
0000: --connection line description--> ,00000 -- 000000000,000 --> 00
000000.

graph LR A-- text -->B



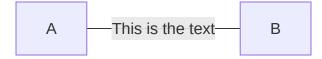
DDDD: |connection line description| ,DD || DDDDDDDDDD.

```
graph LR
 A-->|text|B
```



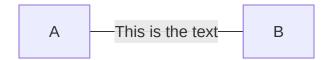
• 00000000

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



DDDD: |connection line description| ,DD || DDDDDDDDDD.

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



• 00000

0000: -.connection line description.-> ,00000 -. 000000000,000 .-> 00
000000.

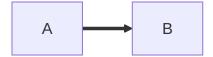
```
graph LR
A-. text .-> B
```



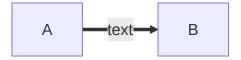
• 000000

0000: ==> ,000000.

graph LR A ==> B



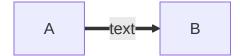
• 000000000



• 0000000000

DDDD: |connection line description| ,DD || DDDDDDDDDD.

graph LR
 A ==>|text| B



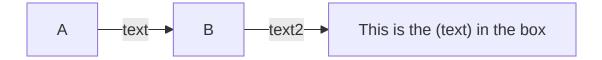
```
+ -->-->

+ &

+ ""

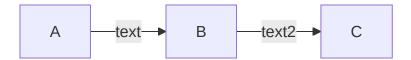
+ %%

+ subgraph
```

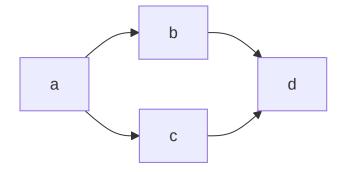


• 0000000

 $\square \square \square \square \square \square \square \square \square, A-->B-->C \square \square \square \square A-->B \square B-->C \square \square.$ 

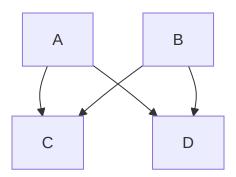


• 0000000



• 0000000

□□□□□□□□□□, A & B --> C & D □□□ A-->C , A-->D , B-->C □ B-->D □□□□□□.



• 00000000

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

This is the (text) in the box

• 00000000

□□ Html □□□□

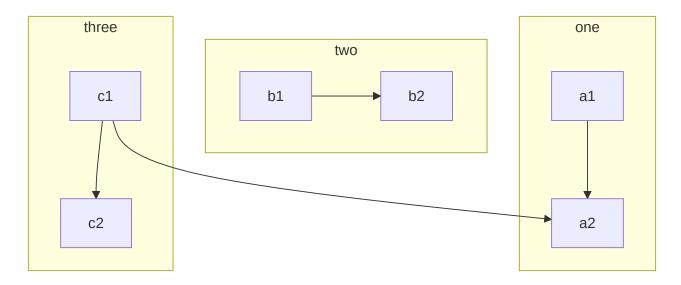
```
graph LR
A["A double quote:#quot;"] -->B["A dec char:#9829;"]
```



• 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
```

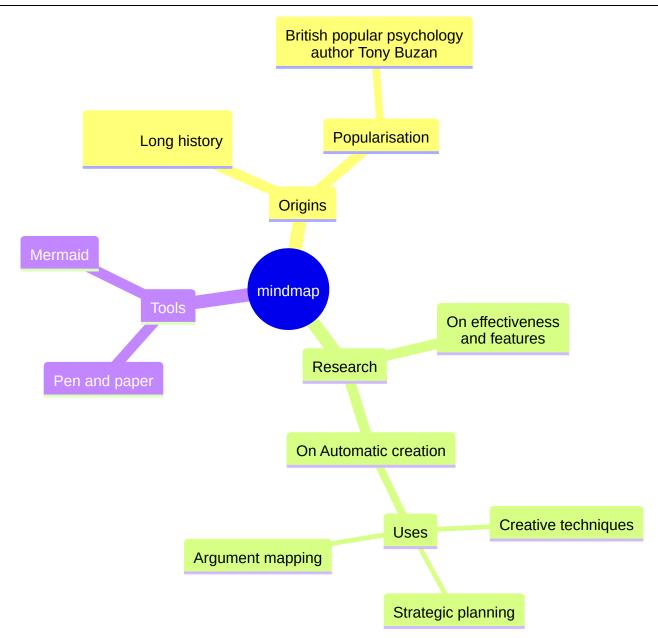


• 0000

000 %% 00000000.



- $\square$



# 

00	00	
graph	graph 000000	
subgraph	subgraph 0000000	
top	TB 0 BT ,000000000000	
bottom	вт О тв,ОООООООООООО	
left	LR 0 RL,00000000000000000000000000000000	
right	RL 0 LR,0000000000000	

# 

#### • 0000

	00	00	
		0000	
()	0000	0000	
{}		0000	
<>		0000	
		00000	
		00000	
==	0000	00000	
=:	0000	00000	

000	00	00	
>	000	00000	
-	000	00000	
	00000000	0000000	
	0000000000	0000000	
	0000000000	0000000	
==	0000000000000	0000000	
=:	0000000000000	0000000	000

#### • 0000

000	00	00	
	000	0000	
[()]	000	0000	
[{}]	000	0000	
(())		0000	
([])	000	0000	
({})		0000	
	000	0000	
{[]}	0000	0000	
{()}		0000	
>	00000	00000	
	00000	00000	
>	00000	00000	

000	00	00	
>	00000	00000	
>	00000	00000	
	00000	00000	
	00000	00000	
==>	000000	00000	
===	000000	00000	
=;>	000000	00000	
=:=>	000000	00000	
=:=	000000	00000	
:=	000000	00000	
	00000000	000000	
connection line description- ->	000000000000 0	000000	
<pre>connection line description&gt;</pre>	000000000000 0	000000	
connection line description- 	000000000000 0	000000	
connection line description	000000000000 0	000000	
==connection line description==>	000000000000 000	000000	
<pre>=:connection line description=:=&gt;</pre>	000000000000 000	000000	
==connection line description===	000000000000 000	000000	

000	00	00	
=:connection line	000000000000	000000	
description=:=			

□□□□: 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, summary, tldr



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

info, todo



#### 1 Info

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

tip, hint, important



#### **★** Tip

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

success, check, done



#### Success

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

question, help, faq



#### Question

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

warning, caution, attention



#### 🛕 Warning

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

failure, fail, missing



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

danger, error

## **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.

example

#### **Example**

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

quote, cite

#### 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.



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.

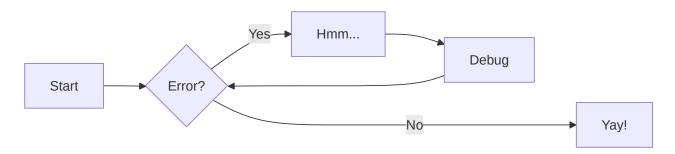
#### Referencing and dereferencing

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



# 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.

# 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:

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

Markdown (requires mdbook-katex):

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

Inspect element and use Sources tab (under Debugger 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

```
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]
```