pybind11 Documentation

Wenzel Jakob

Nov 01, 2026

CONTENTS

1 Changelog 3

2 Upgrade guide 49

3 Installing the library 60

4 First steps 62

5 Object-oriented code 67

6 Build systems 77

7 Functions 89

8 Classes 99

9 Exceptions 122

10 Smart pointers & `py::class_` 128

11 Type conversions 131

12 Python C++ interface 152

13 Embedding the interpreter 167

14 Miscellaneous 177

15 Deprecated 187

16 Frequently asked questions 190

17 Benchmark 196

18 Limitations 199

19 Reference 201

20 CMake helpers 222

Bibliography 227

Index 228

pybind11 is a lightweight header-only library that exposes C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code. Its goals and syntax are similar to the excellent Boost.Python library by David Abrahams: to minimize boilerplate code in traditional extension modules by inferring type information using compile-time introspection.

The main issue with Boost.Python—and the reason for creating such a similar project—is Boost. Boost is an enormously large and complex suite of utility libraries that works with almost every C++ compiler in existence. This compatibility has its cost: arcane template tricks and workarounds are necessary to support the oldest and buggiest of compiler specimens. Now that C++11-compatible compilers are widely available, this heavy machinery has become an excessively large and unnecessary dependency.

Think of this library as a tiny self-contained version of Boost.Python with everything stripped away that isn't relevant for binding generation. Without comments, the core header files only require ~4K lines of code and depend on Python (CPython 3.9+, PyPy, or GraalPy) and the C++ standard library. This compact implementation was possible thanks to some C++11 language features (specifically: tuples, lambda functions and variadic templates). Since its creation, this library has grown beyond Boost.Python in many ways, leading to dramatically simpler binding code in many common situations.

Tutorial and reference documentation is provided at pybind11.readthedocs.io. A PDF version of the manual is available here. And the source code is always available at github.com/pybind/pybind11.

Core features

pybind11 can map the following core C++ features to Python:

- Functions accepting and returning custom data structures per value, reference, or pointer
- Instance methods and static methods
- Overloaded functions
- Instance attributes and static attributes
- Arbitrary exception types
- Enumerations
- Callbacks
- Iterators and ranges
- Custom operators
- Single and multiple inheritance
- STL data structures
- Smart pointers with reference counting like `std::shared_ptr`
- Internal references with correct reference counting
- C++ classes with virtual (and pure virtual) methods can be extended in Python
- Integrated NumPy support (NumPy 2 requires pybind11 2.12+)

Goodies

In addition to the core functionality, pybind11 provides some extra goodies:

- CPython 3.9+, PyPy3 7.3.17+, and GraalPy 24.1+ are supported with an implementation-agnostic interface (see older versions for older CPython and PyPy versions).
- It is possible to bind C++11 lambda functions with captured variables. The lambda capture data is stored inside the resulting Python function object.

- pybind11 uses C++11 move constructors and move assignment operators whenever possible to efficiently transfer custom data types.
- It's easy to expose the internal storage of custom data types through Pythons' buffer protocols. This is handy e.g.for fast conversion between C++ matrix classes like Eigen and NumPy without expensive copy operations.
- pybind11 can automatically vectorize functions so that they are transparently applied to all entries of one or more NumPy array arguments.
- Python's slice-based access and assignment operations can be supported with just a few lines of code.
- Everything is contained in just a few header files; there is no need to link against any additional libraries.
- Binaries are generally smaller by a factor of at least 2 compared to equivalent bindings generated by Boost.Python. A recent pybind11 conversion of PyRosetta, an enormous Boost.Python binding project, reported a binary size reduction of **5.4x** and compile time reduction by **5.8x**.
- Function signatures are precomputed at compile time (using `constexpr`), leading to smaller binaries.
- With little extra effort, C++ types can be pickled and unpickled similar to regular Python objects.

Supported platforms & compilers

pybind11 is exercised in continuous integration across a range of operating systems, Python versions, C++ standards, and toolchains. For an up-to-date view of the combinations we currently test, please see the pybind11 GitHub Actions and AppVeyor logs.

The test matrix naturally evolves over time as older platforms and compilers fall out of use and new ones are added by the community. Closely related versions of a tested compiler or platform will often work as well in practice, but we cannot promise to validate every possible combination. If a configuration you rely on is missing from the matrix or regresses, issues and pull requests to extend coverage are very welcome. At the same time, we need to balance the size of the test matrix with the available CI resources, such as GitHub's limits on concurrent jobs under the free tier.

About

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, Dustin Spicuzza, Boris Staletic, Ethan Steinberg, Patrick Stewart, Ivor Wanders, and Xiaofei Wang.

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

Contributing

See the contributing guide for information on building and contributing to pybind11.

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.

CHAPTER

ONE

CHANGELOG

Starting with version 1.8.0, pybind11 releases use a semantic versioning policy.

Changes will be added here periodically from the "Suggested changelog entry" block in pull request descriptions.

1.1 Version 3.2.0 (IN DEVELOPMENT)

1.2 Version 3.1.0 (August 6, 2026)

New Features:

- Support for Python 3.8 was removed. The minimum supported version is now Python 3.9. #6110
- Support for MSVC 2017 has been dropped. #6110
- Changed strict-mode numeric conversions for PEP 484 compatibility: `float` now accepts `int`, and `complex` now accepts `int` and `float`. #5879
- Small extra features added to the pybind11 command line tool based on python-config. #4272
- Added `subinterpreter_thread_state`, an RAII wrapper that owns a reusable `PyThreadState` for a sub-interpreter, together with a `subinterpreter_scoped_activate` overload that activates it. This lets an OS thread re-enter one or more sub-interpreters without creating and destroying a `PyThreadState` on every activation. #6073
- Add `py::mod_gil_used()` as replacement spelling to `py::mod_gil_not_used(false)`. #5797

Bug fixes:

- Fixed a crash in `py::subinterpreter::create()` when called without a current `PyThreadState`, which its documentation explicitly allows — for example from an embedder that ended initialization with `PyEval_SaveThread()`, or from a worker thread that has never touched Python. `error_scope` is now constructed after a thread state has been attached instead of before. #6127
- Make `py::print` delegate to the current frame/interpreter built-ins `print` entry, fixing handling of `sys.stdout = None`, following the active runtime's stream, keyword, and error semantics, and remaining a no-op if the entry is unavailable during teardown. #6121
- Include the builtin `complex` type in the input annotation of the `std::complex<T>` type caster. #6113
- String views (e.g. `std::string_view`) are now kept alive only when loaded from a transient source (such as a generator), fixing both a use-after-free and a regression where casting a view from a durable object outside a bound function would throw. #6096
- Drop Python 2 prepend to evaluated source, causing errors to be off by one. #6089

- Fix data race on `last_storage_ptr_` cache in `gil_safe_call_once_and_store`. #6087
- Allow user defined `__str__` on `enum_`. #6078
- Fix generated `Callable` type annotations for Python callbacks passed into C++ by inverting the callback's argument/return I/O context relative to the enclosing function signature. #6055
- Correct `__delitem__` for negative-step slices and re-enable contiguous erase fast path. #6088
- Do not pass `-fno-fat-lto-objects` to GCC on macOS, which made all LTO probes fail and silently disabled LTO. #6114
- Fixed cross-compilation to Emscripten/Pyodide with CMake 4.1 by no longer overriding an explicitly set `PYBIND11_USE_CROSSCOMPILING` when `CMAKE_CROSSCOMPILING_EMULATOR` is defined. #6094
- Unset stale `PYTHON_MODULE_DEBUG_POSTFIX` and correct `USE_PYTHON_INCLUDE_DIR` variable. #6086
- Apply `-undefined dynamic_lookup` to all Apple platforms. #6075

Internal:

- Improved `py::enum_` operator performance by restoring type-specific implementations. #5887
- Optimized internal `std::unordered_map`/`std::unordered_set` hashing paths with `noexcept`. #5960

CI:

- Test Python 3.15 in CI. #6107
- Updated the CIBW test workflow to cibuildwheel v4.1 and bumped the Pyodide build to 314. #6095

1.3 Version 3.0.4 (April 18, 2026)

Bug fixes:

- Fixed test builds with installed Eigen 5 by improving `Eigen3` CMake package detection. #6036
- Fixed move semantics of `scoped_ostream_redirect` to preserve buffered output and avoid crashes when moved redirects restore stream buffers. #6033
- Fixed `py::dynamic_attr()` traversal on Python 3.13+ to correctly propagate `PyObject_VisitManagedDict()` results. #6032
- Fixed `std::shared_ptr<T>` fallback casting to avoid unnecessary copy-constructor instantiation in `reference_internal` paths. #6028

CI:

- Updated `setup-uv` to the maintained GitHub Action tag scheme. #6035
- Updated pre-commit hooks. #6029
- Updated GitHub Actions dependencies, including `actions-setup-cmake` and `cibuildwheel`. #6027

1.4 Version 3.0.3 (March 31, 2026)

Bug fixes:

- Fixed TSS key exhaustion in `implicitly_convertible()` when many implicit conversions are registered across large module sets. #6020
- Fixed heap-buffer-overflow in `pythonbuf` with undersized buffers by enforcing a minimum buffer size. #6019
- Fixed virtual-inheritance pointer offset crashes when dispatching inherited methods through virtual bases. #6017
- Fixed `free(): invalid pointer` crashes during interpreter shutdown with `py::enum_<>` by duplicating late-added `def_property_static` argument strings. #6015
- Fixed `function_record` heap-type deallocation to call `PyObject_Free()` and decref the type. #6010
- Hardened `PYBIND11_MODULE_PYINIT` and `get_internals()` against module-initialization crashes. #6018
- Fixed `static_pointer_cast` build failure with virtual inheritance in `holder_caster_foreign_helpers.h`. #6014
- Fixed ambiguous `factory` template specialization that caused compilation failures with nvcc + GCC 14. #6011
- Fixed crash in `def_readwrite` for non-smart-holder properties of smart-holder classes. #6008
- Fixed memory leak for `py::dynamic_attr()` objects on Python 3.13+ by clearing managed `__dict__` contents during deallocation. #5999
- Fixed binding of `noexcept` and ref-qualified (&, &&) methods inherited from unregistered base classes. #5992

Internal:

- Moved `tomlkit` dependency to the dev dependency group. #5990
- Switched to newer public CPython APIs (`PyType_GetFlags` and public vectorcall APIs where available). #6005

Tests:

- Made an async callback test deterministic by replacing fixed sleep with bounded waiting. #5986

CI:

- Re-enabled Android tests in the cibuildwheel workflow. #6001

1.5 Version 3.0.2 (February 16, 2026)

New Features:

- Added helper functions to `py::array` that return shape and strides as `std::span` when available. #5974

Bug fixes:

- Added fallback locking for Python 3.13t where `PyCriticalSection_BeginMutex` is unavailable. #5981
- Fixed race condition in `py::make_key_iterator` with free-threaded Python. #5971
- MSVC 19.16 and earlier were blocked from using `std::launder` due to internal compiler errors. #5968
- Internals destructors were updated to check the owning interpreter before clearing Python objects. #5965
- Internals shutdown handling was refined in two iterations before release: an initial finalization-time cleanup was followed by a safety adjustment to avoid late-shutdown `py::cast` segfaults. #5958 #5972
- Fixed ambiguous `str(handle)` construction for `object`-derived types like `kwargs` or `dict` by templatizing the constructor with SFINAE. #5949

- Fixed concurrency consistency for `internals_pp_manager` under multiple-interpreters. #5947
- Fixed MSVC LNK2001 in C++20 builds when /GL (whole program optimization) is enabled. #5939
- Added per-interpreter storage for `gil_safe_call_once_and_store` to make it safe under multi-interpreters. #5933
- A workaround for a GCC `-Warray-bounds` false positive in `argument_vector` was added. #5908
- Corrected a mistake where support for `__index__` was added, but the type hints did not reflect acceptance of `SupportsIndex` objects. Also fixed a long-standing bug: the complex-caster did not accept `__index__` in `convert` mode. #5891
- Fixed `*args`/`**kwargs` return types. Added type hinting to `py::make_tuple`. #5881
- Fixed compiler error in `type_caster_generic` when casting a `T` implicitly convertible from `T*`. #5873
- Updated `py::native_enum` bindings to unregister enum types on destruction, preventing a use-after-free when returning a destroyed enum instance. #5871
- Fixed undefined behavior that occurred when importing pybind11 modules from non-main threads created by C API modules or embedded python interpreters. #5870
- Fixed dangling pointer in `internals::registered_types_cpp_fast`. #5867
- Added support for `std::shared_ptr<T>` when loading module-local or conduit types from other modules. #5862
- Fixed thread-safety issues if types were concurrently registered while `get_local_type_info()` was called in free threaded Python. #5856
- Fixed py::float_ casting and py::int_ and py::float_ type hints. #5839
- Fixed two `smart_holder` bugs in `shared_ptr` and `unique_ptr` adoption with multiple/virtual inheritance:
  - `shared_ptr` to-Python caster was updated to register the correct subobject pointer (fixes #5786).
  - `unique_ptr` adoption was updated to own the proper object start while aliasing subobject pointers for registration, which fixed MSVC crashes during destruction. #5836
- Constrained `accessor::operator=` templates to avoid obscuring special members. #5832
- Fixed crash that can occur when finalizers acquire and release the GIL. #5828
- Fixed compiler detection in `pybind11/detail/pybind11_namespace_macros.h` for clang-cl on Windows, to address warning suppression macros. #5816
- Fixed compatibility with CMake policy CMP0190 by not always requiring a Python interpreter when cross-compiling. #5829
- Added a static assertion to disallow `keep_alive` and `call_guard` on properties. #5533

Internal:

- CMake policy limit was set to 4.1. #5944
- Improved performance of function calls between Python and C++ by switching to the "vectorcall" calling protocol. #5948
- Many C-style casts were replaced with C++-style casts. #5930
- Added `cast_sources` abstraction to `type_caster_generic`. #5866
- Improved the performance of from-Python conversions of legacy pybind11 enum objects bound by `py::enum_`. #5860
- Reduced size overhead by deduplicating functions' readable signatures and type information. #5857

- Used new Python 3.14 C APIs when available. #5854
- Improved performance of function dispatch and type casting by porting two-level type info lookup strategy from nanobind. #5842
- Updated `.gitignore` to exclude `__pycache__/` directories. #5838
- Changed internals to use `thread_local` instead of `thread_specific_storage` for increased performance. #5834
- Reduced function call overhead by using thread_local for loader_life_support when possible. #5830
- Removed heap allocation for the C++ argument array when dispatching functions with 6 or fewer arguments. #5824

Documentation:

- Fixed docstring for `long double` complex types to use `numpy.clongdouble` instead of the deprecated `numpy.longcomplex` (removed in NumPy 2.0). #5952
- The "Supported compilers" and "Supported platforms" sections in the main `README.rst` were replaced with a new "Supported platforms & compilers" section that points to the CI test matrix as the living source of truth. #5910
- Fixed documentation formatting. #5903
- Updated upgrade notes for `py::native_enum`. #5885
- Clarified in the docs to what extent bindings are global. #5859

Tests:

- Fixed deadlock in a free-threading test by releasing the GIL while waiting on synchronization. #5973
- Calls to `env.deprecated_call()` were replaced with direct calls to `pytest.deprecated_call()`. #5893
- Updated pytest configuration to use `log_level` instead of `log_cli_level`. #5890

CI:

- Added CI tests for windows-11-arm with clang/MSVC (currently python 3.13), windows-11-arm with clang/mingw (currently python 3.12). #5932
- These clang-tidy rules were added: `readability-redundant-casting`, `readability-redundant-inline-specifier`, `readability-redundant-member-init` #5924
- Replaced deprecated macos-13 runners with macos-15-intel in CI. #5916
- Restored `runs-on: windows-latest` in CI. #5835

1.6 Version 3.0.1 (August 22, 2025)

Bug fixes:

- Fixed compilation error in `type_caster_enum_type` when casting pointer-to-enum types. Added pointer overload to handle dereferencing before enum conversion. #5776
- Implement binary version of `make_index_sequence` to reduce template depth requirements for functions with many parameters. #5751
- Subinterpreter-specific exception handling code was removed to resolve segfaults. #5795
- Fixed issue that caused `PYBIND11_MODULE` code to run again if the module was re-imported after being deleted from `sys.modules`. #5782

- Prevent concurrent creation of sub-interpreters as a workaround for stdlib concurrency issues in Python 3.12. #5779
- Fixed potential crash when using `cpp_function` objects with sub-interpreters. #5771
- Fixed non-entrant check in `implicitly_convertible()`. #5777
- Support C++20 on platforms that have older c++ runtimes. #5761
- Fix compilation with clang on msys2. #5757
- Avoid `nullptr` dereference warning with GCC 13.3.0 and python 3.11.13. #5756
- Fix potential warning about number of threads being too large. #5807
- Fix gcc 11.4+ warning about serial compilation using CMake. #5791

Documentation:

- Improve `buffer_info` type checking in numpy docs. #5805
- Replace `robotpy-build` with `semiwrap` in the binding tool list. #5804
- Show nogil in most examples. #5770
- Fix `py::trampoline_self_life_support` visibility in docs. #5766

Tests:

- Avoid a spurious warning about `DOWNLOAD_CATCH` being manually specified. #5803
- Fix an IsolatedConfig test. #5768

CI:

- Add CI testing for Android. #5714

Internal:

- Rename internal variables to avoid the word `slots` (reads better). #5793

1.7 Version 3.0.0 (July 10, 2025)

Pybind11 3.0 includes an ABI bump, the first required bump in many years on Unix (Windows has had required bumps more often). This release contains the smart-holder branch, multi-phase init and subinterpreter support, `py::native_enum`, an interface to warnings, typing improvements, and more. CMake now defaults to FindPython mode. Please check our upgrade guide for more info on upgrading!

Support for Python 3.14, 3.14t, GraalPy, and PyPy 3.11 has been added, while legacy support for Python 3.7, PyPy 3.8/3.9, and CMake <3.15 has been removed. Most deprecated features have been kept for this release, but anything producing a warning in 3.0 may be removed in a future 3.x version. We also now have a deprecation page.

New Features:

- The `smart_holder` branch has been merged, enabling `py::class_<T, py::smart_holder>`, which handles two-way conversion with `std::unique_ptr<T>` and `std::shared_ptr<T>` (simultaneously), disowning a Python object being passed to `std::unique_ptr<T>`, trampoline objects, and `std::enable_shared_from_this`. #5542
  - Added support for `std::shared_ptr<const T>` in `py::init()` when using `py::smart_holder`, complementing existing support for `std::unique_ptr<const T>`. #5731
  - Support const-only smart pointers. #5718

  - Eliminate cross-DSO RTTI reliance from `trampoline_self_life_support` functionality, `smart_holder` deleter detection, and other `smart_holder` bookkeeping. Resolves platform-specific issues on macOS related to cross-DSO `dynamic_cast` and `typeid` mismatches. #5728 (replaces #5700)

- Changed `PYBIND11_MODULE` macro implementation to perform multi-phase module initialization (PEP 489) behind the scenes. #5574 and avoid destruction #5688

- Support for sub-interpreters (both isolated (with separate GILs) and legacy (with a global GIL). Add the `py::multiple_interpreters::per_interpreter_gil()` tag (or, `py::multiple_interpreters::shared_gil()` for legacy interpreter support) to `PYBIND11_MODULE` calls (as the third parameter) to indicate that a module supports running with sub-interpreters. #5564

  - Rename macro `PYBIND11_SUBINTERPRETER_SUPPORT` -> `PYBIND11_HAS_SUBINTERPRETER_SUPPORT` to meet naming convention. #5682

  - Allow subinterpreter support to be disabled if defined to 0. This is mostly an emergency workaround, and is not exposed in CMake. #5708 and #5710

  - Modify internals pointer-to-pointer implementation to not use `thread_local` (better iOS support). #5709

  - Support implementations without subinterpreter support. #5732

- Changed `PYBIND11_EMBEDDED_MODULE` macro implementation to perform multi-phase module initialization (PEP 489) behind the scenes and to support `py::mod_gil_not_used()`, `py::multiple_interpreters::per_interpreter_gil()` and `py::multiple_interpreters::shared_gil()`. #5665 and consolidate code #5670.

- Added API in `pybind11/subinterpreter.h` for embedding sub-interpreters (requires Python 3.12+). #5666

- `py::native_enum` was added, for conversions between Python's native (stdlib) enum types and C++ enums. #5555

  - Add class doc string to `py::native_enum`. #5617.

  - Fix signature for functions with a `native_enum` in the signature. #5619

- Support `py::numpy_scalar<>` / `py::make_scalar()` for NumPy types. #5726

- A `py::release_gil_before_calling_cpp_dtor` option (for `py::class_`) was added to resolve the long-standing issue #1446. #5522

- Add `dtype::normalized_num` and `dtype::num_of`. #5429

- Add support for `array_t<handle>` and `array_t<object>`. #5427

- Added `py::warnings` namespace with `py::warnings::warn` and `py::warnings::new_warning_type` that provides the interface for Python warnings. #5291

- `stl.h` `list|set|map_caster` were made more user friendly: it is no longer necessary to explicitly convert Python iterables to `tuple()`, `set()`, or `map()` in many common situations. #4686

- The `array_caster` in pybind11/stl.h was enhanced to support value types that are not default-constructible. #5305

- `pybind11/conduit/pybind11_platform_abi_id.h` was factored out, to maximize reusability of `PYBIND11_PLATFORM_ABI_ID` (for other Python/C++ binding systems). Separately, a note was added to explain that the conduit feature only covers from-Python-to-C++ conversions. #5375 | #5740

- Added support for finding pybind11 using pkgconf distributed on pypi. #5552

- Support `--extension-suffix` on the pybind11 command. #5360

- Add semi-public API: `pybind11::detail::is_holder_constructed` and update example for `pybind11::custom_type_setup` in documentation. #5669

- Added `py::scoped_critical_section` to support free-threaded mode. #5684 | #5706

New Features / fixes (typing):

- Added option for different arg/return type hints to `type_caster`. Updated `stl/filesystem` to use correct arg/return type hints. Updated `pybind11::typing` to use correct arg/return type hints for nested types. #5450
- Updated type hint for `py::capsule` to `type.CapsuleType`. #5567
- Adds support for `typing.SupportsInt` and `typing.SupportsFloat`. Update `Final` to be narrower type hint. Make `std::function` match `Callable` type. Fix `io_name` bug in `attr_with_type_hint`. #5540
- Rework of arg/return type hints to support `.noconvert()`. #5486
- Add `attr_with_type` for declaring attribute types and `Final`, `ClassVar` type annotations. #5460
- Allow annotate methods with `py::pos_only` when only have the `self` argument. Make arguments for auto-generated dunder methods positional-only. #5403
- Added `py::Args` and `py::KWArgs` to enable custom type hinting of `*args` and `**kwargs` (see PEP 484). #5357
- Switched to `numpy.typing.NDArray` and `numpy.typing.ArrayLike`. #5212
- Use `numpy.object_` instead of `object`. #5571
- Fix module type hint. #5469
- Fix Buffer type hint. #5662
- Added support for `collections.abc` in type hints and convertible checks of STL casters and `py::buffer`. #5566
- Fix `typing` and `collections.abc` type hint ambiguity. #5663
- Add `typing_extensions` alternatives for all types that need them. #5693

Removals:

- Remove support for pybind11 v2 internals versions (4, 5, 6). (The internals version number has been bumped for pybind11 v3.) #5512 | #5530
- Remove `make_simple_namespace` (added in 2.8.0, deprecated in 2.8.1). #5597
- Legacy-mode option `PYBIND11_NUMPY_1_ONLY` has been removed. #5595
- Add a deprecation warning to `.get_type` (deprecated in pybind11 2.6 in 2020). #5596

Bug fixes:

- Set `__file__` on submodules. #5584. Except on embedded modules. #5650
- pybind11-bound functions are now pickleable. #5580
- Fix bug in `attr_with_type_hint` to allow objects to be in `attr_with_type_hint`. #5576
- A `-Wmaybe-uninitialized` warning suppression was added in `pybind11/eigen/matrix.h`. #5516
- `PYBIND11_WARNING_POP` was incorrectly defined as `PYBIND11_PRAGMA(clang diagnostic push)`. #5448
- `PYBIND11_PLATFORM_ABI_ID` (which is used in composing `PYBIND11_INTERNALS_ID`) was modernized to reflect actual ABI compatibility more accurately. #4953 | #5439
- Fix buffer protocol implementation. #5407
- Fix iterator increment operator does not skip first item. #5400
- When getting or deleting an element in a container bound by `bind_map`, print the key in `KeyError` if it does not exist. #5397

- `pybind11::builtin_exception` is now explicitly exported when linked to libc++. #5390
- Allow subclasses of `py::args` and `py::kwargs`. #5381
- Disable false-positive GCC 12 Bound Check warning. #5355
- Update the dict when restoring pickles, instead of assigning a replacement dict. #5658
- Properly define `_DEBUG` macro to 1 instead of defining it without value. #5639
- Fix a missing time cast causing a compile error for newer ICC. #5621
- Change the behavior of the default constructor of `py::slice` to be equivalent to `::` in Python. #5620

Bug fixes and features (CMake):

- Enable FindPython mode by default, with a `COMPAT` mode that sets some of the old variables to ease transition. #5553
- Add an author warning that auto-calculated `PYTHON_MODULE_EXTENSION` may not respect `SETUPTOOLS_EXT_SUFFIX` during cross-compilation. #5495
- Don't strip with `CMAKE_BUILD_TYPE` None. #5392
- Fix an issue with `NO_EXTRAS` adding `pybind11::windows_extras` anyway. #5378
- Fix issue with NEW/OLD message showing up. #5656
- Use CMake's warnings as errors if available (CMake 3.24+). #5612
- Add support for running pybind11's tests via presets in CMake 3.25+. #5655 and support `--fresh`. #5668
- Experimental CMake support for Android. #5733
- Presets now generate `compile_commands.json`. #5685

Bug fixes (free-threading):

- Fix data race in free threaded CPython when accessing a shared static variable. #5494
- A free-threading data race in `all_type_info()` was fixed. #5419
- Added exception translator specific mutex used with `try_translate_exceptions` in the free-threaded build for internal locking. #5362

Internals:

- Consolidated all `PYBIND11_HAS_...` feature macros into `pybind11/detail/common.h` to streamline backward compatibility checks and simplify internal refactoring. This change ensures consistent macro availability regardless of header inclusion order. #5647
- `pybind11/gil_simple.h` was factored out from `pybind11/gil.h`, so that it can easily be reused. #5614
- Use CPython macros to construct `PYBIND11_VERSION_HEX`. #5683

Documentation:

- Improved `reference_internal` policy documentation. #5528
- A new "Double locking, deadlocking, GIL" document was added. #5394
- Add documenting for free-threading and subinterpreters. #5659

Tests:

- Download the final Catch2 2.x release if Catch download is requested. #5568
- Explicitly used `signed char` for two numpy dtype tests. As seen when compiling using `clang` on Linux with the `-funsigned-char` flag. #5545

- CI testing now includes `-Wwrite-strings -Wunreachable-code -Wpointer-arith -Wredundant-decls` in some jobs. #5523
- Add nightly wheels to scientific-python's nightly wheelhouse. #5675
- Expect free-threaded warning when loading a non-free-threaded module. #5680
- Run pytest under Python devmode. #5715

New and removed platforms:

- Support Python 3.14 (beta 1+). #5646
- Added support for GraalPy Python implementation (https://github.com/oracle/graalpython). #5380
- Support and test iOS in CI. #5705
- Support for PyPy 3.11 added. #5508 And test in ci. #5534
- Support for PyPy 3.8 and 3.9 was dropped. #5578
- Support for Python 3.7 was removed. (Official end-of-life: 2023-06-27). #5191
- Support for CMake older than 3.15 removed. CMake 3.15-4.0 supported. #5304 and fix regression #5691.
- Use scikit-build-core for the build backend for the PyPI `pybind11`. The CMake generation has been moved to the sdist->wheel step. `PYBIND11_GLOBAL_SDIST` has been removed. #5598 and updated docs/ci. #5676
- clang 20 tested and used for clang-tidy. #5692
- Drop testing on MSVC 2019 (as it is being removed from GitHub). #5712
- Support Windows C++20 and Linux C++23 in tests. #5707

1.8 Version 2.13.6 (September 13, 2024)

New Features:

- A new `self._pybind11_conduit_v1_()` method is automatically added to all `py::class_`-wrapped types, to enable type-safe interoperability between different independent Python/C++ bindings systems, including pybind11 versions with different `PYBIND11_INTERNALS_VERSION`'s. Supported on pybind11 2.11.2, 2.12.1, and 2.13.6+. #5296

Bug fixes:

- Using `__cpp_nontype_template_args` instead of `__cpp_nontype_template_parameter_class`. #5330
- Properly translate C++ exception to Python exception when creating Python buffer from wrapped object. #5324

Documentation:

- Adds an answer (FAQ) for "What is a highly conclusive and simple way to find memory leaks?". #5340

1.9 Version 2.13.5 (August 22, 2024)

Bug fixes:

- Fix includes when using Windows long paths (\\?\ prefix). #5321
- Support `-Wpedantic` in C++20 mode. #5322
- Fix and test `<ranges>` support for `py::tuple` and `py::list`. #5314

1.10 Version 2.13.4 (August 14, 2024)

Bug fixes:

- Fix paths with spaces, including on Windows. (Replaces regression from #5302) #4874

Documentation:

- Remove repetitive words. #5308

1.11 Version 2.13.3 (August 13, 2024)

Bug fixes:

- Quote paths from pybind11-config #5302
- Fix typo in Emscripten support when in config mode (CMake) #5301

1.12 Version 2.13.2 (August 13, 2024)

New Features:

- A `pybind11::detail::type_caster_std_function_specializations` feature was added, to support specializations for `std::function`'s with return types that require custom to-Python conversion behavior (to primary use case is to catch and convert exceptions). #4597

Changes:

- Use `PyMutex` instead of `std::mutex` for internal locking in the free-threaded build. #5219
- Add a special type annotation for C++ empty tuple. #5214
- When compiling for WebAssembly, add the required exception flags (CMake 3.13+). #5298

Bug fixes:

- Make `gil_safe_call_once_and_store` thread-safe in free-threaded CPython. #5246
- A missing `#include <algorithm>` in pybind11/typing.h was added to fix build errors (in case user code does not already depend on that include). #5208
- Fix regression introduced in #5201 for GCC<10.3 in C++20 mode. #5205
- Remove extra = when assigning flto value in the case for Clang in CMake. #5207

Tests:

- Adding WASM testing to our CI (Pyodide / Emscripten via scikit-build-core). #4745

- clang-tidy (in GitHub Actions) was updated from clang 15 to clang 18. #5272

1.13 Version 2.13.1 (June 26, 2024)

New Features:

- Add support for `Typing.Callable[..., T]`. #5202

Bug fixes:

- Avoid aligned allocation in free-threaded build in order to support macOS versions before 10.14. #5200

1.14 Version 2.13.0 (June 25, 2024)

New Features:

- Support free-threaded CPython (3.13t). Add `py::mod_gil_not_used()` tag to indicate if a module supports running with the GIL disabled. #5148
- Support for Python 3.6 was removed. (Official end-of-life: 2021-12-23). #5177
- `py::list` gained a `.clear()` method. #5153
- Support for `Union`, `Optional`, `type[T]`, `typing.TypeGuard`, `typing.TypeIs`, `typing.Never`, `typing.NoReturn` and `typing.Literal` was added to `pybind11/typing.h`. #5166 #5165 #5194 #5193 #5192
- In CMake, if `PYBIND11_USE_CROSSCOMPILING` is enabled, then `CMAKE_CROSSCOMPILING` will be respected and will keep pybind11 from accessing the interpreter during configuration. Several CMake variables will be required in this case, but can be deduced from the environment variable `SETUPTOOLS_EXT_SUFFIX`. The default (currently `OFF`) may be changed in the future. #5083

Bug fixes:

- A refcount bug (leading to heap-use-after-free) involving trampoline functions with `PyObject *` return type was fixed. #5156
- Return `py::ssize_t` from `.ref_count()` instead of `int`. #5139
- A subtle bug involving C++ types with unusual `operator&` overrides was fixed. #5189
- Support Python 3.13 with minor fix, add to CI. #5127
- Fix mistake affecting old cmake and old boost. #5149

Documentation:

- Build docs updated to feature scikit-build-core and meson-python, and updated setuptools instructions. #5168

Tests:

- Avoid immortal objects in tests. #5150

CI:

- Compile against Python 3.13t in CI.
- Use `macos-13` (Intel) for CI jobs for now (will drop Python 3.7 soon). #5109
- Releases now have artifact attestations, visible at https://github.com/pybind/pybind11/attestations. #5196

Other:

- Some cleanup in preparation for 3.13 support. #5137

- Avoid a warning by ensuring an iterator end check is included in release mode. #5129
- Bump max cmake to 3.29. #5075
- Update docs and noxfile. #5071

1.15 Version 2.12.1 (September 13, 2024)

New Features:

- A new `self._pybind11_conduit_v1_()` method is automatically added to all `py::class_`-wrapped types, to enable type-safe interoperability between different independent Python/C++ bindings systems, including pybind11 versions with different `PYBIND11_INTERNALS_VERSION`'s. Supported on pybind11 2.11.2, 2.12.1, and 2.13.6+. #5296

1.16 Version 2.12.0 (March 27, 2024)

New Features:

- `pybind11` now supports compiling for NumPy 2. Most code shouldn't change (see `upgrade-guide-2.12` for details). However, if you experience issues you can define `PYBIND11_NUMPY_1_ONLY` to disable the new support for now, but this will be removed in the future. #5050
- `pybind11/gil_safe_call_once.h` was added (it needs to be included explicitly). The primary use case is GIL-safe initialization of C++ `static` variables. #4877
- Support move-only iterators in `py::make_iterator`, `py::make_key_iterator`, `py::make_value_iterator`. #4834
- Two simple `py::set_error()` functions were added and the documentation was updated accordingly. In particular, `py::exception<>::operator()` was deprecated (use one of the new functions instead). The documentation for `py::exception<>` was further updated to not suggest code that may result in undefined behavior. #4772

Bug fixes:

- Removes potential for Undefined Behavior during process teardown. #4897
- Improve compatibility with the nvcc compiler (especially CUDA 12.1/12.2). #4893
- `pybind11/numpy.h` now imports NumPy's `multiarray` and `_internal` submodules with paths depending on the installed version of NumPy (for compatibility with NumPy 2). #4857
- Builtins collections names in docstrings are now consistently rendered in lowercase (list, set, dict, tuple), in accordance with PEP 585. #4833
- Added `py::typing::Iterator<T>`, `py::typing::Iterable<T>`. #4832
- Render `py::function` as `Callable` in docstring. #4829
- Also bump `PYBIND11_INTERNALS_VERSION` for MSVC, which unlocks two new features without creating additional incompatibilities. #4819
- Guard against crashes/corruptions caused by modules built with different MSVC versions. #4779
- A long-standing bug in the handling of Python multiple inheritance was fixed. See PR #4762 for the rather complex details. #4762
- Fix `bind_map` with `using` declarations. #4952

- Qualify `py::detail::concat` usage to avoid ADL selecting one from somewhere else, such as modernjson's concat. #4955
- Use new PyCode API on Python 3.12+. #4916
- Minor cleanup from warnings reported by Clazy. #4988
- Remove typing and duplicate `class_` for `KeysView/ValuesView/ItemsView`. #4985
- Use `PyObject_VisitManagedDict()` and `PyObject_ClearManagedDict()` on Python 3.13 and newer. #4973
- Update `make_static_property_type()` to make it compatible with Python 3.13. #4971
- Render typed iterators for `make_iterator`, `make_key_iterator`, `make_value_iterator`. #4876
- Add several missing type name specializations. #5073
- Change docstring render for `py::buffer`, `py::sequence` and `py::handle` (to `Buffer`, `Sequence`, `Any`). #4831
- Fixed `base_enum.__str__` docstring. #4827
- Enforce single line docstring signatures. #4735
- Special 'typed' wrappers now available in `typing.h` to annotate tuple, dict, list, set, and function. #4259
- Create `handle_type_name` specialization to type-hint variable length tuples. #5051
- Setting `PYBIND11_FINDPYTHON` to OFF will force the old FindPythonLibs mechanism to be used. #5042
- Skip empty `PYBIND11_PYTHON_EXECUTABLE_LAST` for the first cmake run. #4856
- Fix FindPython mode exports & avoid `pkg_resources` if `importlib.metadata` available. #4941
- `Python_ADDITIONAL_VERSIONS` (classic search) now includes 3.12. #4909
- `pybind11.pc` is now relocatable by default as long as install destinations are not absolute paths. #4830
- Correctly detect CMake FindPython removal when used as a subdirectory. #4806
- Don't require the libs component on CMake 3.18+ when using PYBIND11_FINDPYTHON (fixes manylinux builds). #4805
- `pybind11_strip` is no longer automatically applied when `CMAKE_BUILD_TYPE` is unset. #4780
- Support `DEBUG_POSFIX` correctly for debug builds. #4761
- Hardcode lto/thin lto for Emscripten cross-compiles. #4642
- Upgrade maximum supported CMake version to 3.27 to fix CMP0148 warnings. #4786

Documentation:

- Small fix to grammar in `functions.rst`. #4791
- Remove upper bound in example pyproject.toml for setuptools. #4774

CI:

- CI: Update NVHPC to 23.5 and Ubuntu 20.04. #4764
- Test on PyPy 3.10. #4714

Other:

- Use Ruff formatter instead of Black. #4912
- An `assert()` was added to help Coverty avoid generating a false positive. #4817

1.17 Version 2.11.2 (September 13, 2024)

New Features:

- A new `self._pybind11_conduit_v1_()` method is automatically added to all `py::class_`-wrapped types, to enable type-safe interoperability between different independent Python/C++ bindings systems, including pybind11 versions with different `PYBIND11_INTERNALS_VERSION`'s. Supported on pybind11 2.11.2, 2.12.1, and 2.13.6+. #5296

1.18 Version 2.11.1 (July 17, 2023)

Changes:

- `PYBIND11_NO_ASSERT_GIL_HELD_INCREF_DECREF` is now provided as an option for disabling the default-on `PyGILState_Check()`'s in `pybind11::handle`'s `inc_ref()` & `dec_ref()`. #4753
- `PYBIND11_ASSERT_GIL_HELD_INCREF_DECREF` was disabled for PyPy in general (not just PyPy Windows). #4751

1.19 Version 2.11.0 (July 14, 2023)

New features:

- The newly added `pybind11::detail::is_move_constructible` trait can be specialized for cases in which `std::is_move_constructible` does not work as needed. This is very similar to the long-established `pybind11::detail::is_copy_constructible`. #4631
- Introduce `recursive_container_traits`. #4623
- `pybind11/type_caster_pyobject_ptr.h` was added to support automatic wrapping of APIs that make use of `PyObject *`. This header needs to included explicitly (i.e. it is not included implicitly with `pybind/pybind11.h`). #4601
- `format_descriptor<>` & `npy_format_descriptor<>` `PyObject *` specializations were added. The latter enables `py::array_t<PyObject *>` to/from-python conversions. #4674
- `buffer_info` gained an `item_type_is_equivalent_to<T>()` member function. #4674
- The `capsule` API gained a user-friendly constructor (`py::capsule(ptr, "name", dtor)`). #4720

Changes:

- `PyGILState_Check()`'s in `pybind11::handle`'s `inc_ref()` & `dec_ref()` are now enabled by default again. #4246
- `py::initialize_interpreter()` using `PyConfig_InitPythonConfig()` instead of `PyConfig_InitIsolatedConfig()`, to obtain complete `sys.path`. #4473
- Cast errors now always include Python type information, even if `PYBIND11_DETAILED_ERROR_MESSAGES` is not defined. This increases binary sizes slightly (~1.5%) but the error messages are much more informative. #4463
- The docstring generation for the `std::array`-list caster was fixed. Previously, signatures included the size of the list in a non-standard, non-spec compliant way. The new format conforms to PEP 593. **Tooling for processing the docstrings may need to be updated accordingly.** #4679

- Setter return values (which are inaccessible for all practical purposes) are no longer converted to Python (only to be discarded). #4621
- Allow lambda specified to function definition to be `noexcept(true)` in C++17. #4593
- Get rid of recursive template instantiations for concatenating type signatures on C++17 and higher. #4587
- Compatibility with Python 3.12 (beta). Note that the minimum pybind11 ABI version for Python 3.12 is version 5. (The default ABI version for Python versions up to and including 3.11 is still version 4.). #4570
- With `PYBIND11_INTERNALS_VERSION 5` (default for Python 3.12+), MSVC builds use `std::hash<std::type_index>` and `std::equal_to<std::type_index>` instead of string-based type comparisons. This resolves issues when binding types defined in the unnamed namespace. #4319
- Python exception `__notes__` (introduced with Python 3.11) are now added to the `error_already_set::what()` output. #4678

Build system improvements:

- CMake 3.27 support was added, CMake 3.4 support was dropped. FindPython will be used if `FindPythonInterp` is not present. #4719
- Update clang-tidy to 15 in CI. #4387
- Moved the linting framework over to Ruff. #4483
- Skip `lto` checks and target generation when `CMAKE_INTERPROCEDURAL_OPTIMIZATION` is defined. #4643
- No longer inject `-stdlib=libc++`, not needed for modern Pythons (macOS 10.9+). #4639
- PyPy 3.10 support was added, PyPy 3.7 support was dropped. #4728
- Testing with Python 3.12 beta releases was added. #4713

1.20 Version 2.10.4 (Mar 16, 2023)

Changes:

- `python3 -m pybind11` gained a `--version` option (prints the version and exits). #4526

Bug Fixes:

- Fix a warning when pydebug is enabled on Python 3.11. #4461
- Ensure `gil_scoped_release` RAII is non-copyable. #4490
- Ensure the tests dir does not show up with new versions of setuptools. #4510
- Better stacklevel for a warning in setuptools helpers. #4516

1.21 Version 2.10.3 (Jan 3, 2023)

Changes:

- Temporarily made our GIL status assertions (added in 2.10.2) disabled by default (re-enable manually by defining `PYBIND11_ASSERT_GIL_HELD_INCREF_DECREF`, will be enabled in 2.11). #4432
- Improved error messages when `inc_ref`/`dec_ref` are called with an invalid GIL state. #4427 #4436

Bug Fixes:

- Some minor touchups found by static analyzers. #4440

1.22 Version 2.10.2 (Dec 20, 2022)

Changes:

- `scoped_interpreter` constructor taking `PyConfig`. #4330
- `pybind11/eigen/tensor.h` adds converters to and from `Eigen::Tensor` and `Eigen::TensorMap`. #4201
- `PyGILState_Check()`'s were integrated to `pybind11::handle inc_ref()` & `dec_ref()`. The added GIL checks are guarded by `PYBIND11_ASSERT_GIL_HELD_INCREF_DECREF`, which is the default only if `NDEBUG` is not defined. (Made non-default in 2.10.3, will be active in 2.11) #4246
- Add option for enable/disable enum members in docstring. #2768
- Fixed typing of `KeysView`, `ValuesView` and `ItemsView` in `bind_map`. #4353

Bug fixes:

- Bug fix affecting only Python 3.6 under very specific, uncommon conditions: move `PyEval_InitThreads()` call to the correct location. #4350
- Fix segfault bug when passing foreign native functions to functional.h. #4254

Build system improvements:

- Support setting PYTHON_LIBRARIES manually for Windows ARM cross-compilation (classic mode). #4406
- Extend IPO/LTO detection for ICX (a.k.a IntelLLVM) compiler. #4402
- Allow calling `find_package(pybind11 CONFIG)` multiple times from separate directories in the same CMake project and properly link Python (new mode). #4401
- `multiprocessing_set_spawn` in pytest fixture for added safety. #4377
- Fixed a bug in two pybind11/tools cmake scripts causing "Unknown arguments specified" errors. #4327

1.23 Version 2.10.1 (Oct 31, 2022)

This is the first version to fully support embedding the newly released Python 3.11.

Changes:

- Allow `pybind11::capsule` constructor to take null destructor pointers. #4221
- `embed.h` was changed so that `PYTHONPATH` is used also with Python 3.11 (established behavior). #4119
- A `PYBIND11_SIMPLE_GIL_MANAGEMENT` option was added (cmake, C++ define), along with many additional tests in `test_gil_scoped.py`. The option may be useful to try when debugging GIL-related issues, to determine if the more complex default implementation is or is not to blame. See #4216 for background. WARNING: Please be careful to not create ODR violations when using the option: everything that is linked together with mutual symbol visibility needs to be rebuilt. #4216
- `PYBIND11_EXPORT_EXCEPTION` was made non-empty only under macOS. This makes Linux builds safer, and enables the removal of warning suppression pragmas for Windows. #4298

Bug fixes:

- Fixed a bug where `UnicodeDecodeError` was not propagated from various `py::str` ctors when decoding surrogate utf characters. #4294
- Revert perfect forwarding for `make_iterator`. This broke at least one valid use case. May revisit later. #4234
- Fix support for safe casts to `void*` (regression in 2.10.0). #4275

- Fix `char8_t` support (regression in 2.9). #4278
- Unicode surrogate character in Python exception message leads to process termination in `error_already_set::what()`. #4297
- Fix MSVC 2019 v.1924 & C++14 mode error for `overload_cast`. #4188
- Make augmented assignment operators non-const for the object-api. Behavior was previously broken for augmented assignment operators. #4065
- Add proper error checking to C++ bindings for Python list append and insert. #4208
- Work-around for Nvidia's CUDA nvcc compiler in versions 11.4.0 - 11.8.0. #4220
- A workaround for PyPy was added in the `py::error_already_set` implementation, related to PR #1895 released with v2.10.0. #4079
- Fixed compiler errors when C++23 `std::forward_like` is available. #4136
- Properly raise exceptions in contains methods (like when an object in unhashable). #4209
- Further improve another error in exception handling. #4232
- `get_local_internals()` was made compatible with `finalize_interpreter()`, fixing potential freezes during interpreter finalization. #4192

Performance and style:

- Reserve space in set and STL map casters if possible. This will prevent unnecessary rehashing / resizing by knowing the number of keys ahead of time for Python to C++ casting. This improvement will greatly speed up the casting of large unordered maps and sets. #4194
- GIL RAII scopes are non-copyable to avoid potential bugs. #4183
- Explicitly default all relevant ctors for pytypes in the `PYBIND11_OBJECT` macros and enforce the clang-tidy checks `modernize-use-equals-default` in macros as well. #4017
- Optimize iterator advancement in C++ bindings. #4237
- Use the modern `PyObject_GenericGetDict` and `PyObject_GenericSetDict` for handling dynamic attribute dictionaries. #4106
- Document that users should use `PYBIND11_NAMESPACE` instead of using `pybind11` when opening namespaces. Using namespace declarations and namespace qualification remain the same as `pybind11`. This is done to ensure consistent symbol visibility. #4098
- Mark `detail::forward_like` as constexpr. #4147
- Optimize unpacking_collector when processing `arg_v` arguments. #4219
- Optimize casting C++ object to `None`. #4269

Build system improvements:

- CMake: revert overwrite behavior, now opt-in with `PYBIND11_PYTHONLIBS_OVERRWRITE OFF`. #4195
- Include a pkg-config file when installing pybind11, such as in the Python package. #4077
- Avoid stripping debug symbols when `CMAKE_BUILD_TYPE` is set to `DEBUG` instead of `Debug`. #4078
- Followup to #3948, fixing vcpkg again. #4123

1.24 Version 2.10.0 (Jul 15, 2022)

Removed support for Python 2.7, Python 3.5, and MSVC 2015. Support for MSVC 2017 is limited due to availability of CI runners; we highly recommend MSVC 2019 or 2022 be used. Initial support added for Python 3.11.

New features:

- `py::anyset` & `py::frozenset` were added, with copying (cast) to `std::set` (similar to `set`). #3901
- Support bytearray casting to string. #3707
- `type_caster<std::monostate>` was added. `std::monostate` is a tag type that allows `std::variant` to act as an optional, or allows default construction of a `std::variant` holding a non-default constructible type. #3818
- `pybind11::capsule::set_name` added to mutate the name of the capsule instance. #3866
- NumPy: dtype constructor from type number added, accessors corresponding to Python API `dtype.num`, `dtype.byteorder`, `dtype.flags` and `dtype.alignment` added. #3868

Changes:

- Python 3.6 is now the minimum supported version. #3688 #3719
- The minimum version for MSVC is now 2017. #3722
- Fix issues with CPython 3.11 betas and add to supported test matrix. #3923
- `error_already_set` is now safer and more performant, especially for exceptions with long tracebacks, by delaying computation. #1895
- Improve exception handling in python `str` bindings. #3826
- The bindings for capsules now have more consistent exception handling. #3825
- `PYBIND11_OBJECT_CVT` and `PYBIND11_OBJECT_CVT_DEFAULT` macro can now be used to define classes in namespaces other than pybind11. #3797
- Error printing code now uses `PYBIND11_DETAILED_ERROR_MESSAGES` instead of requiring `NDEBUG`, allowing use with release builds if desired. #3913
- Implicit conversion of the literal `0` to `pybind11::handle` is now disabled. #4008

Bug fixes:

- Fix exception handling when `pybind11::weakref()` fails. #3739
- `module_::def_submodule` was missing proper error handling. This is fixed now. #3973
- The behavior or `error_already_set` was made safer and the highly opaque "Unknown internal error occurred" message was replaced with a more helpful message. #3982
- `error_already_set::what()` now handles non-normalized exceptions correctly. #3971
- Support older C++ compilers where filesystem is not yet part of the standard library and is instead included in `std::experimental::filesystem`. #3840
- Fix `-Wfree-nonheap-object` warnings produced by GCC by avoiding returning pointers to static objects with `return_value_policy::take_ownership`. #3946
- Fix cast from pytype rvalue to another pytype. #3949
- Ensure proper behavior when garbage collecting classes with dynamic attributes in Python >=3.9. #4051
- A couple long-standing `PYBIND11_NAMESPACE __attribute__((visibility("hidden")))` inconsistencies are now fixed (affects only unusual environments). #4043

- `pybind11::detail::get_internals()` is now resilient to in-flight Python exceptions. #3981
- Arrays with a dimension of size 0 are now properly converted to dynamic Eigen matrices (more common in NumPy 1.23). #4038
- Avoid catching unrelated errors when importing NumPy. #3974

Performance and style:

- Added an accessor overload of `(object &&key)` to reference steal the object when using python types as keys. This prevents unnecessary reference count overhead for attr, dictionary, tuple, and sequence look ups. Added additional regression tests. Fixed a performance bug the caused accessor assignments to potentially perform unnecessary copies. #3970
- Perfect forward all args of `make_iterator`. #3980
- Avoid potential bug in pycapsule destructor by adding an `error_guard` to one of the dtors. #3958
- Optimize dictionary access in `strip_padding` for numpy. #3994
- `stl_bind.h` bindings now take slice args as a const-ref. #3852
- Made slice constructor more consistent, and improve performance of some casters by allowing reference stealing. #3845
- Change numpy dtype from_args method to use const ref. #3878
- Follow rule of three to ensure `PyErr_Restore` is called only once. #3872
- Added missing perfect forwarding for `make_iterator` functions. #3860
- Optimize c++ to python function casting by using the rvalue caster. #3966
- Optimize Eigen sparse matrix casting by removing unnecessary temporary. #4064
- Avoid potential implicit copy/assignment constructors causing double free in `strdup_guard`. #3905
- Enable clang-tidy checks `misc-definitions-in-headers`, `modernize-loop-convert`, and `modernize-use-nullptr`. #3881 #3988

Build system improvements:

- CMake: Fix file extension on Windows with cp36 and cp37 using FindPython. #3919
- CMake: Support multiple Python targets (such as on vcpkg). #3948
- CMake: Fix issue with NVCC on Windows. #3947
- CMake: Drop the bitness check on cross compiles (like targeting WebAssembly via Emscripten). #3959
- Add MSVC builds in debug mode to CI. #3784
- MSVC 2022 C++20 coverage was added to GitHub Actions, including Eigen. #3732, #3741

Backend and tidying up:

- New theme for the documentation. #3109
- Remove idioms in code comments. Use more inclusive language. #3809
- `#include <iostream>` was removed from the `pybind11/stl.h` header. Your project may break if it has a transitive dependency on this include. The fix is to "Include What You Use". #3928
- Avoid `setup.py <command>` usage in internal tests. #3734

1.25 Version 2.9.2 (Mar 29, 2022)

Changes:

- Enum now has an `__index__` method on Python <3.8 too. #3700
- Local internals are now cleared after finalizing the interpreter. #3744

Bug fixes:

- Better support for Python 3.11 alphas. #3694
- `PYBIND11_TYPE_CASTER` now uses fully qualified symbols, so it can be used outside of `pybind11::detail`. #3758
- Some fixes for PyPy 3.9. #3768
- Fixed a potential memleak in PyPy in `get_type_override`. #3774
- Fix usage of `VISIBILITY_INLINES_HIDDEN`. #3721

Build system improvements:

- Uses `sysconfig` module to determine installation locations on Python >= 3.10, instead of `distutils` which has been deprecated. #3764
- Support Catch 2.13.5+ (supporting GLIBC 2.34+). #3679
- Fix test failures with numpy 1.22 by ignoring whitespace when comparing `str()` of dtypes. #3682

Backend and tidying up:

- clang-tidy: added `readability-qualified-auto`, `readability-braces-around-statements`, `cppcoreguidelines-prefer-member-initializer`, `clang-analyzer-optin.performance.Padding`, `cppcoreguidelines-pro-type-static-cast-downcast`, and `readability-inconsistent-declaration-parameter-name`. #3702, #3699, #3716, #3709
- clang-format was added to the pre-commit actions, and the entire code base automatically reformatted (after several iterations preparing for this leap). #3713

1.26 Version 2.9.1 (Feb 2, 2022)

Changes:

- If possible, attach Python exception with `py::raise_from` to `TypeError` when casting from C++ to Python. This will give additional info if Python exceptions occur in the caster. Adds a test case of trying to convert a set from C++ to Python when the hash function is not defined in Python. #3605
- Add a mapping of C++11 nested exceptions to their Python exception equivalent using `py::raise_from`. This attaches the nested exceptions in Python using the `__cause__` field. #3608
- Propagate Python exception traceback using `raise_from` if a pybind11 function runs out of overloads. #3671
- `py::multiple_inheritance` is now only needed when C++ bases are hidden from pybind11. #3650 and #3659

Bug fixes:

- Remove a boolean cast in `numpy.h` that causes MSVC C4800 warnings when compiling against Python 3.10 or newer. #3669
- Render `py::bool_` and `py::float_` as `bool` and `float` respectively. #3622

Build system improvements:

- Fix CMake extension suffix computation on Python 3.10+. #3663
- Allow `CMAKE_ARGS` to override CMake args in pybind11's own `setup.py`. #3577
- Remove a few deprecated c-headers. #3610
- More uniform handling of test targets. #3590
- Add clang-tidy readability check to catch potentially swapped function args. #3611

1.27 Version 2.9.0 (Dec 28, 2021)

This is the last version to support Python 2.7 and 3.5.

New Features:

- Allow `py::args` to be followed by other arguments; the remaining arguments are implicitly keyword-only, as if a `py::kw_only{}` annotation had been used. #3402

Changes:

- Make str/bytes/memoryview more interoperable with `std::string_view`. #3521
- Replace `_` with `const_name` in internals, avoid defining `pybind::_` if `_` defined as macro (common gettext usage) #3423

Bug fixes:

- Fix a rare warning about extra copy in an Eigen constructor. #3486
- Fix caching of the C++ overrides. #3465
- Add missing `std::forward` calls to some `cpp_function` overloads. #3443
- Support PyPy 7.3.7 and the PyPy3.8 beta. Test python-3.11 on PRs with the `python dev` label. #3419
- Replace usage of deprecated `Eigen::MappedSparseMatrix` with `Eigen::Map<Eigen::SparseMatrix<...>>` for Eigen 3.3+. #3499
- Tweaks to support Microsoft Visual Studio 2022. #3497

Build system improvements:

- Nicer CMake printout and IDE organisation for pybind11's own tests. #3479
- CMake: report version type as part of the version string to avoid a spurious space in the package status message. #3472
- Flags starting with `-g` in `$CFLAGS` and `$CPPFLAGS` are no longer overridden by `.Pybind11Extension`. #3436
- Ensure ThreadPool is closed in `setup_helpers`. #3548
- Avoid LTS on `mips64` and `ppc64le` (reported broken). #3557

1.28 v2.8.1 (Oct 27, 2021)

Changes and additions:

- The simple namespace creation shortcut added in 2.8.0 was deprecated due to usage of CPython internal API, and will be removed soon. Use `py::module_::import("types").attr("SimpleNamespace")`. #3374
- Add C++ Exception type to throw and catch `AttributeError`. Useful for defining custom `__setattr__` and `__getattr__` methods. #3387

Fixes:

- Fixed the potential for dangling references when using properties with `std::optional` types. #3376
- Modernize usage of `PyCodeObject` on Python 3.9+ (moving toward support for Python 3.11a1) #3368
- A long-standing bug in `eigen.h` was fixed (originally PR #3343). The bug was unmasked by newly added `static_assert`'s in the Eigen 3.4.0 release. #3352
- Support multiple raw inclusion of CMake helper files (Conan.io does this for multi-config generators). #3420
- Fix harmless warning on upcoming CMake 3.22. #3368
- Fix 2.8.0 regression with MSVC 2017 + C++17 mode + Python 3. #3407
- Fix 2.8.0 regression that caused undefined behavior (typically segfaults) in `make_key_iterator/make_value_iterator` if dereferencing the iterator returned a temporary value instead of a reference. #3348

1.29 v2.8.0 (Oct 4, 2021)

New features:

- Added `py::raise_from` to enable chaining exceptions. #3215
- Allow exception translators to be optionally registered local to a module instead of applying globally across all pybind11 modules. Use `register_local_exception_translator(ExceptionTranslator&& translator)` instead of `register_exception_translator(ExceptionTranslator&& translator)` to keep your exception remapping code local to the module. #2650
- Add `make_simple_namespace` function for instantiating Python `SimpleNamespace` objects. **Deprecated in 2.8.1.** #2840
- `pybind11::scoped_interpreter` and `initialize_interpreter` have new arguments to allow `sys.argv` initialization. #2341
- Allow Python builtins to be used as callbacks in CPython. #1413
- Added `view` to view arrays with a different datatype. #987
- Implemented `reshape` on arrays. #984
- Enable defining custom `__new__` methods on classes by fixing bug preventing overriding methods if they have non-pybind11 siblings. #3265
- Add `make_value_iterator()`, and fix `make_key_iterator()` to return references instead of copies. #3293
- Improve the classes generated by `bind_map`: #3310
  - Change `.items` from an iterator to a dictionary view.
  - Add `.keys` and `.values` (both dictionary views).

  - Allow `__contains__` to take any object.

- `pybind11::custom_type_setup` was added, for customizing the `PyHeapTypeObject` corresponding to a class, which may be useful for enabling garbage collection support, among other things. #3287

Changes:

- Set `__file__` constant when running `eval_file` in an embedded interpreter. #3233
- Python objects and (C++17) `std::optional` now accepted in `py::slice` constructor. #1101
- The pybind11 proxy types `str`, `bytes`, `bytearray`, `tuple`, `list` now consistently support passing `ssize_t` values for sizes and indexes. Previously, only `size_t` was accepted in several interfaces. #3219
- Avoid evaluating `PYBIND11_TLS_REPLACE_VALUE` arguments more than once. #3290

Fixes:

- Bug fix: enum value's `__int__` returning non-int when underlying type is bool or of char type. #1334
- Fixes bug in setting error state in Capsule's pointer methods. #3261
- A long-standing memory leak in `py::cpp_function::initialize` was fixed. #3229
- Fixes thread safety for some `pybind11::type_caster` which require lifetime extension, such as for `std::string_view`. #3237
- Restore compatibility with gcc 4.8.4 as distributed by ubuntu-trusty, linuxmint-17. #3270

Build system improvements:

- Fix regression in CMake Python package config: improper use of absolute path. #3144
- Cached Python version information could become stale when CMake was re-run with a different Python version. The build system now detects this and updates this information. #3299
- Specified UTF8-encoding in setup.py calls of open(). #3137
- Fix a harmless warning from CMake 3.21 with the classic Python discovery. #3220
- Eigen repo and version can now be specified as cmake options. #3324

Backend and tidying up:

- Reduced thread-local storage required for keeping alive temporary data for type conversion to one key per ABI version, rather than one key per extension module. This makes the total thread-local storage required by pybind11 2 keys per ABI version. #3275
- Optimize NumPy array construction with additional moves. #3183
- Conversion to `std::string` and `std::string_view` now avoids making an extra copy of the data on Python >= 3.3. #3257
- Remove const modifier from certain C++ methods on Python collections (`list`, `set`, `dict`) such as (`clear()`, `append()`, `insert()`, etc...) and annotated them with `py-non-const`.
- Enable readability `clang-tidy-const-return` and remove useless consts. #3254 #3194
- The clang-tidy `google-explicit-constructor` option was enabled. #3250
- Mark a pytype move constructor as noexcept (perf). #3236
- Enable clang-tidy check to guard against inheritance slicing. #3210
- Legacy warning suppression pragma were removed from eigen.h. On Unix platforms, please use -isystem for Eigen include directories, to suppress compiler warnings originating from Eigen headers. Note that CMake does this by default. No adjustments are needed for Windows. #3198

- Format pybind11 with isort consistent ordering of imports #3195
- The warnings-suppression "pragma clamp" at the top/bottom of pybind11 was removed, clearing the path to refactoring and IWYU cleanup. #3186
- Enable most bugprone checks in clang-tidy and fix the found potential bugs and poor coding styles. #3166
- Add `clang-tidy-readability` rules to make boolean casts explicit improving code readability. Also enabled other misc and readability clang-tidy checks. #3148
- Move object in `.pop()` for list. #3116

1.30 v2.7.1 (Aug 3, 2021)

Minor missing functionality added:

- Allow Python builtins to be used as callbacks in CPython. #1413

Bug fixes:

- Fix regression in CMake Python package config: improper use of absolute path. #3144
- Fix Mingw64 and add to the CI testing matrix. #3132
- Specified UTF8-encoding in setup.py calls of open(). #3137
- Add clang-tidy-readability rules to make boolean casts explicit improving code readability. Also enabled other misc and readability clang-tidy checks. #3148
- Move object in `.pop()` for list. #3116

Backend and tidying up:

- Removed and fixed warning suppressions. #3127 #3129 #3135 #3141 #3142 #3150 #3152 #3160 #3161

1.31 v2.7.0 (Jul 16, 2021)

New features:

- Enable `py::implicitly_convertible<py::none, ...>` for `py::class_`-wrapped types. #3059
- Allow function pointer extraction from overloaded functions. #2944
- NumPy: added `.char_()` to type which gives the NumPy public `char` result, which also distinguishes types by bit length (unlike `.kind()`). #2864
- Add `pybind11::bytearray` to manipulate `bytearray` similar to `bytes`. #2799
- `pybind11/stl/filesystem.h` registers a type caster that, on C++17/Python 3.6+, converts `std::filesystem::path` to `pathlib.Path` and any `os.PathLike` to `std::filesystem::path`. #2730
- A `PYBIND11_VERSION_HEX` define was added, similar to `PY_VERSION_HEX`. #3120

Changes:

- `py::str` changed to exclusively hold `PyUnicodeObject`. Previously `py::str` could also hold `bytes`, which is probably surprising, was never documented, and can mask bugs (e.g. accidental use of `py::str` instead of `py::bytes`). #2409
- Add a safety guard to ensure that the Python GIL is held when C++ calls back into Python via `object_api<>::operator()` (e.g. `py::function __call__`). (This feature is available for Python 3.6+ only.) #2919

- Catch a missing `self` argument in calls to `__init__()`. #2914
- Use `std::string_view` if available to avoid a copy when passing an object to a `std::ostream`. #3042
- An important warning about thread safety was added to the `iostream.h` documentation; attempts to make `py::scoped_ostream_redirect` thread safe have been removed, as it was only partially effective. #2995

Fixes:

- Performance: avoid unnecessary strlen calls. #3058
- Fix auto-generated documentation string when using `const T` in `pyarray_t`. #3020
- Unify error messages thrown by `simple_collector`/`unpacking_collector`. #3013
- `pybind11::builtin_exception` is now explicitly exported, which means the types included/defined in different modules are identical, and exceptions raised in different modules can be caught correctly. The documentation was updated to explain that custom exceptions that are used across module boundaries need to be explicitly exported as well. #2999
- Fixed exception when printing UTF-8 to a `scoped_ostream_redirect`. #2982
- Pickle support enhancement: `setstate` implementation will attempt to `setattr __dict__` only if the unpickled `dict` object is not empty, to not force use of `py::dynamic_attr()` unnecessarily. #2972
- Allow negative timedelta values to roundtrip. #2870
- Fix unchecked errors could potentially swallow signals/other exceptions. #2863
- Add null pointer check with `std::localtime`. #2846
- Fix the `weakref` constructor from `py::object` to create a new `weakref` on conversion. #2832
- Avoid relying on exceptions in C++17 when getting a `shared_ptr` holder from a `shared_from_this` class. #2819
- Allow the codec's exception to be raised instead of `RuntimeError` when casting from `py::str` to `std::string`. #2903

Build system improvements:

- In `setup_helpers.py`, test for platforms that have some multiprocessing features but lack semaphores, which `ParallelCompile` requires. #3043
- Fix `pybind11_INCLUDE_DIR` in case `CMAKE_INSTALL_INCLUDEDIR` is absolute. #3005
- Fix bug not respecting `WITH_SOABI` or `WITHOUT_SOABI` to CMake. #2938
- Fix the default `Pybind11Extension` compilation flags with a Mingw64 python. #2921
- Clang on Windows: do not pass `/MP` (ignored flag). #2824
- `pybind11.setup_helpers.intree_extensions` can be used to generate `Pybind11Extension` instances from cpp files placed in the Python package source tree. #2831

Backend and tidying up:

- Enable clang-tidy performance, readability, and modernization checks throughout the codebase to enforce best coding practices. #3046, #3049, #3051, #3052, #3080, and #3094
- Checks for common misspellings were added to the pre-commit hooks. #3076
- Changed `Werror` to stricter `Werror-all` for Intel compiler and fixed minor issues. #2948
- Fixed compilation with GCC < 5 when the user defines `_GLIBCXX_USE_CXX11_ABI`. #2956
- Added nox support for easier local testing and linting of contributions. #3101 and #3121

- Avoid RTD style issue with docutils 0.17+. #3119
- Support pipx run, such as `pipx run pybind11 --include` for a quick compile. #3117

1.32 v2.6.2 (Jan 26, 2021)

Minor missing functionality added:

- enum: add missing Enum.value property. #2739
- Allow thread termination to be avoided during shutdown for CPython 3.7+ via `.disarm` for `gil_scoped_acquire/gil_scoped_release`. #2657

Fixed or improved behavior in a few special cases:

- Fix bug where the constructor of `object` subclasses would not throw on being passed a Python object of the wrong type. #2701
- The `type_caster` for integers does not convert Python objects with `__int__` anymore with `noconvert` or during the first round of trying overloads. #2698
- When casting to a C++ integer, `__index__` is always called and not considered as conversion, consistent with Python 3.8+. #2801

Build improvements:

- Setup helpers: `extra_compile_args` and `extra_link_args` automatically set by Pybind11Extension are now prepended, which allows them to be overridden by user-set `extra_compile_args` and `extra_link_args`. #2808
- Setup helpers: Don't trigger unused parameter warning. #2735
- CMake: Support running with `--warn-uninitialized` active. #2806
- CMake: Avoid error if included from two submodule directories. #2804
- CMake: Fix `STATIC` / `SHARED` being ignored in FindPython mode. #2796
- CMake: Respect the setting for `CMAKE_CXX_VISIBILITY_PRESET` if defined. #2793
- CMake: Fix issue with FindPython2/FindPython3 not working with `pybind11::embed`. #2662
- CMake: mixing local and installed pybind11's would prioritize the installed one over the local one (regression in 2.6.0). #2716

Bug fixes:

- Fixed segfault in multithreaded environments when using `scoped_ostream_redirect`. #2675
- Leave docstring unset when all docstring-related options are disabled, rather than set an empty string. #2745
- The module key in builtins that pybind11 uses to store its internals changed from std::string to a python str type (more natural on Python 2, no change on Python 3). #2814
- Fixed assertion error related to unhandled (later overwritten) exception in CPython 3.8 and 3.9 debug builds. #2685
- Fix `py::gil_scoped_acquire` assert with CPython 3.9 debug build. #2683
- Fix issue with a test failing on pytest 6.2. #2741

Warning fixes:

- Fix warning modifying constructor parameter 'flag' that shadows a field of 'set_flag' `[-Wshadow-field-in-constructor-modified]`. #2780

- Suppressed some deprecation warnings about old-style `__init__`/`__setstate__` in the tests. #2759

Valgrind work:

- Fix invalid access when calling a pybind11 `__init__` on a non-pybind11 class instance. #2755
- Fixed various minor memory leaks in pybind11's test suite. #2758
- Resolved memory leak in cpp_function initialization when exceptions occurred. #2756
- Added a Valgrind build, checking for leaks and memory-related UB, to CI. #2746

Compiler support:

- Intel compiler was not activating C++14 support due to a broken define. #2679
- Support ICC and NVIDIA HPC SDK in C++17 mode. #2729
- Support Intel OneAPI compiler (ICC 20.2) and add to CI. #2573

1.33 v2.6.1 (Nov 11, 2020)

- `py::exec`, `py::eval`, and `py::eval_file` now add the builtins module as `"__builtins__"` to their `globals` argument, better matching `exec` and `eval` in pure Python. #2616
- `setup_helpers` will no longer set a minimum macOS version higher than the current version. #2622
- Allow deleting static properties. #2629
- Seal a leak in `def_buffer`, cleaning up the `capture` object after the `class_` object goes out of scope. #2634
- `pybind11_INCLUDE_DIRS` was incorrect, potentially causing a regression if it was expected to include `PYTHON_INCLUDE_DIRS` (please use targets instead). #2636
- Added parameter names to the `py::enum_` constructor and methods, avoiding `arg0` in the generated docstrings. #2637
- Added `needs_recompile` optional function to the `ParallelCompiler` helper, to allow a recompile to be skipped based on a user-defined function. #2643

1.34 v2.6.0 (Oct 21, 2020)

See `upgrade-guide-2.6` for help upgrading to the new version.

New features:

- Keyword-only arguments supported in Python 2 or 3 with `py::kw_only()`. #2100
- Positional-only arguments supported in Python 2 or 3 with `py::pos_only()`. #2459
- `py::is_final()` class modifier to block subclassing (CPython only). #2151
- Added `py::prepend()`, allowing a function to be placed at the beginning of the overload chain. #1131
- Access to the type object now provided with `py::type::of<T>()` and `py::type::of(h)`. #2364
- Perfect forwarding support for methods. #2048
- Added `py::error_already_set::discard_as_unraisable()`. #2372
- `py::hash` is now public. #2217

- `py::class_<union_type>` is now supported. Note that writing to one data member of the union and reading another (type punning) is UB in C++. Thus pybind11-bound enums should never be used for such conversions. #2320.
- Classes now check local scope when registering members, allowing a subclass to have a member with the same name as a parent (such as an enum). #2335

Code correctness features:

- Error now thrown when `__init__` is forgotten on subclasses. #2152
- Throw error if conversion to a pybind11 type if the Python object isn't a valid instance of that type, such as `py::bytes(o)` when `py::object o` isn't a bytes instance. #2349
- Throw if conversion to `str` fails. #2477

API changes:

- `py::module` was renamed `py::module_` to avoid issues with C++20 when used unqualified, but an alias `py::module` is provided for backward compatibility. #2489
- Public constructors for `py::module_` have been deprecated; please use `pybind11::module_::create_extension_module` if you were using the public constructor (fairly rare after `PYBIND11_MODULE` was introduced). #2552
- `PYBIND11_OVERLOAD*` macros and `get_overload` function replaced by correctly-named `PYBIND11_OVERRIDE*` and `get_override`, fixing inconsistencies in the presence of a closing `;` in these macros. `get_type_overload` is deprecated. #2325

Packaging / building improvements:

- The Python package was reworked to be more powerful and useful. #2433
  - `build-setuptools` is easier thanks to a new `pybind11.setup_helpers` module, which provides utilities to use setuptools with pybind11. It can be used via PEP 518, `setup_requires`, or by directly importing or copying `setup_helpers.py` into your project.
  - CMake configuration files are now included in the Python package. Use `pybind11.get_cmake_dir()` or `python -m pybind11 --cmakedir` to get the directory with the CMake configuration files, or include the site-packages location in your `CMAKE_MODULE_PATH`. Or you can use the new `pybind11[global]` extra when you install `pybind11`, which installs the CMake files and headers into your base environment in the standard location.
  - `pybind11-config` is another way to write `python -m pybind11` if you have your PATH set up.
  - Added external typing support to the helper module, code from `import pybind11` can now be type checked. #2588
- Minimum CMake required increased to 3.4. #2338 and #2370
  - Full integration with CMake's C++ standard system and compile features replaces `PYBIND11_CPP_STANDARD`.
  - Generated config file is now portable to different Python/compiler/CMake versions.
  - Virtual environments prioritized if `PYTHON_EXECUTABLE` is not set (`venv`, `virtualenv`, and `conda`) (similar to the new FindPython mode).
  - Other CMake features now natively supported, like `CMAKE_INTERPROCEDURAL_OPTIMIZATION`, `set(CMAKE_CXX_VISIBILITY_PRESET hidden)`.
  - `CUDA` as a language is now supported.
  - Helper functions `pybind11_strip`, `pybind11_extension`, `pybind11_find_import` added, see `cmake/index`.

  - Optional `find-python-mode` and `nopython-mode` with CMake. #2370

- Uninstall target added. #2265 and #2346
- `pybind11_add_module()` now accepts an optional `OPT_SIZE` flag that switches the binding target to size-based optimization if the global build type can not always be fixed to `MinSizeRel` (except in debug mode, where optimizations remain disabled). `MinSizeRel` or this flag reduces binary size quite substantially (~25% on some platforms). #2463

Smaller or developer focused features and fixes:

- Moved `mkdoc.py` to a new repo, pybind11-mkdoc. There are no longer submodules in the main repo.
- `py::memoryview` segfault fix and update, with new `py::memoryview::from_memory` in Python 3, and documentation. #2223
- Fix for `buffer_info` on Python 2. #2503
- If `__eq__` defined but not `__hash__`, `__hash__` is now set to `None`. #2291
- `py::ellipsis` now also works on Python 2. #2360
- Pointer to `std::tuple` & `std::pair` supported in cast. #2334
- Small fixes in NumPy support. `py::array` now uses `py::ssize_t` as first argument type. #2293
- Added missing signature for `py::array`. #2363
- `unchecked_mutable_reference` has access to operator `()` and `[]` when const. #2514
- `py::vectorize` is now supported on functions that return void. #1969
- `py::capsule` supports `get_pointer` and `set_pointer`. #1131
- Fix crash when different instances share the same pointer of the same type. #2252
- Fix for `py::len` not clearing Python's error state when it fails and throws. #2575
- Bugfixes related to more extensive testing, new GitHub Actions CI. #2321
- Bug in timezone issue in Eastern hemisphere midnight fixed. #2438
- `std::chrono::time_point` now works when the resolution is not the same as the system. #2481
- Bug fixed where `py::array_t` could accept arrays that did not match the requested ordering. #2484
- Avoid a segfault on some compilers when types are removed in Python. #2564
- `py::arg::none()` is now also respected when passing keyword arguments. #2611
- PyPy fixes, PyPy 7.3.x now supported, including PyPy3. (Known issue with PyPy2 and Windows #2596). #2146
- CPython 3.9.0 workaround for undefined behavior (macOS segfault). #2576
- CPython 3.9 warning fixes. #2253
- Improved C++20 support, now tested in CI. #2489 #2599
- Improved but still incomplete debug Python interpreter support. #2025
- NVCC (CUDA 11) now supported and tested in CI. #2461
- NVIDIA PGI compilers now supported and tested in CI. #2475
- At least Intel 18 now explicitly required when compiling with Intel. #2577
- Extensive style checking in CI, with pre-commit support. Code modernization, checked by clang-tidy.
- Expanded docs, including new main page, new installing section, and CMake helpers page, along with over a dozen new sections on existing pages.

- In GitHub, new docs for contributing and new issue templates.

1.35 v2.5.0 (Mar 31, 2020)

- Use C++17 fold expressions in type casters, if available. This can improve performance during overload resolution when functions have multiple arguments. #2043.
- Changed include directory resolution in `pybind11/__init__.py` and installation in `setup.py`. This fixes a number of open issues where pybind11 headers could not be found in certain environments. #1995.
- C++20 `char8_t` and `u8string` support. #2026.
- CMake: search for Python 3.9. bb9c91.
- Fixes for MSYS-based build environments. #2087, #2053.
- STL bindings for `std::vector<...>::clear`. #2074.
- Read-only flag for `py::buffer`. #1466.
- Exception handling during module initialization. bf2b031.
- Support linking against a CPython debug build. #2025.
- Fixed issues involving the availability and use of aligned `new` and `delete`. #1988, 759221.
- Fixed a resource leak upon interpreter shutdown. #2020.
- Fixed error handling in the boolean caster. #1976.

1.36 v2.4.3 (Oct 15, 2019)

- Adapt pybind11 to a C API convention change in Python 3.8. #1950.

1.37 v2.4.2 (Sep 21, 2019)

- Replaced usage of a C++14 only construct. #1929.
- Made an ifdef future-proof for Python >= 4. f3109d.

1.38 v2.4.1 (Sep 20, 2019)

- Fixed a problem involving implicit conversion from enumerations to integers on Python 3.8. #1780.

1.39 v2.4.0 (Sep 19, 2019)

- Try harder to keep pybind11-internal data structures separate when there are potential ABI incompatibilities. Fixes crashes that occurred when loading multiple pybind11 extensions that were e.g. compiled by GCC (libstdc++) and Clang (libc++). #1588 and c9f5a.
- Added support for `__await__`, `__aiter__`, and `__anext__` protocols. #1842.
- `pybind11_add_module()`: don't strip symbols when compiling in `RelWithDebInfo` mode. #1980.
- `enum_`: Reproduce Python behavior when comparing against invalid values (e.g. `None`, strings, etc.). Add back support for `__invert__()`. #1912, #1907.
- List insertion operation for `py::list`. Added `.empty()` to all collection types. Added `py::set::contains()` and `py::dict::contains()`. #1887, #1884, #1888.
- `py::details::overload_cast_impl` is available in C++11 mode, can be used like `overload_cast` with an additional set of parentheses. #1581.
- Fixed `get_include()` on Conda. #1877.
- `stl_bind.h`: negative indexing support. #1882.
- Minor CMake fix to add MinGW compatibility. #1851.
- GIL-related fixes. #1836, 8b90b.
- Other very minor/subtle fixes and improvements. #1329, #1910, #1863, #1847, #1890, #1860, #1848, #1821, #1837, #1833, #1748, #1852.

1.40 v2.3.0 (June 11, 2019)

- Significantly reduced module binary size (10-20%) when compiled in C++11 mode with GCC/Clang, or in any mode with MSVC. Function signatures are now always precomputed at compile time (this was previously only available in C++14 mode for non-MSVC compilers). #934.
- Add basic support for tag-based static polymorphism, where classes provide a method to returns the desired type of an instance. #1326.
- Python type wrappers (`py::handle`, `py::object`, etc.) now support map Python's number protocol onto C++ arithmetic operators such as `operator+`, `operator/=`, etc. #1511.
- A number of improvements related to enumerations:
  1. The `enum_` implementation was rewritten from scratch to reduce code bloat. Rather than instantiating a full implementation for each enumeration, most code is now contained in a generic base class. #1511.
  2. The `value()` method of `py::enum_` now accepts an optional docstring that will be shown in the documentation of the associated enumeration. #1160.
  3. check for already existing enum value and throw an error if present. #1453.
- Support for over-aligned type allocation via C++17's aligned `new` statement. #1582.
- Added `py::ellipsis()` method for slicing of multidimensional NumPy arrays #1502.
- Numerous Improvements to the `mkdoc.py` script for extracting documentation from C++ header files. #1788.
- `pybind11_add_module()`: allow including Python as a `SYSTEM` include path. #1416.
- `pybind11/stl.h` does not convert strings to `vector<string>` anymore. #1258.
- Mark static methods as such to fix auto-generated Sphinx documentation. #1732.

- Re-throw forced unwind exceptions (e.g. during pthread termination). #1208.
- Added `__contains__` method to the bindings of maps (`std::map`, `std::unordered_map`). #1767.
- Improvements to `gil_scoped_acquire`. #1211.
- Type caster support for `std::deque<T>`. #1609.
- Support for `std::unique_ptr` holders, whose deleters differ between a base and derived class. #1353.
- Construction of STL array/vector-like data structures from iterators. Added an `extend()` operation. #1709,
- CMake build system improvements for projects that include non-C++ files (e.g. plain C, CUDA) in `pybind11_add_module` et al. #1678.
- Fixed asynchronous invocation and deallocation of Python functions wrapped in `std::function`. #1595.
- Fixes regarding return value policy propagation in STL type casters. #1603.
- Fixed scoped enum comparisons. #1571.
- Fixed iostream redirection for code that releases the GIL. #1368,
- A number of CI-related fixes. #1757, #1744, #1670.

1.41 v2.2.4 (September 11, 2018)

- Use new Python 3.7 Thread Specific Storage (TSS) implementation if available. #1454, #1517.
- Fixes for newer MSVC versions and C++17 mode. #1347, #1462.
- Propagate return value policies to type-specific casters when casting STL containers. #1455.
- Allow ostream-redirection of more than 1024 characters. #1479.
- Set `Py_DEBUG` define when compiling against a debug Python build. #1438.
- Untangle integer logic in number type caster to work for custom types that may only be castable to a restricted set of builtin types. #1442.
- CMake build system: Remember Python version in cache file. #1434.
- Fix for custom smart pointers: use `std::addressof` to obtain holder address instead of `operator&`. #1435.
- Properly report exceptions thrown during module initialization. #1362.
- Fixed a segmentation fault when creating empty-shaped NumPy array. #1371.
- The version of Intel C++ compiler must be >= 2017, and this is now checked by the header files. #1363.
- A few minor typo fixes and improvements to the test suite, and patches that silence compiler warnings.
- Vectors now support construction from generators, as well as `extend()` from a list or generator. #1496.

1.42 v2.2.3 (April 29, 2018)

- The pybind11 header location detection was replaced by a new implementation that no longer depends on `pip` internals (the recently released `pip` 10 has restricted access to this API). #1190.
- Small adjustment to an implementation detail to work around a compiler segmentation fault in Clang 3.3/3.4. #1350.
- The minimal supported version of the Intel compiler was >= 17.0 since pybind11 v2.1. This check is now explicit, and a compile-time error is raised if the compiler meet the requirement. #1363.
- Fixed an endianness-related fault in the test suite. #1287.

1.43 v2.2.2 (February 7, 2018)

- Fixed a segfault when combining embedded interpreter shutdown/reinitialization with external loaded pybind11 modules. #1092.
- Eigen support: fixed a bug where Nx1/1xN numpy inputs couldn't be passed as arguments to Eigen vectors (which for Eigen are simply compile-time fixed Nx1/1xN matrices). #1106.
- Clarified to license by moving the licensing of contributions from `LICENSE` into `CONTRIBUTING.md`: the licensing of contributions is not actually part of the software license as distributed. This isn't meant to be a substantial change in the licensing of the project, but addresses concerns that the clause made the license non-standard. #1109.
- Fixed a regression introduced in 2.1 that broke binding functions with lvalue character literal arguments. #1128.
- MSVC: fix for compilation failures under /permissive-, and added the flag to the appveyor test suite. #1155.
- Fixed `__qualname__` generation, and in turn, fixes how class names (especially nested class names) are shown in generated docstrings. #1171.
- Updated the FAQ with a suggested project citation reference. #1189.
- Added fixes for deprecation warnings when compiled under C++17 with `-Wdeprecated` turned on, and add `-Wdeprecated` to the test suite compilation flags. #1191.
- Fixed outdated PyPI URLs in `setup.py`. #1213.
- Fixed a refcount leak for arguments that end up in a `py::args` argument for functions with both fixed positional and `py::args` arguments. #1216.
- Fixed a potential segfault resulting from possible premature destruction of `py::args`/`py::kwargs` arguments with overloaded functions. #1223.
- Fixed `del map[item]` for a `stl_bind.h` bound stl map. #1229.
- Fixed a regression from v2.1.x where the aggregate initialization could unintentionally end up at a constructor taking a templated `std::initializer_list<T>` argument. #1249.
- Fixed an issue where calling a function with a keep_alive policy on the same nurse/patient pair would cause the internal patient storage to needlessly grow (unboundedly, if the nurse is long-lived). #1251.
- Various other minor fixes.

1.44 v2.2.1 (September 14, 2017)

- Added `py::module_::reload()` member function for reloading a module. #1040.
- Fixed a reference leak in the number converter. #1078.
- Fixed compilation with Clang on host GCC < 5 (old libstdc++ which isn't fully C++11 compliant). #1062.
- Fixed a regression where the automatic `std::vector<bool>` caster would fail to compile. The same fix also applies to any container which returns element proxies instead of references. #1053.
- Fixed a regression where the `py::keep_alive` policy could not be applied to constructors. #1065.
- Fixed a nullptr dereference when loading a `py::module_local` type that's only registered in an external module. #1058.
- Fixed implicit conversion of accessors to types derived from `py::object`. #1076.
- The `name` in `PYBIND11_MODULE(name, variable)` can now be a macro. #1082.
- Relaxed overly strict `py::pickle()` check for matching get and set types. #1064.
- Conversion errors now try to be more informative when it's likely that a missing header is the cause (e.g. forgetting `<pybind11/stl.h>`). #1077.

1.45 v2.2.0 (August 31, 2017)

- Support for embedding the Python interpreter. See the `documentation page </advanced/embedding>` for a full overview of the new features. #774, #889, #892, #920.

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

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

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

- Support for inheriting from multiple C++ bases in Python. #693.

```
from cpp_module import CppBase1, CppBase2


class PyDerived(CppBase1, CppBase2):
    def __init__(self):
        CppBase1.__init__(self)  # C++ bases must be initialized explicitly
        CppBase2.__init__(self)
```

- `PYBIND11_MODULE` is now the preferred way to create module entry points. `PYBIND11_PLUGIN` is deprecated. See `macros` for details. #879.

```
// new
PYBIND11_MODULE(example, m) {
    m.def("add", [](int a, int b) { return a + b; });
}
```

(continues on next page)

(continued from previous page)

```
// old
PYBIND11_PLUGIN(example) {
    py::module m("example");
    m.def("add", [](int a, int b) { return a + b; });
    return m.ptr();
}
```

- pybind11's headers and build system now more strictly enforce hidden symbol visibility for extension modules. This should be seamless for most users, but see the `upgrade` if you use a custom build system. #995.
- Support for `py::module_local` types which allow multiple modules to export the same C++ types without conflicts. This is useful for opaque types like `std::vector<int>`. `py::bind_vector` and `py::bind_map` now default to `py::module_local` if their elements are builtins or local types. See `module_local` for details. #949, #981, #995, #997.
- Custom constructors can now be added very easily using lambdas or factory functions which return a class instance by value, pointer or holder. This supersedes the old placement-new `__init__` technique. See `custom_constructors` for details. #805, #1014.

```
struct Example {
    Example(std::string);
};

py::class_<Example>(m, "Example")
    .def(py::init<std::string>()) // existing constructor
    .def(py::init([](int n) { // custom constructor
        return std::make_unique<Example>(std::to_string(n));
    }));
```

- Similarly to custom constructors, pickling support functions are now bound using the `py::pickle()` adaptor which improves type safety. See the `upgrade` and `pickling` for details. #1038.
- Builtin support for converting C++17 standard library types and general conversion improvements:
  1. C++17 `std::variant` is supported right out of the box. C++11/14 equivalents (e.g. `boost::variant`) can also be added with a simple user-defined specialization. See `cpp17_container_casters` for details. #811, #845, #989.
  2. Out-of-the-box support for C++17 `std::string_view`. #906.
  3. Improved compatibility of the builtin `optional` converter. #874.
  4. The `bool` converter now accepts `numpy.bool_` and types which define `__bool__` (Python 3.x) or `__nonzero__` (Python 2.7). #925.
  5. C++-to-Python casters are now more efficient and move elements out of rvalue containers whenever possible. #851, #936, #938.
  6. Fixed `bytes` to `std::string/char*` conversion on Python 3. #817.
  7. Fixed lifetime of temporary C++ objects created in Python-to-C++ conversions. #924.
- Scope guard call policy for RAII types, e.g. `py::call_guard<py::gil_scoped_release>()`, `py::call_guard<py::scoped_ostream_redirect>()`. See `call_policies` for details. #740.
- Utility for redirecting C++ streams to Python (e.g. `std::cout -> sys.stdout`). Scope guard `py::scoped_ostream_redirect` in C++ and a context manager in Python. See `ostream_redirect`. #1009.

- Improved handling of types and exceptions across module boundaries. #915, #951, #995.
- Fixed destruction order of `py::keep_alive` nurse/patient objects in reference cycles. #856.
- NumPy and buffer protocol related improvements:
  1. Support for negative strides in Python buffer objects/numpy arrays. This required changing integers from unsigned to signed for the related C++ APIs. Note: If you have compiler warnings enabled, you may notice some new conversion warnings after upgrading. These can be resolved with `static_cast`. #782.
  2. Support `std::complex` and arrays inside `PYBIND11_NUMPY_DTYPE`. #831, #832.
  3. Support for constructing `py::buffer_info` and `py::arrays` using arbitrary containers or iterators instead of requiring a `std::vector`. #788, #822, #860.
  4. Explicitly check numpy version and require >= 1.7.0. #819.
- Support for allowing/prohibiting `None` for specific arguments and improved `None` overload resolution order. See `none_arguments` for details. #843. #859.
- Added `py::exec()` as a shortcut for `py::eval<py::eval_statements>()` and support for C++11 raw string literals as input. See `eval`. #766, #827.
- `py::vectorize()` ignores non-vectorizable arguments and supports member functions. #762.
- Support for bound methods as callbacks (`pybind11/functional.h`). #815.
- Allow aliasing pybind11 methods: `cls.attr("foo") = cls.attr("bar")`. #802.
- Don't allow mixed static/non-static overloads. #804.
- Fixed overriding static properties in derived classes. #784.
- Added support for write only properties. #1144.
- Improved deduction of member functions of a derived class when its bases aren't registered with pybind11. #855.

```
struct Base {
    int foo() { return 42; }
}

struct Derived : Base {}

// Now works, but previously required also binding `Base`
py::class_<Derived>(m, "Derived")
    .def("foo", &Derived::foo); // function is actually from `Base`
```

- The implementation of `py::init<>` now uses C++11 brace initialization syntax to construct instances, which permits binding implicit constructors of aggregate types. #1015.

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

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

- Fixed issues with multiple inheritance with offset base/derived pointers. #812, #866, #960.
- Fixed reference leak of type objects. #1030.
- Improved support for the `/std:c++14` and `/std:c++latest` modes on MSVC 2017. #841, #999.

- Fixed detection of private operator new on MSVC. #893, #918.
- Intel C++ compiler compatibility fixes. #937.
- Fixed implicit conversion of `py::enum_` to integer types on Python 2.7. #821.
- Added `py::hash` to fetch the hash value of Python objects, and `.def(hash(py::self))` to provide the C++ `std::hash` as the Python `__hash__` method. #1034.
- Fixed `__truediv__` on Python 2 and `__itruediv__` on Python 3. #867.
- `py::capsule` objects now support the `name` attribute. This is useful for interfacing with `scipy.LowLevelCallable`. #902.
- Fixed `py::make_iterator`'s `__next__()` for past-the-end calls. #897.
- Added `error_already_set::matches()` for checking Python exceptions. #772.
- Deprecated `py::error_already_set::clear()`. It's no longer needed following a simplification of the `py::error_already_set` class. #954.
- Deprecated `py::handle::operator==()` in favor of `py::handle::is()` #825.
- Deprecated `py::object::borrowed/py::object::stolen`. Use `py::object::borrowed_t{}/py::object::stolen_t{}` instead. #771.
- Changed internal data structure versioning to avoid conflicts between modules compiled with different revisions of pybind11. #1012.
- Additional compile-time and run-time error checking and more informative messages. #786, #794, #803.
- Various minor improvements and fixes. #764, #791, #795, #840, #844, #846, #849, #858, #862, #871, #872, #881, #888, #899, #928, #931, #944, #950, #952, #962, #965, #970, #978, #979, #986, #1020, #1027, #1037.
- Testing improvements. #798, #882, #898, #900, #921, #923, #963.

1.46 v2.1.1 (April 7, 2017)

- Fixed minimum version requirement for MSVC 2015u3 #773.

1.47 v2.1.0 (March 22, 2017)

- pybind11 now performs function overload resolution in two phases. The first phase only considers exact type matches, while the second allows for implicit conversions to take place. A special `noconvert()` syntax can be used to completely disable implicit conversions for specific arguments. #643, #634, #650.
- Fixed a regression where static properties no longer worked with classes using multiple inheritance. The `py::metaclass` attribute is no longer necessary (and deprecated as of this release) when binding classes with static properties. #679,
- Classes bound using `pybind11` can now use custom metaclasses. #679,
- `py::args` and `py::kwargs` can now be mixed with other positional arguments when binding functions using pybind11. #611.
- Improved support for C++11 unicode string and character types; added extensive documentation regarding pybind11's string conversion behavior. #624, #636, #715.
- pybind11 can now avoid expensive copies when converting Eigen arrays to NumPy arrays (and vice versa). #610.

- The "fast path" in `py::vectorize` now works for any full-size group of C or F-contiguous arrays. The non-fast path is also faster since it no longer performs copies of the input arguments (except when type conversions are necessary). #610.
- Added fast, unchecked access to NumPy arrays via a proxy object. #746.
- Transparent support for class-specific `operator new` and `operator delete` implementations. #755.
- Slimmer and more efficient STL-compatible iterator interface for sequence types. #662.
- Improved custom holder type support. #607.
- `nullptr` to `None` conversion fixed in various builtin type casters. #732.
- `enum_` now exposes its members via a special `__members__` attribute. #666.
- `std::vector` bindings created using `stl_bind.h` can now optionally implement the buffer protocol. #488.
- Automated C++ reference documentation using doxygen and breathe. #598.
- Added minimum compiler version assertions. #727.
- Improved compatibility with C++1z. #677.
- Improved `py::capsule` API. Can be used to implement cleanup callbacks that are involved at module destruction time. #752.
- Various minor improvements and fixes. #595, #588, #589, #603, #619, #648, #695, #720, #723, #729, #724, #742, #753.

1.48 v2.0.1 (Jan 4, 2017)

- Fix pointer to reference error in type_caster on MSVC #583.
- Fixed a segmentation in the test suite due to a typo cd7eac.

1.49 v2.0.0 (Jan 1, 2017)

- Fixed a reference counting regression affecting types with custom metaclasses (introduced in v2.0.0-rc1). #571.
- Quenched a CMake policy warning. #570.

1.50 v2.0.0-rc1 (Dec 23, 2016)

The pybind11 developers are excited to issue a release candidate of pybind11 with a subsequent v2.0.0 release planned in early January next year.

An incredible amount of effort by went into pybind11 over the last ~5 months, leading to a release that is jam-packed with exciting new features and numerous usability improvements. The following list links PRs or individual commits whenever applicable.

Happy Christmas!

- Support for binding C++ class hierarchies that make use of multiple inheritance. #410.

- PyPy support: pybind11 now supports nightly builds of PyPy and will interoperate with the future 5.7 release. No code changes are necessary, everything "just" works as usual. Note that we only target the Python 2.7 branch for now; support for 3.x will be added once its `cpyext` extension support catches up. A few minor features remain unsupported for the time being (notably dynamic attributes in custom types). #527.
- Significant work on the documentation – in particular, the monolithic `advanced.rst` file was restructured into a easier to read hierarchical organization. #448.
- Many NumPy-related improvements:
  1. Object-oriented API to access and modify NumPy `ndarray` instances, replicating much of the corresponding NumPy C API functionality. #402.
  2. NumPy array `dtype` array descriptors are now first-class citizens and are exposed via a new class `py::dtype`.
  3. Structured dtypes can be registered using the `PYBIND11_NUMPY_DTYPE()` macro. Special `array` constructors accepting dtype objects were also added.

     One potential caveat involving this change: format descriptor strings should now be accessed via `format_descriptor::format()` (however, for compatibility purposes, the old syntax `format_descriptor::value` will still work for non-structured data types). #308.
  4. Further improvements to support structured dtypes throughout the system. #472, #474, #459, #453, #452, and #505.
  5. Fast access operators. #497.
  6. Constructors for arrays whose storage is owned by another object. #440.
  7. Added constructors for `array` and `array_t` explicitly accepting shape and strides; if strides are not provided, they are deduced assuming C-contiguity. Also added simplified constructors for 1-dimensional case.
  8. Added buffer/NumPy support for `char[N]` and `std::array<char, N>` types.
  9. Added `memoryview` wrapper type which is constructible from `buffer_info`.
- Eigen: many additional conversions and support for non-contiguous arrays/slices. #427, #315, #316, #312, and #267
- Incompatible changes in `class_<...>::class_()`:
  1. Declarations of types that provide access via the buffer protocol must now include the `py::buffer_protocol()` annotation as an argument to the `class_` constructor.
  2. Declarations of types that require a custom metaclass (i.e. all classes which include static properties via commands such as `def_readwrite_static()`) must now include the `py::metaclass()` annotation as an argument to the `class_` constructor.

     These two changes were necessary to make type definitions in pybind11 future-proof, and to support PyPy via its cpyext mechanism. #527.
  3. This version of pybind11 uses a redesigned mechanism for instantiating trampoline classes that are used to override virtual methods from within Python. This led to the following user-visible syntax change: instead of

     ```
     py::class_<TrampolineClass>("MyClass")
       .alias<MyClass>()
       ....
     ```

     write

```
py::class_<MyClass, TrampolineClass>("MyClass")
  ....
```

Importantly, both the original and the trampoline class are now specified as an arguments (in arbitrary order) to the `py::class_` template, and the `alias<..>()` call is gone. The new scheme has zero overhead in cases when Python doesn't override any functions of the underlying C++ class. rev. 86d825.

- Added `eval` and `eval_file` functions for evaluating expressions and statements from a string or file. rev. 0d3fc3.
- pybind11 can now create types with a modifiable dictionary. #437 and #444.
- Support for translation of arbitrary C++ exceptions to Python counterparts. #296 and #273.
- Report full backtraces through mixed C++/Python code, better reporting for import errors, fixed GIL management in exception processing. #537, #494, rev. e72d95, and rev. 099d6e.
- Support for bit-level operations, comparisons, and serialization of C++ enumerations. #503, #508, #380, #309. #311.
- The `class_` constructor now accepts its template arguments in any order. #385.
- Attribute and item accessors now have a more complete interface which makes it possible to chain attributes as in `obj.attr("a")[key].attr("b").attr("method")(1, 2, 3)`. #425.
- Major redesign of the default and conversion constructors in `pytypes.h`. #464.
- Added built-in support for `std::shared_ptr` holder type. It is no longer necessary to to include a declaration of the form `PYBIND11_DECLARE_HOLDER_TYPE(T, std::shared_ptr<T>)` (though continuing to do so won't cause an error). #454.
- New `py::overload_cast` casting operator to select among multiple possible overloads of a function. An example:

```
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");
```

This feature only works on C++14-capable compilers. #541.

- C++ types are automatically cast to Python types, e.g. when assigning them as an attribute. For instance, the following is now legal:

```
py::module m = /* ... */
m.attr("constant") = 123;
```

(Previously, a `py::cast` call was necessary to avoid a compilation error.) #551.

- Redesigned `pytest`-based test suite. #321.
- Instance tracking to detect reference leaks in test suite. #324
- pybind11 can now distinguish between multiple different instances that are located at the same memory address, but which have different types. #329.
- Improved logic in `move` return value policy. #510, #297.
- Generalized unpacking API to permit calling Python functions from C++ using notation such as `foo(a1, a2, *args, "ka"_a=1, "kb"_a=2, **kwargs)`. #372.
- `py::print()` function whose behavior matches that of the native Python `print()` function. #372.

- Added `py::dict` keyword constructor:`auto d = dict("number"_a=42, "name"_a="World");`. #372.
- Added `py::str::format()` method and `_s` literal: `py::str s = "1 + 2 = {}"_s.format(3);`. #372.
- Added `py::repr()` function which is equivalent to Python's builtin `repr()`. #333.
- Improved construction and destruction logic for holder types. It is now possible to reference instances with smart pointer holder types without constructing the holder if desired. The `PYBIND11_DECLARE_HOLDER_TYPE` macro now accepts an optional second parameter to indicate whether the holder type uses intrusive reference counting. #533 and #561.
- Mapping a stateless C++ function to Python and back is now "for free" (i.e. no extra indirections or argument conversion overheads). rev. 954b79.
- Bindings for `std::valarray<T>`. #545.
- Improved support for C++17 capable compilers. #562.
- Bindings for `std::optional<t>`. #475, #476, #479, #499, and #501.
- `stl_bind.h`: general improvements and support for `std::map` and `std::unordered_map`. #490, #282, #235.
- The `std::tuple`, `std::pair`, `std::list`, and `std::vector` type casters now accept any Python sequence type as input. rev. 107285.
- Improved CMake Python detection on multi-architecture Linux. #532.
- Infrastructure to selectively disable or enable parts of the automatically generated docstrings. #486.
- `reference` and `reference_internal` are now the default return value properties for static and non-static properties, respectively. #473. (the previous defaults were `automatic`). #473.
- Support for `std::unique_ptr` with non-default deleters or no deleter at all (`py::nodelete`). #384.
- Deprecated `handle::call()` method. The new syntax to call Python functions is simply `handle()`. It can also be invoked explicitly via `handle::operator<X>()`, where `X` is an optional return value policy.
- Print more informative error messages when `make_tuple()` or `cast()` fail. #262.
- Creation of holder types for classes deriving from `std::enable_shared_from_this<>` now also works for `const` values. #260.
- `make_iterator()` improvements for better compatibility with various types (now uses prefix increment operator); it now also accepts iterators with different begin/end types as long as they are equality comparable. #247.
- `arg()` now accepts a wider range of argument types for default values. #244.
- Support `keep_alive` where the nurse object may be `None`. #341.
- Added constructors for `str` and `bytes` from zero-terminated char pointers, and from char pointers and length. Added constructors for `str` from `bytes` and for `bytes` from `str`, which will perform UTF-8 decoding/encoding as required.
- Many other improvements of library internals without user-visible changes

1.51 1.8.1 (July 12, 2016)

- Fixed a rare but potentially very severe issue when the garbage collector ran during pybind11 type creation.

1.52 1.8.0 (June 14, 2016)

- Redesigned CMake build system which exports a convenient `pybind11_add_module` function to parent projects.
- `std::vector<>` type bindings analogous to Boost.Python's `indexing_suite`
- Transparent conversion of sparse and dense Eigen matrices and vectors (`eigen.h`)
- Added an `ExtraFlags` template argument to the NumPy `array_t<>` wrapper to disable an enforced cast that may lose precision, e.g. to create overloads for different precisions and complex vs real-valued matrices.
- Prevent implicit conversion of floating point values to integral types in function arguments
- Fixed incorrect default return value policy for functions returning a shared pointer
- Don't allow registering a type via `class_` twice
- Don't allow casting a `None` value into a C++ lvalue reference
- Fixed a crash in `enum_::operator==` that was triggered by the `help()` command
- Improved detection of whether or not custom C++ types can be copy/move-constructed
- Extended `str` type to also work with `bytes` instances
- Added a `"name"_a` user defined string literal that is equivalent to `py::arg("name")`.
- When specifying function arguments via `py::arg`, the test that verifies the number of arguments now runs at compile time.
- Added `[[noreturn]]` attribute to `pybind11_fail()` to quench some compiler warnings
- List function arguments in exception text when the dispatch code cannot find a matching overload
- Added `PYBIND11_OVERLOAD_NAME` and `PYBIND11_OVERLOAD_PURE_NAME` macros which can be used to override virtual methods whose name differs in C++ and Python (e.g. `__call__` and `operator()`)
- Various minor `iterator` and `make_iterator()` improvements
- Transparently support `__bool__` on Python 2.x and Python 3.x
- Fixed issue with destructor of unpickled object not being called
- Minor CMake build system improvements on Windows
- New `pybind11::args` and `pybind11::kwargs` types to create functions which take an arbitrary number of arguments and keyword arguments
- New syntax to call a Python function from C++ using `*args` and `*kwargs`
- The functions `def_property_*` now correctly process docstring arguments (these formerly caused a segmentation fault)
- Many `mkdoc.py` improvements (enumerations, template arguments, `DOC()` macro accepts more arguments)
- Cygwin support
- Documentation improvements (pickling support, `keep_alive`, macro usage)

1.53 1.7 (April 30, 2016)

- Added a new `move` return value policy that triggers C++11 move semantics. The automatic return value policy falls back to this case whenever a rvalue reference is encountered
- Significantly more general GIL state routines that are used instead of Python's troublesome `PyGILState_Ensure` and `PyGILState_Release` API
- Redesign of opaque types that drastically simplifies their usage
- Extended ability to pass values of type `[const] void *`
- `keep_alive` fix: don't fail when there is no patient
- `functional.h`: acquire the GIL before calling a Python function
- Added Python RAII type wrappers `none` and `iterable`
- Added `*args` and `*kwargs` pass-through parameters to `pybind11.get_include()` function
- Iterator improvements and fixes
- Documentation on return value policies and opaque types improved

1.54 1.6 (April 30, 2016)

- Skipped due to upload to PyPI gone wrong and inability to recover (https://github.com/pypa/packaging-problems/issues/74)

1.55 1.5 (April 21, 2016)

- For polymorphic types, use RTTI to try to return the closest type registered with pybind11
- Pickling support for serializing and unserializing C++ instances to a byte stream in Python
- Added a convenience routine `make_iterator()` which turns a range indicated by a pair of C++ iterators into a iterable Python object
- Added `len()` and a variadic `make_tuple()` function
- Addressed a rare issue that could confuse the current virtual function dispatcher and another that could lead to crashes in multi-threaded applications
- Added a `get_include()` function to the Python module that returns the path of the directory containing the installed pybind11 header files
- Documentation improvements: import issues, symbol visibility, pickling, limitations
- Added casting support for `std::reference_wrapper<>`

1.56 1.4 (April 7, 2016)

- Transparent type conversion for `std::wstring` and `wchar_t`
- Allow passing `nullptr`-valued strings
- Transparent passing of `void *` pointers using capsules
- Transparent support for returning values wrapped in `std::unique_ptr<>`
- Improved docstring generation for compatibility with Sphinx
- Nicer debug error message when default parameter construction fails
- Support for "opaque" types that bypass the transparent conversion layer for STL containers
- Redesigned type casting interface to avoid ambiguities that could occasionally cause compiler errors
- Redesigned property implementation; fixes crashes due to an unfortunate default return value policy
- Anaconda package generation support

1.57 1.3 (March 8, 2016)

- Added support for the Intel C++ compiler (v15+)
- Added support for the STL unordered set/map data structures
- Added support for the STL linked list data structure
- NumPy-style broadcasting support in `pybind11::vectorize`
- pybind11 now displays more verbose error messages when `arg::operator=()` fails
- pybind11 internal data structures now live in a version-dependent namespace to avoid ABI issues
- Many, many bugfixes involving corner cases and advanced usage

1.58 1.2 (February 7, 2016)

- Optional: efficient generation of function signatures at compile time using C++14
- Switched to a simpler and more general way of dealing with function default arguments. Unused keyword arguments in function calls are now detected and cause errors as expected
- New `keep_alive` call policy analogous to Boost.Python's `with_custodian_and_ward`
- New `pybind11::base<>` attribute to indicate a subclass relationship
- Improved interface for RAII type wrappers in `pytypes.h`
- Use RAII type wrappers consistently within pybind11 itself. This fixes various potential refcount leaks when exceptions occur
- Added new `bytes` RAII type wrapper (maps to `string` in Python 2.7)
- Made handle and related RAII classes const correct, using them more consistently everywhere now
- Got rid of the ugly `__pybind11__` attributes on the Python side—they are now stored in a C++ hash table that is not visible in Python
- Fixed refcount leaks involving NumPy arrays and bound functions

- Vastly improved handling of shared/smart pointers
- Removed an unnecessary copy operation in `pybind11::vectorize`
- Fixed naming clashes when both pybind11 and NumPy headers are included
- Added conversions for additional exception types
- Documentation improvements (using multiple extension modules, smart pointers, other minor clarifications)
- unified infrastructure for parsing variadic arguments in `class_` and cpp_function
- Fixed license text (was: ZLIB, should have been: 3-clause BSD)
- Python 3.2 compatibility
- Fixed remaining issues when accessing types in another plugin module
- Added enum comparison and casting methods
- Improved SFINAE-based detection of whether types are copy-constructible
- Eliminated many warnings about unused variables and the use of `offsetof()`
- Support for `std::array`<> conversions

1.59 1.1 (December 7, 2015)

- Documentation improvements (GIL, wrapping functions, casting, fixed many typos)
- Generalized conversion of integer types
- Improved support for casting function objects
- Improved support for `std::shared_ptr`<> conversions
- Initial support for `std::set`<> conversions
- Fixed type resolution issue for types defined in a separate plugin module
- CMake build system improvements
- Factored out generic functionality to non-templated code (smaller code size)
- Added a code size / compile time benchmark vs Boost.Python
- Added an appveyor CI script

1.60 1.0 (October 15, 2015)

- Initial release

CHAPTER

TWO

UPGRADE GUIDE

This is a companion guide to the *Changelog*. While the changelog briefly lists all of the new features, improvements and bug fixes, this upgrade guide focuses only the subset which directly impacts your experience when upgrading to a new version. But it goes into more detail. This includes things like deprecated APIs and their replacements, build system changes, general code modernization and other useful information.

2.1 v3.0

pybind11 v3.0 introduces major new features, but the vast majority of existing extensions are expected to build and run without modification. Minor adjustments may be needed in rare cases, and any such changes can be easily wrapped in preprocessor conditionals to maintain compatibility with the 2.x series.

However, due to new features and modernizations, extensions built with pybind11 v3.0 are not ABI-compatible with those built using v2.13. To ensure cross-extension-module compatibility, it is recommended to rebuild all pybind11-based extensions with v3.0.

CMake support now defaults to the modern FindPython module. If you haven't updated yet, we provide some backward compatibility for `PYTHON_*` variables, but you should switch to using `Python_*` variables instead. Note that setting `PYTHON_*` variables no longer affects the build.

A major new feature in this release is the integration of `py::smart_holder`, which improves support for `std::unique_ptr` and `std::shared_ptr`, resolving several long-standing issues. See *py::smart_holder* for details. Closely related is the addition of `py::trampoline_self_life_support`, documented under *Overriding virtual functions in Python*.

This release includes a major modernization of cross-extension-module ABI compatibility handling. The new implementation reflects actual ABI compatibility much more accurately than in previous versions. The details are subtle and complex; see #4953 and #5439.

Also new in v3.0 is `py::native_enum`, a modern API for exposing C++ enumerations as native Python types — typically standard-library `enum.Enum` or related subclasses. This provides improved integration with Python's enum system, compared to the older (now deprecated) `py::enum_`. See #5555 for details. Note that `#include <pybind11/native_enum.h>` is not included automatically and must be added explicitly.

Functions exposed with pybind11 are now pickleable. This removes a long-standing obstacle when using pybind11-bound functions with Python features that rely on pickling, such as multiprocessing and caching tools. See #5580 for details.

Anything producing a deprecation warning in the 2.x series may be removed in a future minor release of 3.x. Most of these are still present in 3.0 in order to ease transition. The new *Deprecated* page details deprecations.

2.1.1 Migration Recommendations

We recommend migrating to pybind11 v3.0 promptly, while keeping initial changes to a minimum. Most projects can upgrade simply by updating the pybind11 version, without altering existing binding code.

After a short stabilization period — enough to surface any subtle issues — you may incrementally adopt new features where appropriate:

- Use `py::smart_holder` and `py::trampoline_self_life_support` as needed, or to improve code health. Note that `py::classh` is available as a shortcut — for example, `py::classh<Pet>` is shorthand for `py::class_<Pet, py::smart_holder>`. This is designed to enable easy experimentation with `py::smart_holder` without introducing distracting whitespace changes. In many cases, a global replacement of `py::class_` with `py::classh` can be an effective first step. Build failures will quickly identify places where `std::shared_ptr<...>` holders need to be removed. Runtime failures (assuming good unit test coverage) will highlight base-and-derived class situations that require coordinated changes.

  Note that `py::bind_vector` and `py::bind_map` (in pybind11/stl_bind.h) have a `holder_type` template parameter that defaults to `std::unique_ptr`. If `py::smart_holder` functionality is desired or required, use e.g. `py::bind_vector<VecType, py::smart_holder>`.

- Gradually migrate from `py::enum_` to `py::native_enum` to improve integration with Python's standard enum types.

There is no urgency to refactor existing, working bindings — adopt new features as the need arises or as part of ongoing maintenance efforts.

If you are using CMake, update to FindPython variables (mostly changing variables from `PYTHON_*` -> `Python_*`). You should see if you can use `set(PYBIND11_FINDPYTHON ON)`, which has been supported for years and will avoid setting the compatibility mode variables (and will avoid a warning).

2.1.2 Potential stumbling blocks when migrating to v3.0

The following issues are very unlikely to arise, and easy to work around:

- In rare cases, a C++ enum may be bound to Python via a *custom type caster*. In such cases, a template specialization like this may be required:

  ```
  #if defined(PYBIND11_HAS_NATIVE_ENUM)
  namespace pybind11::detail {
  template <typename FancyEnum>
  struct type_caster_enum_type_enabled<
      FancyEnum,
      enable_if_t<is_fancy_enum<FancyEnum>::value>> : std::false_type {};
  }
  #endif
  ```

  This specialization is needed only if the custom type caster is templated.

  The `PYBIND11_HAS_NATIVE_ENUM` guard is needed only if backward compatibility with pybind11v2 is required.

- Similarly, template specializations like the following may be required if there are custom

  - `pybind11::detail::copyable_holder_caster` or
  - `pybind11::detail::move_only_holder_caster`

  implementations that are used for `std::shared_ptr` or `std::unique_ptr` conversions:

```
#if defined(PYBIND11_HAS_INTERNALS_WITH_SMART_HOLDER_SUPPORT)
namespace pybind11::detail {
template <typename ExampleType>
struct copyable_holder_caster_shared_ptr_with_smart_holder_support_enabled<
    ExampleType,
    enable_if_t<is_example_type<ExampleType>::value>> : std::false_type {};
}
#endif
```

```
#if defined(PYBIND11_HAS_INTERNALS_WITH_SMART_HOLDER_SUPPORT)
namespace pybind11::detail {
template <typename ExampleType>
struct move_only_holder_caster_unique_ptr_with_smart_holder_support_enabled<
    ExampleType,
    enable_if_t<is_example_type<ExampleType>::value>> : std::false_type {};
}
#endif
```

The `PYBIND11_HAS_INTERNALS_WITH_SMART_HOLDER_SUPPORT` guard is needed only if backward compatibility with pybind11v2 is required.

(Note that `copyable_holder_caster` and `move_only_holder_caster` are not documented, although they existed since 2017.)

2.2 v2.12

NumPy support has been upgraded to support the 2.x series too. The two relevant changes are that:

- `dtype.flags()` is now a `uint64` and `dtype.alignment()` an `ssize_t` (and NumPy may return an larger than integer value for `itemsize()` in NumPy 2.x).
- The long deprecated NumPy function `PyArray_GetArrayParamsFromObject` function is not available anymore.

Due to NumPy changes, you may experience difficulties updating to NumPy 2. Please see the NumPy 2 migration guide for details. For example, a more direct change could be that the default integer `"int_"` (and `"uint"`) is now `ssize_t` and not `long` (affects 64bit windows).

If you want to only support NumPy 1.x for now and are having problems due to the two internal changes listed above, you can define `PYBIND11_NUMPY_1_ONLY` to disable the new support for now. Make sure you define this on all pybind11 compile units, since it could be a source of ODR violations if used inconsistently. This option will be removed in the future, so adapting your code is highly recommended.

2.3 v2.11

- The minimum version of CMake is now 3.5. A future version will likely move to requiring something like CMake 3.15. Note that CMake 3.27 is removing the long-deprecated support for `FindPythonInterp` if you set 3.27 as the minimum or maximum supported version. To prepare for that future, CMake 3.15+ using `FindPython` or setting `PYBIND11_FINDPYTHON` is highly recommended, otherwise pybind11 will automatically switch to using `FindPython` if `FindPythonInterp` is not available.

2.4 v2.9

- Any usage of the recently added `py::make_simple_namespace` should be converted to using `py::module_::import("types").attr("SimpleNamespace")` instead.
- The use of `_` in custom type casters can now be replaced with the more readable `const_name` instead. The old `_` shortcut has been retained unless it is being used as a macro (like for gettext).

2.5 v2.7

Before v2.7, `py::str` can hold `PyUnicodeObject` or `PyBytesObject`, and `py::isinstance<str>()` is `true` for both `py::str` and `py::bytes`. Starting with v2.7, `py::str` exclusively holds `PyUnicodeObject` (#2409), and `py::isinstance<str>()` is `true` only for `py::str`. To help in the transition of user code, the `PYBIND11_STR_LEGACY_PERMISSIVE` macro is provided as an escape hatch to go back to the legacy behavior. This macro will be removed in future releases. Two types of required fixes are expected to be common:

- Accidental use of `py::str` instead of `py::bytes`, masked by the legacy behavior. These are probably very easy to fix, by changing from `py::str` to `py::bytes`.
- Reliance on py::isinstance<str>(obj) being `true` for `py::bytes`. This is likely to be easy to fix in most cases by adding `|| py::isinstance<bytes>(obj)`, but a fix may be more involved, e.g. if `py::isinstance<T>` appears in a template. Such situations will require careful review and custom fixes.

2.6 v2.6

Usage of the `PYBIND11_OVERLOAD*` macros and `get_overload` function should be replaced by `PYBIND11_OVERRIDE*` and `get_override`. In the future, the old macros may be deprecated and removed.

`py::module` has been renamed `py::module_`, but a backward compatible typedef has been included. This change was to avoid a language change in C++20 that requires unqualified `module` not be placed at the start of a logical line. Qualified usage is unaffected and the typedef will remain unless the C++ language rules change again.

The public constructors of `py::module_` have been deprecated. Use `PYBIND11_MODULE` or `module_::create_extension_module` instead.

An error is now thrown when `__init__` is forgotten on subclasses. This was incorrect before, but was not checked. Add a call to `__init__` if it is missing.

A `py::type_error` is now thrown when casting to a subclass (like `py::bytes` from `py::object`) if the conversion is not valid. Make a valid conversion instead.

The undocumented `h.get_type()` method has been deprecated and replaced by `py::type::of(h)`.

Enums now have a `__str__` method pre-defined; if you want to override it, the simplest fix is to add the new `py::prepend()` tag when defining `"__str__"`.

If `__eq__` defined but not `__hash__`, `__hash__` is now set to `None`, as in normal CPython. You should add `__hash__` if you intended the class to be hashable, possibly using the new `py::hash` shortcut.

The constructors for `py::array` now always take signed integers for size, for consistency. This may lead to compiler warnings on some systems. Cast to `py::ssize_t` instead of `std::size_t`.

The `tools/clang` submodule and `tools/mkdoc.py` have been moved to a standalone package, pybind11-mkdoc. If you were using those tools, please use them via a pip install from the new location.

The pybind11 package on PyPI no longer fills the wheel "headers" slot - if you were using the headers from this slot, they are available by requesting the global extra, that is, pip install "pybind11[global]". (Most users will be unaffected, as the pybind11/include location is reported by python -m pybind11 --includes and pybind11.get_include() is still correct and has not changed since 2.5).

2.6.1 CMake support:

The minimum required version of CMake is now 3.4. Several details of the CMake support have been deprecated; warnings will be shown if you need to change something. The changes are:

- PYBIND11_CPP_STANDARD=<platform-flag> is deprecated, please use CMAKE_CXX_STANDARD=<number> instead, or any other valid CMake CXX or CUDA standard selection method, like target_compile_features.
- If you do not request a standard, pybind11 targets will compile with the compiler default, but not less than C++11, instead of forcing C++14 always. If you depend on the old behavior, please use set(CMAKE_CXX_STANDARD 14 CACHE STRING "") instead.
- Direct pybind11::module usage should always be accompanied by at least set(CMAKE_CXX_VISIBILITY_PRESET hidden) or similar - it used to try to manually force this compiler flag (but not correctly on all compilers or with CUDA).
- pybind11_add_module's SYSTEM argument is deprecated and does nothing; linking now behaves like other imported libraries consistently in both config and submodule mode, and behaves like a SYSTEM library by default.
- If PYTHON_EXECUTABLE is not set, virtual environments (venv, virtualenv, and conda) are prioritized over the standard search (similar to the new FindPython mode).

In addition, the following changes may be of interest:

- CMAKE_INTERPROCEDURAL_OPTIMIZATION will be respected by pybind11_add_module if set instead of linking to pybind11::lto or pybind11::thin_lto.
- Using find_package(Python COMPONENTS Interpreter Development) before pybind11 will cause pybind11 to use the new Python mechanisms instead of its own custom search, based on a patched version of classic FindPythonInterp / FindPythonLibs. In the future, this may become the default. A recent (3.15+ or 3.18.2+) version of CMake is recommended.

2.7 v2.5

The Python package now includes the headers as data in the package itself, as well as in the "headers" wheel slot. pybind11 --includes and pybind11.get_include() report the new location, which is always correct regardless of how pybind11 was installed, making the old user= argument meaningless. If you are not using the function to get the location already, you are encouraged to switch to the package location.

2.8 v2.2

2.8.1 Deprecation of the PYBIND11_PLUGIN macro

PYBIND11_MODULE is now the preferred way to create module entry points. The old macro emits a compile-time deprecation warning.

```
// old
PYBIND11_PLUGIN(example) {
    py::module m("example", "documentation string");

    m.def("add", [](int a, int b) { return a + b; });

    return m.ptr();
}

// new
PYBIND11_MODULE(example, m) {
    m.doc() = "documentation string"; // optional

    m.def("add", [](int a, int b) { return a + b; });
}
```

2.8.2 New API for defining custom constructors and pickling functions

The old placement-new custom constructors have been deprecated. The new approach uses `py::init()` and factory functions to greatly improve type safety.

Placement-new can be called accidentally with an incompatible type (without any compiler errors or warnings), or it can initialize the same object multiple times if not careful with the Python-side `__init__` calls. The new-style custom constructors prevent such mistakes. See *Custom constructors* for details.

```
// old -- deprecated (runtime warning shown only in debug mode)
py::class<Foo>(m, "Foo")
    .def("__init__", [](Foo &self, ...) {
        new (&self) Foo(...); // uses placement-new
    });

// new
py::class<Foo>(m, "Foo")
    .def(py::init([](...) { // Note: no `self` argument
        return new Foo(...); // return by raw pointer
        // or: return std::make_unique<Foo>(...); // return by holder
        // or: return Foo(...); // return by value (move constructor)
    }));
```

Warning: Deprecated placement-new `__init__` and `__setstate__` callbacks receive access to raw storage before the C++ object's lifetime begins. For compatibility, pybind11 retains this behavior for the complete constructor overload chain whenever the chain contains such a callback. This compatibility feature has important caveats: other loads triggered during argument conversion or callback execution may receive a C++ pointer to the storage even though no C++ object has been constructed there yet. Accessing the storage through such a pointer as though it contained a live C++ object results in undefined behavior.

Consequently, while such a constructor overload chain is active, the binding must not otherwise load or inspect an unconstructed value slot as a C++ object. Unsafe access can occur through reentrant argument conversion or callback code, nested initialization, another C++ base in a Python multiple-inheritance instance, or concurrent access. Mixing old- and new-style constructor overloads does not narrow the window. Such access may treat unconstructed storage as a live object and result in undefined behavior. To avoid these hazards, use `py::init()`

factories and `py::pickle()` for new bindings, and migrate existing placement-new callbacks wherever practical.

Mirroring the custom constructor changes, `py::pickle()` is now the preferred way to get and set object state. See *Pickling support* for details.

```
// old -- deprecated (runtime warning shown only in debug mode)
py::class<Foo>(m, "Foo")
    ...
    .def("__getstate__", [](const Foo &self) {
        return py::make_tuple(self.value1(), self.value2(), ...);
    })
    .def("__setstate__", [](Foo &self, py::tuple t) {
        new (&self) Foo(t[0].cast<std::string>(), ...);
    });

// new
py::class<Foo>(m, "Foo")
    ...
    .def(py::pickle(
        [](const Foo &self) { // __getstate__
            return py::make_tuple(self.value1(), self.value2(), ...); // unchanged
        },
        [](py::tuple t) { // __setstate__, note: no `self` argument
            return new Foo(t[0].cast<std::string>(), ...);
            // or: return std::make_unique<Foo>(...); // return by holder
            // or: return Foo(...); // return by value (move constructor)
        }
    ));
```

For both the constructors and pickling, warnings are shown at module initialization time (on import, not when the functions are called). They're only visible when compiled in debug mode. Sample warning:

```
pybind11-bound class 'mymodule.Foo' is using an old-style placement-new '__init__'
which has been deprecated. See the upgrade guide in pybind11's docs.
```

2.8.3 Stricter enforcement of hidden symbol visibility for pybind11 modules

pybind11 now tries to actively enforce hidden symbol visibility for modules. If you're using either one of pybind11's *CMake or Python build systems* (the two example repositories) and you haven't been exporting any symbols, there's nothing to be concerned about. All the changes have been done transparently in the background. If you were building manually or relied on specific default visibility, read on.

Setting default symbol visibility to *hidden* has always been recommended for pybind11 (see *How can I create smaller binaries?*). On Linux and macOS, hidden symbol visibility (in conjunction with the `strip` utility) yields much smaller module binaries. CPython's extension docs also recommend hiding symbols by default, with the goal of avoiding symbol name clashes between modules. Starting with v2.2, pybind11 enforces this more strictly: (1) by declaring all symbols inside the `pybind11` namespace as hidden and (2) by including the `-fvisibility=hidden` flag on Linux and macOS (only for extension modules, not for embedding the interpreter).

The namespace-scope hidden visibility is done automatically in pybind11's headers and it's generally transparent to users. It ensures that:

- Modules compiled with different pybind11 versions don't clash with each other.

- Some new features, like `py::module_local` bindings, can work as intended.

The `-fvisibility=hidden` flag applies the same visibility to user bindings outside of the `pybind11` namespace. It's now set automatic by pybind11's CMake and Python build systems, but this needs to be done manually by users of other build systems. Adding this flag:

- Minimizes the chances of symbol conflicts between modules. E.g. if two unrelated modules were statically linked to different (ABI-incompatible) versions of the same third-party library, a symbol clash would be likely (and would end with unpredictable results).
- Produces smaller binaries on Linux and macOS, as pointed out previously.

Within pybind11's CMake build system, `pybind11_add_module` has always been setting the `-fvisibility=hidden` flag in release mode. From now on, it's being applied unconditionally, even in debug mode and it can no longer be opted out of with the `NO_EXTRAS` option. The `pybind11::module` target now also adds this flag to its interface. The `pybind11::embed` target is unchanged.

The most significant change here is for the `pybind11::module` target. If you were previously relying on default visibility, i.e. if your Python module was doubling as a shared library with dependents, you'll need to either export symbols manually (recommended for cross-platform libraries) or factor out the shared library (and have the Python module link to it like the other dependents). As a temporary workaround, you can also restore default visibility using the CMake code below, but this is not recommended in the long run:

```
target_link_libraries(mymodule PRIVATE pybind11::module)

add_library(restore_default_visibility INTERFACE)
target_compile_options(restore_default_visibility INTERFACE -fvisibility=default)
target_link_libraries(mymodule PRIVATE restore_default_visibility)
```

2.8.4 Local STL container bindings

Previous pybind11 versions could only bind types globally – all pybind11 modules, even unrelated ones, would have access to the same exported types. However, this would also result in a conflict if two modules exported the same C++ type, which is especially problematic for very common types, e.g. `std::vector<int>`. *Module-local class bindings* were added to resolve this (see that section for a complete usage guide).

`py::class_` still defaults to global bindings (because these types are usually unique across modules), however in order to avoid clashes of opaque types, `py::bind_vector` and `py::bind_map` will now bind STL containers as `py::module_local` if their elements are: builtins (`int`, `float`, etc.), not bound using `py::class_`, or bound as `py::module_local`. For example, this change allows multiple modules to bind `std::vector<int>` without causing conflicts. See *Binding STL containers* for more details.

When upgrading to this version, if you have multiple modules which depend on a single global binding of an STL container, note that all modules can still accept foreign `py::module_local` types in the direction of Python-to-C++. The locality only affects the C++-to-Python direction. If this is needed in multiple modules, you'll need to either:

- Add a copy of the same STL binding to all of the modules which need it.
- Restore the global status of that single binding by marking it `py::module_local(false)`.

The latter is an easy workaround, but in the long run it would be best to localize all common type bindings in order to avoid conflicts with third-party modules.

2.8.5 Negative strides for Python buffer objects and numpy arrays

Support for negative strides required changing the integer type from unsigned to signed in the interfaces of `py::buffer_info` and `py::array`. If you have compiler warnings enabled, you may notice some new conversion warnings after upgrading. These can be resolved using `static_cast`.

2.8.6 Deprecation of some `py::object` APIs

To compare `py::object` instances by pointer, you should now use `obj1.is(obj2)` which is equivalent to `obj1 is obj2` in Python. Previously, pybind11 used `operator==` for this (`obj1 == obj2`), but that could be confusing and is now deprecated (so that it can eventually be replaced with proper rich object comparison in a future release).

For classes which inherit from `py::object`, `borrowed` and `stolen` were previously available as protected constructor tags. Now the types should be used directly instead: `borrowed_t{}` and `stolen_t{}` (#771).

2.8.7 Stricter compile-time error checking

Some error checks have been moved from run time to compile time. Notably, automatic conversion of `std::shared_ptr<T>` is not possible when `T` is not directly registered with `py::class_<T>` (e.g. `std::shared_ptr<int>` or `std::shared_ptr<std::vector<T>>` are not automatically convertible). Attempting to bind a function with such arguments now results in a compile-time error instead of waiting to fail at run time.

`py::init<...>()` constructor definitions are also stricter and now prevent bindings which could cause unexpected behavior:

```
struct Example {
    Example(int &);
};

py::class_<Example>(m, "Example")
    .def(py::init<int &>()); // OK, exact match
    // .def(py::init<int>()); // compile-time error, mismatch
```

A non-`const` lvalue reference is not allowed to bind to an rvalue. However, note that a constructor taking `const T &` can still be registered using `py::init<T>()` because a `const` lvalue reference can bind to an rvalue.

2.9 v2.1

2.9.1 Minimum compiler versions are enforced at compile time

The minimums also apply to v2.0 but the check is now explicit and a compile-time error is raised if the compiler does not meet the requirements:

- GCC >= 4.8
- clang >= 3.3 (appleclang >= 5.0)
- MSVC >= 2015u3
- Intel C++ >= 15.0

2.9.2 The `py::metaclass` attribute is not required for static properties

Binding classes with static properties is now possible by default. The zero-parameter version of `py::metaclass()` is deprecated. However, a new one-parameter `py::metaclass(python_type)` version was added for rare cases when a custom metaclass is needed to override pybind11's default.

```
// old -- emits a deprecation warning
py::class_<Foo>(m, "Foo", py::metaclass())
    .def_property_readonly_static("foo", ...);

// new -- static properties work without the attribute
py::class_<Foo>(m, "Foo")
    .def_property_readonly_static("foo", ...);

// new -- advanced feature, override pybind11's default metaclass
py::class_<Bar>(m, "Bar", py::metaclass(custom_python_type))
    ...
```

2.10 v2.0

2.10.1 Breaking changes in `py::class_`

These changes were necessary to make type definitions in pybind11 future-proof, to support PyPy via its `cpyext` mechanism (#527), and to improve efficiency (rev. 86d825).

1. Declarations of types that provide access via the buffer protocol must now include the `py::buffer_protocol()` annotation as an argument to the `py::class_` constructor.

   ```
   py::class_<Matrix>("Matrix", py::buffer_protocol())
       .def(py::init<...>())
       .def_buffer(...);
   ```

2. Classes which include static properties (e.g. `def_readwrite_static()`) must now include the `py::metaclass()` attribute. Note: this requirement has since been removed in v2.1. If you're upgrading from 1.x, it's recommended to skip directly to v2.1 or newer.

3. This version of pybind11 uses a redesigned mechanism for instantiating trampoline classes that are used to override virtual methods from within Python. This led to the following user-visible syntax change:

   ```
   // old v1.x syntax
   py::class_<TrampolineClass>("MyClass")
       .alias<MyClass>()
       ...

   // new v2.x syntax
   py::class_<MyClass, TrampolineClass>("MyClass")
       ...
   ```

   Importantly, both the original and the trampoline class are now specified as arguments to the `py::class_` template, and the `alias<..>()` call is gone. The new scheme has zero overhead in cases when Python doesn't override any functions of the underlying C++ class. rev. 86d825.

   The class type must be the first template argument given to `py::class_` while the trampoline can be mixed in arbitrary order with other arguments (see the following section).

2.10.2 Deprecation of the py::base<T>() attribute

`py::base<T>()` was deprecated in favor of specifying `T` as a template argument to `py::class_`. This new syntax also supports multiple inheritance. Note that, while the type being exported must be the first argument in the `py::class_<Class, ...>` template, the order of the following types (bases, holder and/or trampoline) is not important.

```
// old v1.x
py::class_<Derived>("Derived", py::base<Base>());

// new v2.x
py::class_<Derived, Base>("Derived");

// new -- multiple inheritance
py::class_<Derived, Base1, Base2>("Derived");

// new -- apart from `Derived` the argument order can be arbitrary
py::class_<Derived, Base1, Holder, Base2, Trampoline>("Derived");
```

2.10.3 Out-of-the-box support for std::shared_ptr

The relevant type caster is now built in, so it's no longer necessary to include a declaration of the form:

```
PYBIND11_DECLARE_HOLDER_TYPE(T, std::shared_ptr<T>)
```

Continuing to do so won't cause an error or even a deprecation warning, but it's completely redundant.

2.10.4 Deprecation of a few py::object APIs

All of the old-style calls emit deprecation warnings.

Old syntax	New syntax
`obj.call(args...)`	`obj(args...)`
`obj.str()`	`py::str(obj)`
`auto l = py::list(obj); l.check()`	`py::isinstance<py::list>(obj)`
`py::object(ptr, true)`	`py::reinterpret_borrow<py::object>(ptr)`
`py::object(ptr, false)`	`py::reinterpret_steal<py::object>(ptr)`
`if (obj.attr("foo"))`	`if (py::hasattr(obj, "foo"))`
`if (obj["bar"])`	`if (obj.contains("bar"))`

CHAPTER

THREE

INSTALLING THE LIBRARY

There are several ways to get the pybind11 source, which lives at pybind/pybind11 on GitHub. The pybind11 developers recommend one of the first three ways listed here, submodule, PyPI, or conda-forge, for obtaining pybind11.

3.1 Include as a submodule

When you are working on a project in Git, you can use the pybind11 repository as a submodule. From your git repository, use:

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

This assumes you are placing your dependencies in `extern/`, and that you are using GitHub; if you are not using GitHub, use the full https or ssh URL instead of the relative URL `../../pybind/pybind11` above. Some other servers also require the `.git` extension (GitHub does not).

From here, you can now include `extern/pybind11/include`, or you can use the various integration tools (see *Build systems*) pybind11 provides directly from the local folder.

3.2 Include with PyPI

You can download the sources and CMake files as a Python package from PyPI using Pip. Just use:

```
pip install pybind11
```

This will provide pybind11 in a standard Python package format. If you want pybind11 available directly in your environment root, you can use:

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

This is not recommended if you are installing with your system Python, as it will add files to `/usr/local/include/pybind11` and `/usr/local/share/cmake/pybind11`, so unless that is what you want, it is recommended only for use in virtual environments or your `pyproject.toml` file (see *Build systems*).

3.3 Include with conda-forge

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

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

3.4 Include with vcpkg

You can download and install pybind11 using the Microsoft vcpkg dependency manager:

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

The pybind11 port in vcpkg is kept up to date by Microsoft team members and community contributors. If the version is out of date, please create an issue or pull request on the vcpkg repository.

3.5 Global install with brew

The brew package manager (Homebrew on macOS, or Linuxbrew on Linux) has a pybind11 package. To install:

```
brew install pybind11
```

3.6 Other options

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

CHAPTER

FOUR

FIRST STEPS

This sections demonstrates the basic features of pybind11. Before getting started, make sure that development environment is set up to compile the included set of test cases.

4.1 Compiling the test cases

4.1.1 Linux/macOS

On Linux you'll need to install the **python-dev** or **python3-dev** packages as well as **cmake**. On macOS, the included python version works out of the box, but **cmake** must still be installed.

After installing the prerequisites, run

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

The last line will both compile and run the tests.

4.1.2 Windows

On Windows, only **Visual Studio 2019** and newer are supported.

Note: The `/permissive-` flag to enforce standard conformance is not necessary, but it is still advised.

To compile and run the tests:

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

This will create a Visual Studio project, compile and run the target, all from the command line.

Note: If all tests fail, make sure that the Python binary and the testcases are compiled for the same processor type and bitness (i.e. either **i386** or **x86_64**). You can specify **x86_64** as the target architecture for the generated Visual Studio project using `cmake -A x64 ...`.

See also:

Advanced users who are already familiar with Boost.Python may want to skip the tutorial and look at the test cases in the `tests` directory, which exercise all features of pybind11.

4.2 Header and namespace conventions

For brevity, all code examples assume that the following two lines are present:

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

namespace py = pybind11;
```

Note: `pybind11/pybind11.h` includes `Python.h`, as such it must be the first file included in any source file or header for the same reasons as Python.h.

Some features may require additional headers, but those will be specified as needed.

4.3 Creating bindings for a simple function

Let's start by creating Python bindings for an extremely simple function, which adds two numbers and returns their result:

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

For simplicity[1], we'll put both this function and the binding code into a file named `example.cpp` with the following contents:

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

namespace py = pybind11;

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

PYBIND11_MODULE(example, m, py::mod_gil_not_used()) {
    m.doc() = "pybind11 example plugin"; // optional module docstring

    m.def("add", &add, "A function that adds two numbers");
}
```

The `PYBIND11_MODULE()` macro creates a function that will be called when an `import` statement is issued from within Python. The module name (`example`) is given as the first macro argument (it should not be in quotes). The second argument (`m`) defines a variable of type `py::module_` which is the main interface for creating bindings. The method *module_::def()* generates binding code that exposes the `add()` function to Python.

[1] In practice, implementation and binding code will generally be located in separate files.

Note: Notice how little code was needed to expose our function to Python: all details regarding the function's parameters and return value were automatically inferred using template metaprogramming. This overall approach and the used syntax are borrowed from Boost.Python, though the underlying implementation is very different.

pybind11 is a header-only library, hence it is not necessary to link against any special libraries and there are no intermediate (magic) translation steps. On Linux, the above example can be compiled using the following command:

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

Note: If you used *Include as a submodule* to get the pybind11 source, then use `$(python3-config --includes) -Iextern/pybind11/include` instead of `$(python3 -m pybind11 --includes)` in the above compilation, as explained in *Building manually*.

For more details on the required compiler flags on Linux and macOS, see *Building manually*. For complete cross-platform compilation instructions, refer to the *Build systems* page.

The python_example and cmake_example repositories are also a good place to start. They are both complete project examples with cross-platform build systems. The only difference between the two is that python_example uses Python's `setuptools` to build the module, while cmake_example uses CMake (which may be preferable for existing C++ projects).

Building the above C++ code will produce a binary module file that can be imported to Python. Assuming that the compiled module is located in the current directory, the following interactive Python session shows how to load and execute the example:

```
$ python
Python 3.9.10 (main, Jan 15 2022, 11:48:04)
[Clang 13.0.0 (clang-1300.0.29.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import example
>>> example.add(1, 2)
3
>>>
```

4.4 Keyword arguments

With a simple code modification, it is possible to inform Python about the names of the arguments ("i" and "j" in this case).

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

arg is one of several special tag classes which can be used to pass metadata into *module_::def()*. With this modified binding code, we can now call the function using keyword arguments, which is a more readable alternative particularly for functions taking many parameters:

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

The keyword names also appear in the function signatures within the documentation.

```
>>> help(example)

....

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

        A function which adds two numbers
```

A shorter notation for named arguments is also available:

```
// 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);
```

The `_a` suffix forms a C++11 literal which is equivalent to *arg*. Note that the literal operator must first be made visible with the directive `using namespace pybind11::literals`. This does not bring in anything else from the `pybind11` namespace except for literals.

4.5 Default arguments

Suppose now that the function to be bound has default arguments, e.g.:

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

Unfortunately, pybind11 cannot automatically extract these parameters, since they are not part of the function's type information. However, they are simple to specify using an extension of *arg*:

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

The default values also appear within the documentation.

```
>>> help(example)

....

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

        A function which adds two numbers
```

The shorthand notation is also available for default arguments:

```
// 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 Exporting variables

To expose a value from C++, use the `attr` function to register it in a module as shown below. Built-in types and general objects (more on that later) are automatically converted when assigned as attributes, and can be explicitly converted using the function `py::cast`.

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

These are then accessible from Python:

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

4.7 Supported data types

A large number of data types are supported out of the box and can be used seamlessly as functions arguments, return values or with `py::cast` in general. For a full overview, see the *Type conversions* section.

CHAPTER

FIVE

OBJECT-ORIENTED CODE

5.1 Creating bindings for a custom type

Let's now look at a more complex example where we'll create bindings for a custom C++ data structure named `Pet`. Its definition is given below:

```
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;
};
```

The binding code for `Pet` looks as follows:

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

namespace py = pybind11;

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

`py::class_` creates bindings for a C++ *class* or *struct*-style data structure. `init()` is a convenience function that takes the types of a constructor's parameters as template arguments and wraps the corresponding constructor (see the *Custom constructors* section for details).

Note: Starting with pybind11v3, it is recommended to include `py::smart_holder` in most situations for safety, especially if you plan to support conversions to C++ smart pointers. See *py::smart_holder* for more information.

An interactive Python session demonstrating this example is shown below:

```
% python
>>> import example
>>> p = example.Pet("Molly")
>>> print(p)
```

(continues on next page)

(continued from previous page)

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

See also:

Static member functions can be bound in the same way using `class_::def_static()`.

Note: Binding C++ types in unnamed namespaces (also known as anonymous namespaces) works reliably on many platforms, but not all. The `XFAIL_CONDITION` in tests/test_unnamed_namespace_a.py encodes the currently known conditions. For background see #4319. If portability is a concern, it is therefore not recommended to bind C++ types in unnamed namespaces. It will be safest to manually pick unique namespace names.

5.2 Keyword and default arguments

It is possible to specify keyword and default arguments using the syntax discussed in the previous chapter. Refer to the sections *Keyword arguments* and *Default arguments* for details.

5.3 Binding lambda functions

Note how `print(p)` produced a rather useless summary of our data structure in the example above:

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

To address this, we could bind a utility function that returns a human-readable summary to the special method slot named `__repr__`. Unfortunately, there is no suitable functionality in the `Pet` data structure, and it would be nice if we did not have to change it. This can easily be accomplished by binding a Lambda function instead:

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

Both stateless[1] and stateful lambda closures are supported by pybind11. With the above change, the same Python code now produces the following output:

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

[1] Stateless closures are those with an empty pair of brackets `[]` as the capture object.

5.4 Instance and static fields

We can also directly expose the `name` field using the `class_::def_readwrite()` method. A similar `class_::def_readonly()` method also exists for `const` fields.

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

This makes it possible to write

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

Now suppose that `Pet::name` was a private internal variable that can only be accessed via setters and getters.

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

In this case, the method `class_::def_property()` (`class_::def_property_readonly()` for read-only data) can be used to provide a field-like interface within Python that will transparently call the setter and getter functions:

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

Write only properties can be defined by passing `nullptr` as the input for the read function.

See also:

Similar functions `class_::def_readwrite_static()`, `class_::def_readonly_static()` `class_::def_property_static()`, and `class_::def_property_readonly_static()` are provided for binding static variables and properties. Please also see the section on *Static properties* in the advanced part of the documentation.

5.5 Dynamic attributes

Native Python classes can pick up new attributes dynamically:

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

By default, classes exported from C++ do not support this and the only writable attributes are the ones explicitly defined using `class_::def_readwrite()` or `class_::def_property()`.

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

Trying to set any other attribute results in an error:

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

To enable dynamic attributes for C++ classes, the `py::dynamic_attr` tag must be added to the `py::class_` constructor:

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

Now everything works as expected:

```
>>> 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}
```

Note that there is a small runtime cost for a class with dynamic attributes. Not only because of the addition of a `__dict__`, but also because of more expensive garbage collection tracking which must be activated to resolve possible circular references. Native Python classes incur this same cost by default, so this is not anything to worry about. By default, pybind11 classes are more efficient than native Python classes. Enabling dynamic attributes just brings them on par.

5.6 Inheritance and automatic downcasting

Suppose now that the example consists of two data structures with an inheritance relationship:

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

There are two different ways of indicating a hierarchical relationship to pybind11: the first specifies the C++ base class as an extra template parameter of the `py::class_`:

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

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

Alternatively, we can also assign a name to the previously bound `Pet` `py::class_` object and reference it when binding the `Dog` class:

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

// 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);
```

Functionality-wise, both approaches are equivalent. Afterwards, instances will expose fields and methods of both types:

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

The C++ classes defined above are regular non-polymorphic types with an inheritance relationship. This is reflected in Python:

```
// Return a base pointer to a derived instance
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'
```

The function returned a `Dog` instance, but because it's a non-polymorphic type behind a base pointer, Python only sees a `Pet`. In C++, a type is only considered polymorphic if it has at least one virtual function and pybind11 will automatically recognize this:

```
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()
'woof!'
```

Given a pointer to a polymorphic base, pybind11 performs automatic downcasting to the actual derived type. Note that this goes beyond the usual situation in C++: we don't just get access to the virtual functions of the base, we get the concrete derived type including functions and attributes that the base type may not even be aware of.

See also:

For more information about polymorphic behavior see *Overriding virtual functions in Python*.

5.7 Overloaded methods

Sometimes there are several overloaded C++ methods with the same name taking different kinds of input arguments:

```
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_; }
```

(continues on next page)

(continued from previous page)

```
    std::string name;
    int age;
};
```

Attempting to bind `Pet::set` will cause an error since the compiler does not know which method the user intended to select. We can disambiguate by casting them to function pointers. Binding multiple functions to the same Python name automatically creates a chain of function overloads that will be tried in sequence.

```
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");
```

The overload signatures are also visible in the method's docstring:

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

class Pet(__builtin__.object)
 |  Methods defined here:
 |
 |  __init__(...)
 |      Signature : (Pet, str, int) -> NoneType
 |
 |  set(...)
 |      1. Signature : (Pet, int) -> NoneType
 |
 |      Set the pet's age
 |
 |      2. Signature : (Pet, str) -> NoneType
 |
 |      Set the pet's name
```

If you have a C++14 compatible compiler[2], you can use an alternative syntax to cast the overloaded function:

```
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");
```

Here, `py::overload_cast` only requires the parameter types to be specified. The return type and class are deduced. This avoids the additional noise of `void (Pet::*)()` as seen in the raw cast. If a function is overloaded based on constness, the `py::const_` tag should be used:

```
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_));
```

[2] A compiler which supports the `-std=c++14` flag.

If you prefer the `py::overload_cast` syntax but have a C++11 compatible compiler only, you can use `py::detail::overload_cast_impl` with an additional set of parentheses:

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

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

Note: To define multiple overloaded constructors, simply declare one after the other using the `.def(py::init<...>())` syntax. The existing machinery for specifying keyword and default arguments also works.

5.8 Pitfalls with raw pointers and shared ownership

`py::class_`-wrapped objects automatically manage the lifetime of the wrapped C++ object, in collaboration with the chosen holder type (see *Smart pointers & py::class_*). When wrapping C++ functions involving raw pointers, care needs to be taken to not accidentally undermine automatic lifetime management. For example, ownership is inadvertently transferred here:

```
class Child { };

class Parent {
public:
   Parent() : child(std::make_shared<Child>()) { }
   Child *get_child() { return child.get(); }  /* DANGER */
private:
    std::shared_ptr<Child> child;
};

PYBIND11_MODULE(example, m, py::mod_gil_not_used()) {
    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);  /* PROBLEM */
}
```

The following Python code leads to undefined behavior, likely resulting in a segmentation fault.

```
from example import Parent

print(Parent().get_child())
```

Part of the `/* PROBLEM */` here is that pybind11 falls back to using `return_value_policy::take_ownership` as the default (see *Return value policies*). The fact that the `Child` instance is already managed by `std::shared_ptr<Child>` is lost. Therefore pybind11 will create a second independent `std::shared_ptr<Child>` that also claims ownership of the pointer, eventually leading to heap-use-after-free or double-free errors.

There are various ways to resolve this issue, either by changing the `Child` or `Parent` C++ implementations (e.g. using `std::enable_shared_from_this<Child>` as a base class for `Child`, or adding a

member function to `Parent` that returns `std::shared_ptr<Child>`), or if that is not feasible, by using `return_value_policy::reference_internal`. What is the best approach depends on the exact situation.

A highly effective way to stay in the clear — even in pure C++, but especially when binding C++ code to Python — is to consistently prefer `std::shared_ptr` or `std::unique_ptr` over passing raw pointers.

5.9 Enumerations and internal types

Let's now suppose that the example class contains internal types like enumerations, e.g.:

```
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;
};
```

The binding code for this example looks as follows:

```
#include <pybind11/native_enum.h> // Not already included with pybind11.h

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::native_enum<Pet::Kind>(pet, "Kind", "enum.Enum")
    .value("Dog", Pet::Kind::Dog)
    .value("Cat", Pet::Kind::Cat)
    .export_values()
    .finalize();

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

To ensure that the nested types `Kind` and `Attributes` are created within the scope of `Pet`, the `pet` `py::class_` instance must be supplied to the `py::native_enum` and `py::class_` constructors. The `.export_values()` function is available for exporting the enum entries into the parent scope, if desired.

`py::native_enum` was introduced with pybind11v3. It binds C++ enum types to native Python enum types, typically types in Python's stdlib enum module, which are PEP 435 compatible. This is the recommended way to bind C++

enums. The older `py::enum_` is not PEP 435 compatible (see issue #2332) but remains supported indefinitely for backward compatibility. New bindings should prefer `py::native_enum`.

Note: The deprecated `py::enum_` is *documented here*.

The `.finalize()` call above is needed because Python's native enums cannot be built incrementally — all name/value pairs need to be passed at once. To achieve this, `py::native_enum` acts as a buffer to collect the name/value pairs. The `.finalize()` call uses the accumulated name/value pairs to build the arguments for constructing a native Python enum type.

The `py::native_enum` constructor takes a third argument, `native_type_name`, which specifies the fully qualified name of the Python base class to use — e.g., `"enum.Enum"` or `"enum.IntEnum"`. A fourth optional argument, `class_doc`, provides the docstring for the generated class.

For example:

```
py::native_enum<Pet::Kind>(pet, "Kind", "enum.IntEnum", "Constant specifying the kind of␣
↪pet")
```

You may use any fully qualified Python name for `native_type_name`. The only requirement is that the named type is similar to enum.Enum in these ways:

- Has a constructor similar to that of enum.Enum:

  ```
  Colors = enum.Enum("Colors", (("Red", 0), ("Green", 1)))
  ```

- A C++ underlying enum value can be passed to the constructor for the Python enum value:

  ```
  red = Colors(0)
  ```

- The enum values have a `.value` property yielding a value that can be cast to the C++ underlying type:

  ```
  underlying = red.value
  ```

As of Python 3.13, the compatible types in the stdlib enum module are: `Enum`, `IntEnum`, `Flag`, `IntFlag`.

Note: In rare cases, a C++ enum may be bound to Python via a *custom type caster*. In such cases, a template specialization like this may be required:

```
#if defined(PYBIND11_HAS_NATIVE_ENUM)
namespace pybind11::detail {
template <typename FancyEnum>
struct type_caster_enum_type_enabled<
    FancyEnum,
    enable_if_t<is_fancy_enum<FancyEnum>::value>> : std::false_type {};
}
#endif
```

This specialization is needed only if the custom type caster is templated.

The `PYBIND11_HAS_NATIVE_ENUM` guard is needed only if backward compatibility with pybind11v2 is required.

CHAPTER

SIX

BUILD SYSTEMS

For an overview of Python packaging including compiled packaging with a pybind11 example, along with a cookiecutter that includes several pybind11 options, see the Scientific Python Development Guide.

6.1 Modules with CMake

A Python extension module can be created with just a few lines of code:

```
cmake_minimum_required(VERSION 3.15...4.2)
project(example LANGUAGES CXX)

set(PYBIND11_FINDPYTHON ON)
find_package(pybind11 CONFIG REQUIRED)

pybind11_add_module(example example.cpp)
install(TARGETS example DESTINATION .)
```

(You use the `add_subdirectory` instead, see the example in *Building with CMake*.) In this example, the code is located in a file named `example.cpp`. Either method will import the pybind11 project which provides the `pybind11_add_module` function. It will take care of all the details needed to build a Python extension module on any platform.

To build with pip, build, cibuildwheel, uv, or other Python tools, you can add a `pyproject.toml` file like this:

```
[build-system]
requires = ["scikit-build-core", "pybind11"]
build-backend = "scikit_build_core.build"

[project]
name = "example"
version = "0.1.0"
```

You don't need setuptools files like `MANIFEST.in`, `setup.py`, or `setup.cfg`, as this is not setuptools. See scikit-build-core for details. For projects you plan to upload to PyPI, be sure to fill out the `[project]` table with other important metadata as well (see Writing pyproject.toml).

A working sample project can be found in the [scikit_build_example] repository. An older and harder-to-maintain method is in [cmake_example]. More details about our cmake support can be found below in *Building with CMake*.

6.2 Modules with meson-python

You can also build a package with Meson using meson-python, if you prefer that. Your `meson.build` file would look something like this:

```
project(
    'example',
    'cpp',
    version: '0.1.0',
    default_options: [
        'cpp_std=c++11',
    ],
)

py = import('python').find_installation(pure: false)
pybind11_dep = dependency('pybind11')

py.extension_module('example',
    'example.cpp',
    install: true,
    dependencies : [pybind11_dep],
)
```

And you would need a `pyproject.toml` file like this:

```
[build-system]
requires = ["meson-python", "pybind11"]
build-backend = "mesonpy"
```

Meson-python *requires* your project to be in git (or mercurial) as it uses it for the SDist creation. For projects you plan to upload to PyPI, be sure to fill out the `[project]` table as well (see Writing pyproject.toml).

6.3 Modules with setuptools

For projects on PyPI, a historically popular option is setuptools. Sylvain Corlay has kindly provided an example project which shows how to set up everything, including automatic generation of documentation using Sphinx. Please refer to the [python_example] repository.

A helper file is provided with pybind11 that can simplify usage with setuptools.

To use pybind11 inside your `setup.py`, you have to have some system to ensure that `pybind11` is installed when you build your package. There are four possible ways to do this, and pybind11 supports all four: You can ask all users to install pybind11 beforehand (bad), you can use *Build requirements* (good), `setup_requires=` (discouraged), or you can *Copy manually* (works but you have to manually sync your copy to get updates). Third party packagers like conda-forge generally strongly prefer the `pyproject.toml` method, as it gives them control over the `pybind11` version, and they may apply patches, etc.

An example of a `setup.py` using pybind11's helpers:

```
from glob import glob
from setuptools import setup
from pybind11.setup_helpers import Pybind11Extension
```

(continues on next page)

(continued from previous page)

```
ext_modules = [
    Pybind11Extension(
        "python_example",
        sorted(glob("src/*.cpp")),  # Sort source files for reproducibility
    ),
]

setup(..., ext_modules=ext_modules)
```

If you want to do an automatic search for the highest supported C++ standard, that is supported via a `build_ext` command override; it will only affect `Pybind11Extensions`:

```
from glob import glob
from setuptools import setup
from pybind11.setup_helpers import Pybind11Extension, build_ext

ext_modules = [
    Pybind11Extension(
        "python_example",
        sorted(glob("src/*.cpp")),
    ),
]

setup(..., cmdclass={"build_ext": build_ext}, ext_modules=ext_modules)
```

If you have single-file extension modules that are directly stored in the Python source tree (`foo.cpp` in the same directory as where a `foo.py` would be located), you can also generate `Pybind11Extensions` using `setup_helpers.intree_extensions`: `intree_extensions(["path/to/foo.cpp", ...])` returns a list of `Pybind11Extensions` which can be passed to `ext_modules`, possibly after further customizing their attributes (`libraries`, `include_dirs`, etc.). By doing so, a `foo.*.so` extension module will be generated and made available upon installation.

`intree_extension` will automatically detect if you are using a `src`-style layout (as long as no namespace packages are involved), but you can also explicitly pass `package_dir` to it (as in `setuptools.setup`).

Since pybind11 does not require NumPy when building, a light-weight replacement for NumPy's parallel compilation distutils tool is included. Use it like this:

```
from pybind11.setup_helpers import ParallelCompile

# Optional multithreaded build
ParallelCompile("NPY_NUM_BUILD_JOBS").install()

setup(...)
```

The argument is the name of an environment variable to control the number of threads, such as `NPY_NUM_BUILD_JOBS` (as used by NumPy), though you can set something different if you want; `CMAKE_BUILD_PARALLEL_LEVEL` is another choice a user might expect. You can also pass `default=N` to set the default number of threads (0 will take the number of threads available) and `max=N`, the maximum number of threads; if you have a large extension you may want set this to a memory dependent number.

If you are developing rapidly and have a lot of C++ files, you may want to avoid rebuilding files that have not changed. For simple cases were you are using `pip install -e .` and do not have local headers, you can skip the rebuild if an object file is newer than its source (headers are not checked!) with the following:

```
from pybind11.setup_helpers import ParallelCompile, naive_recompile

ParallelCompile("NPY_NUM_BUILD_JOBS", needs_recompile=naive_recompile).install()
```

If you have a more complex build, you can implement a smarter function and pass it to `needs_recompile`, or you can use [Ccache] instead. `CXX="cache g++" pip install -e .` would be the way to use it with GCC, for example. Unlike the simple solution, this even works even when not compiling in editable mode, but it does require Ccache to be installed.

Keep in mind that Pip will not even attempt to rebuild if it thinks it has already built a copy of your code, which it deduces from the version number. One way to avoid this is to use [setuptools_scm], which will generate a version number that includes the number of commits since your last tag and a hash for a dirty directory. Another way to force a rebuild is purge your cache or use Pip's `--no-cache-dir` option.

You also need a `MANIFEST.in` file to include all relevant files so that you can make an SDist. If you use pypa-build, that will build an SDist then a wheel from that SDist by default, so you can look inside those files (wheels are just zip files with a `.whl` extension) to make sure you aren't missing files. check-manifest (setuptools specific) or check-sdist (general) are CLI tools that can compare the SDist contents with your source control.

6.3.1 Build requirements

With a `pyproject.toml` file, you can ensure that `pybind11` is available during the compilation of your project. When this file exists, Pip will make a new virtual environment, download just the packages listed here in `requires=`, and build a wheel (binary Python package). It will then throw away the environment, and install your wheel.

Your `pyproject.toml` file will likely look something like this:

```
[build-system]
requires = ["setuptools", "pybind11"]
build-backend = "setuptools.build_meta"
```

6.3.2 Copy manually

You can also copy `setup_helpers.py` directly to your project; it was designed to be usable standalone, like the old example `setup.py`. You can set `include_pybind11=False` to skip including the pybind11 package headers, so you can use it with git submodules and a specific git version. If you use this, you will need to import from a local file in `setup.py` and ensure the helper file is part of your MANIFEST.

Closely related, if you include pybind11 as a subproject, you can run the `setup_helpers.py` inplace. If loaded correctly, this should even pick up the correct include for pybind11, though you can turn it off as shown above if you want to input it manually.

Suggested usage if you have pybind11 as a submodule in `extern/pybind11`:

```
DIR = os.path.abspath(os.path.dirname(__file__))

sys.path.append(os.path.join(DIR, "extern", "pybind11"))
from pybind11.setup_helpers import Pybind11Extension  # noqa: E402

del sys.path[-1]
```

Changed in version 2.6: Added `setup_helpers` file.

6.4 Building with cppimport

[cppimport] is a small Python import hook that determines whether there is a C++ source file whose name matches the requested module. If there is, the file is compiled as a Python extension using pybind11 and placed in the same folder as the C++ source file. Python is then able to find the module and load it.

6.5 Building with CMake

For C++ codebases that have an existing CMake-based build system, a Python extension module can be created with just a few lines of code, as seen above in the module section. Pybind11 currently defaults to the old mechanism, though be aware that CMake 3.27 removed the old mechanism, so pybind11 will automatically switch if the old mechanism is not available. Please opt into the new mechanism if at all possible. Our default may change in future versions. This is the minimum required:

Changed in version 2.6: CMake 3.4+ is required.

Changed in version 2.11: CMake 3.5+ is required.

Changed in version 2.14: CMake 3.15+ is required.

Further information can be found at *CMake helpers*.

6.5.1 pybind11_add_module

To ease the creation of Python extension modules, pybind11 provides a CMake function with the following signature:

```
pybind11_add_module(<name> [MODULE | SHARED] [EXCLUDE_FROM_ALL]
                    [NO_EXTRAS] [THIN_LTO] [OPT_SIZE] [PRECOMPILE | NO_PRECOMPILE]
                    source1 [source2 ...])
```

This function behaves very much like CMake's builtin `add_library` (in fact, it's a wrapper function around that command). It will add a library target called `<name>` to be built from the listed source files. In addition, it will take care of all the Python-specific compiler and linker flags as well as the OS- and Python-version-specific file extension. The produced target `<name>` can be further manipulated with regular CMake commands.

`MODULE` or `SHARED` may be given to specify the type of library. If no type is given, `MODULE` is used by default which ensures the creation of a Python-exclusive module. Specifying `SHARED` will create a more traditional dynamic library which can also be linked from elsewhere. `EXCLUDE_FROM_ALL` removes this target from the default build (see CMake docs for details).

Since pybind11 is a template library, `pybind11_add_module` adds compiler flags to ensure high quality code generation without bloat arising from long symbol names and duplication of code in different translation units. It sets default visibility to *hidden*, which is required for some pybind11 features and functionality when attempting to load multiple pybind11 modules compiled under different pybind11 versions. It also adds additional flags enabling LTO (Link Time Optimization) and strip unneeded symbols. See the *FAQ entry* for a more detailed explanation. These latter optimizations are never applied in `Debug` mode. If `NO_EXTRAS` is given, they will always be disabled, even in `Release` mode. However, this will result in code bloat and is generally not recommended.

As stated above, LTO is enabled by default. Some newer compilers also support different flavors of LTO such as ThinLTO. Setting `THIN_LTO` will cause the function to prefer this flavor if available. The function falls back to regular LTO if `-flto=thin` is not available. If `CMAKE_INTERPROCEDURAL_OPTIMIZATION` is set (either `ON` or `OFF`), then that will be respected instead of the built-in flag search.

Note: If you want to set the property form on targets or the `CMAKE_INTERPROCEDURAL_OPTIMIZATION_<CONFIG>` versions of this, you should still use `set(CMAKE_INTERPROCEDURAL_OPTIMIZATION OFF)` (otherwise a no-op) to disable pybind11's ipo flags.

The `OPT_SIZE` flag enables size-based optimization equivalent to the standard `/Os` or `-Os` compiler flags and the `MinSizeRel` build type, which avoid optimizations that can substantially increase the size of the resulting binary. This flag is particularly useful in projects that are split into performance-critical parts and associated bindings. In this case, we can compile the project in release mode (and hence, optimize performance globally), and specify `OPT_SIZE` for the binding target, where size might be the main concern as performance is often less critical here. A ~25% size reduction has been observed in practice. This flag only changes the optimization behavior at a per-target level and takes precedence over the global CMake build type (`Release`, `RelWithDebInfo`) except for `Debug` builds, where optimizations remain disabled.

6.5.2 Pre-compiling part of pybind11

pybind11 is header-only by default: every translation unit compiles its own copy of the non-template implementation. The opt-in *precompiled* mode compiles that implementation once, into a static library built inside your own project with your own flags. This reduces the build time, most of all for projects with many translation units or many modules in one build.

```
pybind11_add_module(example PRECOMPILE example.cpp)
```

The first `PRECOMPILE` target creates the library target `pybind11::precompiled`; further targets reuse it. Set the CMake variable `PYBIND11_PRECOMPILE` to make it the default for all `pybind11_add_module` calls; use `NO_PRECOMPILE` on a target to opt back out. For targets you create yourself, call the `pybind11_precompile()` function and link `pybind11::precompiled` PRIVATE; the target carries the required `PYBIND11_PRECOMPILED` compile definition PUBLIC, so your sources also get it.

Requirements and caveats:

- The library and every module linking it must agree on the configuration macros `PYBIND11_INTERNALS_VERSION`, `Py_GIL_DISABLED`, `PYBIND11_SIMPLE_GIL_MANAGEMENT`, `PYBIND11_DETAILED_ERROR_MESSAGES` (defaults on in debug builds), `PYBIND11_HAS_SUBINTERPRETER_SUPPORT`, and `PYBIND11_BACKWARD_COMPATIBILITY_TP_DICTOFFSET`. A mismatch produces one readable undefined symbol at link time referencing `pybind11_precompiled_config`.
- Configuration macros that only change code inside the library (for example `PYBIND11_DISABLE_NEW_STYLE_INIT_WARNING`) must be defined when the library is compiled; a definition only on your module has no effect. This also applies to a custom `PYBIND11_NAMESPACE` visibility attribute: set it at the directory level (`add_compile_definitions`) so that the library gets it too.
- The library picks up your directory-level flags and C++ standard when it is first created, so set those before the first `PRECOMPILE` target. A status message reports the directory that created the library.
- The library is not compiled with link-time optimization, and the per-target `THIN_LTO` and `OPT_SIZE` options of `pybind11_add_module` do not apply to it. In a Release build this adds a few nanoseconds to each call of a bound function (up to about 10% for a function that does nothing, depending on the compiler; a few percent with LTO on the library). To change this, call `pybind11_precompile()` yourself and set the properties on the created target, `pybind11_precompiled` (the real target behind the `pybind11::precompiled` alias; CMake does not let you set properties through an alias):

```
pybind11_precompile()
set_target_properties(pybind11_precompiled PROPERTIES
                      INTERPROCEDURAL_OPTIMIZATION ON)
```

- The library is static and per-build-tree; it is never installed or shared between projects. Each extension module links its own copy, which keeps pybind11's per-module state the same as in header-only mode.
- Not available with `PYBIND11_NOPYTHON` (the library needs Python headers).

For build systems other than CMake, the same sources ship with the pybind11 package: compile `pybind11_combined.cpp` from the directory reported by `python -m pybind11 --srcdir` (also available as `pybind11.get_source_dir()` and the `srcdir` pkg-config variable) into a static library or into your extension, and define `PYBIND11_PRECOMPILED` for every translation unit.

With Meson, build the library once per build tree and link it into each extension module, the same as the CMake path:

```
pybind11_dep = dependency('pybind11')
pybind11_src = run_command(py, ['-m', 'pybind11', '--srcdir'],
                           check : true).stdout().strip()

pybind11_precompiled = static_library('pybind11_precompiled',
    pybind11_src / 'pybind11_combined.cpp',
    cpp_args : ['-DPYBIND11_PRECOMPILED'],
    gnu_symbol_visibility : 'hidden',
    dependencies : [pybind11_dep, py.dependency()])

py.extension_module('example', 'example.cpp',
    cpp_args : ['-DPYBIND11_PRECOMPILED'],
    link_with : pybind11_precompiled,
    dependencies : [pybind11_dep])
```

The configuration-macro rules above apply here too: the static library and every module that links it must be compiled with the same configuration macros, and `-DPYBIND11_PRECOMPILED` must appear in both `cpp_args` lists. (`pybind11_dep.get_variable('srcdir')` also reports the source directory when Meson finds pybind11 through pkg-config.)

6.5.3 Configuration variables

By default, pybind11 will compile modules with the compiler default or the minimum standard required by pybind11, whichever is higher. You can set the standard explicitly with CMAKE_CXX_STANDARD:

```
set(CMAKE_CXX_STANDARD 14 CACHE STRING "C++ version selection")  # or 11, 14, 17, 20
set(CMAKE_CXX_STANDARD_REQUIRED ON)  # optional, ensure standard is supported
set(CMAKE_CXX_EXTENSIONS OFF)  # optional, keep compiler extensions off
```

The variables can also be set when calling CMake from the command line using the `-D<variable>=<value>` flag. You can also manually set `CXX_STANDARD` on a target or use `target_compile_features` on your targets - anything that CMake supports.

Classic Python support: The target Python version can be selected by setting `PYBIND11_PYTHON_VERSION` or an exact Python installation can be specified with `PYTHON_EXECUTABLE`. For example:

```
cmake -DPYBIND11_PYTHON_VERSION=3.9 ..
```

(continues on next page)

(continued from previous page)

```
# Another method:
cmake -DPYTHON_EXECUTABLE=/path/to/python ..

# This often is a good way to get the current Python, works in environments:
cmake -DPYTHON_EXECUTABLE=$(python3 -c "import sys; print(sys.executable)") ..
```

6.5.4 find_package vs. add_subdirectory

For CMake-based projects that don't include the pybind11 repository internally, an external installation can be detected through `find_package(pybind11)`. See the Config file docstring for details of relevant CMake variables.

```
cmake_minimum_required(VERSION 3.15...4.2)
project(example LANGUAGES CXX)

find_package(pybind11 REQUIRED)
pybind11_add_module(example example.cpp)
```

Note that `find_package(pybind11)` will only work correctly if pybind11 has been correctly installed on the system, e. g. after downloading or cloning the pybind11 repository :

```
# Classic CMake
cd pybind11
mkdir build
cd build
cmake ..
make install

# CMake 3.15+
cd pybind11
cmake -S . -B build
cmake --build build -j 2  # Build on 2 cores
cmake --install build
```

Once detected, the aforementioned `pybind11_add_module` can be employed as before. The function usage and configuration variables are identical no matter if pybind11 is added as a subdirectory or found as an installed package. You can refer to the same [cmake_example] repository for a full sample project – just swap out `add_subdirectory` for `find_package`.

6.5.5 FindPython mode

Modern CMake (3.18.2+ ideal) added a new module called FindPython that had a highly improved search algorithm and modern targets and tools. If you use FindPython, pybind11 will detect this and use the existing targets instead:

```
cmake_minimum_required(VERSION 3.15...4.2)
project(example LANGUAGES CXX)

find_package(Python 3.9 COMPONENTS Interpreter Development REQUIRED)
find_package(pybind11 CONFIG REQUIRED)
# or add_subdirectory(pybind11)

pybind11_add_module(example example.cpp)
```

You can also use the targets (as listed below) with FindPython. If you define `PYBIND11_FINDPYTHON`, pybind11 will perform the FindPython step for you (mostly useful when building pybind11's own tests, or as a way to change search algorithms from the CMake invocation, with `-DPYBIND11_FINDPYTHON=ON`.

Warning: If you use FindPython to multi-target Python versions, use the individual targets listed below, and avoid targets that directly include Python parts.

There are many ways to hint or force a discovery of a specific Python installation), setting `Python_ROOT_DIR` may be the most common one (though with virtualenv/venv support, and Conda support, this tends to find the correct Python version more often than the old system did).

Warning: When the Python libraries (i.e. `libpythonXX.a` and `libpythonXX.so` on Unix) are not available, as is the case on a manylinux image, the `Development` component will not be resolved by `FindPython`. When not using the embedding functionality, CMake 3.18+ allows you to specify `Development.Module` instead of `Development` to resolve this issue.

New in version 2.6.

6.5.6 Advanced: interface library targets

Pybind11 supports modern CMake usage patterns with a set of interface targets, available in all modes. The targets provided are:

pybind11::headers
Just the pybind11 headers and minimum compile requirements

pybind11::pybind11
Python headers + `pybind11::headers`

pybind11::python_link_helper
Just the "linking" part of pybind11:module

pybind11::module
Everything for extension modules - `pybind11::pybind11` + `Python::Module` (FindPython) or `pybind11::python_link_helper`

pybind11::embed
Everything for embedding the Python interpreter - `pybind11::pybind11` + `Python::Python` (FindPython) or Python libs

pybind11::lto / pybind11::thin_lto
An alternative to `INTERPROCEDURAL_OPTIMIZATION` for adding link-time optimization.

pybind11::windows_extras
`/bigobj` and `/mp` for MSVC.

pybind11::opt_size
`/Os` for MSVC, `-Os` for other compilers. Does nothing for debug builds.

Two helper functions are also provided:

pybind11_strip(target)
Strips a target (uses `CMAKE_STRIP` after the target is built)

pybind11_extension(target)
Sets the correct extension (with SOABI) for a target.

You can use these targets to build complex applications. For example, the `add_python_module` function is identical to:

```
cmake_minimum_required(VERSION 3.15...4.2)
project(example LANGUAGES CXX)

find_package(pybind11 REQUIRED)  # or add_subdirectory(pybind11)

add_library(example MODULE main.cpp)

target_link_libraries(example PRIVATE pybind11::module pybind11::lto pybind11::windows_
↪extras)

pybind11_extension(example)
if(NOT MSVC AND NOT ${CMAKE_BUILD_TYPE} MATCHES Debug|RelWithDebInfo)
    # Strip unnecessary sections of the binary on Linux/macOS
    pybind11_strip(example)
endif()

set_target_properties(example PROPERTIES CXX_VISIBILITY_PRESET "hidden"
                                         CUDA_VISIBILITY_PRESET "hidden")
```

Instead of setting properties, you can set `CMAKE_*` variables to initialize these correctly.

Warning: Since pybind11 is a metatemplate library, it is crucial that certain compiler flags are provided to ensure high quality code generation. In contrast to the `pybind11_add_module()` command, the CMake interface provides a *composable* set of targets to ensure that you retain flexibility. It can be especially important to provide or set these properties; the *FAQ* contains an explanation on why these are needed.

New in version 2.6.

6.5.7 Advanced: NOPYTHON mode

If you want complete control, you can set `PYBIND11_NOPYTHON` to completely disable Python integration (this also happens if you run `FindPython2` and `FindPython3` without running `FindPython`). This gives you complete freedom to integrate into an existing system (like Scikit-Build's `PythonExtensions`). `pybind11_add_module` and `pybind11_extension` will be unavailable, and the targets will be missing any Python specific behavior.

New in version 2.6.

6.5.8 Embedding the Python interpreter

In addition to extension modules, pybind11 also supports embedding Python into a C++ executable or library. In CMake, simply link with the `pybind11::embed` target. It provides everything needed to get the interpreter running. The Python headers and libraries are attached to the target. Unlike `pybind11::module`, there is no need to manually set any additional properties here. For more information about usage in C++, see *Embedding the interpreter*.

```
cmake_minimum_required(VERSION 3.15...4.2)
project(example LANGUAGES CXX)

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

(continues on next page)

(continued from previous page)

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

6.6 Building manually

pybind11 is a header-only library, hence it is not necessary to link against any special libraries and there are no intermediate (magic) translation steps.

On Linux, you can compile an example such as the one given in *Creating bindings for a simple function* using the following command:

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

The `python3 -m pybind11 --includes` command fetches the include paths for both pybind11 and Python headers. This assumes that pybind11 has been installed using `pip` or `conda`. If it hasn't, you can also manually specify `-I <path-to-pybind11>/include` together with the Python includes path `python3-config --includes`.

On macOS: the build command is almost the same but it also requires passing the `-undefined dynamic_lookup` flag so as to ignore missing symbols when building the module:

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

For quick tests, the command line tool can also produce the full set of flags for you, based on `python-config`:

```
$ c++ $(python3 -m pybind11 --cflags) example.cpp $(python3 -m pybind11 --ldflags) -o␣
↪example$(python3 -m pybind11 --extension-suffix)
```

Or, shorter still, `--file` prints everything after the compiler for a given source file, including the output name (add `--embed` for a program that embeds the interpreter instead of an extension):

```
$ c++ $(python3 -m pybind11 --file=example.cpp)
```

These helpers target Unix-style compilers (GCC/Clang) and are intended for quick tests, not production builds; `--file` places the output next to the source file.

In general, it is advisable to include several additional build parameters that can considerably reduce the size of the created binary. Refer to section *Building with CMake* for a detailed example of a suitable cross-platform CMake-based build system that works on all platforms including Windows.

Note: On Linux and macOS, it's better to (intentionally) not link against `libpython`. The symbols will be resolved when the extension library is loaded into a Python binary. This is preferable because you might have several different installations of a given Python version (e.g. the system-provided Python, and one that ships with a piece of commercial software). In this way, the plugin will work with both versions, instead of possibly importing a second Python library into a process that already contains one (which will lead to a segfault).

6.7 Building with Bazel

You can build with the Bazel build system using the pybind11_bazel repository.

6.8 Building with Meson

You can use Meson, which has support for `pybind11` as a dependency (internally relying on our `pkg-config` support). See the *module example above*.

6.9 Generating binding code automatically

The `Binder` project is a tool for automatic generation of pybind11 binding code by introspecting existing C++ codebases using LLVM/Clang. See the [binder] documentation for details.

[AutoWIG] is a Python library that wraps automatically compiled libraries into high-level languages. It parses C++ code using LLVM/Clang technologies and generates the wrappers using the Mako templating engine. The approach is automatic, extensible, and applies to very complex C++ libraries, composed of thousands of classes or incorporating modern meta-programming constructs.

[semiwrap] is a build tool that makes it simpler to wrap C/C++ libraries with pybind11 by automating large portions of the wrapping process and handling some of the more complex aspects of creating pybind11 based wrappers (especially with trampolines to allow inheriting from C++ classes from Python). It includes a hatchling plugin that autogenerates meson.build files that can be built using meson, and those build files parse your wrapped headers and generate/compile pybind11 based wrappers into python extension modules.

[litgen] is an automatic python bindings generator with a focus on generating documented and discoverable bindings: bindings will nicely reproduce the documentation found in headers. It is based on srcML (srcml.org), a highly scalable, multi-language parsing tool with a developer centric approach. The API that you want to expose to python must be C++14 compatible (but your implementation can use more modern constructs).

CHAPTER

SEVEN

FUNCTIONS

Before proceeding with this section, make sure that you are already familiar with the basics of binding functions and classes, as explained in *First steps* and *Object-oriented code*. The following guide is applicable to both free and member functions, i.e. *methods* in Python.

7.1 Return value policies

Python and C++ use fundamentally different ways of managing the memory and lifetime of objects managed by them. This can lead to issues when creating bindings for functions that return a non-trivial type. Just by looking at the type information, it is not clear whether Python should take charge of the returned value and eventually free its resources, or if this is handled on the C++ side. For this reason, pybind11 provides several *return value policy* annotations that can be passed to the *module_::def()* and `class_::def()` functions. The default policy is `return_value_policy::automatic`.

Return value policies are tricky, and it's very important to get them right. Just to illustrate what can go wrong, consider the following simple example:

```
/* 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
```

What's going on here? When `get_data()` is called from Python, the return value (a native C++ type) must be wrapped to turn it into a usable Python type. In this case, the default return value policy (`return_value_policy::automatic`) causes pybind11 to assume ownership of the static `_data` instance.

When Python's garbage collector eventually deletes the Python wrapper, pybind11 will also attempt to delete the C++ instance (via `operator delete()`) due to the implied ownership. At this point, the entire application will come crashing down, though errors could also be more subtle and involve silent data corruption.

In the above example, the policy `return_value_policy::reference` should have been specified so that the global data instance is only *referenced* without any implied transfer of ownership, i.e.:

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

On the other hand, this is not the right policy for many other situations, where ignoring ownership could lead to resource leaks. As a developer using pybind11, it's important to be familiar with the different return value policies, including which situation calls for which one of them. The following table provides an overview of available policies:

Return value policy	Description
`return_value_policy::take_ownership`	Reference an existing object (i.e. do not create a new copy) and take ownership. Python will call the destructor and delete operator when the object's reference count reaches zero. Undefined behavior ensues when the C++ side does the same, or when the data was not dynamically allocated.
`return_value_policy::copy`	Create a new copy of the returned object, which will be owned by Python. This policy is comparably safe because the lifetimes of the two instances are decoupled.
`return_value_policy::move`	Use `std::move` to move the return value contents into a new instance that will be owned by Python. This policy is comparably safe because the lifetimes of the two instances (move source and destination) are decoupled.
`return_value_policy::reference`	Reference an existing object, but do not take ownership. The C++ side is responsible for managing the object's lifetime and deallocating it when it is no longer used. Warning: undefined behavior will ensue when the C++ side deletes an object that is still referenced and used by Python.
`return_value_policy::reference_internal`	If the return value is an lvalue reference or a pointer, the parent object (the implicit `this`, or `self` argument of the called method or property) is kept alive for at least the lifespan of the return value. **Otherwise this policy falls back to** `return_value_policy::move` (see #5528). Internally, this policy works just like `return_value_policy::reference` but additionally applies a `keep_alive<0, 1>` *call policy* (described in the next section) that prevents the parent object from being garbage collected as long as the return value is referenced by Python. This is the default policy for property getters created via `def_property`, `def_readwrite`, etc.
`return_value_policy::automatic`	This policy falls back to the policy `return_value_policy::take_ownership` when the return value is a pointer. Otherwise, it uses `return_value_policy::move` or `return_value_policy::copy` for rvalue and lvalue references, respectively. See above for a description of what all of these different policies do. This is the default policy for `py::class_`-wrapped types.
`return_value_policy::automatic_reference`	As above, but use policy `return_value_policy::reference` when the return value is a pointer. This is the default conversion policy for function arguments when calling Python functions manually from C++ code (i.e. via `handle::operator()`) and the casters in `pybind11/stl.h`. You probably won't need to use this explicitly.

Return value policies can also be applied to properties:

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

Technically, the code above applies the policy to both the getter and the setter function, however, the setter doesn't really care about *return* value policies which makes this a convenient terse syntax. Alternatively, targeted arguments can be passed through the `cpp_function` constructor:

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

Warning: Code with invalid return value policies might access uninitialized memory or free data structures multiple times, which can lead to hard-to-debug non-determinism and segmentation faults, hence it is worth spending the time to understand all the different options in the table above.

Note: One important aspect of the above policies is that they only apply to instances which pybind11 has *not* seen before, in which case the policy clarifies essential questions about the return value's lifetime and ownership. When pybind11 knows the instance already (as identified by its type and address in memory), it will return the existing Python object wrapper rather than creating a new copy.

Note: The next section on *Additional call policies* discusses *call policies* that can be specified *in addition* to a return value policy from the list above. Call policies indicate reference relationships that can involve both return values and parameters of functions.

Note: As an alternative to elaborate call policies and lifetime management logic, consider using smart pointers (see the section on *Custom smart pointers* for details). Smart pointers can tell whether an object is still referenced from C++ or Python, which generally eliminates the kinds of inconsistencies that can lead to crashes or undefined behavior. For functions returning smart pointers, it is not necessary to specify a return value policy.

7.2 Additional call policies

In addition to the above return value policies, further *call policies* can be specified to indicate dependencies between parameters or ensure a certain state for the function call.

7.2.1 Keep alive

In general, this policy is required when the C++ object is any kind of container and another object is being added to the container. `keep_alive<Nurse, Patient>` indicates that the argument with index `Patient` should be kept alive at least until the argument with index `Nurse` is freed by the garbage collector. Argument indices start at one, while zero refers to the return value. For methods, index `1` refers to the implicit `this` pointer, while regular arguments begin at index `2`. Arbitrarily many call policies can be specified. When a `Nurse` with value `None` is detected at runtime, the call policy does nothing.

When the nurse is not a pybind11-registered type, the implementation internally relies on the ability to create a *weak reference* to the nurse object. When the nurse object is not a pybind11-registered type and does not support weak references, an exception will be thrown.

If you use an incorrect argument index, you will get a `RuntimeError` saying `Could not activate keep_alive!`. You should review the indices you're using.

Consider the following example: here, the binding code for a list append operation ties the lifetime of the newly added element to the underlying container:

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

For consistency, the argument indexing is identical for constructors. Index `1` still refers to the implicit `this` pointer, i.e. the object which is being constructed. Index `0` refers to the return type which is presumed to be `void` when a constructor is viewed like a function. The following example ties the lifetime of the constructor element to the constructed object:

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

Note: `keep_alive` is analogous to the `with_custodian_and_ward` (if Nurse, Patient != 0) and `with_custodian_and_ward_postcall` (if Nurse/Patient == 0) policies from Boost.Python.

7.2.2 Call guard

The `call_guard<T>` policy allows any scope guard type `T` to be placed around the function call. For example, this definition:

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

is equivalent to the following pseudocode:

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

Argument conversion, including extraction from type casters, finishes before the guards are constructed. A candidate rejected during argument conversion does not construct its guards. Call-policy `precall` hooks also run before guard construction, and the guards are destroyed before Python return-value conversion.

The only requirement is that `T` is default-constructible, but otherwise any scope guard will work. This is very useful in combination with `gil_scoped_release`. See *Global Interpreter Lock (GIL)*.

Multiple guards can also be specified as `py::call_guard<T1, T2, T3...>`. The constructor order is left to right and destruction happens in reverse.

See also:

The file `tests/test_call_policies.cpp` contains a complete example that demonstrates using *keep_alive* and *call_guard* in more detail.

7.3 Python objects as arguments

pybind11 exposes all major Python types using thin C++ wrapper classes. These wrapper classes can also be used as parameters of functions in bindings, which makes it possible to directly work with native Python types on the C++ side. For instance, the following statement iterates over a Python `dict`:

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

It can be exported:

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

And used in Python as usual:

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

For more information on using Python objects in C++, see *Python C++ interface*.

7.4 Accepting *args and **kwargs

Python provides a useful mechanism to define functions that accept arbitrary numbers of arguments and keyword arguments:

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

Such functions can also be created using pybind11:

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

(continues on next page)

(continued from previous page)

```
        /// .. do something with kwargs
}

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

The class `py::args` derives from `py::tuple` and `py::kwargs` derives from `py::dict`.

You may also use just one or the other, and may combine these with other arguments. Note, however, that `py::kwargs` must always be the last argument of the function, and `py::args` implies that any further arguments are keyword-only (see *Keyword-only arguments*).

Please refer to the other examples for details on how to iterate over these, and on how to cast their entries into C++ objects. A demonstration is also available in `tests/test_kwargs_and_defaults.cpp`.

Note: When combining *args or **kwargs with *Keyword arguments* you should *not* include `py::arg` tags for the `py::args` and `py::kwargs` arguments.

7.5 Default arguments revisited

The section on *Default arguments* previously discussed basic usage of default arguments using pybind11. One noteworthy aspect of their implementation is that default arguments are converted to Python objects right at declaration time. Consider the following example:

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

In this case, pybind11 must already be set up to deal with values of the type `SomeType` (via a prior instantiation of `py::class_<SomeType>`), or an exception will be thrown.

Another aspect worth highlighting is that the "preview" of the default argument in the function signature is generated using the object's `__repr__` method. If not available, the signature may not be very helpful, e.g.:

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

The first way of addressing this is by defining `SomeType.__repr__`. Alternatively, it is possible to specify the human-readable preview of the default argument manually using the `arg_v` notation:

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

Sometimes it may be necessary to pass a null pointer value as a default argument. In this case, remember to cast it to the underlying type in question, like so:

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

7.6 Keyword-only arguments

Python implements keyword-only arguments by specifying an unnamed * argument in a function definition:

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


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

Pybind11 provides a `py::kw_only` object that allows you to implement the same behaviour by specifying the object between positional and keyword-only argument annotations when registering the function:

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

New in version 2.6.

A `py::args` argument implies that any following arguments are keyword-only, as if `py::kw_only()` had been specified in the same relative location of the argument list as the `py::args` argument. The `py::kw_only()` may be included to be explicit about this, but is not required.

Changed in version 2.9: This can now be combined with `py::args`. Before, `py::args` could only occur at the end of the argument list, or immediately before a `py::kwargs` argument at the end.

7.7 Positional-only arguments

Python 3.8 introduced a new positional-only argument syntax, using `/` in the function definition (note that this has been a convention for CPython positional arguments, such as in `pow()`, since Python 2). You can do the same thing in any version of Python using `py::pos_only()`:

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

You now cannot give argument `a` by keyword. This can be combined with keyword-only arguments, as well.

New in version 2.6.

7.8 Non-converting arguments

Certain argument types may support conversion from one type to another. Some examples of conversions are:

- *Implicit conversions* declared using `py::implicitly_convertible<A,B>()`
- Passing an argument that implements `__float__` or `__index__` to `float` or `double`.
- Passing an argument that implements `__int__` or `__index__` to `int`.
- Passing an argument that implements `__complex__`, `__float__`, or `__index__` to `std::complex<float>`. (Requires the optional `pybind11/complex.h` header).

- Calling a function taking an Eigen matrix reference with a numpy array of the wrong type or of an incompatible data layout. (Requires the optional `pybind11/eigen.h` header).

This behaviour is sometimes undesirable: the binding code may prefer to raise an error rather than convert the argument. This behaviour can be obtained through `py::arg` by calling the `.noconvert()` method of the `py::arg` object, such as:

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

`supports_float` will accept any argument that implements `__float__` or `__index__`. `only_float` will only accept a float or int argument. Anything else will fail with a `TypeError`:

Note: The noconvert behaviour of float, double and complex has changed to match PEP 484. A float/double argument marked noconvert will accept float or int. A std::complex<float> argument will accept complex, float or int.

```
class MyFloat:
    def __init__(self, value: float) -> None:
        self._value = float(value)
    def __repr__(self) -> str:
        return f"MyFloat({self._value})"
    def __float__(self) -> float:
        return self._value

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

Invoked with: MyFloat(4)
```

You may, of course, combine this with the `_a` shorthand notation (see *Keyword arguments*) and/or *Default arguments*. It is also permitted to omit the argument name by using the `py::arg()` constructor without an argument name, i.e. by specifying `py::arg().noconvert()`.

Note: When specifying `py::arg` options it is necessary to provide the same number of options as the bound function has arguments. Thus if you want to enable no-convert behaviour for just one of several arguments, you will need to specify a `py::arg()` annotation for each argument with the no-convert argument modified to `py::arg().noconvert()`.

7.9 Allow/Prohibiting None arguments

When a C++ type registered with `py::class_` is passed as an argument to a function taking the instance as pointer or shared holder (e.g. `shared_ptr` or a custom, copyable holder as described in *Custom smart pointers*), pybind allows `None` to be passed from Python which results in calling the C++ function with `nullptr` (or an empty holder) for the argument.

To explicitly enable or disable this behaviour, using the `.none` method of the `py::arg` object:

```
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));
```

With the above, the Python call `bark(None)` will return the string `"(no dog)"`, while attempting to call `meow(None)` will raise a `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
```

The default behaviour when the tag is unspecified is to allow `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 Overload resolution order

When a function or method with multiple overloads is called from Python, pybind11 determines which overload to call in two passes. The first pass attempts to call each overload without allowing argument conversion (as if every argument had been specified as `py::arg().noconvert()` as described above).

If no overload succeeds in the no-conversion first pass, a second pass is attempted in which argument conversion is allowed (except where prohibited via an explicit `py::arg().noconvert()` attribute in the function definition).

If the second pass also fails a `TypeError` is raised.

Within each pass, overloads are tried in the order they were registered with pybind11. If the `py::prepend()` tag is added to the definition, a function can be placed at the beginning of the overload sequence instead, allowing user overloads to proceed built in functions.

What this means in practice is that pybind11 will prefer any overload that does not require conversion of arguments to an overload that does, but otherwise prefers earlier-defined overloads to later-defined ones.

Note: pybind11 does *not* further prioritize based on the number/pattern of overloaded arguments. That is, pybind11 does not prioritize a function requiring one conversion over one requiring three, but only prioritizes overloads requiring no conversion at all to overloads that require conversion of at least one argument.

New in version 2.6: The `py::prepend()` tag.

7.11 Binding functions with template parameters

You can bind functions that have template parameters. Here's a function:

```
template <typename T>
void set(T t);
```

C++ templates cannot be instantiated at runtime, so you cannot bind the non-instantiated function:

```
// BROKEN (this will not compile)
m.def("set", &set);
```

You must bind each instantiated function template separately. You may bind each instantiation with the same name, which will be treated the same as an overloaded function:

```
m.def("set", &set<int>);
m.def("set", &set<std::string>);
```

Sometimes it's more clear to bind them with separate names, which is also an option:

```
m.def("setInt", &set<int>);
m.def("setString", &set<std::string>);
```

CHAPTER

EIGHT

CLASSES

This section presents advanced binding code for classes and it is assumed that you are already familiar with the basics from *Object-oriented code*.

8.1 Overriding virtual functions in Python

Suppose that a C++ class or interface has a virtual function that we'd like to override from within Python (we'll focus on the class `Animal`; `Dog` is given as a specific example of how one would do this with traditional C++ code).

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

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

Let's also suppose that we are given a plain function which calls the function `go()` on an arbitrary `Animal` instance.

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

Normally, the binding code for these classes would look as follows:

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

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

(continues on next page)

(continued from previous page)

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

However, these bindings are impossible to extend: `Animal` is not constructible, and we clearly require some kind of "trampoline" that redirects virtual calls back to Python.

Defining a new type of `Animal` from within Python is possible but requires a helper class that is defined as follows:

```
class PyAnimal : public Animal, public py::trampoline_self_life_support {
public:
    /* Inherit the constructors */
    using Animal::Animal;

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

The `py::trampoline_self_life_support` base class is needed to ensure that a `std::unique_ptr` can safely be passed between Python and C++. To help you steer clear of notorious pitfalls (e.g. inheritance slicing), pybind11 enforces that trampoline classes inherit from `py::trampoline_self_life_support` if used in in combination with `py::smart_holder`.

Note: For completeness, the base class has no effect if a holder other than `py::smart_holder` used, including the default `std::unique_ptr<T>`. To avoid confusion, pybind11 will fail to compile bindings that combine `py::trampoline_self_life_support` with a holder other than `py::smart_holder`.

Please think twice, though, before deciding to not use the safer `py::smart_holder`. The pitfalls associated with avoiding it are very real, and the overhead for using it is very likely in the noise.

The macro *PYBIND11_OVERRIDE_PURE* should be used for pure virtual functions, and *PYBIND11_OVERRIDE* should be used for functions which have a default implementation. There are also two alternate macros *PYBIND11_OVERRIDE_PURE_NAME* and *PYBIND11_OVERRIDE_NAME* which take a string-valued name argument between the *Parent class* and *Name of the function* slots, which defines the name of function in Python. This is required when the C++ and Python versions of the function have different names, e.g. `operator()` vs `__call__`.

The binding code also needs a few minor adaptations (highlighted):

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

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

(continues on next page)

(continued from previous page)

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

Importantly, pybind11 is made aware of the trampoline helper class by specifying it as an extra template argument to `py::class_`. (This can also be combined with other template arguments such as a custom holder type; the order of template types does not matter). Following this, we are able to define a constructor as usual.

Bindings should be made against the actual class, not the trampoline helper class.

```
py::class_<Animal, PyAnimal /* <--- trampoline */, py::smart_holder>(m, "Animal");
    .def(py::init<>())
    .def("go", &Animal::go); /* <--- DO NOT USE &PyAnimal::go HERE */
```

Note, however, that the above is sufficient for allowing python classes to extend `Animal`, but not `Dog`: see *Combining virtual functions and inheritance* for the necessary steps required to providing proper overriding support for inherited classes.

The Python session below shows how to override `Animal::go` and invoke it via a virtual method call.

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

If you are defining a custom constructor in a derived Python class, you *must* ensure that you explicitly call the bound C++ constructor using `__init__`, *regardless* of whether it is a default constructor or not. Otherwise, the memory for the C++ portion of the instance will be left uninitialized, which will generally leave the C++ instance in an invalid state and cause undefined behavior if the C++ instance is subsequently used.

Changed in version 2.6: The default pybind11 metaclass will throw a `TypeError` when it detects that `__init__` was not called by a derived class.

Here is an example:

```
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 that a direct `__init__` constructor *should be called*, and `super()` should not be used. For simple cases of linear inheritance, `super()` may work, but once you begin mixing Python and C++ multiple inheritance, things will fall apart due to differences between Python's MRO and C++'s mechanisms.

Please take a look at the *General notes regarding convenience macros* before using this feature.

Note: When the overridden type returns a reference or pointer to a type that pybind11 converts from Python (for example, numeric values, std::string, and other built-in value-converting types), there are some limitations to be aware of:

- 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.

Warning: The *PYBIND11_OVERRIDE* and accompanying macros used to be called `PYBIND11_OVERLOAD` up until pybind11 v2.5.0, and *get_override()* used to be called `get_overload`. This naming was corrected and the older macro and function names may soon be deprecated, in order to reduce confusion with overloaded functions and methods and `py::overload_cast` (see *Object-oriented code*).

See also:

The file `tests/test_virtual_functions.cpp` contains a complete example that demonstrates how to override virtual functions using pybind11 in more detail.

8.2 Combining virtual functions and inheritance

When combining virtual methods with inheritance, you need to be sure to provide an override for each method for which you want to allow overrides from derived python classes. For example, suppose we extend the above `Animal`/`Dog` example as follows:

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

then the trampoline class for `Animal` must, as described in the previous section, override `go()` and `name()`, but in order to allow python code to inherit properly from `Dog`, we also need a trampoline class for `Dog` that overrides both the added `bark()` method *and* the `go()` and `name()` methods inherited from `Animal` (even though `Dog` doesn't directly override the `name()` method):

```
class PyAnimal : public Animal, public py::trampoline_self_life_support {
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 py::trampoline_self_life_support {
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, ); }
};
```

Note: Note the trailing commas in the `PYBIND11_OVERRIDE` calls to `name()` and `bark()`. These are needed to portably implement a trampoline for a function that does not take any arguments. For functions that take a nonzero number of arguments, the trailing comma must be omitted.

A registered class derived from a pybind11-registered class with virtual methods requires a similar trampoline class, *even if* it doesn't explicitly declare or override any virtual methods itself:

```
class Husky : public Dog {};
class PyHusky : public Husky, public py::trampoline_self_life_support {
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, ); }
};
```

There is, however, a technique that can be used to avoid this duplication (which can be especially helpful for a base class with several virtual methods). The technique involves using template trampoline classes, as follows:

```
template <class AnimalBase = Animal>
class PyAnimal : public AnimalBase, public py::trampoline_self_life_support {
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 py::trampoline_self_life_support {
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, ); }
```

(continues on next page)

(continued from previous page)

```
};
```

This technique has the advantage of requiring just one trampoline method to be declared per virtual method and pure virtual method override. It does, however, require the compiler to generate at least as many methods (and possibly more, if both pure virtual and overridden pure virtual methods are exposed, as above).

The classes are then registered with pybind11 using:

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

Note that `Husky` did not require a dedicated trampoline template class at all, since it neither declares any new virtual methods nor provides any pure virtual method implementations.

With either the repeated-virtuals or templated trampoline methods in place, you can now create a python class that inherits from `Dog`:

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

See also:

See the file `tests/test_virtual_functions.cpp` for complete examples using both the duplication and templated trampoline approaches.

8.3 Extended trampoline class functionality

8.3.1 Forced trampoline class initialisation

The trampoline classes described in the previous sections are, by default, only initialized when needed. More specifically, they are initialized when a python class actually inherits from a registered type (instead of merely creating an instance of the registered type), or when a registered constructor is only valid for the trampoline class but not the registered class. This is primarily for performance reasons: when the trampoline class is not needed for anything except virtual method dispatching, not initializing the trampoline class improves performance by avoiding needing to do a run-time check to see if the inheriting python instance has an overridden method.

Sometimes, however, it is useful to always initialize a trampoline class as an intermediate class that does more than just handle virtual method dispatching. For example, such a class might perform extra class initialization, extra destruction operations, and might define new members and methods to enable a more python-like interface to a class.

In order to tell pybind11 that it should *always* initialize the trampoline class when creating new instances of a type, the class constructors should be declared using `py::init_alias<Args, ...>()` instead of the usual `py::init<Args, ...>()`. This forces construction via the trampoline class, ensuring member initialization and (eventual) destruction.

See also:

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

8.3.2 Different method signatures

The macro's introduced in *Overriding virtual functions in Python* cover most of the standard use cases when exposing C++ classes to Python. Sometimes it is hard or unwieldy to create a direct one-on-one mapping between the arguments and method return type.

An example would be when the C++ signature contains output arguments using references (See also *Limitations involving reference arguments*). Another way of solving this is to use the method body of the trampoline class to do conversions to the input and return of the Python method.

The main building block to do so is the *get_override()*, this function allows retrieving a method implemented in Python from within the trampoline's methods. Consider for example a C++ method which has the signature `bool myMethod(int32_t& value)`, where the return indicates whether something should be done with the `value`. This can be made convenient on the Python side by allowing the Python function to return `None` or an `int`:

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

8.3.3 Avoiding Inheritance Slicing and `std::weak_ptr` surprises

When working with classes that use virtual functions and are subclassed in Python, special care must be taken when converting Python objects to `std::shared_ptr<T>`. Depending on whether the class uses a plain `std::shared_ptr` holder or `py::smart_holder`, the resulting `shared_ptr` may either allow inheritance slicing or lead to potentially surprising behavior when constructing `std::weak_ptr` instances.

This section explains how `std::shared_ptr` and `py::smart_holder` manage object lifetimes differently, how these differences affect trampoline-derived objects, and what options are available to achieve the situation-specific desired behavior.

When using `std::shared_ptr` as the holder type, converting a Python object to a `std::shared_ptr<T>` (e.g., `obj.cast<std::shared_ptr<T>>()`, or simply passing the Python object as an argument to a `.def()`-ed function) returns a `shared_ptr` that shares ownership with the original `class_` holder, usually preserving object lifetime. However, for Python classes that derive from a trampoline, if the Python object is destroyed, only the base C++ object may remain alive, leading to inheritance slicing (see #1333).

In contrast, with `py::smart_holder`, converting a Python object to a `std::shared_ptr<T>` returns a new `shared_ptr` with an independent control block that keeps the derived Python object alive. This avoids inheritance slicing but can lead to unintended behavior when creating `std::weak_ptr` instances (see #5623).

If it is necessary to obtain a `std::weak_ptr` that shares the control block with the `smart_holder`—at the cost of reintroducing potential inheritance slicing—you can use `py::potentially_slicing_weak_ptr<T>(obj)`.

When precise lifetime management of derived Python objects is important, using a Python-side `weakref` is the most reliable approach, as it avoids both inheritance slicing and unintended interactions with `std::weak_ptr` semantics in C++.

See also:

- `potentially_slicing_weak_ptr()` C++ documentation
- `tests/test_potentially_slicing_weak_ptr.cpp`

8.4 Custom constructors

The syntax for binding constructors was previously introduced, but it only works when a constructor of the appropriate arguments actually exists on the C++ side. To extend this to more general cases, pybind11 makes it possible to bind factory functions as constructors. For example, suppose you have a class like this:

```
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));
```

While it is possible to create a straightforward binding of the static `create` method, it may sometimes be preferable to expose it as a constructor on the Python side. This can be accomplished by calling `.def(py::init(...))` with the function reference returning the new instance passed as an argument. It is also possible to use this approach to bind a function returning a new instance by raw pointer or by the holder (e.g. `std::unique_ptr`).

The following example shows the different approaches:

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

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

py::class_<Example>(m, "Example")
    // Bind the factory function as a constructor:
    .def(py::init(&Example::create))
    // Bind a lambda function returning a pointer wrapped in a holder:
    .def(py::init([](std::string arg) {
        return std::unique_ptr<Example>(new Example(arg));
    }))
```

(continues on next page)

(continued from previous page)

```
    // 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>())
    ;
```

When the constructor is invoked from Python, pybind11 will call the factory function and store the resulting C++ instance in the Python instance.

When combining factory functions constructors with *virtual function trampolines* there are two approaches. The first is to add a constructor to the alias class that takes a base value by rvalue-reference. If such a constructor is available, it will be used to construct an alias instance from the value returned by the factory function. The second option is to provide two factory functions to `py::init()`: the first will be invoked when no alias class is required (i.e. when the class is being used but not inherited from in Python), and the second will be invoked when an alias is required.

You can also specify a single factory function that always returns an alias instance: this will result in behaviour similar to `py::init_alias<...>()`, as described in the *extended trampoline class documentation*.

The following example shows the different factory approaches for a class with an alias:

```
#include <pybind11/factory.h>
class Example {
public:
    // ...
    virtual ~Example() = default;
};
class PyExample : public Example, public py::trampoline_self_life_support {
public:
    using Example::Example;
    PyExample(Example &&base) : Example(std::move(base)) {}
};
py::class_<Example, PyExample, py::smart_holder>(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(); }))
    ;
```

8.4.1 Brace initialization

`pybind11::init<>` internally uses C++11 brace initialization to call the constructor of the target class. This means that it can be used to bind *implicit* constructors as well:

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

(continues on next page)

(continued from previous page)

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

Note: Note that brace initialization preferentially invokes constructor overloads taking a `std::initializer_list`. In the rare event that this causes an issue, you can work around it by using `py::init(...)` with a lambda function that constructs the new object as desired.

8.5 Non-public destructors

If a class has a private or protected destructor (as might e.g. be the case in a singleton pattern), a compile error will occur when creating bindings via pybind11. The underlying issue is that the `std::unique_ptr` holder type that is responsible for managing the lifetime of instances will reference the destructor even if no deallocations ever take place. In order to expose classes with private or protected destructors, it is possible to override the holder type via a holder type argument to `py::class_`. Pybind11 provides a helper class `py::nodelete` that disables any destructor invocations. In this case, it is crucial that instances are deallocated on the C++ side to avoid memory leaks.

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

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

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

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

8.6 Destructors that call Python

If a Python function is invoked from a C++ destructor, an exception may be thrown of type *error_already_set*. If this error is thrown out of a class destructor, `std::terminate()` will be called, terminating the process. Class destructors must catch all exceptions of type *error_already_set* to discard the Python exception using *error_already_set::discard_as_unraisable()*.

Every Python function should be treated as *possibly throwing*. When a Python generator stops yielding items, Python will throw a `StopIteration` exception, which can pass though C++ destructors if the generator's stack frame holds the last reference to C++ objects.

For more information, see *the documentation on exceptions*.

```
class MyClass {
public:
    ~MyClass() {
        try {
            py::print("Even printing is dangerous in a destructor");
            py::exec("raise ValueError('This is an unraisable exception')");
        } catch (py::error_already_set &e) {
```

(continues on next page)

(continued from previous page)

```
            // error_context should be information about where/why the occurred,
            // e.g. use __func__ to get the name of the current function
            e.discard_as_unraisable(__func__);
        }
    }
};
```

Note: pybind11 does not support C++ destructors marked `noexcept(false)`.

New in version 2.6.

8.7 Implicit conversions

Suppose that instances of two types `A` and `B` are used in a project, and that an `A` can easily be converted into an instance of type `B` (examples of this could be a fixed and an arbitrary precision number type).

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

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

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

To invoke the function `func` using a variable `a` containing an `A` instance, we'd have to write `func(B(a))` in Python. On the other hand, C++ will automatically apply an implicit type conversion, which makes it possible to directly write `func(a)`.

In this situation (i.e. where `B` has a constructor that converts from `A`), the following statement enables similar implicit conversions on the Python side:

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

Note: Implicit conversions from `A` to `B` only work when `B` is a custom data type that is exposed to Python via pybind11.

To prevent runaway recursion, implicit conversions are non-reentrant: an implicit conversion invoked as part of another implicit conversion of the same type (i.e. from `A` to `B`) will fail.

8.8 Static properties

The section on *Instance and static fields* discussed the creation of instance properties that are implemented in terms of C++ getters and setters.

Static properties can also be created in a similar way to expose getters and setters of static class attributes. Note that the implicit `self` argument also exists in this case and is used to pass the Python `type` subclass instance. This parameter will often not be needed by the C++ side, and the following example illustrates how to instantiate a lambda getter function that ignores it:

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

8.9 Operator overloading

Suppose that we're given the following `Vector2` class with a vector addition and scalar multiplication operation, all implemented using overloaded operators in C++.

```
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;
};
```

The following snippet shows how the above operators can be conveniently exposed to Python.

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

PYBIND11_MODULE(example, m, py::mod_gil_not_used()) {
    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)
```

(continues on next page)

(continued from previous page)

```
        .def("__repr__", &Vector2::toString);
}
```

Note that a line like

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

is really just short hand notation for

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

This can be useful for exposing additional operators that don't exist on the C++ side, or to perform other types of customization. The `py::is_operator` flag marker is needed to inform pybind11 that this is an operator, which returns `NotImplemented` when invoked with incompatible arguments rather than throwing a type error.

Note: To use the more convenient `py::self` notation, the additional header file `pybind11/operators.h` must be included.

See also:

The file `tests/test_operator_overloading.cpp` contains a complete example that demonstrates how to work with overloaded operators in more detail.

8.10 Pickling support

Python's `pickle` module provides a powerful facility to serialize and de-serialize a Python object graph into a binary data stream. To pickle and unpickle C++ classes using pybind11, a `py::pickle()` definition must be provided. Suppose the class in question has the following signature:

```
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;
};
```

Pickling support in Python is enabled by defining the `__setstate__` and `__getstate__` methods[1]. For pybind11 classes, use `py::pickle()` to bind these two functions:

```
py::class_<Pickleable>(m, "Pickleable")
    .def(py::init<std::string>())
    .def("value", &Pickleable::value)
```

(continues on next page)

[1] http://docs.python.org/3/library/pickle.html#pickling-class-instances

(continued from previous page)

```
    .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;
        }
    ));
```

The `__setstate__` part of the `py::pickle()` definition follows the same rules as the single-argument version of `py::init()`. The return type can be a value, pointer or holder type. See *Custom constructors* for details.

Calling `__new__` directly creates the Python wrapper without constructing its C++ value. Outside deprecated placement-new constructor dispatch, passing such an uninitialized wrapper to bound C++ code raises `ValueError`. Calling its `__init__` or a pickle-generated `__setstate__` can still finish construction normally. See *New API for defining custom constructors and pickling functions* for the compatibility behavior and safety limitations of deprecated placement-new callbacks.

An instance can now be pickled as follows:

```
import pickle

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

Note: If given, the second argument to `dumps` must be 2 or larger - 0 and 1 are 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.

See also:

The file `tests/test_pickling.cpp` contains a complete example that demonstrates how to pickle and unpickle types using pybind11 in more detail.

8.11 Deepcopy support

Python normally uses references in assignments. Sometimes a real copy is needed to prevent changing all copies. The `copy` module[2] provides these capabilities.

A class with pickle support is automatically also (deep)copy compatible. However, performance can be improved by adding custom `__copy__` and `__deepcopy__` methods.

For simple classes (deep)copy can be enabled by using the copy constructor, which should look as follows:

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

Note: Dynamic attributes will not be copied in this example.

8.12 Multiple Inheritance

pybind11 can create bindings for types that derive from multiple base types (aka. *multiple inheritance*). To do so, specify all bases in the template arguments of the `py::class_` declaration:

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

The base types can be specified in arbitrary order, and they can even be interspersed with alias types and holder types (discussed earlier in this document)—pybind11 will automatically find out which is which. The only requirement is that the first template argument is the type to be declared.

It is also permitted to inherit multiply from exported C++ classes in Python, as well as inheriting from multiple Python and/or pybind11-exported classes.

There is one caveat regarding the implementation of this feature:

When only one base type is specified for a C++ type that actually has multiple bases, pybind11 will assume that it does not participate in multiple inheritance, which can lead to undefined behavior. In such cases, add the tag `multiple_inheritance` to the class constructor:

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

The tag is redundant and does not need to be specified when multiple base types are listed.

[2] https://docs.python.org/3/library/copy.html

8.13 Module-local class bindings

When creating a binding for a class, pybind11 by default makes that binding "global" across modules. What this means is that instances whose type is defined with a `py::class_` statement in one module can be passed to or returned from a function defined in any other module that is "ABI compatible" with the first, i.e., that was built with sufficiently similar versions of pybind11 and of the C++ compiler and C++ standard library. The internal data structures that pybind11 uses to keep track of its types and instances are shared just as they would be if everything were in the same module. For example, this allows the following:

```
// 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'
```

When writing binding code for a library, this is usually desirable: this allows, for example, splitting up a complex library into multiple Python modules.

In some cases, however, this can cause conflicts. For example, suppose two unrelated modules make use of an external C++ library and each provide custom bindings for one of that library's classes. This will result in an error when a Python program attempts to import both modules (directly or indirectly) because of conflicting definitions on the external type:

```
// 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!
```

To get around this, you can tell pybind11 to keep the external class binding localized to the module by passing the `py::module_local()` attribute into the `py::class_` constructor:

```
// 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);
```

This makes the Python-side `dogs.Pet` and `cats.Pet` into distinct classes, avoiding the conflict and allowing both modules to be loaded. C++ code in the `dogs` module that casts or returns a `Pet` instance will result in a `dogs.Pet` Python instance, while C++ code in the `cats` module will result in a `cats.Pet` Python instance.

This does come with two caveats, however: First, external modules cannot return or cast a `Pet` instance to Python (unless they also provide their own local bindings). Second, from the Python point of view they are two distinct classes.

Note that the locality only applies in the C++ -> Python direction. When passing such a `py::module_local` type into a C++ function, the module-local classes are still considered. This means that if the following function is added to any module (including but not limited to the `cats` and `dogs` modules above) it will be callable with either a `dogs.Pet` or `cats.Pet` argument:

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

For example, suppose the above function is added to each of `cats.cpp`, `dogs.cpp` and `frogs.cpp` (where `frogs.cpp` is some other module that does *not* bind `Pets` at all).

```
>>> 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')
```

It is possible to use `py::module_local()` registrations in one module even if another module registers the same type globally: within the module with the module-local definition, all C++ instances will be cast to the associated bound Python type. In other modules any such values are converted to the global Python type created elsewhere.

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.

Note: 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.

See also:

The file `tests/test_local_bindings.cpp` contains additional examples that demonstrate how `py::module_local()` works.

8.14 Binding protected member functions

It's normally not possible to expose `protected` member functions to Python:

```
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'
```

On one hand, this is good because non-`public` members aren't meant to be accessed from the outside. But we may want to make use of `protected` functions in derived Python classes.

The following pattern makes this possible:

```
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
```

This works because `&Publicist::foo` is exactly the same function as `&A::foo` (same signature and address), just with a different access modifier. The only purpose of the `Publicist` helper class is to make the function name `public`.

If the intent is to expose `protected` `virtual` functions which can be overridden in Python, the publicist pattern can be combined with the previously described trampoline:

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

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

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

(continues on next page)

(continued from previous page)

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

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

8.15 Binding final classes

Some classes may not be appropriate to inherit from. In C++11, classes can use the `final` specifier to ensure that a class cannot be inherited from. The `py::is_final` attribute can be used to ensure that Python classes cannot inherit from a specified type. The underlying C++ type does not need to be declared final.

```
class IsFinal final {};

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

When you try to inherit from such a class in Python, you will now get this error:

```
>>> class PyFinalChild(IsFinal):
...     pass
...
TypeError: type 'IsFinal' is not an acceptable base type
```

Note: This attribute is currently ignored on PyPy

New in version 2.6.

8.16 Binding classes with template parameters

pybind11 can also wrap classes that have template parameters. Consider these classes:

```
struct Cat {};
struct Dog {};

template <typename PetType>
struct Cage {
    Cage(PetType& pet);
    PetType& get();
};
```

C++ templates may only be instantiated at compile time, so pybind11 can only wrap instantiated templated classes. You cannot wrap a non-instantiated template:

```
// BROKEN (this will not compile)
py::class_<Cage>(m, "Cage");
    .def("get", &Cage::get);
```

You must explicitly specify each template/type combination that you want to wrap separately.

```
// ok
py::class_<Cage<Cat>>(m, "CatCage")
    .def("get", &Cage<Cat>::get);

// ok
py::class_<Cage<Dog>>(m, "DogCage")
    .def("get", &Cage<Dog>::get);
```

If your class methods have template parameters you can wrap those as well, but once again each instantiation must be explicitly specified:

```
typename <typename T>
struct MyClass {
    template <typename V>
    T fn(V v);
};

py::class_<MyClass<int>>(m, "MyClassT")
    .def("fn", &MyClass<int>::fn<std::string>);
```

8.17 Custom automatic downcasters

As explained in *Inheritance and automatic downcasting*, pybind11 comes with built-in understanding of the dynamic type of polymorphic objects in C++; that is, returning a Pet to Python produces a Python object that knows it's wrapping a Dog, if Pet has virtual methods and pybind11 knows about Dog and this Pet is in fact a Dog. Sometimes, you might want to provide this automatic downcasting behavior when creating bindings for a class hierarchy that does not use standard C++ polymorphism, such as LLVM[3]. As long as there's some way to determine at runtime whether a downcast is safe, you can proceed by specializing the `pybind11::polymorphic_type_hook` template:

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

namespace PYBIND11_NAMESPACE {
    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);
```

(continues on next page)

[3] https://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html

(continued from previous page)

```
            return static_cast<const Dog*>(src);
        }
        return src;
    }
};
} // namespace PYBIND11_NAMESPACE
```

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

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

Note: pybind11's standard support for downcasting objects whose types have virtual methods is implemented using `polymorphic_type_hook` too, using the standard C++ ability to determine the most-derived type of a polymorphic object using `typeid()` and to cast a base pointer to that most-derived type (even if you don't know what it is) using `dynamic_cast<void*>`.

See also:

The file `tests/test_tagbased_polymorphic.cpp` contains a more complete example, including a demonstration of how to provide automatic downcasting for an entire class hierarchy without writing one get() function for each class.

8.18 Accessing the type object

You can get the type object from a C++ class that has already been registered using:

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

You can directly use `py::type::of(ob)` to get the type object from any python object, just like `type(ob)` in Python.

Note: Other types, like `py::type::of<int>()`, do not work, see *Type conversions*.

New in version 2.6.

8.19 Custom type setup

For advanced use cases, such as enabling garbage collection support, you may wish to directly manipulate the `PyHeapTypeObject` corresponding to a `py::class_` definition.

You can do that using `py::custom_type_setup`:

```
struct ContainerOwnsPythonObjects {
    std::vector<py::object> list;

    void append(const py::object &obj) { list.emplace_back(obj); }
    py::object at(py::ssize_t index) const {
        if (index >= size() || index < 0) {
            throw py::index_error("Index out of range");
        }
        return list.at(py::size_t(index));
    }
    py::ssize_t size() const { return py::ssize_t_cast(list.size()); }
    void clear() { list.clear(); }
};

py::class_<ContainerOwnsPythonObjects> cls(
    m, "ContainerOwnsPythonObjects", py::custom_type_setup([](PyHeapTypeObject *heap_
↪type) {
        auto *type = &heap_type->ht_type;
        type->tp_flags |= Py_TPFLAGS_HAVE_GC;
        type->tp_traverse = [](PyObject *self_base, visitproc visit, void *arg) {
            // https://docs.python.org/3/c-api/typeobj.html#c.PyTypeObject.tp_traverse
            Py_VISIT(Py_TYPE(self_base));
            if (py::detail::is_holder_constructed(self_base)) {
                auto &self = py::cast<ContainerOwnsPythonObjects &>(py::handle(self_
↪base));
                for (auto &item : self.list) {
                    Py_VISIT(item.ptr());
                }
            }
            return 0;
        };
        type->tp_clear = [](PyObject *self_base) {
            if (py::detail::is_holder_constructed(self_base)) {
                auto &self = py::cast<ContainerOwnsPythonObjects &>(py::handle(self_
↪base));
                for (auto &item : self.list) {
                    Py_CLEAR(item.ptr());
                }
                self.list.clear();
            }
            return 0;
        };
    }));
cls.def(py::init<>());
cls.def("append", &ContainerOwnsPythonObjects::append);
cls.def("at", &ContainerOwnsPythonObjects::at);
```

(continues on next page)

(continued from previous page)

```
cls.def("size", &ContainerOwnsPythonObjects::size);
cls.def("clear", &ContainerOwnsPythonObjects::clear);
```

Note: The `py::detail::is_holder_constructed()` guards above are required. During garbage collection, `tp_traverse` and `tp_clear` may be handed an instance whose C++ value has not been constructed yet – for example one created with `__new__` before `__init__` has run. Casting such an instance raises `ValueError`, and an exception must not be allowed to escape either of these slots.

New in version 2.8.

CHAPTER

NINE

EXCEPTIONS

9.1 Built-in C++ to Python exception translation

When Python calls C++ code through pybind11, pybind11 provides a C++ exception handler that will trap C++ exceptions, translate them to the corresponding Python exception, and raise them so that Python code can handle them.

pybind11 defines translations for `std::exception` and its standard subclasses, and several special exception classes that translate to specific Python exceptions. Note that these are not actually Python exceptions, so they cannot be examined using the Python C API. Instead, they are pure C++ objects that pybind11 will translate the corresponding Python exception when they arrive at its exception handler.

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`
`pybind11::stop_iteration`	`StopIteration` (used to implement custom iterators)
`pybind11::index_error`	`IndexError` (used to indicate out of bounds access in `__getitem__`, `__setitem__`, etc.)
`pybind11::key_error`	`KeyError` (used to indicate out of bounds access in `__getitem__`, `__setitem__` in dict-like objects, etc.)
`pybind11::value_error`	`ValueError` (used to indicate wrong value passed in `container.remove(...)`)
`pybind11::type_error`	`TypeError`
`pybind11::buffer_error`	`BufferError`
`pybind11::import_error`	`ImportError`
`pybind11::attribute_error`	`AttributeError`
Any other exception	`RuntimeError`

Exception translation is not bidirectional. That is, *catching* the C++ exceptions defined above will not trap exceptions that originate from Python. For that, catch `pybind11::error_already_set`. See *below* for further details.

There is also a special exception `cast_error` that is thrown by `handle::call()` when the input arguments cannot be converted to Python objects.

9.2 Registering custom translators

If the default exception conversion policy described above is insufficient, pybind11 also provides support for registering custom exception translators. Similar to pybind11 classes, exception translators can be local to the module they are defined in or global to the entire python session. To register a simple exception conversion that translates a C++ exception into a new Python exception using the C++ exception's `what()` method, a helper function is available:

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

This call creates a Python exception class with the name `PyExp` in the given module and automatically converts any encountered exceptions of type `CppExp` into Python exceptions of type `PyExp`.

A matching function is available for registering a local exception translator:

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

It is possible to specify base class for the exception using the third parameter, a `handle`:

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

Then `PyExp` can be caught both as `PyExp` and `RuntimeError`.

The class objects of the built-in Python exceptions are listed in the Python documentation on Standard Exceptions. The default base class is `PyExc_Exception`.

When more advanced exception translation is needed, the functions `py::register_exception_translator(translator)` and `py::register_local_exception_translator(translator)` can be used to register functions that can translate arbitrary exception types (and which may include additional logic to do so). The functions takes a stateless callable (e.g. a function pointer or a lambda function without captured variables) with the call signature `void(std::exception_ptr)`.

When a C++ exception is thrown, the registered exception translators are tried in reverse order of registration (i.e. the last registered translator gets the first shot at handling the exception). All local translators will be tried before a global translator is tried.

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 `py::set_error()` (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:

```
PYBIND11_CONSTINIT static py::gil_safe_call_once_and_store<py::object> exc_storage;
exc_storage.call_once_and_store_result(
    [&]() { return py::exception<MyCustomException>(m, "MyCustomError"); });
py::register_exception_translator([](std::exception_ptr p) {
    try {
        if (p) std::rethrow_exception(p);
    } catch (const MyCustomException &e) {
        py::set_error(exc_storage.get_stored(), e.what());
    } catch (const OtherException &e) {
```

(continues on next page)

(continued from previous page)

```
        py::set_error(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.

See also:

The file `tests/test_exceptions.cpp` contains examples of various custom exception translators and custom exception types.

Note: Call `py::set_error()` for every exception caught in a custom exception translator. Failure to do so will cause Python to crash with `SystemError: error return without exception set`.

Exceptions that you do not plan to handle should simply not be caught, or may be explicitly (re-)thrown to delegate it to the other, previously-declared existing exception translators.

Note that `libc++` and `libstdc++` behave differently under macOS with `-fvisibility=hidden`. Therefore exceptions that are used across ABI boundaries need to be explicitly exported, as exercised in `tests/test_exceptions.h`. See also: "Problems with C++ exceptions" under GCC Wiki.

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) {
      py::set_error(PyExc_ArgumentError, "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) {
      py::set_error(PyExc_ArgumentError, "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 Handling exceptions from Python in C++

When C++ calls Python functions, such as in a callback function or when manipulating Python objects, and Python raises an `Exception`, pybind11 converts the Python exception into a C++ exception of type `pybind11::error_already_set` whose payload contains a C++ string textual summary and the actual Python exception. `error_already_set` is used to propagate Python exception back to Python (or possibly, handle them in C++).

Exception raised in Python	Thrown as C++ exception type
Any Python `Exception`	`pybind11::error_already_set`

For example:

```
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;
    }
}
```

Note that C++ to Python exception translation does not apply here, since that is a method for translating C++ exceptions to Python, not vice versa. The error raised from Python is always `error_already_set`.

This example illustrates this behavior:

```
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
```

(continues on next page)

(continued from previous page)

```
    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 Handling errors from the Python C API

Where possible, use *pybind11 wrappers* instead of calling the Python C API directly. When calling the Python C API directly, in addition to manually managing reference counts, one must follow the pybind11 error protocol, which is outlined here.

After calling the Python C API, if Python returns an error, `throw py::error_already_set();`, which allows pybind11 to deal with the exception and pass it back to the Python interpreter. This includes calls to the error setting functions such as `py::set_error()`.

```
py::set_error(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");
```

Alternately, to ignore the error, call PyErr_Clear.

Any Python error must be thrown or cleared, or Python/pybind11 will be left in an invalid state.

9.6 Handling warnings from the Python C API

Wrappers for handling Python warnings are provided in `pybind11/warnings.h`. This header must be included explicitly; it is not transitively included via `pybind11/pybind11.h`.

Warnings can be raised with the `warn` function:

```
py::warnings::warn("This is a warning!", PyExc_Warning);

// Optionally, a `stack_level` can be specified.
py::warnings::warn("Another one!", PyExc_DeprecationWarning, 3);
```

New warning types can be registered at the module level using `new_warning_type`:

```
py::warnings::new_warning_type(m, "CustomWarning", PyExc_RuntimeWarning);
```

9.7 Chaining exceptions ('raise from')

Python has a mechanism for indicating that exceptions were caused by other exceptions:

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

To do a similar thing in pybind11, you can use the `py::raise_from` function. It sets the current python error indicator, so to continue propagating the exception you should `throw py::error_already_set()`.

```
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();
}
```

New in version 2.8.

9.8 Handling unraisable exceptions

If a Python function invoked from a C++ destructor or any function marked `noexcept(true)` (collectively, "noexcept functions") throws an exception, there is no way to propagate the exception, as such functions may not throw. Should they throw or fail to catch any exceptions in their call graph, the C++ runtime calls `std::terminate()` to abort immediately.

Similarly, Python exceptions raised in a class's `__del__` method do not propagate, but `sys.unraisablehook()` is triggered and an auditing event is logged.

Any noexcept function should have a try-catch block that traps class:*error_already_set* (or any other exception that can occur). Note that pybind11 wrappers around Python exceptions such as `pybind11::value_error` are *not* Python exceptions; they are C++ exceptions that pybind11 catches and converts to Python exceptions. Noexcept functions cannot propagate these exceptions either. A useful approach is to convert them to Python exceptions and then `discard_as_unraisable` as shown below.

```
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);
    }
}
```

New in version 2.6.

CHAPTER

TEN

SMART POINTERS & PY::CLASS_

The binding generator for classes, py::class_, can be passed a template type that denotes a special *holder* type that is used to manage references to the object. If no such holder type template argument is given, the default for a type T is std::unique_ptr<T>.

Note: A py::class_ for a given C++ type T — and all its derived types — can only use a single holder type.

10.1 py::smart_holder

Starting with pybind11v3, py::smart_holder is built into pybind11. It is the recommended py::class_ holder for most situations. However, for backward compatibility it is **not** the default holder, and there are no plans to make it the default holder in the future.

It is extremely easy to use the safer and more versatile py::smart_holder: simply add py::smart_holder to py::class_:

- py::class_<T> to
- py::class_<T, py::smart_holder>.

Note: A shorthand, py::classh<T>, is provided for py::class_<T, py::smart_holder>. The h in py::classh stands for **smart_holder** but is shortened for brevity, ensuring it has the same number of characters as py::class_. This design choice facilitates easy experimentation with py::smart_holder without introducing distracting whitespace noise in diffs.

The py::smart_holder functionality includes the following:

- Support for **two-way** Python/C++ conversions for both std::unique_ptr<T> and std::shared_ptr<T> **simultaneously**.
- Passing a Python object back to C++ via std::unique_ptr<T>, safely **disowning** the Python object.
- Safely passing "trampoline" objects (objects with C++ virtual function overrides implemented in Python, see *Overriding virtual functions in Python*) via std::unique_ptr<T> or std::shared_ptr<T> back to C++: associated Python objects are automatically kept alive for the lifetime of the smart-pointer.
- Full support for std::enable_shared_from_this (cppreference).

10.2 std::unique_ptr

This is the default py::class_ holder and works as expected in most situations. However, handling base-and-derived classes involves a reinterpret_cast, which is, strictly speaking, undefined behavior. Also note that the std::unique_ptr holder only supports passing a std::unique_ptr from C++ to Python, but not the other way around. For example, the following code works as expected with py::class_<Example>:

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

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

However, this will fail with py::class_<Example> (but works with py::class_<Example, py::smart_holder>):

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

Note: The reinterpret_cast mentioned above is here. For completeness: The same cast is also applied to py::smart_holder, but that is safe, because py::smart_holder is not templated.

10.3 std::shared_ptr

It is possible to use std::shared_ptr as the holder, for example:

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

Compared to using py::class_<Example, py::smart_holder>, there are two noteworthy disadvantages:

- Because a py::class_ for a given C++ type T can only use a single holder type, std::unique_ptr<T> cannot even be passed from C++ to Python. This will become apparent only at runtime, often through a segmentation fault.
- Similar to the std::unique_ptr holder, the handling of base-and-derived classes involves a reinterpret_cast that has strictly speaking undefined behavior, although it works as expected in most situations.

10.4 Custom smart pointers

For custom smart pointers (e.g. c10::intrusive_ptr in pytorch), transparent conversions can be enabled using a macro invocation similar to the following. It must be declared at the top namespace level before any binding code:

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

The first argument of PYBIND11_DECLARE_HOLDER_TYPE() should be a placeholder name that is used as a template parameter of the second argument. Thus, feel free to use any identifier, but use it consistently on both sides; also, don't use the name of a type that already exists in your codebase.

The macro also accepts a third optional boolean parameter that is set to false by default. Specify

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

if `SmartPtr<T>` can always be initialized from a `T*` pointer without the risk of inconsistencies (such as multiple independent `SmartPtr` instances believing that they are the sole owner of the `T*` pointer). A common situation where `true` should be passed is when the `T` instances use *intrusive* reference counting.

Please take a look at the *General notes regarding convenience macros* before using this feature.

By default, pybind11 assumes that your custom smart pointer has a standard interface, i.e. provides a `.get()` member function to access the underlying raw pointer. If this is not the case, pybind11's `holder_helper` must be specialized:

```
// 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 { namespace detail {
    template <typename T>
    struct holder_helper<SmartPtr<T>> { // <-- specialization
        static const T *get(const SmartPtr<T> &p) { return p.getPointer(); }
    };
}}
```

The above specialization informs pybind11 that the custom `SmartPtr` class provides `.get()` functionality via `.getPointer()`.

Note: The two noteworthy disadvantages mentioned under the `std::shared_ptr` section apply similarly to custom smart pointer holders, but there is no established safe alternative in this case.

See also:

The file `tests/test_smart_ptr.cpp` contains a complete example that demonstrates how to work with custom reference-counting holder types in more detail.

CHAPTER

ELEVEN

TYPE CONVERSIONS

Apart from enabling cross-language function calls, a fundamental problem that a binding tool like pybind11 must address is to provide access to native Python types in C++ and vice versa. There are three fundamentally different ways to do this—which approach is preferable for a particular type depends on the situation at hand.

1. Use a native C++ type everywhere. In this case, the type must be wrapped using pybind11-generated bindings so that Python can interact with it.
2. Use a native Python type everywhere. It will need to be wrapped so that C++ functions can interact with it.
3. Use a native C++ type on the C++ side and a native Python type on the Python side. pybind11 refers to this as a *type conversion*.

   Type conversions are the most "natural" option in the sense that native (non-wrapped) types are used everywhere. The main downside is that a copy of the data must be made on every Python ⟷ C++ transition: this is needed since the C++ and Python versions of the same type generally won't have the same memory layout.

   pybind11 can perform many kinds of conversions automatically. An overview is provided in the table "*List of all builtin conversions*".

The following subsections discuss the differences between these options in more detail. The main focus in this section is on type conversions, which represent the last case of the above list.

11.1 Overview

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

Exposing a custom C++ type using `py::class_` was covered in detail in the *Object-oriented code* section. There, the underlying data structure is always the original C++ class while the `py::class_` wrapper provides a Python interface. Internally, when an object like this is sent from C++ to Python, pybind11 will just add the outer wrapper layer over the native C++ object. Getting it back from Python is just a matter of peeling off the wrapper.

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

This is the exact opposite situation. Now, we have a type which is native to Python, like a `tuple` or a `list`. One way to get this data into C++ is with the `py::object` family of wrappers. These are explained in more detail in the *Python types* section. We'll just give a quick example here:

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

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

The Python `list` is not converted in any way – it's just wrapped in a C++ `py::list` class. At its core it's still a Python object. Copying a `py::list` will do the usual reference-counting like in Python. Returning the object to Python will just remove the thin wrapper.

3. Converting between native C++ and Python types

In the previous two cases we had a native type in one language and a wrapper in the other. Now, we have native types on both sides and we convert between them.

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

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

In this case, pybind11 will construct a new `std::vector<int>` and copy each element from the Python `list`. The newly constructed object will be passed to `print_vector`. The same thing happens in the other direction: a new `list` is made to match the value returned from C++.

Lots of these conversions are supported out of the box, as shown in the table below. They are very convenient, but keep in mind that these conversions are fundamentally based on copying data. This is perfectly fine for small immutable types but it may become quite expensive for large data structures. This can be avoided by overriding the automatic conversion with a custom wrapper (i.e. the above-mentioned approach 1). This requires some manual effort and more details are available in the *Making opaque types* section.

11.1.1 List of all builtin conversions

The following basic data types are supported out of the box (some may require an additional extension header to be included). To pass other data structures as arguments and return values, refer to the section on binding *Object-oriented code*.

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`
`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`

continues on next page

Table 1 – continued from previous page

Data type	Description	Header file
`const char32_t *`	UTF-32 string literal	`pybind11/pybind11.h`
`const wchar_t *`	Wide string literal	`pybind11/pybind11.h`
`std::string`	STL dynamic UTF-8 string	`pybind11/pybind11.h`
`std::u16string`	STL dynamic UTF-16 string	`pybind11/pybind11.h`
`std::u32string`	STL dynamic UTF-32 string	`pybind11/pybind11.h`
`std::wstring`	STL dynamic wide string	`pybind11/pybind11.h`
`std::string_view`, `std::u16string_view`, etc.	STL C++17 string views	`pybind11/pybind11.h`
`std::pair<T1, T2>`	Pair of two custom types	`pybind11/pybind11.h`
`std::tuple<...>`	Arbitrary tuple of types	`pybind11/pybind11.h`
`std::reference_wrapper<...>`	Reference type wrapper	`pybind11/pybind11.h`
`std::complex<T>`	Complex numbers	`pybind11/complex.h`
`std::array<T, Size>`	STL static array	`pybind11/stl.h`
`std::vector<T>`	STL dynamic array	`pybind11/stl.h`
`std::deque<T>`	STL double-ended queue	`pybind11/stl.h`
`std::valarray<T>`	STL value array	`pybind11/stl.h`
`std::list<T>`	STL linked list	`pybind11/stl.h`
`std::map<T1, T2>`	STL ordered map	`pybind11/stl.h`
`std::unordered_map<T1, T2>`	STL unordered map	`pybind11/stl.h`
`std::set<T>`	STL ordered set	`pybind11/stl.h`
`std::unordered_set<T>`	STL unordered set	`pybind11/stl.h`
`std::optional<T>`	STL optional type (C++17)	`pybind11/stl.h`
`std::experimental::optional<T`	STL optional type (exp.)	`pybind11/stl.h`
`std::variant<...>`	Type-safe union (C++17)	`pybind11/stl.h`
`std::filesystem::path`	STL path (C++17)[1]	`pybind11/stl/filesystem.h`
`std::function<...>`	STL polymorphic function	`pybind11/functional.h`
`std::chrono::duration<...>`	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 memory	`pybind11/eigen.h`
`Eigen::SparseMatrix<...>`	Eigen: sparse matrix	`pybind11/eigen.h`

11.2 Strings, bytes and Unicode conversions

11.2.1 Passing Python strings to C++

When a Python `str` is passed from Python to a C++ function that accepts `std::string` or `char *` as arguments, pybind11 will encode the Python string to UTF-8. All Python `str` can be encoded in UTF-8, so this operation does not fail.

The C++ language is encoding agnostic. It is the responsibility of the programmer to track encodings. It's often easiest to simply use UTF-8 everywhere.

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

(continues on next page)

[1] `std::filesystem::path` is converted to `pathlib.Path` and can be loaded from `os.PathLike`, `str`, and `bytes`.

(continued from previous page)

```
        cout << s;
    }
);
m.def("utf8_charptr",
    [](const char *s) {
        cout << "My favorite food is\n";
        cout << s;
    }
);
```

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

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

Note: Some terminal emulators do not support UTF-8 or emoji fonts and may not display the example above correctly.

The results are the same whether the C++ function accepts arguments by value or reference, and whether or not `const` is used.

Passing bytes to C++

A Python `bytes` object will be passed to C++ functions that accept `std::string` or `char*` *without* conversion. In order to make a function *only* accept `bytes` (and not `str`), declare it as taking a `py::bytes` argument.

11.2.2 Returning C++ strings to Python

When a C++ function returns a `std::string` or `char*` to a Python caller, **pybind11 will assume that the string is valid UTF-8** and will decode it to a native Python `str`, using the same API as Python uses to perform `bytes.decode('utf-8')`. If this implicit conversion fails, pybind11 will raise a `UnicodeDecodeError`.

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

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

Because UTF-8 is inclusive of pure ASCII, there is never any issue with returning a pure ASCII string to Python. If there is any possibility that the string is not pure ASCII, it is necessary to ensure the encoding is valid UTF-8.

Warning: Implicit conversion assumes that a returned `char *` is null-terminated. If there is no null terminator a buffer overrun will occur.

Explicit conversions

If some C++ code constructs a `std::string` that is not a UTF-8 string, one can perform a explicit conversion and return a `py::str` object. Explicit conversion has the same overhead as implicit conversion.

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

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

The Python C API provides several built-in codecs. Note that these all return *new* references, so use *reinterpret_steal()* when converting them to a *str*.

One could also use a third party encoding library such as libiconv to transcode to UTF-8.

Return C++ strings without conversion

If the data in a C++ `std::string` does not represent text and should be returned to Python as `bytes`, then one can return the data as a `py::bytes` object.

```
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'
```

Note the asymmetry: pybind11 will convert `bytes` to `std::string` without encoding, but cannot convert `std::string` back to `bytes` implicitly.

```
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
```

(continues on next page)

(continued from previous page)

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xba in position 0: invalid start␣
↪byte
```

11.2.3 Wide character strings

When a Python `str` is passed to a C++ function expecting `std::wstring`, `wchar_t*`, `std::u16string` or `std::u32string`, the `str` will be encoded to UTF-16 or UTF-32 depending on how the C++ compiler implements each type, in the platform's native endianness. When strings of these types are returned, they are assumed to contain valid UTF-16 or UTF-32, and will be decoded to Python `str`.

```
#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;
    }
);
```

Strings in multibyte encodings such as Shift-JIS must transcoded to a UTF-8/16/32 before being returned to Python.

11.2.4 Character literals

C++ functions that accept character literals as input will receive the first character of a Python `str` as their input. If the string is longer than one Unicode character, trailing characters will be ignored.

When a character literal is returned from C++ (such as a `char` or a `wchar_t`), it will be converted to a `str` that represents the single character.

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

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

While C++ will cast integers to character types (`char c = 0x65;`), pybind11 does not convert Python integers to characters implicitly. The Python function `chr()` can be used to convert integers to characters.

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

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

If the desire is to work with an 8-bit integer, use `int8_t` or `uint8_t` as the argument type.

Grapheme clusters

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

```
>>> example.pass_wchar("é")
'é'

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

>>> combining_e_acute
'e''

>>> combining_e_acute == "é"
False

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

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

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

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

11.2.5 C++17 string views

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).

A string view does not own its character data. When a view is loaded as an argument to a pybind11-bound function, pybind11 keeps the Python object that provides the data alive until the function returns. This also applies to views nested in automatically converted STL containers. The C++ function must not retain any such view after it returns unless it separately guarantees that the backing storage remains alive and valid.

Lifetime support keeps the Python object alive, but does not prevent its storage from being invalidated. For example, if C++ releases the GIL or calls back into Python, resizing a backing `bytearray` while the view is in use can invalidate the view.

A direct Python-to-C++ `py::cast()` made when no bound-function call is active has no such lifetime support. When a cast to a non-owning view succeeds, the caller must keep the backing Python object alive, with its storage unchanged,

for as long as the view is used. For a container of views, this requirement applies to every element: retain the elements directly or through an unmodified owning container, and do not cast an iterable that creates temporary elements.

Some view conversions require temporary backing storage, for example to encode text. Outside a bound-function call, such conversions raise `cast_error` instead of returning a dangling view.

11.2.6 References

- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)
- C++ - Using STL Strings at Win32 API Boundaries

11.3 STL containers

11.3.1 Automatic conversion

When including the additional header file `pybind11/stl.h`, conversions between `std::vector<>`/`std::deque<>`/`std::list<>`/`std::array<>`/`std::valarray<>`, `std::set<>`/`std::unordered_set<>`, and `std::map<>`/`std::unordered_map<>` and the Python `list`, `set` and `dict` data structures are automatically enabled. The types `std::pair<>` and `std::tuple<>` are already supported out of the box with just the core `pybind11/pybind11.h` header.

The major downside of these implicit conversions is that containers must be converted (i.e. copied) on every Python->C++ and C++->Python transition, which can have implications on the program semantics and performance. Please read the next sections for more details and alternative approaches that avoid this.

Copying the container does not make non-owning element types own their data. In particular, containers of C++ string views have additional *C++17 string views* requirements.

Note: Arbitrary nesting of any of these types is possible.

See also:

The file `tests/test_stl.cpp` contains a complete example that demonstrates how to pass STL data types in more detail.

11.3.2 C++17 library containers

The `pybind11/stl.h` header also includes support for `std::optional<>` and `std::variant<>`. These require a C++17 compiler and standard library. In C++14 mode, `std::experimental::optional<>` is supported if available.

Various versions of these containers also exist for C++11 (e.g. in Boost). pybind11 provides an easy way to specialize the `type_caster` for such types:

```
// `boost::optional` as an example -- can be any `std::optional`-like container
namespace PYBIND11_NAMESPACE { namespace detail {
    template <typename T>
    struct type_caster<boost::optional<T>> : optional_caster<boost::optional<T>> {};
}}
```

The above should be placed in a header file and included in all translation units where automatic conversion is needed. Similarly, a specialization can be provided for custom variant types:

```
// `boost::variant` as an example -- can be any `std::variant`-like container
namespace PYBIND11_NAMESPACE { 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_NAMESPACE::detail
```

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

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

Note: pybind11 only supports the modern implementation of `boost::variant` which makes use of variadic templates. This requires Boost 1.56 or newer.

11.3.3 Making opaque types

pybind11 heavily relies on a template matching mechanism to convert parameters and return values that are constructed from STL data types such as vectors, linked lists, hash tables, etc. This even works in a recursive manner, for instance to deal with lists of hash maps of pairs of elementary and custom types, etc.

However, a fundamental limitation of this approach is that internal conversions between Python and C++ types involve a copy operation that prevents pass-by-reference semantics. What does this mean?

Suppose we bind the following function

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

and call it from Python, the following happens:

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

As you can see, when passing STL data structures by reference, modifications are not propagated back the Python side. A similar situation arises when exposing STL data structures using the `def_readwrite` or `def_readonly` functions:

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

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

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

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

In this case, properties can be read and written in their entirety. However, an `append` operation involving such a list type has no effect:

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

Finally, the involved copy operations can be costly when dealing with very large lists. To deal with all of the above situations, pybind11 provides a macro named `PYBIND11_MAKE_OPAQUE(T)` that disables the template-based conversion machinery of types, thus rendering them *opaque*. The contents of opaque objects are never inspected or extracted, hence they *can* be passed by reference. For instance, to turn `std::vector<int>` into an opaque type, add the declaration

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

before any binding code (e.g. invocations to `class_::def()`, etc.). This macro must be specified at the top level (and outside of any namespaces), since it adds a template instantiation of `type_caster`. If your binding code consists of multiple compilation units, it must be present in every file (typically via a common header) preceding any usage of `std::vector<int>`. Opaque types must also have a corresponding `py::class_` declaration to associate them with a name in Python, and to define a set of available operations, e.g.:

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

See also:

The file `tests/test_opaque_types.cpp` contains a complete example that demonstrates how to create and expose opaque types using pybind11 in more detail.

11.3.4 Binding STL containers

The ability to expose STL containers as native Python objects is a fairly common request, hence pybind11 also provides an optional header file named `pybind11/stl_bind.h` that does exactly this. The mapped containers try to match the behavior of their native Python counterparts as much as possible.

The following example showcases usage of `pybind11/stl_bind.h`:

```
// 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");
```

When binding STL containers pybind11 considers the types of the container's elements to decide whether the container should be confined to the local module (via the *Module-local class bindings* feature). If the container element types are anything other than already-bound custom types bound without `py::module_local()` the container binding will have `py::module_local()` applied. This includes converting types such as numeric types, strings, Eigen types; and types that have not yet been bound at the time of the stl container binding. This module-local binding is designed to avoid potential conflicts between module bindings (for example, from two separate modules each attempting to bind `std::vector<int>` as a python type).

It is possible to override this behavior to force a definition to be either module-local or global. To do so, you can pass the attributes `py::module_local()` (to make the binding module-local) or `py::module_local(false)` (to make the binding global) into the `py::bind_vector` or `py::bind_map` arguments:

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

Note, however, that such a global binding would make it impossible to load this module at the same time as any other pybind module that also attempts to bind the same container type (`std::vector<int>` in the above example).

See *Module-local class bindings* for more details on module-local bindings.

See also:

The file `tests/test_stl_binders.cpp` shows how to use the convenience STL container wrappers.

11.4 Functional

The following features must be enabled by including `pybind11/functional.h`.

11.4.1 Callbacks and passing anonymous functions

The C++11 standard brought lambda functions and the generic polymorphic function wrapper `std::function<>` to the C++ programming language, which enable powerful new ways of working with functions. Lambda functions come in two flavors: stateless lambda function resemble classic function pointers that link to an anonymous piece of code, while stateful lambda functions additionally depend on captured variables that are stored in an anonymous *lambda closure object*.

Here is a simple example of a C++ function that takes an arbitrary function (stateful or stateless) with signature `int -> int` as an argument and runs it with the value 10.

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

The example below is more involved: it takes a function of signature `int -> int` and returns another function of the same kind. The return value is a stateful lambda function, which stores the value `f` in the capture object and adds 1 to its return value upon execution.

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

This example demonstrates using python named parameters in C++ callbacks which requires using `py::cpp_function` as a wrapper. Usage is similar to defining methods of classes:

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

After including the extra header file `pybind11/functional.h`, it is almost trivial to generate binding code for all of these functions.

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

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

The following interactive session shows how to call them from Python.

```
$ python
>>> import example
>>> def square(i):
...     return i * i
...
>>> example.func_arg(square)
100L
>>> square_plus_1 = example.func_ret(square)
>>> square_plus_1(4)
```

(continues on next page)

(continued from previous page)

```
17L
>>> plus_1 = func_cpp()
>>> plus_1(number=43)
44L
```

Warning: Keep in mind that passing a function from C++ to Python (or vice versa) will instantiate a piece of wrapper code that translates function invocations between the two languages. Naturally, this translation increases the computational cost of each function call somewhat. A problematic situation can arise when a function is copied back and forth between Python and C++ many times in a row, in which case the underlying wrappers will accumulate correspondingly. The resulting long sequence of C++ -> Python -> C++ -> ... roundtrips can significantly decrease performance.

There is one exception: pybind11 detects case where a stateless function (i.e. a function pointer or a lambda function without captured variables) is passed as an argument to another C++ function exposed in Python. In this case, there is no overhead. Pybind11 will extract the underlying C++ function pointer from the wrapped function to sidestep a potential C++ -> Python -> C++ roundtrip. This is demonstrated in `tests/test_callbacks.cpp`.

Note: This functionality is very useful when generating bindings for callbacks in C++ libraries (e.g. GUI libraries, asynchronous networking libraries, etc.).

The file `tests/test_callbacks.cpp` contains a complete example that demonstrates how to work with callbacks and anonymous functions in more detail.

11.5 Chrono

When including the additional header file `pybind11/chrono.h` conversions from C++11 chrono datatypes to python datetime objects are automatically enabled. This header also enables conversions of python floats (often from sources such as `time.monotonic()`, `time.perf_counter()` and `time.process_time()`) into durations.

11.5.1 An overview of clocks in C++11

A point of confusion when using these conversions is the differences between clocks provided in C++11. There are three clock types defined by the C++11 standard and users can define their own if needed. Each of these clocks have different properties and when converting to and from python will give different results.

The first clock defined by the standard is `std::chrono::system_clock`. This clock measures the current date and time. However, this clock changes with to updates to the operating system time. For example, if your time is synchronised with a time server this clock will change. This makes this clock a poor choice for timing purposes but good for measuring the wall time.

The second clock defined in the standard is `std::chrono::steady_clock`. This clock ticks at a steady rate and is never adjusted. This makes it excellent for timing purposes, however the value in this clock does not correspond to the current date and time. Often this clock will be the amount of time your system has been on, although it does not have to be. This clock will never be the same clock as the system clock as the system clock can change but steady clocks cannot.

The third clock defined in the standard is `std::chrono::high_resolution_clock`. This clock is the clock that has the highest resolution out of the clocks in the system. It is normally a typedef to either the system clock or the steady clock but can be its own independent clock. This is important as when using these conversions as the types you get in

python for this clock might be different depending on the system. If it is a typedef of the system clock, python will get datetime objects, but if it is a different clock they will be timedelta objects.

11.5.2 Provided conversions

C++ to Python

- `std::chrono::system_clock::time_point` → `datetime.datetime`
  System clock times are converted to python datetime instances. They are in the local timezone, but do not have any timezone information attached to them (they are naive datetime objects). The `fold` attribute distinguishes the two occurrences of an ambiguous local time when clocks move backward.
- `std::chrono::duration` → `datetime.timedelta`
  Durations are converted to timedeltas, any precision in the duration greater than microseconds is lost by rounding towards zero.
- `std::chrono::[other_clocks]::time_point` → `datetime.timedelta`
  Any clock time that is not the system clock is converted to a time delta. This timedelta measures the time from the clocks epoch to now.

Python to C++

- `datetime.datetime` or `datetime.date` or `datetime.time` → `std::chrono::system_clock::time_point`
  Date/time objects are converted into system clock timepoints. Any timezone information is ignored and the type is treated as a naive object in the local timezone. The `fold` attribute selects the earlier (`0`) or later (`1`) occurrence of an ambiguous local time. Nonexistent local times when clocks move forward follow Python's `datetime.timestamp()` behavior and cannot round-trip unchanged.
- `datetime.timedelta` → `std::chrono::duration`
  Time delta are converted into durations with microsecond precision.
- `datetime.timedelta` → `std::chrono::[other_clocks]::time_point`
  Time deltas that are converted into clock timepoints are treated as the amount of time from the start of the clocks epoch.
- `float` → `std::chrono::duration`
  Floats that are passed to C++ as durations be interpreted as a number of seconds. These will be converted to the duration using `duration_cast` from the float.
- `float` → `std::chrono::[other_clocks]::time_point`
  Floats that are passed to C++ as time points will be interpreted as the number of seconds from the start of the clocks epoch.

11.6 Eigen

Eigen is C++ header-based library for dense and sparse linear algebra. Due to its popularity and widespread adoption, pybind11 provides transparent conversion and limited mapping support between Eigen and Scientific Python linear algebra data types.

To enable the built-in Eigen support you must include the optional header file `pybind11/eigen.h`.

11.6.1 Pass-by-value

When binding a function with ordinary Eigen dense object arguments (for example, `Eigen::MatrixXd`), pybind11 will accept any input value that is already (or convertible to) a `numpy.ndarray` with dimensions compatible with the Eigen type, copy its values into a temporary Eigen variable of the appropriate type, then call the function with this temporary variable.

Sparse matrices are similarly copied to or from `scipy.sparse.csr_matrix/scipy.sparse.csc_matrix` objects.

11.6.2 Pass-by-reference

One major limitation of the above is that every data conversion implicitly involves a copy, which can be both expensive (for large matrices) and disallows binding functions that change their (Matrix) arguments. Pybind11 allows you to work around this by using Eigen's `Eigen::Ref<MatrixType>` class much as you would when writing a function taking a generic type in Eigen itself (subject to some limitations discussed below).

When calling a bound function accepting a `Eigen::Ref<const MatrixType>` type, pybind11 will attempt to avoid copying by using an `Eigen::Map` object that maps into the source `numpy.ndarray` data: this requires both that the data types are the same (e.g. `dtype='float64'` and `MatrixType::Scalar` is `double`); and that the storage is layout compatible. The latter limitation is discussed in detail in the section below, and requires careful consideration: by default, numpy matrices and Eigen matrices are *not* storage compatible.

If the numpy matrix cannot be used as is (either because its types differ, e.g. passing an array of integers to an Eigen parameter requiring doubles, or because the storage is incompatible), pybind11 makes a temporary copy and passes the copy instead.

When a bound function parameter is instead `Eigen::Ref<MatrixType>` (note the lack of `const`), pybind11 will only allow the function to be called if it can be mapped *and* if the numpy array is writeable (that is `a.flags.writeable` is true). Any access (including modification) made to the passed variable will be transparently carried out directly on the `numpy.ndarray`.

This means you can write code such as the following and have it work as expected:

```
void scale_by_2(Eigen::Ref<Eigen::VectorXd> v) {
    v *= 2;
}
```

Note, however, that you will likely run into limitations due to numpy and Eigen's difference default storage order for data; see the below section on *Storage orders* for details on how to bind code that won't run into such limitations.

Note: Passing by reference is not supported for sparse types.

11.6.3 Returning values to Python

When returning an ordinary dense Eigen matrix type to numpy (e.g. `Eigen::MatrixXd` or `Eigen::RowVectorXf`) pybind11 keeps the matrix and returns a numpy array that directly references the Eigen matrix: no copy of the data is performed. The numpy array will have `array.flags.owndata` set to `False` to indicate that it does not own the data, and the lifetime of the stored Eigen matrix will be tied to the returned `array`.

If you bind a function with a non-reference, `const` return type (e.g. `const Eigen::MatrixXd`), the same thing happens except that pybind11 also sets the numpy array's `writeable` flag to false.

If you return an lvalue reference or pointer, the usual pybind11 rules apply, as dictated by the binding function's return value policy (see the documentation on *Return value policies* for full details). That means, without an explicit return

value policy, lvalue references will be copied and pointers will be managed by pybind11. In order to avoid copying, you should explicitly specify an appropriate return value policy, as in the following example:

```
class MyClass {
    Eigen::MatrixXd big_mat = Eigen::MatrixXd::Zero(10000, 10000);
public:
    Eigen::MatrixXd &getMatrix() { return big_mat; }
    const Eigen::MatrixXd &viewMatrix() { return big_mat; }
};

// Later, in binding code:
py::class_<MyClass>(m, "MyClass")
    .def(py::init<>())
    .def("copy_matrix", &MyClass::getMatrix) // Makes a copy!
    .def("get_matrix", &MyClass::getMatrix, py::return_value_policy::reference_internal)
    .def("view_matrix", &MyClass::viewMatrix, py::return_value_policy::reference_
↪internal)
    ;
```

```
a = MyClass()
m = a.get_matrix()  # flags.writeable = True,  flags.owndata = False
v = a.view_matrix()  # flags.writeable = False, flags.owndata = False
c = a.copy_matrix()  # flags.writeable = True,  flags.owndata = True
# m[5,6] and v[5,6] refer to the same element, c[5,6] does not.
```

Note in this example that `py::return_value_policy::reference_internal` is used to tie the life of the MyClass object to the life of the returned arrays.

You may also return an `Eigen::Ref`, `Eigen::Map` or other map-like Eigen object (for example, the return value of `matrix.block()` and related methods) that map into a dense Eigen type. When doing so, the default behaviour of pybind11 is to simply reference the returned data: you must take care to ensure that this data remains valid! You may ask pybind11 to explicitly *copy* such a return value by using the `py::return_value_policy::copy` policy when binding the function. You may also use `py::return_value_policy::reference_internal` or a `py::keep_alive` to ensure the data stays valid as long as the returned numpy array does.

When returning such a reference of map, pybind11 additionally respects the readonly-status of the returned value, marking the numpy array as non-writeable if the reference or map was itself read-only.

Note: Sparse types are always copied when returned.

11.6.4 Storage orders

Passing arguments via `Eigen::Ref` has some limitations that you must be aware of in order to effectively pass matrices by reference. First and foremost is that the default `Eigen::Ref<MatrixType>` class requires contiguous storage along columns (for column-major types, the default in Eigen) or rows if `MatrixType` is specifically an `Eigen::RowMajor` storage type. The former, Eigen's default, is incompatible with `numpy`'s default row-major storage, and so you will not be able to pass numpy arrays to Eigen by reference without making one of two changes.

(Note that this does not apply to vectors (or column or row matrices): for such types the "row-major" and "column-major" distinction is meaningless).

The first approach is to change the use of `Eigen::Ref<MatrixType>` to the more general `Eigen::Ref<MatrixType, 0, Eigen::Stride<Eigen::Dynamic, Eigen::Dynamic>>` (or similar type with a fully dynamic stride type in the

third template argument). Since this is a rather cumbersome type, pybind11 provides a `py::EigenDRef<MatrixType>` type alias for your convenience (along with EigenDMap for the equivalent Map, and EigenDStride for just the stride type).

This type allows Eigen to map into any arbitrary storage order. This is not the default in Eigen for performance reasons: contiguous storage allows vectorization that cannot be done when storage is not known to be contiguous at compile time. The default `Eigen::Ref` stride type allows non-contiguous storage along the outer dimension (that is, the rows of a column-major matrix or columns of a row-major matrix), but not along the inner dimension.

This type, however, has the added benefit of also being able to map numpy array slices. For example, the following (contrived) example uses Eigen with a numpy slice to multiply by 2 all coefficients that are both on even rows (0, 2, 4, ...) and in columns 2, 5, or 8:

```
m.def("scale", [](py::EigenDRef<Eigen::MatrixXd> m, double c) { m *= c; });
```

```
# a = np.array(...)
scale_by_2(myarray[0::2, 2:9:3])
```

The second approach to avoid copying is more intrusive: rearranging the underlying data types to not run into the non-contiguous storage problem in the first place. In particular, that means using matrices with `Eigen::RowMajor` storage, where appropriate, such as:

```
using RowMatrixXd = Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic,␣
↪Eigen::RowMajor>;
// Use RowMatrixXd instead of MatrixXd
```

Now bound functions accepting `Eigen::Ref<RowMatrixXd>` arguments will be callable with numpy's (default) arrays without involving a copying.

You can, alternatively, change the storage order that numpy arrays use by adding the `order='F'` option when creating an array:

```
myarray = np.array(source, order="F")
```

Such an object will be passable to a bound function accepting an `Eigen::Ref<MatrixXd>` (or similar column-major Eigen type).

One major caveat with this approach, however, is that it is not entirely as easy as simply flipping all Eigen or numpy usage from one to the other: some operations may alter the storage order of a numpy array. For example, `a2 = array.transpose()` results in `a2` being a view of `array` that references the same data, but in the opposite storage order!

While this approach allows fully optimized vectorized calculations in Eigen, it cannot be used with array slices, unlike the first approach.

When *returning* a matrix to Python (either a regular matrix, a reference via `Eigen::Ref<>`, or a map/block into a matrix), no special storage consideration is required: the created numpy array will have the required stride that allows numpy to properly interpret the array, whatever its storage order.

11.6.5 Failing rather than copying

The default behaviour when binding `Eigen::Ref<const MatrixType>` Eigen references is to copy matrix values when passed a numpy array that does not conform to the element type of `MatrixType` or does not have a compatible stride layout. If you want to explicitly avoid copying in such a case, you should bind arguments using the `py::arg().noconvert()` annotation (as described in the *Non-converting arguments* documentation).

The following example shows an example of arguments that don't allow data copying to take place:

```
// The method and function to be bound:
class MyClass {
    // ...
    double some_method(const Eigen::Ref<const MatrixXd> &matrix) { /* ... */ }
};
float some_function(const Eigen::Ref<const MatrixXf> &big,
                    const Eigen::Ref<const MatrixXf> &small) {
    // ...
}

// The associated binding code:
using namespace pybind11::literals; // for "arg"_a
py::class_<MyClass>(m, "MyClass")
    // ... other class definitions
    .def("some_method", &MyClass::some_method, py::arg().noconvert());

m.def("some_function", &some_function,
    "big"_a.noconvert(), // <- Don't allow copying for this arg
    "small"_a            // <- This one can be copied if needed
);
```

With the above binding code, attempting to call the `some_method(m)` method on a `MyClass` object, or attempting to call `some_function(m, m2)` will raise a `RuntimeError` rather than making a temporary copy of the array. It will, however, allow the `m2` argument to be copied into a temporary if necessary.

Note that explicitly specifying `.noconvert()` is not required for *mutable* Eigen references (e.g. `Eigen::Ref<MatrixXd>` without `const` on the `MatrixXd`): mutable references will never be called with a temporary copy.

11.6.6 Vectors versus column/row matrices

Eigen and numpy have fundamentally different notions of a vector. In Eigen, a vector is simply a matrix with the number of columns or rows set to 1 at compile time (for a column vector or row vector, respectively). NumPy, in contrast, has comparable 2-dimensional 1xN and Nx1 arrays, but *also* has 1-dimensional arrays of size N.

When passing a 2-dimensional 1xN or Nx1 array to Eigen, the Eigen type must have matching dimensions: That is, you cannot pass a 2-dimensional Nx1 numpy array to an Eigen value expecting a row vector, or a 1xN numpy array as a column vector argument.

On the other hand, pybind11 allows you to pass 1-dimensional arrays of length N as Eigen parameters. If the Eigen type can hold a column vector of length N it will be passed as such a column vector. If not, but the Eigen type constraints will accept a row vector, it will be passed as a row vector. (The column vector takes precedence when both are supported, for example, when passing a 1D numpy array to a MatrixXd argument). Note that the type need not be explicitly a vector: it is permitted to pass a 1D numpy array of size 5 to an Eigen `Matrix<double, Dynamic, 5>`: you would end up with a 1x5 Eigen matrix. Passing the same to an `Eigen::MatrixXd` would result in a 5x1 Eigen matrix.

When returning an Eigen vector to numpy, the conversion is ambiguous: a row vector of length 4 could be returned as either a 1D array of length 4, or as a 2D array of size 1x4. When encountering such a situation, pybind11 compromises by considering the returned Eigen type: if it is a compile-time vector–that is, the type has either the number of rows or columns set to 1 at compile time–pybind11 converts to a 1D numpy array when returning the value. For instances that are a vector only at run-time (e.g. `MatrixXd`, `Matrix<float, Dynamic, 4>`), pybind11 returns the vector as a 2D array to numpy. If this isn't want you want, you can use `array.reshape(...)` to get a view of the same data in the desired dimensions.

See also:

The file `tests/test_eigen.cpp` contains a complete example that shows how to pass Eigen sparse and dense data types in more detail.

11.7 Custom type casters

Some applications may prefer custom type casters that convert between existing Python types and C++ types, similar to the `list` ⟷ `std::vector` and `dict` ⟷ `std::map` conversions which are built into pybind11. Implementing custom type casters is fairly advanced usage. While it is recommended to use the pybind11 API as much as possible, more complex examples may require familiarity with the intricacies of the Python C API. You can refer to the Python/C API Reference Manual for more information.

The following snippets demonstrate how this works for a very simple `Point2D` type. We want this type to be convertible to C++ from Python types implementing the `Sequence` protocol and having two elements of type `float`. When returned from C++ to Python, it should be converted to a Python `tuple[float, float]`. For this type we could provide Python bindings for different arithmetic functions implemented in C++ (here demonstrated by a simple `negate` function).

```
namespace user_space {

struct Point2D {
    double x;
    double y;
};

Point2D negate(const Point2D &point) { return Point2D{-point.x, -point.y}; }

} // namespace user_space
```

The following Python snippet demonstrates the intended usage of `negate` from the Python side:

```
from my_math_module import docs_advanced_cast_custom as m

point1 = [1.0, -1.0]
point2 = m.negate(point1)
assert point2 == (-1.0, 1.0)
```

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

```
namespace pybind11 {
namespace detail {

template <>
```

(continues on next page)

(continued from previous page)

```
struct type_caster<user_space::Point2D> {
    // This macro inserts a lot of boilerplate code and sets the type hint.
    // `io_name` is used to specify different type hints for arguments and return values.
    // The signature of our negate function would then look like:
    // `negate(Sequence[float]) -> tuple[float, float]`
    PYBIND11_TYPE_CASTER(user_space::Point2D, io_name("Sequence[float]", "tuple[float,␣
↪float]"));

    // C++ -> Python: convert `Point2D` to `tuple[float, float]`. The second and third␣
↪arguments
    // are used to indicate the return value policy and parent object (for
    // return_value_policy::reference_internal) and are often ignored by custom casters.
    // The return value should reflect the type hint specified by the second argument of␣
↪`io_name`.
    static handle
    cast(const user_space::Point2D &number, return_value_policy /*policy*/, handle /
↪*parent*/) {
        return py::make_tuple(number.x, number.y).release();
    }

    // Python -> C++: convert a `PyObject` into a `Point2D` and return false upon failure.
↪ The
    // second argument indicates whether implicit conversions should be allowed.
    // The accepted types should reflect the type hint specified by the first argument of
    // `io_name`.
    bool load(handle src, bool /*convert*/) {
        // Check if handle is a Sequence
        if (!py::isinstance<py::sequence>(src)) {
            return false;
        }
        auto seq = py::reinterpret_borrow<py::sequence>(src);
        // Check if exactly two values are in the Sequence
        if (seq.size() != 2) {
            return false;
        }
        // Check if each element is either a float or an int
        for (auto item : seq) {
            if (!py::isinstance<py::float_>(item) && !py::isinstance<py::int_>(item)) {
                return false;
            }
        }
        value.x = seq[0].cast<double>();
        value.y = seq[1].cast<double>();
        return true;
    }
};

} // namespace detail
} // namespace pybind11

// Bind the negate function
PYBIND11_MODULE(docs_advanced_cast_custom, m, py::mod_gil_not_used()) { m.def("negate",␣
```

(continues on next page)

(continued from previous page)

```
↪user_space::negate); }
```

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).

Note: For further information on the `return_value_policy` argument of `cast` refer to *Return value policies*. To learn about the `convert` argument of `load` see *Non-converting arguments*.

Note: If a custom `load()` produces a non-owning view of storage owned by a Python object, and keeping that object alive is sufficient to keep the storage valid, register the owner (often `src`) with `pybind11::detail::loader_life_support::try_add_patient(owner)`. During a bound-function call, this keeps the owner alive until the call returns. With no active call frame, it returns `false` and the caller remains responsible for the lifetime.

If `load()` instead creates a temporary Python object to own the view's storage, use the strict `loader_life_support::add_patient(owner)`. Outside a bound-function call, it raises `cast_error` rather than allowing a dangling view. Neither form permits C++ code to retain the view after the call without separate lifetime management. These `detail` helpers are intended only for type-caster implementations. See *C++17 string views* for a concrete example.

Warning: When using custom type casters, it's important to declare them consistently in every compilation unit of the Python extension module to satisfy the C++ One Definition Rule (ODR). Otherwise, undefined behavior can ensue.

Note: Using the type hint `Sequence[float]` signals to static type checkers, that not only tuples may be passed, but any type implementing the Sequence protocol, e.g., `list[float]`. Unfortunately, that loses the length information `tuple[float, float]` provides. One way of still providing some length information in type hints is using `typing.Annotated`, e.g., `Annotated[Sequence[float], 2]`, or further add libraries like annotated-types.

CHAPTER

TWELVE

PYTHON C++ INTERFACE

pybind11 exposes Python types and functions using thin C++ wrappers, which makes it possible to conveniently call Python code from C++ without resorting to Python's C API.

12.1 Python types

12.1.1 Available wrappers

All major Python types are available as thin C++ wrapper classes. These can also be used as function parameters – see *Python objects as arguments*.

Available types include *handle*, *object*, *bool_*, *int_*, *float_*, *str*, *bytes*, *tuple*, *list*, *dict*, *slice*, *none*, *capsule*, *iterable*, *iterator*, *function*, *buffer*, `array`, and `array_t`.

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

12.1.2 Instantiating compound Python types from C++

Dictionaries can be initialized in the *dict* constructor:

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

A tuple of python objects can be instantiated using `py::make_tuple()`:

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

Each element is converted to a supported Python type.

A simple namespace can be instantiated using

```
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);
```

Attributes on a namespace can be modified with the `py::delattr()`, `py::getattr()`, and `py::setattr()` functions. Simple namespaces can be useful as lightweight stand-ins for class instances.

12.1.3 Casting back and forth

In this kind of mixed code, it is often necessary to convert arbitrary C++ types to Python, which can be done using `py::cast()`:

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

The reverse direction uses the following syntax:

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

When conversion fails, both directions throw the exception `cast_error`.

When casting to a non-owning type such as `std::string_view`, the Python source may need to remain alive after a successful cast. See *C++17 string views* for details.

12.1.4 Accessing Python libraries from C++

It is also possible to import objects defined in the Python standard library or available in the current Python environment (`sys.path`) and work with these in C++.

This example obtains a reference to the Python `Decimal` class.

```
// 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 Calling Python functions

It is also possible to call Python classes, functions and methods via `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 Calling Python methods

To call an object's method, one can again use `.attr` to obtain access to the Python method.

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

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

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

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

12.1.7 Keyword arguments

Keyword arguments are also supported. In Python, there is the usual call syntax:

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


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

In C++, the same call can be made using:

```
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 Unpacking arguments

Unpacking of `*args` and `**kwargs` is also possible and can be mixed with other arguments:

```
// * 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);
```

See also:

The file `tests/test_pytypes.cpp` contains a complete example that demonstrates passing native Python types in more detail. The file `tests/test_callbacks.cpp` presents a few examples of calling Python functions from C++, including keywords arguments and unpacking.

12.1.9 Implicit casting

When using the C++ interface for Python types, or calling Python functions, objects of type *object* are returned. It is possible to invoke implicit conversions to subclasses like *dict*. The same holds for the proxy objects returned by `operator[]` or `obj.attr()`. Casting to subtypes improves code readability and allows values to be passed to C++ functions that require a specific subtype rather than a generic *object*.

```
#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)));
```

These implicit conversions are available for subclasses of *object*; there is no need to call `obj.cast()` explicitly as for custom classes, see *Casting back and forth*.

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 Handling exceptions

Python exceptions from wrapper classes will be thrown as a `py::error_already_set`. See *Handling exceptions from Python in C++* for more information on handling exceptions raised when calling C++ wrapper classes.

12.1.11 Gotchas

Default-Constructed Wrappers

When a wrapper type is default-constructed, it is **not** a valid Python object (i.e. it is not `py::none()`). It is simply the same as `PyObject*` null pointer. To check for this, use `static_cast<bool>(my_wrapper)`.

Assigning py::none() to wrappers

You may be tempted to use types like `py::str` and `py::dict` in C++ signatures (either pure C++, or in bound signatures), and assign them default values of `py::none()`. However, in a best case scenario, it will fail fast because `None` is not convertible to that type (e.g. `py::dict`), or in a worse case scenario, it will silently work but corrupt the types you want to work with (e.g. `py::str(py::none())` will yield `"None"` in Python).

12.2 NumPy

12.2.1 Buffer protocol

Python supports an extremely general and convenient approach for exchanging data between plugin libraries. Types can expose a buffer view[1], which provides fast direct access to the raw internal data representation. Suppose we want to bind the following simplistic Matrix class:

```
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;
};
```

The following binding code exposes the `Matrix` contents as a buffer object, making it possible to cast Matrices into NumPy arrays. It is even possible to completely avoid copy operations with Python expressions like `np.array(matrix_instance, copy = False)`.

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

(continues on next page)

[1] http://docs.python.org/3/c-api/buffer.html

(continued from previous page)

```
↪*/
                sizeof(float) }
        );
    });
```

Supporting the buffer protocol in a new type involves specifying the special `py::buffer_protocol()` tag in the `py::class_` constructor and calling the `def_buffer()` method with a lambda function that creates a `py::buffer_info` description record on demand describing a given matrix instance. The contents of `py::buffer_info` mirror the Python buffer protocol specification.

```
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;
};
```

To create a C++ function that can take a Python buffer object as an argument, simply use the type `py::buffer` as one of its arguments. Buffers can exist in a great variety of configurations, hence some safety checks are usually necessary in the function body. Below, you can see a basic example on how to define a custom constructor for the Eigen double precision matrix (`Eigen::MatrixXd`) type, which supports initialization from compatible buffer objects (e.g. a NumPy matrix).

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

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

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

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

        /* Some basic validation checks ... */
        if (!info.item_type_is_equivalent_to<Scalar>())
            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);
```

(continues on next page)

(continued from previous page)

```
        return Matrix(map);
    }));
```

For reference, the `def_buffer()` call for this Eigen data type should look as follows:

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

For a much easier approach of binding Eigen types (although with some limitations), refer to the section on *Eigen*.

See also:

The file `tests/test_buffers.cpp` contains a complete example that demonstrates using the buffer protocol with pybind11 in more detail.

12.2.2 Arrays

By exchanging `py::buffer` with `py::array` in the above snippet, we can restrict the function so that it only accepts NumPy arrays (rather than any type of Python object satisfying the buffer protocol).

In many situations, we want to define a function which only accepts a NumPy array of a certain data type. This is possible via the `py::array_t<T>` template. For instance, the following function requires the argument to be a NumPy array containing double precision values.

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

When it is invoked with a different type (e.g. an integer or a list of integers), the binding code will attempt to cast the input into a NumPy array of the requested type. This feature requires the `pybind11/numpy.h` header to be included. Note that `pybind11/numpy.h` does not depend on the NumPy headers, and thus can be used without declaring a build-time dependency on NumPy; NumPy>=1.7.0 is a runtime dependency.

Data in NumPy arrays is not guaranteed to packed in a dense manner; furthermore, entries can be separated by arbitrary column and row strides. Sometimes, it can be useful to require a function to only accept dense arrays using either the C (row-major) or Fortran (column-major) ordering. This can be accomplished via a second template argument with values `py::array::c_style` or `py::array::f_style`.

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

The `py::array::forcecast` argument is the default value of the second template parameter, and it ensures that non-conforming arguments are converted into an array satisfying the specified requirements instead of trying the next function overload.

There are several methods on arrays; the methods listed below under references work, as well as the following functions based on the NumPy API:

- `.dtype()` returns the type of the contained values.
- `.strides()` returns a pointer to the strides of the array (optionally pass an integer axis to get a number).
- `.flags()` returns the flag settings. `.writable()` and `.owndata()` are directly available.
- `.offset_at()` returns the offset (optionally pass indices).
- `.squeeze()` returns a view with length-1 axes removed.
- `.view(dtype)` returns a view of the array with a different dtype.
- `.reshape({i, j, ...})` returns a view of the array with a different shape. `.resize({...})` is also available.
- `.index_at(i, j, ...)` gets the count from the beginning to a given index.

There are also several methods for getting references (described below).

12.2.3 Structured types

In order for `py::array_t` to work with structured (record) types, we first need to register the memory layout of the type. This can be done via `PYBIND11_NUMPY_DTYPE` macro, called in the plugin definition code, which expects the type followed by field names:

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

struct B {
    int z;
    A a;
};

// ...
PYBIND11_MODULE(test, m, py::mod_gil_not_used()) {
    // ...

    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 */
}
```

The structure should consist of fundamental arithmetic types, `std::complex`, previously registered substructures, and arrays of any of the above. Both C++ arrays and `std::array` are supported. While there is a static assertion to prevent many types of unsupported structures, it is still the user's responsibility to use only "plain" structures that can be safely manipulated as raw memory without violating invariants.

Types whose spelling contains a comma must be wrapped in `PYBIND11_TYPE`: `PYBIND11_NUMPY_DTYPE(PYBIND11_TYPE(C<int, double>), x, y)`. See *General notes regarding convenience macros*.

12.2.4 Scalar types

In some cases we may want to accept or return NumPy scalar values such as `np.float32` or `np.float64`. We hope to be able to handle single-precision and double-precision on the C-side. However, both are bound to Python's double-precision builtin float by default, so they cannot be processed separately. We used the `py::buffer` trick to implement the previous approach, which will cause the readability of the code to drop significantly.

Luckily, there's a helper type for this occasion - `py::numpy_scalar`:

```
m.def("add", [](py::numpy_scalar<float> a, py::numpy_scalar<float> b) {
    return py::make_scalar(a + b);
});
m.def("add", [](py::numpy_scalar<double> a, py::numpy_scalar<double> b) {
    return py::make_scalar(a + b);
});
```

This type is trivially convertible to and from the type it wraps; currently supported scalar types are NumPy arithmetic types: `bool_`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `float32`, `float64`, `complex64`, `complex128`, all of them mapping to respective C++ counterparts.

Note: `py::numpy_scalar<T>` strictly matches NumPy scalar types. For example, `py::numpy_scalar<int64_t>` will accept `np.int64(123)`, but **not** a regular Python `int` like `123`.

Note: Native C types are mapped to NumPy types in a platform specific way: for instance, `char` may be mapped to either `np.int8` or `np.uint8` and `long` may use 4 or 8 bytes depending on the platform. Unless you clearly understand the difference and your needs, please use `<cstdint>`.

12.2.5 Vectorizing functions

Suppose we want to bind a function with the following signature to Python so that it can process arbitrary NumPy array arguments (vectors, matrices, general N-D arrays) in addition to its normal arguments:

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

After including the `pybind11/numpy.h` header, this is extremely simple:

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

Invoking the function like below causes 4 calls to be made to `my_func` with each of the array elements. The significant advantage of this compared to solutions like `numpy.vectorize()` is that the loop over the elements runs entirely on the C++ side and can be crunched down into a tight, optimized loop by the compiler. The result is returned as a NumPy array of type `numpy.dtype.float64`.

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

The scalar argument `z` is transparently replicated 4 times. The input arrays `x` and `y` are automatically converted into the right types (they are of type `numpy.dtype.int64` but need to be `numpy.dtype.int32` and `numpy.dtype.float32`, respectively).

Note: Only arithmetic, complex, and POD types passed by value or by `const &` reference are vectorized; all other arguments are passed through as-is. Functions taking rvalue reference arguments cannot be vectorized.

In cases where the computation is too complicated to be reduced to `vectorize`, it will be necessary to create and access the buffer contents manually. The following snippet contains a complete example that shows how this works (the code is somewhat contrived, since it could have been done more simply using `vectorize`).

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

namespace py = pybind11;

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

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

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

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

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

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

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

    return result;
}

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

See also:

The file `tests/test_numpy_vectorize.cpp` contains a complete example that demonstrates using `vectorize()` in more detail.

12.2.6 Direct access

For performance reasons, particularly when dealing with very large arrays, it is often desirable to directly access array elements without internal checking of dimensions and bounds on every access when indices are known to be already valid. To avoid such checks, the `array` class and `array_t<T>` template class offer an unchecked proxy object that can be used for this unchecked access through the `unchecked<N>` and `mutable_unchecked<N>` methods, where `N` gives the required dimensionality of the array:

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

To obtain the proxy from an `array` object, you must specify both the data type and number of dimensions as template arguments, such as `auto r = myarray.mutable_unchecked<float, 2>()`.

If the number of dimensions is not known at compile time, you can omit the dimensions template parameter (i.e. calling `arr_t.unchecked()` or `arr.unchecked<T>()`. This will give you a proxy object that works in the same way, but results in less optimizable code and thus a small efficiency loss in tight loops.

Note that the returned proxy object directly references the array's data, and only reads its shape, strides, and writeable flag when constructed. You must take care to ensure that the referenced array is not destroyed or reshaped for the duration of the returned object, typically by limiting the scope of the returned instance.

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

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

See also:

The file `tests/test_numpy_array.cpp` contains additional examples demonstrating the use of this feature.

12.2.7 Ellipsis

Python 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.

```
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)];
```

12.2.8 Memory view

For a case when we simply want to provide a direct accessor to C/C++ buffer without a concrete class object, we can return a `memoryview` object. Suppose we wish to expose a `memoryview` for 2x4 uint8_t array, we can do the following:

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

This approach is meant for providing a `memoryview` for a C/C++ buffer not managed by Python. The user is responsible for managing the lifetime of the buffer. Using a `memoryview` created in this way after deleting the buffer in C++ side results in undefined behavior.

We can also use `memoryview::from_memory` for a simple 1D contiguous buffer:

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

Changed in version 2.6: `memoryview::from_memory` added.

12.3 Utilities

12.3.1 Using Python's print function in C++

The usual way to write output in C++ is using `std::cout` while in Python one would use `print`. Since these methods use different buffers, mixing them can lead to output order issues. To resolve this, pybind11 modules can use the `py::print()` function, which writes through Python's printing machinery.

`py::print()` delegates each call to the `print` entry in the current execution frame's built-ins (normally `builtins.print`). When no Python frame is executing, it uses the active interpreter's built-ins. With the standard built-in, optional keyword arguments `sep`, `end`, `file`, and `flush` work as they do in Python:

```
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 <-
```

With the standard built-in, omitting `file` or passing `"file"_a = py::none()` uses the current `sys.stdout`. Other output and error behavior is likewise supplied by the active Python runtime.

12.3.2 Capturing standard output from ostream

Often, a library will use the streams `std::cout` and `std::cerr` to print, but this does not play well with Python's standard `sys.stdout` and `sys.stderr` redirection. Replacing a library's printing with `py::print` <print> may not be feasible. This can be fixed using a guard around the library function that redirects output to the corresponding Python streams:

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

...

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

Warning: The implementation in `pybind11/iostream.h` is NOT thread safe. Multiple threads writing to a redirected ostream concurrently cause data races and potentially buffer overflows. Therefore it is currently a requirement that all (possibly) concurrent redirected ostream writes are protected by a mutex. #HelpAppreciated: Work on iostream.h thread safety. For more background see the discussions under PR #2982 and PR #2995.

This method respects flushes on the output streams and will flush if needed when the scoped guard is destroyed. This allows the output to be redirected in real time, such as to a Jupyter notebook. The two arguments, the C++ stream and the Python output, are optional, and default to standard output if not given. An extra type, `py::scoped_estream_redirect` <scoped_estream_redirect>, is identical except for defaulting to `std::cerr`

and `sys.stderr`; this can be useful with `py::call_guard`, which allows multiple items, but uses the default constructor:

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

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

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

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

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

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

Note: The above methods will not redirect C-level output to file descriptors, such as `fprintf`. For those cases, you'll need to redirect the file descriptors either directly in C or with Python's `os.dup2` function in an operating-system dependent way.

12.3.3 Evaluating Python expressions from strings and files

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
);
```

Note: `eval` and `eval_file` accept a template parameter that describes how the string/file should be interpreted. Possible choices include `eval_expr` (isolated expression), `eval_single_statement` (a single statement, return value is always `none`), and `eval_statements` (sequence of statements, return value is always `none`). `eval` defaults to `eval_expr`, `eval_file` defaults to `eval_statements` and `exec` is just a shortcut for `eval<eval_statements>`.

CHAPTER

THIRTEEN

EMBEDDING THE INTERPRETER

While pybind11 is mainly focused on extending Python using C++, it's also possible to do the reverse: embed the Python interpreter into a C++ program. All of the other documentation pages still apply here, so refer to them for general pybind11 usage. This section will cover a few extra things required for embedding.

13.1 Getting started

A basic executable with an embedded interpreter can be created with just a few lines of CMake and the `pybind11::embed` target, as shown below. For more information, see *Build systems*.

```
cmake_minimum_required(VERSION 3.15...4.2)
project(example)

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

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

The essential structure of the `main.cpp` file looks like this:

```
#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
}
```

The interpreter must be initialized before using any Python API, which includes all the functions and classes in pybind11. The RAII guard class `scoped_interpreter` takes care of the interpreter lifetime. After the guard is destroyed, the interpreter shuts down and clears its memory. No Python functions can be called after this.

13.2 Executing Python code

There are a few different ways to run Python code. One option is to use `eval`, `exec` or `eval_file`, as explained in *Evaluating Python expressions from strings and files*. Here is a quick example in the context of an executable with an embedded interpreter:

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

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

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

Alternatively, similar results can be achieved using pybind11's API (see *Python C++ interface* for more details).

```
#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);
}
```

The two approaches can also be combined:

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

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

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

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

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

13.3 Importing modules

Python modules can be imported using `module_::import()`:

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

For convenience, the current working directory is included in `sys.path` when embedding the interpreter. This makes it easy to import local Python files:

```
"""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);
```

Modules can be reloaded using `module_::reload()` if the source is modified e.g. by an external process. This can be useful in scenarios where the application imports a user defined data processing script which needs to be updated after changes by the user. Note that this function does not reload modules recursively.

13.4 Adding embedded modules

Embedded binary modules can be added using the `PYBIND11_EMBEDDED_MODULE` macro. Note that the definition must be placed at global scope. They can be imported like any other module.

```
#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:

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

a = cpp_module.a
b = a + 1
```

```cpp
#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");
}
```

`PYBIND11_EMBEDDED_MODULE` also accepts `py::mod_gil_not_used()`, `py::multiple_interpreters::per_interpreter_gil(`
and `py::multiple_interpreters::shared_gil()` tags just like `PYBIND11_MODULE`. See *Sub-interpreter support*
and *Free-threading support* for more information.

13.5 Interpreter lifetime

The Python interpreter shuts down when `scoped_interpreter` is destroyed. After this, creating a new instance will restart the interpreter. Alternatively, the `initialize_interpreter` / `finalize_interpreter` pair of functions can be used to directly set the state at any time.

Modules created with pybind11 can be safely re-initialized after the interpreter has been restarted. However, this may not apply to third-party extension modules. The issue is that Python itself cannot completely unload extension modules and there are several caveats with regard to interpreter restarting. In short, not all memory may be freed, either due to Python reference cycles or user-created global data. All the details can be found in the CPython documentation.

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. Use `scoped_subinterpreter` to create a sub-interpreter. See *Embedding Sub-interpreters* for important details on sub-interpreters.

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.

13.6 Embedding Sub-interpreters

A sub-interpreter is a separate interpreter instance which provides a separate, isolated interpreter environment within the same process as the main interpreter. Sub-interpreters are created and managed with a separate API from the main interpreter. Beginning in Python 3.12, sub-interpreters each have their own Global Interpreter Lock (GIL), which means that running a sub-interpreter in a separate thread from the main interpreter can achieve true concurrency.

pybind11's sub-interpreter API can be found in `pybind11/subinterpreter.h`.

pybind11 `subinterpreter` instances can be safely moved and shared between threads as needed. However, managing multiple threads and the lifetimes of multiple interpreters and their GILs can be challenging. Proceed with caution (and lots of testing)!

The main interpreter must be initialized before creating a sub-interpreter, and the main interpreter must outlive all sub-interpreters. Sub-interpreters are managed through a different API than the main interpreter.

The `subinterpreter` class manages the lifetime of sub-interpreters. Instances are movable, but not copyable. Default constructing this class does *not* create a sub-interpreter (it creates an empty holder). To create a sub-interpreter, call `subinterpreter::create()`.

Warning: Sub-interpreter creation acquires (and subsequently releases) the main interpreter GIL. If another thread holds the main GIL, the function will block until the main GIL can be acquired.

Sub-interpreter destruction temporarily activates the sub-interpreter. The sub-interpreter must not be active (on any threads) at the time the `subinterpreter` destructor is called.

Both actions will re-acquire any interpreter's GIL that was held prior to the call before returning (or return to no active interpreter if none was active at the time of the call).

Each sub-interpreter will import a separate copy of each `PYBIND11_EMBEDDED_MODULE` when those modules specify a `multiple_interpreters` tag. If a module does not specify a `multiple_interpreters` tag, then Python will report an `ImportError` if it is imported in a sub-interpreter.

pybind11 also has a `scoped_subinterpreter` class, which creates and activates a sub-interpreter when it is constructed, and deactivates and deletes it when it goes out of scope.

13.6.1 Activating a Sub-interpreter

Once a sub-interpreter is created, you can "activate" it on a thread (and acquire its GIL) by creating a `subinterpreter_scoped_activate` instance and passing it the sub-interpreter to be activated. The function will acquire the sub-interpreter's GIL and make the sub-interpreter the current active interpreter on the current thread for the lifetime of the instance. When the `subinterpreter_scoped_activate` instance goes out of scope, the sub-interpreter GIL is released and the prior interpreter that was active on the thread (if any) is reactivated and it's GIL is re-acquired.

When using `subinterpreter_scoped_activate`:

1. If the thread holds any interpreter's GIL: - That GIL is released
2. The new sub-interpreter's GIL is acquired

3. The new sub-interpreter is made active.
4. **When the scope ends:**
    - The sub-interpreter's GIL is released
    - **If there was a previous interpreter:**
        - The old interpreter's GIL is re-acquired
        - The old interpreter is made active
    - Otherwise, no interpreter is currently active and no GIL is held.

Example:

```
py::initialize_interpreter();
// Main GIL is held
{
    py::subinterpreter sub = py::subinterpreter::create();
    // Main interpreter is still active, main GIL re-acquired
    {
        py::subinterpreter_scoped_activate guard(sub);
        // Sub-interpreter active, thread holds sub's GIL
        {
            py::subinterpreter_scoped_activate main_guard(py);
            // Sub's GIL was automatically released
            // Main interpreter active, thread holds main's GIL
        }
        // Back to sub-interpreter, thread holds sub's GIL again
    }
    // Main interpreter is active, main's GIL is held
}
```

13.6.2 Reusing a thread state across activations

By default, `subinterpreter_scoped_activate` creates a fresh `PyThreadState` on entry and destroys it on exit. A thread that repeatedly re-enters the same sub-interpreter therefore allocates and frees a thread state every time, and does **not** preserve any per-thread interpreter state between activations.

For workloads where a single OS thread re-enters one or more sub-interpreters many times, pybind11 provides `subinterpreter_thread_state` — an RAII object that owns a `PyThreadState` and lets you swap it in for the duration of each `subinterpreter_scoped_activate` scope without destroying it between activations:

```
// Create the PyThreadState once. It is created in a "released" state:
// not current, no GIL acquired.
thread_local py::subinterpreter_thread_state ts(sub);

{
    // Swap it in; the subinterpreter's GIL is acquired.
    py::subinterpreter_scoped_activate guard(ts);
    // ... use the sub-interpreter ...
}
// Swap-out only; the PyThreadState is kept alive in `ts`.

{
```

(continues on next page)

(continued from previous page)

```
    py::subinterpreter_scoped_activate guard(ts);
    // The same PyThreadState is reused; its per-thread interpreter state
    // is preserved across activations.
}
```

This composes naturally with multiple sub-interpreters on the same OS thread: hold one `subinterpreter_thread_state` per sub-interpreter and alternate between them. Each `PyThreadState` is independent and is preserved across activations.

```
thread_local py::subinterpreter_thread_state ts_a(sub_a);
thread_local py::subinterpreter_thread_state ts_b(sub_b);

{ py::subinterpreter_scoped_activate guard(ts_a); /* in sub_a */ }
{ py::subinterpreter_scoped_activate guard(ts_b); /* in sub_b */ }
{ py::subinterpreter_scoped_activate guard(ts_a); /* same PyThreadState as before */ }
```

The default behavior is unchanged: the `subinterpreter_scoped_activate(subinterpreter const&)` overload still creates and destroys a transient `PyThreadState` per scope, and it never shares a thread state with any `subinterpreter_thread_state` that may also exist for the same sub-interpreter on the same thread.

Warning: Lifetime and threading requirements for `subinterpreter_thread_state`:

- It must be constructed and destroyed on the **same OS thread**. A `PyThreadState` is bound to its creating thread; deleting it on another thread is undefined behavior. Holding the object as a `thread_local` satisfies this automatically.
- It must only be activated (with `subinterpreter_scoped_activate`) on the **same OS thread** that constructed it. Activating it on a different thread is illegal.
- It must be destroyed while its sub-interpreter is still alive.
- It must **not** be destroyed while a `subinterpreter_scoped_activate` referring to it is alive — the activator holds a reference into it.

13.6.3 GIL API for sub-interpreters

`gil_scoped_release` and `gil_scoped_acquire` can be used to manage the GIL of a sub-interpreter just as they do for the main interpreter. They both manage the GIL of the currently active interpreter, without the programmer having to do anything special or different. There is one important caveat:

Note: When no interpreter is active through a `subinterpreter_scoped_activate` instance (such as on a new thread), `gil_scoped_acquire` will acquire the **main** GIL and activate the **main** interpreter.

13.6.4 Full Sub-interpreter example

Here is an example showing how to create and activate sub-interpreters:

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

namespace py = pybind11;

PYBIND11_EMBEDDED_MODULE(printer, m, py::multiple_interpreters::per_interpreter_gil()) {
    m.def("which", [](const std::string& when) {
        std::cout << when << "; Current Interpreter is "
                << py::subinterpreter::current().id()
                << std::endl;
    });
}

int main() {
    py::scoped_interpreter main_interp;

    py::module_::import("printer").attr("which")("First init");

    {
        py::subinterpreter sub = py::subinterpreter::create();

        py::module_::import("printer").attr("which")("Created sub");

        {
            py::subinterpreter_scoped_activate guard(sub);
            try {
                py::module_::import("printer").attr("which")("Activated sub");
            }
            catch (py::error_already_set &e) {
                std::cerr << "EXCEPTION " << e.what() << std::endl;
                return 1;
            }
        }

        py::module_::import("printer").attr("which")("Deactivated sub");

        {
            py::gil_scoped_release nogil;
            {
                py::subinterpreter_scoped_activate guard(sub);
                try {
                    {
                        py::subinterpreter_scoped_activate main_
↪guard(py::subinterpreter::main());
                        try {
                            py::module_::import("printer").attr("which")("Main within sub
↪");
                        }
```

(continues on next page)

(continued from previous page)

```
                        catch (py::error_already_set &e) {
                            std::cerr << "EXCEPTION " << e.what() << std::endl;
                            return 1;
                        }
                    }
                    py::module_::import("printer").attr("which")("After Main, still␣
↪within sub");
                }
                catch (py::error_already_set &e) {
                    std::cerr << "EXCEPTION " << e.what() << std::endl;
                    return 1;
                }
            }
        }
    }

    py::module_::import("printer").attr("which")("At end");

    return 0;
}
```

Expected output:

```
First init; Current Interpreter is 0
Created sub; Current Interpreter is 0
Activated sub; Current Interpreter is 1
Deactivated sub; Current Interpreter is 0
Main within sub; Current Interpreter is 0
After Main, still within sub; Current Interpreter is 1
At end; Current Interpreter is 0
```

Warning: In Python 3.12 sub-interpreters must be destroyed in the same OS thread that created them. Failure to follow this rule may result in deadlocks or crashes when destroying the sub-interpreter on the wrong thread.

This constraint is not present in Python 3.13+.

13.6.5 Best Practices for sub-interpreter safety

- Never share Python objects across different interpreters.
- *error_already_set* objects contain a reference to the Python exception type, and *error_already_set::what()* acquires the GIL. So Python exceptions must **never** be allowed to propagate past the enclosing `subinterpreter_scoped_activate` instance! (So your try/catch should be *just inside* the scope covered by the `subinterpreter_scoped_activate`.)
- Avoid global/static state whenever possible. Instead, keep state within each interpreter, such as within the interpreter state dict, which can be accessed via `subinterpreter::current().state_dict()`, or within instance members and tied to Python objects.
- Avoid trying to "cache" Python objects in C++ variables across function calls (this is an easy way to accidentally introduce sub-interpreter bugs). In the code example above, note that we did not save the result of

module_::import(), in order to avoid accidentally using the resulting Python object when the wrong interpreter was active.

- Avoid moving or disarming RAII objects managing GIL and sub-interpreter lifetimes. Doing so can lead to confusion about lifetimes. (For example, accidentally extending a `subinterpreter_scoped_activate` past the lifetime of it's `subinterpreter`.)
- While sub-interpreters each have their own GIL, there can now be multiple independent GILs in one program so you need to consider the possibility of deadlocks caused by multiple GILs and/or the interactions of the GIL(s) and your C++ code's own locking.
- When using multiple threads to run independent sub-interpreters, the independent GILs allow concurrent calls from different interpreters into the same C++ code from different threads. So you must still consider the thread safety of your C++ code. Remember, in Python 3.12 sub-interpreters must be destroyed on the same thread that they were created on.
- When using sub-interpreters in free-threaded python builds, note that creating and destroying sub-interpreters may initiate a "stop-the-world". Be sure to detach long-running C++ threads from Python thread state (similar to releasing the GIL) to avoid deadlocks.
- Familiarize yourself with *Concurrency and Parallelism in Python with pybind11*.

CHAPTER

FOURTEEN

MISCELLANEOUS

14.1 General notes regarding convenience macros

pybind11 provides a few convenience macros such as `PYBIND11_DECLARE_HOLDER_TYPE()` and `PYBIND11_OVERRIDE_*`. Since these are "just" macros that are evaluated in the preprocessor (which has no concept of types), they *will* get confused by commas in a template argument; for example, consider:

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

The limitation of the C preprocessor interprets this as five arguments (with new arguments beginning after each comma) rather than three. To get around this, there are two alternatives: you can use a type alias, or you can wrap the type using the `PYBIND11_TYPE` macro:

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

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

The `PYBIND11_MAKE_OPAQUE` macro does *not* require the above workarounds.

14.2 Global Interpreter Lock (GIL)

The Python C API dictates that the Global Interpreter Lock (GIL) must always be held by the current thread to safely access Python objects. As a result, when Python calls into C++ via pybind11 the GIL must be held, and pybind11 will never implicitly release the GIL.

```
void my_function() {
    /* GIL is held when this function is called from Python */
}

PYBIND11_MODULE(example, m) {
    m.def("my_function", &my_function);
}
```

pybind11 will ensure that the GIL is held when it knows that it is calling Python code. For example, if a Python callback is passed to C++ code via `std::function`, when C++ code calls the function the built-in wrapper will acquire the

GIL before calling the Python callback. Similarly, the `PYBIND11_OVERRIDE` family of macros will acquire the GIL before calling back into Python.

When writing C++ code that is called from other C++ code, if that code accesses Python state, it must explicitly acquire and release the GIL. A separate document on deadlocks[1] elaborates on a particularly subtle interaction with C++'s block-scope static variable initializer guard mutexes.

The classes `gil_scoped_release` and `gil_scoped_acquire` can be used to acquire and release the global interpreter lock in the body of a C++ function call. In this way, long-running C++ code can be parallelized using multiple Python threads, **but great care must be taken** when any `gil_scoped_release` appear: if there is any way that the C++ code can access Python objects, `gil_scoped_acquire` should be used to reacquire the GIL. Taking *Overriding virtual functions in Python* as an example, this could be realized as follows (important changes highlighted):

```
class PyAnimal : public Animal, public py::trampoline_self_life_support {
public:
    /* Inherit the constructors */
    using Animal::Animal;

    /* Trampoline (need one for each virtual function) */
    std::string go(int n_times) {
        /* PYBIND11_OVERRIDE_PURE will acquire the GIL before accessing Python state */
        PYBIND11_OVERRIDE_PURE(
            std::string, /* Return type */
            Animal,      /* Parent class */
            go,          /* Name of function */
            n_times      /* Argument(s) */
        );
    }
};

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

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

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

The `call_go` wrapper can also be simplified using the `call_guard` policy (see *Additional call policies*) which yields the same result:

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

[1] See docs/advanced/deadlock.md

14.3 Common Sources Of Global Interpreter Lock Errors

Failing to properly hold the Global Interpreter Lock (GIL) is one of the more common sources of bugs within code that uses pybind11. If you are running into GIL related errors, we highly recommend you consult the following checklist.

- Do you have any global variables that are pybind11 objects or invoke pybind11 functions in either their constructor or destructor? You are generally not allowed to invoke any Python function in a global static context. We recommend using lazy initialization and then intentionally leaking at the end of the program.
- Do you have any pybind11 objects that are members of other C++ structures? One commonly overlooked requirement is that pybind11 objects have to increase their reference count whenever their copy constructor is called. Thus, you need to be holding the GIL to invoke the copy constructor of any C++ class that has a pybind11 member. This can sometimes be very tricky to track for complicated programs Think carefully when you make a pybind11 object a member in another struct.
- C++ destructors that invoke Python functions can be particularly troublesome as destructors can sometimes get invoked in weird and unexpected circumstances as a result of exceptions.
- C++ static block-scope variable initialization that calls back into Python can cause deadlocks; see[Page 178, 1] for a detailed discussion.
- You should try running your code in a debug build. That will enable additional assertions within pybind11 that will throw exceptions on certain GIL handling errors (reference counting operations).

14.4 Free-threading support

pybind11 supports the experimental free-threaded builds of Python versions 3.13+. pybind11's internal data structures are thread safe. To enable your modules to be used with free-threading, pass the `mod_gil_not_used` tag as the third argument to `PYBIND11_MODULE`.

For example:

```
PYBIND11_MODULE(example, m, py::mod_gil_not_used()) {
    py::class_<Animal> animal(m, "Animal");
    // etc
}
```

Importantly, enabling your module to be used with free-threading is also your promise that your code is thread safe. Modules must still be built against the Python free-threading branch to enable free-threading, even if they specify this tag. Adding this tag does not break compatibility with non-free-threaded Python.

14.5 Sub-interpreter support

pybind11 supports isolated sub-interpreters, which are stable in Python 3.12+. pybind11's internal data structures are sub-interpreter safe. To enable your modules to be imported in isolated sub-interpreters, pass the `multiple_interpreters::per_interpreter_gil()` tag as the third or later argument to `PYBIND11_MODULE`.

For example:

```
PYBIND11_MODULE(example, m, py::multiple_interpreters::per_interpreter_gil()) {
    py::class_<Animal> animal(m, "Animal");
    // etc
}
```

Best Practices for Sub-interpreter Safety:

- Your initialization function will run for each interpreter that imports your module.
- Never share Python objects across different sub-interpreters.
- Avoid global/static state whenever possible. Instead, keep state within each interpreter, such as in instance members tied to Python objects, *globals()*, and the interpreter state dict.
- Modules without any global/static state in their C++ code may already be sub-interpreter safe without any additional work!
- Avoid trying to "cache" Python objects in C++ variables across function calls (this is an easy way to accidentally introduce sub-interpreter bugs).
- While sub-interpreters each have their own GIL, there can now be multiple independent GILs in one program, so concurrent calls into a module from two different sub-interpreters are still possible. Therefore, your module still needs to consider thread safety.

pybind11 also supports "legacy" sub-interpreters which shared a single global GIL. You can enable legacy-only behavior by using the `multiple_interpreters::shared_gil()` tag in `PYBIND11_MODULE`.

You can explicitly disable sub-interpreter support in your module by using the `multiple_interpreters::not_supported()` tag. This is the default behavior if you do not specify a multiple_interpreters tag.

14.6 Concurrency and Parallelism in Python with pybind11

Sub-interpreter support does not imply free-threading support or vice versa. Free-threading safe modules can still have global/static state (as long as access to them is thread-safe), but sub-interpreter safe modules cannot. Likewise, sub-interpreter safe modules can still rely on the GIL, but free-threading safe modules cannot.

Here is a simple example module which has a function that calculates a value and returns the result of the previous calculation.

```
PYBIND11_MODULE(example, m) {
    static size_t seed = 0;
    m.def("calc_next", []() {
        auto old = seed;
        seed = (seed + 1) * 10;
        return old;
    });
```

This module is not free-threading safe because there is no synchronization on the number variable. It is relatively easy to make this free-threading safe. One way is by using atomics, like this:

```
PYBIND11_MODULE(example, m, py::mod_gil_not_used()) {
    static std::atomic<size_t> seed(0);
    m.def("calc_next", []() {
        size_t old, next;
        do {
            old = seed.load();
            next = (old + 1) * 10;
        } while (!seed.compare_exchange_weak(old, next));
        return old;
    });
}
```

The atomic variable and the compare-exchange guarantee a consistent behavior from this function even when called currently from multiple threads at the same time.

However, the global/static integer is not sub-interpreter safe, because the calls in one sub-interpreter will change what is seen in another. To fix it, the state needs to be specific to each interpreter. One way to do that is by storing the state on another Python object, such as a member of a class. For this simple example, we will store it in *globals()*.

```
PYBIND11_MODULE(example, m, py::multiple_interpreters::per_interpreter_gil()) {
    m.def("calc_next", []() {
        if (!py::globals().contains("myseed"))
            py::globals()["myseed"] = 0;
        size_t old = py::globals()["myseed"];
        py::globals()["myseed"] = (old + 1) * 10;
        return old;
    });
}
```

This module is sub-interpreter safe, for both `shared_gil` ("legacy") and `per_interpreter_gil` ("default") varieties. Multiple sub-interpreters could each call this same function concurrently from different threads. This is safe because each sub-interpreter's GIL protects it's own Python objects from concurrent access.

However, the module is no longer free-threading safe, for the same reason as before, because the calculation is not synchronized. We can synchronize it using a Python critical section. This will do nothing if not in free-threaded Python. You can have it lock one or two Python objects. You cannot nest it.

Warning: When using a `py::scoped_critical_section`, make sure it is not nested and that no other synchronization primitives (such as a `std::mutex`) are held, which could lead to deadlocks. In 3.13, taking the same lock causes it to release then reacquire, which means you can't use it to, for example, read and write to a dictionary, because the dictionary uses a critical section internally in CPython. Use a `std::mutex` instead if you need this on Python 3.13. In 3.14, taking a lock on a locked object no longer releases and relocks as an optimization, which also fixes this case.

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

PYBIND11_MODULE(example, m, py::multiple_interpreters::per_interpreter_gil(), py::mod_
↪gil_not_used()) {
    m.def("calc_next", []() {
        size_t old;
        py::dict g = py::globals();
        py::scoped_critical_section guard(g);
        if (!g.contains("myseed"))
            g["myseed"] = 0;
        old = g["myseed"];
        g["myseed"] = (old + 1) * 10;
        return old;
    });
}
```

The module is now both sub-interpreter safe and free-threading safe.

14.7 Binding sequence data types, iterators, the slicing protocol, etc.

Please refer to the supplemental example for details.

See also:

The file `tests/test_sequences_and_iterators.cpp` contains a complete example that shows how to bind a sequence data type, including length queries (`__len__`), iterators (`__iter__`), the slicing protocol and other kinds of useful operations.

14.8 Partitioning code over multiple extension modules

It's straightforward to split binding code over multiple extension modules, while referencing types that are declared elsewhere. Everything "just" works without any special precautions. One exception to this rule occurs when extending a type declared in another extension module. Recall the basic example from Section *Inheritance and automatic downcasting*.

```
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);
```

Suppose now that `Pet` bindings are defined in a module named `basic`, whereas the `Dog` bindings are defined somewhere else. The challenge is of course that the variable `pet` is not available anymore though it is needed to indicate the inheritance relationship to the constructor of `py::class_<Dog>`. However, it can be acquired as follows:

```
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);
```

Alternatively, you can specify the base class as a template parameter option to `py::class_`, which performs an automated lookup of the corresponding Python type. Like the above code, however, this also requires invoking the `import` function once to ensure that the pybind11 binding code of the module `basic` has been executed:

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

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

Naturally, both methods will fail when there are cyclic dependencies.

Note that pybind11 code compiled with hidden-by-default symbol visibility (e.g. via the command line flag `-fvisibility=hidden` on GCC/Clang), which is required for proper pybind11 functionality, can interfere with the ability to access types defined in another extension module. Working around this requires manually exporting types that are accessed by multiple extension modules; pybind11 provides a macro to do just this:

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

Note also that it is possible (although would rarely be required) to share arbitrary C++ objects between extension modules at runtime. Internal library data is shared between modules using capsule machinery[2] which can be also utilized for storing, modifying and accessing user-defined data. Note that an extension module will "see" other extensions' data if and only if they were built with the same pybind11 version. Consider the following example:

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

If the above snippet was used in several separately compiled extension modules, the first one to be imported would create a `MyData` instance and associate a `"mydata"` key with a pointer to it. Extensions that are imported later would be then able to access the data behind the same pointer.

14.9 Module Destructors

pybind11 does not provide an explicit mechanism to invoke cleanup code at module destruction time. In rare cases where such functionality is required, it is possible to emulate it using Python capsules or weak references with a destruction callback.

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

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

This approach has the potential downside that instances of classes exposed within the module may still be alive when the cleanup callback is invoked (whether this is acceptable will generally depend on the application).

Alternatively, the capsule may also be stashed within a type object, which ensures that it not called before all instances of that type have been collected:

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

Both approaches also expose a potentially dangerous `_cleanup` attribute in Python, which may be undesirable from an API standpoint (a premature explicit call from Python might lead to undefined behavior). Yet another approach that avoids this issue involves weak reference with a cleanup callback:

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

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

(continues on next page)

[2] https://docs.python.org/3/extending/extending.html#using-capsules

(continued from previous page)

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

Note: PyPy does not garbage collect objects when the interpreter exits. An alternative approach (which also works on CPython) is to use the `atexit` module[3], for example:

```
auto atexit = py::module_::import("atexit");
atexit.attr("register")(py::cpp_function([]() {
    // perform cleanup here -- this function is called with the GIL held
}));
```

14.10 Generating documentation using Sphinx

Sphinx[4] has the ability to inspect the signatures and documentation strings in pybind11-based extension modules to automatically generate beautiful documentation in a variety formats. The python_example repository[5] contains a simple example repository which uses this approach.

There are two potential gotchas when using this approach: first, make sure that the resulting strings do not contain any `TAB` characters, which break the docstring parsing routines. You may want to use C++11 raw string literals, which are convenient for multi-line comments. Conveniently, any excess indentation will be automatically be removed by Sphinx. However, for this to work, it is important that all lines are indented consistently, i.e.:

```
// ok
m.def("foo", &foo, R"mydelimiter(
    The foo function

    Parameters
    ----------
)mydelimiter");

// *not ok*
m.def("foo", &foo, R"mydelimiter(The foo function

    Parameters
    ----------
)mydelimiter");
```

By default, pybind11 automatically generates and prepends a signature to the docstring of a function registered with `module_::def()` and `class_::def()`. Sometimes this behavior is not desirable, because you want to provide your own signature or remove the docstring completely to exclude the function from the Sphinx documentation. The class `options` allows you to selectively suppress auto-generated signatures:

```
PYBIND11_MODULE(example, m) {
    py::options options;
    options.disable_function_signatures();
```

(continues on next page)

[3] https://docs.python.org/3/library/atexit.html
[4] http://www.sphinx-doc.org
[5] http://github.com/pybind/python_example

(continued from previous page)

```
    m.def("add", [](int a, int b) { return a + b; }, "A function which adds two numbers
↪");
}
```

pybind11 also appends all members of an enum to the resulting enum docstring. This default behavior can be disabled by using the `disable_enum_members_docstring()` function of the `options` class.

With `disable_user_defined_docstrings()` all user defined docstrings of `module_::def()`, `class_::def()` and `enum_()` are disabled, but the function signatures and enum members are included in the docstring, unless they are disabled separately.

Note that changes to the settings affect only function bindings created during the lifetime of the `options` instance. When it goes out of scope at the end of the module's init function, the default settings are restored to prevent unwanted side effects.

14.11 Avoiding C++ types in docstrings

Docstrings are generated at the time of the declaration, e.g. when `.def(...)` is called. At this point parameter and return types should be known to pybind11. If a custom type is not exposed yet through a `py::class_` constructor or a custom type caster, its C++ type name will be used instead to generate the signature in the docstring:

```
|  __init__(...)
|      __init__(self: example.Foo, arg0: ns::Bar) -> None
                                          ^^^^^^^
```

This limitation can be circumvented by ensuring that C++ classes are registered with pybind11 before they are used as a parameter or return type of a function:

```
PYBIND11_MODULE(example, m) {

    auto pyFoo = py::class_<ns::Foo>(m, "Foo");
    auto pyBar = py::class_<ns::Bar>(m, "Bar");

    pyFoo.def(py::init<const ns::Bar&>());
    pyBar.def(py::init<const ns::Foo&>());
}
```

14.12 Setting inner type hints in docstrings

When you use pybind11 wrappers for `list`, `dict`, and other generic python types, the docstring will just display the generic type. You can convey the inner types in the docstring by using a special 'typed' version of the generic type.

```
PYBIND11_MODULE(example, m) {
    m.def("pass_list_of_str", [](py::typing::List<py::str> arg) {
        // arg can be used just like py::list
    ));
}
```

The resulting docstring will be `pass_list_of_str(arg0: list[str]) -> None`.

The following special types are available in `pybind11/typing.h`:

- py::Tuple<Args...>
- py::Dict<K, V>
- py::List<V>
- py::Set<V>
- py::Callable<Signature>

Warning: Just like in python, these are merely hints. They don't actually enforce the types of their contents at runtime or compile time.

CHAPTER

FIFTEEN

DEPRECATED

Support for C++11 is deprecated and will be removed in a future version. Please use at least C++14.

Support for FindPythonLibs (not available in CMake 3.26+ mode) is deprecated and will be removed in a future version. The default mode is also going to change to `"new"` from `"compat"` in the future.

The following features were deprecated before pybind11 3.0, and may be removed in minor releases of pybind11 3.x.

Table 1: Deprecated Features

Feature	Deprecated Version	Year
`py::metaclass()`	2.1	2017
`PYBIND11_PLUGIN`	2.2	2017
`py::set_error()` replacing `operator()`	2.12	2024
`get_type_overload`	2.6	2020
`call()`	2.0	2016
`.str()`	?	
`.get_type()`	2.6	
== and !=	2.2	2017
`.check()`	?	
`object(handle, bool)`	?	
`error_already_set.clear()`	2.2	2017
`obj.attr(...)` as `bool`	?	
`.contains`	? (maybe 2.4)	
`py::capsule` two-argument with destructor	?	

15.1 py::enum_

This is the original documentation for `py::enum_`, which is deprecated because it is not PEP 435 compatible (see also #2332). Please prefer `py::native_enum` (added with pybind11v3) when writing new bindings. See *Enumerations and internal types* for more information.

Let's suppose that we have an example class that contains internal types like enumerations, e.g.:

```
struct Pet {
    enum Kind {
        Dog = 0,
        Cat
    };

    struct Attributes {
```

(continues on next page)

(continued from previous page)

```
        float age = 0;
    };

    Pet(const std::string &name, Kind type) : name(name), type(type) { }

    std::string name;
    Kind type;
    Attributes attr;
};
```

The binding code for this example looks as follows:

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

To ensure that the nested types `Kind` and `Attributes` are created within the scope of `Pet`, the `pet` `py::class_` instance must be supplied to the `enum_` and `py::class_` constructor. The `enum_::export_values()` function exports the enum entries into the parent scope, which should be skipped for newer C++11-style strongly typed enums.

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

The entries defined by the enumeration type are exposed in the `__members__` property:

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

The `name` property returns the name of the enum value as a unicode string.

Note: It is also possible to use `str(enum)`, however these accomplish different goals. The following shows how these two approaches differ.

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

(continues on next page)

(continued from previous page)

```
>>> str(pet_type)
'Pet.Cat'
>>> pet_type.name
'Cat'
```

Note: When the special tag `py::arithmetic()` is specified to the `enum_` constructor, pybind11 creates an enumeration that also supports rudimentary arithmetic and bit-level operations like comparisons, and, or, xor, negation, etc.

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

By default, these are omitted to conserve space.

Warning: Contrary to Python customs, enum values from the wrappers should not be compared using `is`, but with == (see #1177 for background).

CHAPTER

SIXTEEN

FREQUENTLY ASKED QUESTIONS

16.1 "ImportError: dynamic module does not define init function"

1. Make sure that the name specified in PYBIND11_MODULE is identical to the filename of the extension library (without suffixes such as `.so`).

2. If the above did not fix the issue, you are likely using an incompatible version of Python that does not match what you compiled with.

16.2 "Symbol not found: `__Py_ZeroStruct` / `_PyInstanceMethod_Type`"

See the first answer.

16.3 "SystemError: dynamic module not initialized properly"

See the first answer.

16.4 The Python interpreter immediately crashes when importing my module

See the first answer.

16.5 Limitations involving reference arguments

In C++, it's fairly common to pass arguments using mutable references or mutable pointers, which allows both read and write access to the value supplied by the caller. This is sometimes done for efficiency reasons, or to realize functions that have multiple return values. Here are two very basic examples:

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

In Python, all arguments are passed by reference, so there is no general issue in binding such code from Python.

However, certain basic Python types (like `str`, `int`, `bool`, `float`, etc.) are **immutable**. This means that the following attempt to port the function to Python doesn't have the same effect on the value provided by the caller – in fact, it does nothing at all.

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

pybind11 is also affected by such language-level conventions, which means that binding `increment` or `increment_ptr` will also create Python functions that don't modify their arguments.

Although inconvenient, one workaround is to encapsulate the immutable types in a custom type that does allow modifications.

An other alternative involves binding a small wrapper lambda function that returns a tuple with all output arguments (see the remainder of the documentation for examples on binding lambda functions). An example:

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

and the binding code

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

16.6 How can I reduce the build time?

First, consider the opt-in precompiled mode: it compiles the non-template part of pybind11 once per project instead of once for each translation unit. In CMake, this is one keyword on `pybind11_add_module`. See *Pre-compiling part of pybind11*.

It's also good practice to split binding code over multiple files, as in the following example:

`example.cpp`:

```
void init_ex1(py::module_ &);
void init_ex2(py::module_ &);
/* ... */

PYBIND11_MODULE(example, m, py::mod_gil_not_used()) {
    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
```

As shown above, the various `init_ex` functions should be contained in separate files that can be compiled independently from one another, and then linked together into the same final shared object. Following this approach will:

1. reduce memory requirements per compilation unit.
2. enable parallel builds (if desired).
3. allow for faster incremental builds. For instance, when a single class definition is changed, only a subset of the binding code will generally need to be recompiled.

16.7 "recursive template instantiation exceeded maximum depth of 256"

If you receive an error about excessive recursive template evaluation, try specifying a larger value, e.g. `-ftemplate-depth=1024` on GCC/Clang. The culprit is generally the generation of function signatures at compile time using C++14 template metaprogramming.

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

This error typically indicates that you are compiling without the required `-fvisibility` flag. pybind11 code internally forces hidden visibility on all internal code, but if non-hidden (and thus *exported*) code attempts to include a pybind type (for example, `py::object` or `py::list`) you can run into this warning.

To avoid it, make sure you are specifying `-fvisibility=hidden` when compiling pybind code.

As to why `-fvisibility=hidden` is necessary, because pybind modules could have been compiled under different versions of pybind itself, it is also important that the symbols defined in one module do not clash with the potentially-incompatible symbols defined in another. While Python extension modules are usually loaded with localized symbols (under POSIX systems typically using `dlopen` with the `RTLD_LOCAL` flag), this Python default can be changed, but even if it isn't it is not always enough to guarantee complete independence of the symbols involved when not using `-fvisibility=hidden`.

Additionally, `-fvisibility=hidden` can deliver considerably binary size savings. (See the following section for more details.)

16.9 How can I create smaller binaries?

To do its job, pybind11 extensively relies on a programming technique known as *template metaprogramming*, which is a way of performing computation at compile time using type information. Template metaprogramming usually instantiates code involving significant numbers of deeply nested types that are either completely removed or reduced to just a few instructions during the compiler's optimization phase. However, due to the nested nature of these types, the resulting symbol names in the compiled extension library can be extremely long. For instance, the included test suite contains the following symbol:

```
__ZN8pybind1112cpp_functionC1Iv8Example2JRNSt3__16vectorINS3_12basic_stringIwNS3_11char_
↪traitsIwEENS3_9allocatorIwEEEENS8_ISA_EEEEEJNS_4nameENS_7siblingENS_9is_methodEA28_
↪cEEEMT0_FT_DpT1_EDpRKT2_
```

which is the mangled form of the following function type:

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

The memory needed to store just the mangled name of this function (196 bytes) is larger than the actual piece of code (111 bytes) it represents! On the other hand, it's silly to even give this function a name – after all, it's just a tiny cog in a bigger piece of machinery that is not exposed to the outside world. So we'll generally only want to export symbols for those functions which are actually called from the outside.

This can be achieved by specifying the parameter `-fvisibility=hidden` to GCC and Clang, which sets the default symbol visibility to *hidden*, which has a tremendous impact on the final binary size of the resulting extension library. (On Visual Studio, symbols are already hidden by default, so nothing needs to be done there.)

In addition to decreasing binary size, `-fvisibility=hidden` also avoids potential serious issues when loading multiple modules and is required for proper pybind operation. See the previous FAQ entry for more details.

16.10 How can I properly handle Ctrl-C in long-running functions?

Ctrl-C is received by the Python interpreter, and holds it until the GIL is released, so a long-running function won't be interrupted.

To interrupt from inside your function, you can use the `PyErr_CheckSignals()` function, that will tell if a signal has been raised on the Python side. This function merely checks a flag, so its impact is negligible. When a signal has been received, you must either explicitly interrupt execution by throwing `py::error_already_set` (which will propagate the existing `KeyboardInterrupt`), or clear the error (which you usually will not want):

```
PYBIND11_MODULE(example, m, py::mod_gil_not_used()) {
    m.def("long running_func", []()
    {
        for (;;) {
            if (PyErr_CheckSignals() != 0)
                throw py::error_already_set();
```

(continues on next page)

(continued from previous page)

```
            // Long running iteration
        }
    });
}
```

16.11 What is a highly conclusive and simple way to find memory leaks (e.g. in pybind11 bindings)?

Use `while True` & `top` (Linux, macOS).

For example, locally change tests/test_type_caster_pyobject_ptr.py like this:

```
 def test_return_list_pyobject_ptr_reference():
+  while True:
     vec_obj = m.return_list_pyobject_ptr_reference(ValueHolder)
     assert [e.value for e in vec_obj] == [93, 186]
     # Commenting out the next `assert` will leak the Python references.
     # An easy way to see evidence of the leaks:
     # Insert `while True:` as the first line of this function and monitor the
     # process RES (Resident Memory Size) with the Unix top command.
-    assert m.dec_ref_each_pyobject_ptr(vec_obj) == 2
+    # assert m.dec_ref_each_pyobject_ptr(vec_obj) == 2
```

Then run the test as you would normally do, which will go into the infinite loop.

In another shell, but on the same machine run:

```
top
```

This will show:

```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
1266095 rwgk      20   0 5207496 611372  45696 R 100.0   0.3   0:08.01 test_type_caste
```

Look for the number under `RES` there. You'll see it going up very quickly.

Don't forget to Ctrl-C the test command before your machine becomes unresponsive due to swapping.

This method only takes a couple minutes of effort and is very conclusive. What you want to see is that the `RES` number is stable after a couple seconds.

16.12 CMake doesn't detect the right Python version

The CMake-based build system will try to automatically detect the installed version of Python and link against that. When this fails, or when there are multiple versions of Python and it finds the wrong one, delete `CMakeCache.txt` and then add `-DPYTHON_EXECUTABLE=$(which python)` to your CMake configure line. (Replace `$(which python)` with a path to python if your prefer.)

You can alternatively try `-DPYBIND11_FINDPYTHON=ON`, which will activate the new CMake FindPython support instead of pybind11's custom search. Newer CMake, like, 3.18.2+, is recommended. You can set this in your `CMakeLists.txt` before adding or finding pybind11, as well.

16.13 Inconsistent detection of Python version in CMake and pybind11

The functions `find_package(PythonInterp)` and `find_package(PythonLibs)` provided by CMake for Python version detection are modified by pybind11 due to unreliability and limitations that make them unsuitable for pybind11's needs. Instead pybind11 provides its own, more reliable Python detection CMake code. Conflicts can arise, however, when using pybind11 in a project that *also* uses the CMake Python detection in a system with several Python versions installed.

This difference may cause inconsistencies and errors if *both* mechanisms are used in the same project.

There are three possible solutions:

1. Avoid using `find_package(PythonInterp)` and `find_package(PythonLibs)` from CMake and rely on pybind11 in detecting Python version. If this is not possible, the CMake machinery should be called *before* including pybind11.
2. Set `PYBIND11_FINDPYTHON` to `True` or use `find_package(Python COMPONENTS Interpreter Development)` on modern CMake (3.18.2+ best). Pybind11 in these cases uses the new CMake FindPython instead of the old, deprecated search tools, and these modules are much better at finding the correct Python. If FindPythonLibs/Interp are not available (CMake 3.27+), then this will be ignored and FindPython will be used.
3. Set `PYBIND11_NOPYTHON` to `TRUE`. Pybind11 will not search for Python. However, you will have to use the target-based system, and do more setup yourself, because it does not know about or include things that depend on Python, like `pybind11_add_module`. This might be ideal for integrating into an existing system, like scikit-build's Python helpers.

16.14 How to cite this project?

We suggest the following BibTeX template to cite pybind11 in scientific discourse:

```
@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}
}
```

CHAPTER

SEVENTEEN

BENCHMARK

The following is the result of a synthetic benchmark comparing both compilation time and module size of pybind11 against Boost.Python. A detailed report about a Boost.Python to pybind11 conversion of a real project is available here:[1].

17.1 Setup

A python script (see the `docs/benchmark.py` file) was used to generate a set of files with dummy classes whose count increases for each successive benchmark (between 1 and 2048 classes in powers of two). Each class has four methods with a randomly generated signature with a return value and four arguments. (There was no particular reason for this setup other than the desire to generate many unique function signatures whose count could be controlled in a simple way.)

Here is an example of the binding code for one class:

```
...
class cl034 {
public:
    cl279 *fn_000(cl084 *, cl057 *, cl065 *, cl042 *);
    cl025 *fn_001(cl098 *, cl262 *, cl414 *, cl121 *);
    cl085 *fn_002(cl445 *, cl297 *, cl145 *, cl421 *);
    cl470 *fn_003(cl200 *, cl323 *, cl332 *, cl492 *);
};
...

PYBIND11_MODULE(example, m, py::mod_gil_not_used()) {
    ...
    py::class_<cl034>(m, "cl034")
        .def("fn_000", &cl034::fn_000)
        .def("fn_001", &cl034::fn_001)
        .def("fn_002", &cl034::fn_002)
        .def("fn_003", &cl034::fn_003)
    ...
}
```

The Boost.Python version looks almost identical except that a return value policy had to be specified as an argument to `def()`. For both libraries, compilation was done with

```
Apple LLVM version 7.0.2 (clang-700.1.81)
```

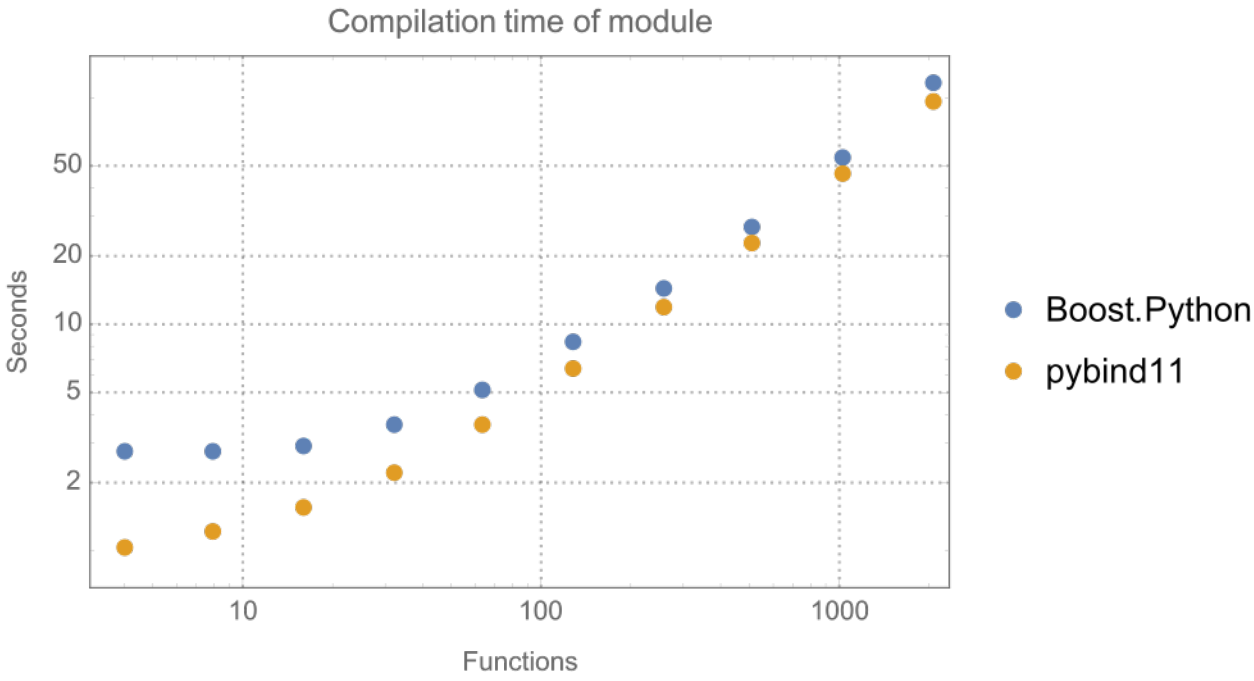
[1] http://graylab.jhu.edu/RosettaCon2016/PyRosetta-4.pdf

and the following compilation flags

```
g++ -Os -shared -rdynamic -undefined dynamic_lookup -fvisibility=hidden -std=c++14
```

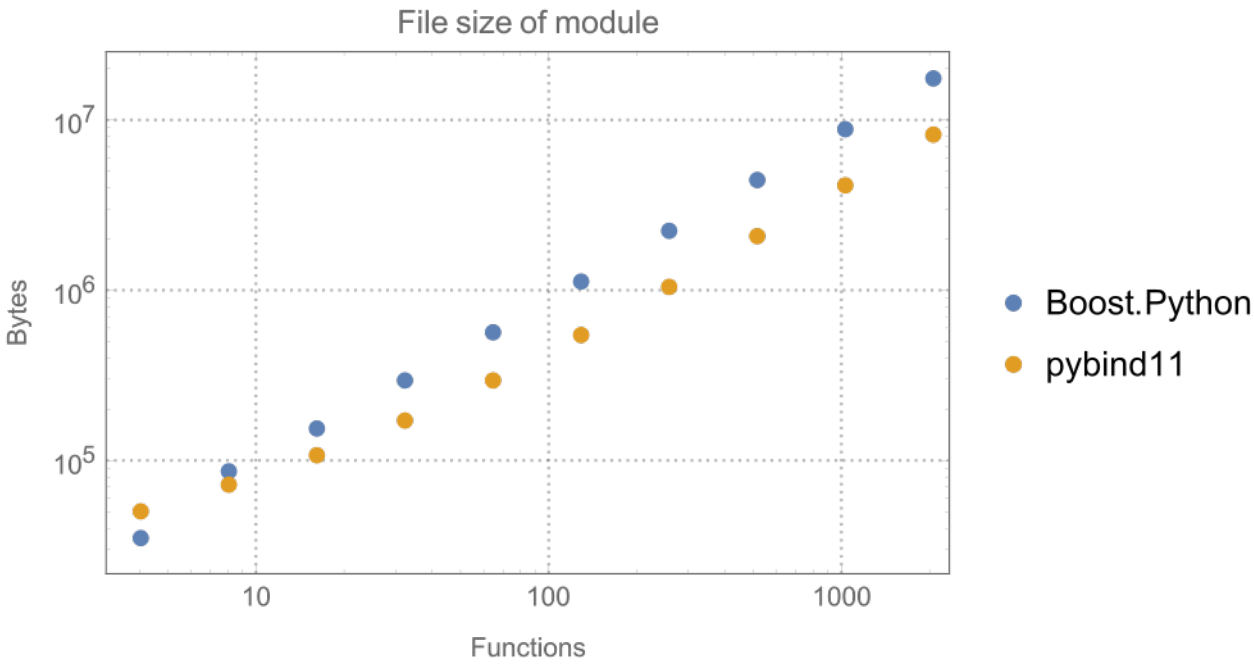
17.2 Compilation time

The following log-log plot shows how the compilation time grows for an increasing number of class and function declarations. pybind11 includes many fewer headers, which initially leads to shorter compilation times, but the performance is ultimately fairly similar (pybind11 is 19.8 seconds faster for the largest largest file with 2048 classes and a total of 8192 methods – a modest **1.2x** speedup relative to Boost.Python, which required 116.35 seconds).



17.3 Module size

Differences between the two libraries become much more pronounced when considering the file size of the generated Python plugin: for the largest file, the binary generated by Boost.Python required 16.8 MiB, which was **2.17 times / 9.1 megabytes** larger than the output generated by pybind11. For very small inputs, Boost.Python has an edge in the plot below – however, note that it stores many definitions in an external library, whose size was not included here, hence the comparison is slightly shifted in Boost.Python's favor.


File size of module
Bytes
10^7
10^6
10^5
10
100
1000
Functions
Boost.Python
pybind11

CHAPTER

EIGHTEEN

LIMITATIONS

18.1 Design choices

pybind11 strives to be a general solution to binding generation, but it also has certain limitations:

- pybind11 casts away `const`-ness in function arguments and return values. This is in line with the Python language, which has no concept of `const` values. This means that some additional care is needed to avoid bugs that would be caught by the type checker in a traditional C++ program.
- The NumPy interface `pybind11::array` greatly simplifies accessing numerical data from C++ (and vice versa), but it's not a full-blown array class like `Eigen::Array` or `boost.multi_array`. `Eigen` objects are directly supported, however, with `pybind11/eigen.h`.

Large but useful features could be implemented in pybind11 but would lead to a significant increase in complexity. Pybind11 strives to be simple and compact. Users who require large new features are encouraged to write an extension to pybind11; see pybind11_json for an example.

18.2 Known bugs

These are issues that hopefully will one day be fixed, but currently are unsolved. If you know how to help with one of these issues, contributions are welcome!

- Intel 20.2 is currently having an issue with the test suite. #2573
- Debug mode Python does not support 1-5 tests in the test suite currently. #2422
- PyPy3 7.3.1 and 7.3.2 have issues with several tests on 32-bit Windows.

18.3 Known limitations

These are issues that are probably solvable, but have not been fixed yet. A clean, well written patch would likely be accepted to solve them.

- Type casters are not kept alive recursively. #2527 One consequence is that containers of `char *` are currently not supported. #2245

18.4 Python 3.9.0 warning

Combining older versions of pybind11 (< 2.6.0) with Python on exactly 3.9.0 will trigger undefined behavior that typically manifests as crashes during interpreter shutdown (but could also destroy your data. **You have been warned**).

This issue was fixed in Python. As a mitigation for this bug, pybind11 2.6.0 or newer includes a workaround specifically when Python 3.9.0 is detected at runtime, leaking about 50 bytes of memory when a callback function is garbage collected. For reference, the pybind11 test suite has about 2,000 such callbacks, but only 49 are garbage collected before the end-of-process. Wheels (even if built with Python 3.9.0) will correctly avoid the leak when run in Python 3.9.1, and this does not affect other 3.X versions.

Warning: Please be advised that the reference documentation discussing pybind11 internals is currently incomplete. Please refer to the previous sections and the pybind11 header files for the nitty gritty details.

CHAPTER

NINETEEN

REFERENCE

19.1 Macros

PYBIND11_MODULE(name, variable, ...)

This macro creates the entry point that will be invoked when the Python interpreter imports an extension module. The module name is given as the first argument and it should not be in quotes. The second macro argument defines a variable of type `py::module_` which can be used to initialize the module.

The entry point is marked as "maybe unused" to aid dead-code detection analysis: since the entry point is typically only looked up at runtime and not referenced during translation, it would otherwise appear as unused ("dead") code.

```
PYBIND11_MODULE(example, m) {
    m.doc() = "pybind11 example module";

    // Add bindings here
    m.def("foo", []() {
        return "Hello, World!";
    });
}
```

The third and subsequent macro arguments are optional (available since 2.13.0), and can be used to mark the extension module as supporting various Python features.

- `mod_gil_not_used()`
- `multiple_interpreters::per_interpreter_gil()`
- `multiple_interpreters::shared_gil()`
- `multiple_interpreters::not_supported()`

```
PYBIND11_MODULE(example, m, py::mod_gil_not_used()) {
    m.doc() = "pybind11 example module safe to run without the GIL";
    m.def("foo", []() {
        return "Hello, Free-threaded World!";
    });
}
```

19.2 Convenience classes for arbitrary Python types

19.2.1 Common member functions

template<typename **Derived**>

class **object_api** : public pyobject_tag

A mixin class which adds common functions to *handle*, *object* and various accessors. The only requirement for *Derived* is to implement `PyObject *Derived::ptr() const`.

Public Functions

iterator **begin**() const

Return an iterator equivalent to calling `iter()` in Python. The object must be a collection which supports the iteration protocol.

iterator **end**() const

Return a sentinel which ends iteration.

item_accessor **operator[]**(*handle* key) const

Return an internal functor to invoke the object's sequence protocol. Casting the returned `detail::item_accessor` instance to a *handle* or *object* subclass causes a corresponding call to `__getitem__`. Assigning a *handle* or *object* subclass causes a call to `__setitem__`.

item_accessor **operator[]**(*object* &&key) const

See above (the only difference is that the key's reference is stolen)

item_accessor **operator[]**(const char *key) const

See above (the only difference is that the key is provided as a string literal)

obj_attr_accessor **attr**(*handle* key) const

Return an internal functor to access the object's attributes. Casting the returned `detail::obj_attr_accessor` instance to a *handle* or *object* subclass causes a corresponding call to `getattr`. Assigning a *handle* or *object* subclass causes a call to `setattr`.

obj_attr_accessor **attr**(*object* &&key) const

See above (the only difference is that the key's reference is stolen)

str_attr_accessor **attr**(const char *key) const

See above (the only difference is that the key is provided as a string literal)

template<typename **T**>
obj_attr_accessor **attr_with_type_hint**(*handle* key) const

Similar to the above attr functions with the difference that the templated Type is used to set the `__annotations__` dict value to the corresponding key. Worth noting that attr_with_type_hint is implemented in cast.h.

template<typename **T**>
str_attr_accessor **attr_with_type_hint**(const char *key) const

See above (the only difference is that the key is provided as a string literal)

args_proxy **operator***() const

Matches * unpacking in Python, e.g. to unpack arguments out of a `tuple` or `list` for a function call. Applying another * to the result yields ** unpacking, e.g. to unpack a dict as function keyword arguments. See *Calling Python functions*.

template<typename **T**>
bool **contains**(*T* &&item) const

Check if the given item is contained within this object, i.e. `item in obj`.

template<return_value_policy **policy** = return_value_policy::automatic_reference, typename ...**Args**>
object **operator()**(*Args*&&... args) const

Assuming the Python object is a function or implements the `__call__` protocol, `operator()` invokes the underlying function, passing an arbitrary set of parameters. The result is returned as a *object* and may need to be converted back into a Python object using *handle::cast()*.

When some of the arguments cannot be converted to Python objects, the function will throw a `cast_error` exception. When the Python function call fails, a *error_already_set* exception is thrown.

inline bool **is**(*object_api* const &other) const

Equivalent to `obj is other` in Python.

inline bool **is_none**() const

Equivalent to `obj is None` in Python.

inline bool **equal**(*object_api* const &other) const

Equivalent to obj == other in Python.

str_attr_accessor **doc**() const

Get or set the object's docstring, i.e. `obj.__doc__`.

object **annotations**() const

Get or set the object's annotations, i.e. `obj.__annotations__`.

inline ssize_t **ref_count**() const

Return the object's current reference count.

19.2.2 Without reference counting

class **handle** : public detail::object_api<*handle*>

Holds a reference to a Python object (no reference counting)

The *handle* class is a thin wrapper around an arbitrary Python object (i.e. a `PyObject *` in Python's C API). It does not perform any automatic reference counting and merely provides a basic C++ interface to various Python API functions.

See also:

The *object* class inherits from *handle* and adds automatic reference counting features.

Subclassed by args_proxy, kwargs_proxy, *object*

Public Functions

handle() = default

The default constructor creates a handle with a `nullptr`-valued pointer.

template<typename **T**, detail::enable_if_t<detail::is_pyobj_ptr_or_nullptr_t<*T*>::value, int> = 0>
inline **handle**(*T* ptr)

Enable implicit conversion from `PyObject *` and `nullptr`. Not using *handle(PyObject *ptr)* to avoid implicit conversion from `0`.

template<typename **T**, detail::enable_if_t<detail::all_of<detail::none_of<std::is_base_of<*handle*, *T*>, detail::is_pyobj_ptr_or_nullptr_t<*T*>>, std::is_convertible<*T*, PyObject*>>::value, int> = 0>
inline **handle**(*T* &obj)

Enable implicit conversion through `T::operator PyObject *()`.

inline PyObject ***ptr**() const

Return the underlying `PyObject *` pointer.

inline const *handle* &**inc_ref**() const &

Manually increase the reference count of the Python object. Usually, it is preferable to use the *object* class which derives from *handle* and calls this function automatically. Returns a reference to itself.

inline const *handle* &**dec_ref**() const &

Manually decrease the reference count of the Python object. Usually, it is preferable to use the *object* class which derives from *handle* and calls this function automatically. Returns a reference to itself.

template<typename **T**>
T **cast**() const

Attempt to cast the Python object into the given C++ type. A `cast_error` will be throw upon failure.

inline explicit **operator bool**() const

Return `true` when the `handle` wraps a valid Python object.

inline bool **operator==**(const *handle* &h) const

Deprecated: Check that the underlying pointers are the same. Equivalent to `obj1 is obj2` in Python.

19.2.3 With reference counting

class **object** : public *handle*

Holds a reference to a Python object (with reference counting)

Like *handle*, the *object* class is a thin wrapper around an arbitrary Python object (i.e. a `PyObject *` in Python's C API). In contrast to *handle*, it optionally increases the object's reference count upon construction, and it *always* decreases the reference count when the *object* instance goes out of scope and is destructed. When using *object* instances consistently, it is much easier to get reference counting right at the first attempt.

Subclassed by ClassVar< T >, Final< T >, Optional< T >, Union< Types >, *anyset*, *bool_*, *buffer*, *bytearray*, *bytes*, *capsule*, *dict*, dtype, *ellipsis*, exception< type >, *float_*, *function*, generic_type, *int_*, *iterable*, *iterator*, *list*, *memoryview*, *module_*, *none*, *sequence*, *slice*, *staticmethod*, *str*, *tuple*, *type*, *weakref*

Public Functions

inline **object**(const *object* &o)

Copy constructor; always increases the reference count.

inline **object**(*object* &&other) noexcept

Move constructor; steals the object from `other` and preserves its reference count.

inline **~object**()

Destructor; automatically calls *handle::dec_ref()*

inline *handle* **release**()

Resets the internal pointer to `nullptr` without decreasing the object's reference count. The function returns a raw handle to the original Python object.

template<typename **T**>
T **reinterpret_borrow**(*handle* h)

Declare that a *handle* or `PyObject *` is a certain type and borrow the reference. The target type `T` must be *object* or one of its derived classes. The function doesn't do any conversions or checks. It's up to the user to make sure that the target type is correct.

```
PyObject *p = PyList_GetItem(obj, index);
py::object o = reinterpret_borrow<py::object>(p);
// or
py::tuple t = reinterpret_borrow<py::tuple>(p); // <-- `p` must be already be a␣
↪`tuple`
```

template<typename **T**>
T **reinterpret_steal**(*handle* h)

Like *reinterpret_borrow()*, but steals the reference.

```
PyObject *p = PyObject_Str(obj);
py::str s = reinterpret_steal<py::str>(p); // <-- `p` must be already be a␣
↪`str`
```

19.3 Convenience classes for specific Python types

class **module_** : public *object*

Wrapper for Python extension modules.

Public Functions

inline explicit **module_**(const char *name, const char *doc = nullptr)

Create a new top-level Python module with the given name and docstring.

template<typename **Func**, typename ...**Extra**>
inline *module_* &**def**(const char *name_, *Func* &&f, const *Extra*&... extra)

Create Python binding for a new function within the module scope. `Func` can be a plain C++ function, a function pointer, or a lambda function. For details on the `Extra&& ... extra` argument, see section *Passing extra arguments to def or py::class_*.

*module_* **def_submodule**(const char *name, const char *doc = nullptr)

Create and return a new Python submodule with the given name and docstring. This also works recursively, i.e.

```
py::module_ m("example", "pybind11 example plugin");
py::module_ m2 = m.def_submodule("sub", "A submodule of 'example'");
py::module_ m3 = m2.def_submodule("subsub", "A submodule of 'example.sub'");
```

void **reload**()

Reload the module or throws *error_already_set*.

void **add_object**(const char *name, *handle* obj, bool overwrite = false)

Adds an object to the module using the given name. Throws if an object with the given name already exists.

`overwrite` should almost always be false: attempting to overwrite objects that pybind11 has established will, in most cases, break things.

Public Static Functions

static *module_* **import**(const char *name)

Import and return a module or throws *error_already_set*.

static *module_* **create_extension_module**(const char *name, const char *doc, PyModuleDef *def, mod_gil_not_used gil_not_used = mod_gil_used())

Create a new top-level module that can be used as the main module of a C extension.

`def` should point to a statically allocated PyModuleDef.

group **pytypes**

Functions

template<typename **Unsigned**>
Unsigned **as_unsigned**(PyObject *o)

template<typename ...**Args**>
constexpr bool **args_are_all_keyword_or_ds**()

class **iterator** : public *object*

#include <pytypes.h>

Wraps a Python iterator so that it can also be used as a C++ input iterator

Caveat: copying an iterator does not (and cannot) clone the internal state of the Python iterable. This also applies to the post-increment operator. This iterator should only be used to retrieve the current value using `operator*()`.

Subclassed by Iterator< T >

Public Static Functions

static inline *iterator* **sentinel**()

The value which marks the end of the iteration. `it == iterator::sentinel()` is equivalent to catching `StopIteration` in Python.

```
void foo(py::iterator it) {
    while (it != py::iterator::sentinel()) {
       // use `*it`
       ++it;
    }
}
```

class **type** : public *object*

Subclassed by Type< T >

Public Static Functions

static inline *handle* **handle_of**(*handle* h)

Return a type handle from a handle or an object.

static inline *type* **of**(*handle* h)

Return a type object from a handle or an object.

template<typename **T**>
static *handle* **handle_of**()

Convert C++ type to handle if previously registered. Does not convert standard types, like int, float. etc. yet. See https://github.com/pybind/pybind11/issues/2486

template<typename **T**>
static inline *type* **of**()

Convert C++ type to type if previously registered. Does not convert standard types, like int, float. etc. yet. See https://github.com/pybind/pybind11/issues/2486

class **iterable** : public *object*

Subclassed by Iterable< T >

class **str** : public *object*

Public Functions

template<typename **T**, detail::enable_if_t<!std::is_base_of<*object*, detail::remove_cvref_t<*T*>>::value && std::is_constructible<*handle*, *T*>::value, int> = 0>
inline explicit **str**(*T* &&h)

Return a string representation of the object. This is analogous to the `str()` function in Python.

class **bytes** : public *object*

class **bytearray** : public *object*

class **none** : public *object*

Subclassed by Never, NoReturn

class **ellipsis** : public *object*

class **bool_** : public *object*

Subclassed by TypeGuard< T >, TypeIs< T >

class **int_** : public *object*

class **float_** : public *object*

class **weakref** : public *object*

class **slice** : public *object*

class **capsule** : public *object*

class **tuple** : public *object*

Subclassed by *args*, typing::Tuple< Types >

class **dict** : public *object*

Subclassed by Dict< K, V >, *kwargs*

class **sequence** : public *object*

class **list** : public *object*

Subclassed by List< T >

class **args** : public *tuple*

Subclassed by *Args< T >*

class **kwargs** : public *dict*

Subclassed by *KWArgs< T >*

template<typename **T**>

class **Args** : public *args*

template<typename **T**>

class **KWArgs** : public *kwargs*

class **anyset** : public *object*

Subclassed by *frozenset*, *set*

class **set** : public *anyset*

Subclassed by Set< T >

class **frozenset** : public *anyset*

class **function** : public *object*

Subclassed by Callable< Return(Args...)>, cpp_function

class **staticmethod** : public *object*

class **buffer** : public *object*

Subclassed by array

class **memoryview** : public *object*

Public Functions

inline explicit **memoryview**(const buffer_info &info)

Creates `memoryview` from `buffer_info`.

`buffer_info` must be created from `buffer::request()`. Otherwise throws an exception.

For creating a `memoryview` from objects that support buffer protocol, use `memoryview(const object& obj)` instead of this constructor.

Public Static Functions

static *memoryview* **from_buffer**(void *ptr, ssize_t itemsize, const char *format, detail::any_container<ssize_t> shape, detail::any_container<ssize_t> strides, bool readonly = false)

Creates `memoryview` from static buffer.

This method is meant for providing a `memoryview` for C/C++ buffer not managed by Python. The caller is responsible for managing the lifetime of `ptr` and `format`, which MUST outlive the memoryview constructed here.

See also: Python C API documentation for PyMemoryView_FromBuffer.

Parameters

- **ptr** – Pointer to the buffer.
- **itemsize** – Byte size of an element.
- **format** – Pointer to the null-terminated format string. For homogeneous Buffers, this should be set to `format_descriptor<T>::value`.
- **shape** – Shape of the tensor (1 entry per dimension).
- **strides** – Number of bytes between adjacent entries (for each per dimension).
- **readonly** – Flag to indicate if the underlying storage may be written to.

static inline *memoryview* **from_memory**(void *mem, ssize_t size, bool readonly = false)

Creates `memoryview` from static memory.

This method is meant for providing a `memoryview` for C/C++ buffer not managed by Python. The caller is responsible for managing the lifetime of `mem`, which MUST outlive the memoryview constructed here.

See also: Python C API documentation for PyMemoryView_FromBuffer.

19.4 Convenience functions converting to Python types

template<return_value_policy **policy** = return_value_policy::automatic_reference, typename ...**Args**>
typing::Tuple<*Args*...> **make_tuple**(*Args*&&... args_)

template<return_value_policy **Policy** = return_value_policy::reference_internal, typename **Iterator**, typename **Sentinel**, typename **ValueType** = typename detail::iterator_access<*Iterator*>::result_type, typename ...**Extra**>
typing::Iterator<*ValueType*> **make_iterator**(*Iterator* first, *Sentinel* last, *Extra*&&... extra)

Makes a python iterator from a first and past-the-end C++ InputIterator.

Warning: doxygenfunction: Unable to resolve function "make_iterator" with arguments (Type&, Extra&&...) in doxygen xml output for project "pybind11" from directory: .build/doxygenxml/. Potential matches:

```
- template<return_value_policy Policy = return_value_policy::reference_internal,␣
↪typename Iterator, typename Sentinel, typename ValueType = typename␣
↪detail::iterator_access<Iterator>::result_type, typename ...Extra> typing::Iterator
↪<ValueType> make_iterator(Iterator first, Sentinel last, Extra&&... extra)
- template<return_value_policy Policy = return_value_policy::reference_internal,␣
↪typename Type, typename ValueType = typename detail::iterator_access
↪<decltype(std::begin(std::declval<Type&>()))>::result_type, typename ...Extra>␣
↪typing::Iterator<ValueType> make_iterator(Type &value, Extra&&... extra)
```

template<return_value_policy **Policy** = return_value_policy::reference_internal, typename **Iterator**, typename **Sentinel**, typename **KeyType** = typename detail::iterator_key_access<*Iterator*>::result_type, typename ...**Extra**>
typing::Iterator<*KeyType*> **make_key_iterator**(*Iterator* first, *Sentinel* last, *Extra*&&... extra)

Makes a python iterator over the keys (`.first`) of a iterator over pairs from a first and past-the-end InputIterator.

Warning: doxygenfunction: Unable to resolve function "make_key_iterator" with arguments (Type&, Extra&&...) in doxygen xml output for project "pybind11" from directory: .build/doxygenxml/. Potential matches:

```
- template<return_value_policy Policy = return_value_policy::reference_internal,␣
↪typename Iterator, typename Sentinel, typename KeyType = typename detail::iterator_
↪key_access<Iterator>::result_type, typename ...Extra> typing::Iterator<KeyType>␣
↪make_key_iterator(Iterator first, Sentinel last, Extra&&... extra)
- template<return_value_policy Policy = return_value_policy::reference_internal,␣
↪typename Type, typename KeyType = typename detail::iterator_key_access
↪<decltype(std::begin(std::declval<Type&>()))>::result_type, typename ...Extra>␣
↪typing::Iterator<KeyType> make_key_iterator(Type &value, Extra&&... extra)
```

template<return_value_policy **Policy** = return_value_policy::reference_internal, typename **Iterator**, typename **Sentinel**, typename **ValueType** = typename detail::iterator_value_access<*Iterator*>::result_type, typename ...**Extra**>
typing::Iterator<*ValueType*> **make_value_iterator**(*Iterator* first, *Sentinel* last, *Extra*&&... extra)

Makes a python iterator over the values (`.second`) of a iterator over pairs from a first and past-the-end InputIterator.

Warning: doxygenfunction: Unable to resolve function "make_value_iterator" with arguments (Type&, Extra&&...) in doxygen xml output for project "pybind11" from directory: .build/doxygenxml/. Potential matches:

```
- template<return_value_policy Policy = return_value_policy::reference_internal,␣
↪typename Iterator, typename Sentinel, typename ValueType = typename␣
↪detail::iterator_value_access<Iterator>::result_type, typename ...Extra>␣
↪typing::Iterator<ValueType> make_value_iterator(Iterator first, Sentinel last,␣
↪Extra&&... extra)
- template<return_value_policy Policy = return_value_policy::reference_internal,␣
↪typename Type, typename ValueType = typename detail::iterator_value_access
↪<decltype(std::begin(std::declval<Type&>()))>::result_type, typename ...Extra>␣
↪typing::Iterator<ValueType> make_value_iterator(Type &value, Extra&&... extra)
```

19.5 Passing extra arguments to `def` or `py::class_`

group **annotations**

struct **is_method**

#include <attr.h> Annotation for methods.

struct **is_setter**

#include <attr.h> Annotation for setters.

struct **is_operator**

#include <attr.h> Annotation for operators.

struct **is_final**

#include <attr.h> Annotation for classes that cannot be subclassed.

struct **scope**

#include <attr.h> Annotation for parent scope.

struct **doc**

#include <attr.h> Annotation for documentation.

struct **name**

#include <attr.h> Annotation for function names.

struct **sibling**

#include <*attr.h*> Annotation indicating that a function is an overload associated with a given "sibling".

template<typename **T**>

struct **base**

#include <*attr.h*> Annotation indicating that a class derives from another given type.

template<size_t **Nurse**, size_t **Patient**>

struct **keep_alive**

#include <*attr.h*> Keep patient alive while nurse lives.

struct **multiple_inheritance**

#include <*attr.h*> Annotation indicating that a class is involved in a multiple inheritance relationship.

struct **dynamic_attr**

#include <*attr.h*> Annotation which enables dynamic attributes, i.e. adds `__dict__` to a class.

struct **buffer_protocol**

#include <*attr.h*> Annotation which enables the buffer protocol for a type.

struct **release_gil_before_calling_cpp_dtor**

#include <*attr.h*> Annotation which enables releasing the GIL before calling the C++ destructor of wrapped instances (pybind/pybind11#1446).

struct **metaclass**

#include <*attr.h*> Annotation which requests that a special metaclass is created for a type.

Public Functions

inline explicit **metaclass**(*handle* value)

Override pybind11's default metaclass.

struct **custom_type_setup**

#include <*attr.h*> Specifies a custom callback with signature `void (PyHeapTypeObject*)` that may be used to customize the Python type.

The callback is invoked immediately before `PyType_Ready`.

Note: This is an advanced interface, and uses of it may require changes to work with later versions of pybind11. You may wish to consult the implementation of `make_new_python_type` in `detail/classes.h` to understand the context in which the callback will be run.

struct **module_local**

#include <*attr.h*> Annotation that marks a class as local to the module:

struct **arithmetic**

#include <*attr.h*> Annotation to mark enums as an arithmetic type.

struct **prepend**

#include <attr.h> Mark a function for addition at the beginning of the existing overload chain instead of the end.

template<typename ...**Ts**>

struct **call_guard**

A call policy which places one or more guard variables (`Ts...`) around the function call.

For example, this definition:

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

is equivalent to the following pseudocode:

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

template<>

struct **call_guard**<>

template<typename **T**>

struct **call_guard**<*T*>

template<typename **T**, typename ...**Ts**>

struct **call_guard**<*T*, *Ts*...>

struct **type**

struct **arg**

#include <cast.h> Annotation for arguments

Subclassed by *arg_v*

Public Functions

inline explicit constexpr **arg**(const char *name = nullptr)

Constructs an argument with the name of the argument; if null or omitted, this is a positional argument.

template<typename **T**>
arg_v **operator=**(*T* &&value) const

Assign a value to this argument.

inline *arg* &**noconvert**(bool flag = true)

Indicate that the type should not be converted in the type caster.

inline *arg* &**none**(bool flag = true)

Indicates that the argument should/shouldn't allow None (e.g. for nullable pointer args)

Public Members

const char ***name**

If non-null, this is a named kwargs argument.

bool **flag_noconvert**

If set, do not allow conversion (requires a supporting type caster!)

bool **flag_none**

If set (the default), allow None to be passed to this argument.

struct **arg_v** : public *arg*

#include <*cast.h*> Annotation for arguments with values

Public Functions

template<typename **T**>
inline **arg_v**(const char *name, *T* &&x, const char *descr = nullptr)

Direct construction with name, default, and description.

template<typename **T**>
inline **arg_v**(const *arg* &base, *T* &&x, const char *descr = nullptr)

Called internally when invoking `py::arg("a") = value`

inline *arg_v* &**noconvert**(bool flag = true)

Same as *arg::noconvert()*, but returns *this as *arg_v*&, not arg&.

inline *arg_v* &**none**(bool flag = true)

Same as `arg::nonone()`, but returns *this as *arg_v*&, not arg&.

Public Members

object **value**

The default value.

const char ***descr**

The (optional) description of the default value.

std::string **type**

The C++ type name of the default value (only available when compiled in debug mode)

struct **kw_only**

#include <*cast.h*> Annotation indicating that all following arguments are keyword-only; the is the equivalent of an unnamed '*' argument

struct **pos_only**

#include <*cast.h*> Annotation indicating that all previous arguments are positional-only; the is the equivalent of an unnamed '/' argument

19.6 Embedding the interpreter

PYBIND11_EMBEDDED_MODULE(name, variable, ...)

Add a new module to the table of builtins for the interpreter. Must be defined in global scope. The first macro parameter is the name of the module (without quotes). The second parameter is the variable which will be used as the interface to add functions and classes to the module.

```
PYBIND11_EMBEDDED_MODULE(example, m) {
    // ... initialize functions and classes here
    m.def("foo", []() {
        return "Hello, World!";
    });
}
```

The third and subsequent macro arguments are optional, and can be used to mark the module as supporting various Python features.

- mod_gil_not_used()
- multiple_interpreters::per_interpreter_gil()
- multiple_interpreters::shared_gil()
- multiple_interpreters::not_supported()

```
PYBIND11_EMBEDDED_MODULE(example, m, py::mod_gil_not_used()) {
    m.def("foo", []() {
        return "Hello, Free-threaded World!";
    });
}
```

Warning: doxygenfunction: Unable to resolve function "initialize_interpreter" with arguments None in doxygen xml output for project "pybind11" from directory: .build/doxygenxml/. Potential matches:

```
- void initialize_interpreter(PyConfig *config, int argc = 0, const char *const *argv␣
↪= nullptr, bool add_program_dir_to_path = true)
- void initialize_interpreter(bool init_signal_handlers = true, int argc = 0, const␣
↪char *const *argv = nullptr, bool add_program_dir_to_path = true)
```

inline void **finalize_interpreter**()

Shut down the Python interpreter. No pybind11 or CPython API functions can be called after this. In addition, pybind11 objects must not outlive the interpreter:

```
{ // BAD
    py::initialize_interpreter();
    auto hello = py::str("Hello, World!");
    py::finalize_interpreter();
} // <-- BOOM, hello's destructor is called after interpreter shutdown

{ // GOOD
    py::initialize_interpreter();
    { // scoped
```

(continues on next page)

(continued from previous page)

```
        auto hello = py::str("Hello, World!");
    } // <-- OK, hello is cleaned up properly
    py::finalize_interpreter();
}

{ // BETTER
    py::scoped_interpreter guard{};
    auto hello = py::str("Hello, World!");
}
```

Warning: The interpreter can be restarted by calling `initialize_interpreter` again. Modules created using pybind11 can be safely re-initialized. However, Python itself cannot completely unload binary extension modules and there are several caveats with regard to interpreter restarting. All the details can be found in the CPython documentation. In short, not all interpreter memory may be freed, either due to reference cycles or user-created global data.

class **scoped_interpreter**

Scope guard version of `initialize_interpreter` and *finalize_interpreter()*. This a move-only guard and only a single instance can exist.

See `initialize_interpreter` for a discussion of its constructor arguments.

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

int main() {
    py::scoped_interpreter guard{};
    py::print(Hello, World!);
} // <-- interpreter shutdown
```

19.7 Redirecting C++ streams

class **scoped_ostream_redirect**

This a move-only guard that redirects output.

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

...

{
    py::scoped_ostream_redirect output;
    std::cout << "Hello, World!"; // Python stdout
} // <-- return std::cout to normal
```

You can explicitly pass the c++ stream and the python object, for example to guard stderr instead.

```
{
    py::scoped_ostream_redirect output{
        std::cerr, py::module::import("sys").attr("stderr")};
    std::cout << "Hello, World!";
}
```

Subclassed by *scoped_estream_redirect*

class **scoped_estream_redirect** : public *scoped_ostream_redirect*

Like *scoped_ostream_redirect*, but redirects cerr by default. This class is provided primary to make `py::call_guard` easier to make.

```
m.def("noisy_func", &noisy_func,
      py::call_guard<scoped_ostream_redirect,
                     scoped_estream_redirect>());
```

inline class_<detail::OstreamRedirect> **add_ostream_redirect**(*module_* m, const std::string &name = "ostream_redirect")

This is a helper function to add a C++ redirect context manager to Python instead of using a C++ guard. To use it, add the following to your binding code:

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

...

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

You now have a Python context manager that redirects your output:

```
with m.ostream_redirect():
    m.print_to_cout_function()
```

This manager can optionally be told which streams to operate on:

```
with m.ostream_redirect(stdout=true, stderr=true):
    m.noisy_function_with_error_printing()
```

19.8 Python built-in functions

group **python_builtins**

Unless stated otherwise, the following C++ functions behave the same as their Python counterparts.

Functions

dict **globals**()

Return a dictionary representing the global variables in the current execution frame, or `__main__.__dict__` if there is no frame (usually when the interpreter is embedded).

template<typename **T**, detail::enable_if_t<std::is_base_of<*object*, *T*>::value, int> = 0>
bool **isinstance**(*handle* obj)

Return true if `obj` is an instance of `T`. Type `T` must be a subclass of `object` or a class which was exposed to Python as `py::class_<T>`.

inline bool **isinstance**(*handle* obj, *handle* type)

Return true if `obj` is an instance of the `type`.

inline bool **hasattr**(*handle* obj, *handle* name)

inline bool **hasattr**(*handle* obj, const char *name)

inline void **delattr**(*handle* obj, *handle* name)

inline void **delattr**(*handle* obj, const char *name)

inline *object* **getattr**(*handle* obj, *handle* name)

inline *object* **getattr**(*handle* obj, const char *name)

inline *object* **getattr**(*handle* obj, *handle* name, *handle* default_)

inline *object* **getattr**(*handle* obj, const char *name, *handle* default_)

inline void **setattr**(*handle* obj, *handle* name, *handle* value)

inline void **setattr**(*handle* obj, const char *name, *handle* value)

inline ssize_t **hash**(*handle* obj)

inline size_t **len**(*handle* h)

Get the length of a Python object.

inline size_t **len_hint**(*handle* h)

Get the length hint of a Python object. Returns 0 when this cannot be determined.

inline *str* **repr**(*handle* h)

inline *iterator* **iter**(*handle* obj)

19.9 Inheritance

See *Object-oriented code* and *Classes* for more detail.

PYBIND11_OVERRIDE(ret_type, cname, fn, ...)

Macro to populate the virtual method in the trampoline class. This macro tries to look up the method from the Python side, deals with the *Global Interpreter Lock (GIL)* and necessary argument conversions to call this method and return the appropriate type. This macro should be used if the method name in C and in Python are identical. See *Overriding virtual functions in Python* for more information.

```
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 (ret_type)
            Animal,      // Parent class (cname)
            go,          // Name of function in C++ (must match Python name) (fn)
            n_times      // Argument(s) (...)
        );
    }
};
```

PYBIND11_OVERRIDE_PURE(ret_type, cname, fn, ...)

Macro for pure virtual functions, this function is identical to *PYBIND11_OVERRIDE*, except that it throws if no override can be found.

PYBIND11_OVERRIDE_NAME(ret_type, cname, name, fn, ...)

Macro to populate the virtual method in the trampoline class. This macro tries to look up a method named 'fn' from the Python side, deals with the *Global Interpreter Lock (GIL)* and necessary argument conversions to call this method and return the appropriate type. See *Overriding virtual functions in Python* for more information. This macro should be used when the method name in C is not the same as the method name in Python. For example with `__str__`.

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

PYBIND11_OVERRIDE_PURE_NAME(ret_type, cname, name, fn, ...)

Macro for pure virtual functions, this function is identical to *PYBIND11_OVERRIDE_NAME*, except that it throws if no override can be found.

template<class **T**>
function **get_override**(const *T* *this_ptr, const char *name)

Try to retrieve a python method by the provided name from the instance pointed to by the this_ptr.

This_ptr
The pointer to the object the overridden method should be retrieved for. This should be the first non-trampoline class encountered in the inheritance chain.

Name
The name of the overridden Python method to retrieve.

Returns
The Python method by this name from the object or an empty function wrapper.

19.10 Exceptions

class **error_already_set** : public std::exception

Fetch and hold an error which was already set in Python. An instance of this is typically thrown to propagate python-side errors back through C++ which can either be caught manually or else falls back to the function dispatcher (which then raises the captured error back to python).

Public Functions

error_already_set()

Fetches the current Python exception (using PyErr_Fetch()), which will clear the current Python error indicator.

const char ***what**() const noexcept override

The *what()* result is built lazily on demand. WARNING: This member function needs to acquire the Python GIL. This can lead to crashes (undefined behavior) if the Python interpreter is finalizing.

inline void **restore**()

Restores the currently-held Python error (which will clear the Python error indicator first if already set). NOTE: This member function will always restore the normalized exception, which may or may not be the original Python exception. WARNING: The GIL must be held when this member function is called!

inline void **discard_as_unraisable**(*object* err_context)

If it is impossible to raise the currently-held error, such as in a destructor, we can write it out using Python's unraisable hook (`sys.unraisablehook`). The error context should be some object whose `repr()` helps identify the location of the error. Python already knows the type and value of the error, so there is no need to repeat that.

inline void **discard_as_unraisable**(const char *err_context)

An alternate version of *discard_as_unraisable()*, where a string provides information on the location of the error. For example, `__func__` could be helpful. WARNING: The GIL must be held when this member function is called!

inline bool **matches**(*handle* exc) const

Check if the currently trapped error type matches the given Python exception class (or a subclass thereof). May also be passed a tuple to search for any exception class matches in the given tuple.

class **builtin_exception** : public std::runtime_error

C++ bindings of builtin Python exceptions.

Subclassed by attribute_error, buffer_error, cast_error, import_error, index_error, key_error, reference_cast_error, stop_iteration, type_error, value_error

Public Functions

virtual void **set_error**() const = 0

Set the error using the Python C API.

19.11 Literals

namespace **literals**

CHAPTER

TWENTY

CMAKE HELPERS

Pybind11 can be used with `add_subdirectory(extern/pybind11)`, or from an install with `find_package(pybind11 CONFIG)`. The interface provided in either case is functionally identical.

20.1 pybind11Config.cmake

20.1.1 Exported variables

This module sets the following variables in your project:

pybind11_FOUND
: true if pybind11 and all required components found on the system

pybind11_VERSION
: pybind11 version in format Major.Minor.Release

pybind11_VERSION_TYPE
: pybind11 version type (`dev*` or empty for a release)

pybind11_INCLUDE_DIRS
: Directories where pybind11 and python headers are located.

pybind11_INCLUDE_DIR
: Directory where pybind11 headers are located.

pybind11_SRC_DIR
: Directory where the library sources for the opt-in precompiled mode are located (used by `pybind11_precompile`).

pybind11_DEFINITIONS
: Definitions necessary to use pybind11, namely USING_pybind11.

pybind11_LIBRARIES
: Compile flags and python libraries (as needed) to link against.

pybind11_LIBRARY
: Empty.

Available components: None

20.1.2 Exported targets

If pybind11 is found, this module defines the following `IMPORTED` interface library targets:

pybind11::module
for extension modules.

pybind11::embed
for embedding the Python interpreter.

Python headers, libraries (as needed by platform), and the C++ standard are attached to the target.

Advanced targets are also supplied - these are primary for users building complex applications, and they are available in all modes:

pybind11::headers
Just the pybind11 headers and minimum compile requirements.

pybind11::pybind11
Python headers too.

pybind11::python_link_helper
Just the "linking" part of `pybind11:module`, for CMake < 3.15.

pybind11::thin_lto
An alternative to `INTERPROCEDURAL_OPTIMIZATION`.

pybind11::lto
An alternative to `INTERPROCEDURAL_OPTIMIZATION` (also avoids thin LTO on clang).

pybind11::windows_extras
Adds bigobj and mp for MSVC.

20.1.3 Modes

There are two modes provided; classic, which is built on the old Python discovery packages in CMake, or the new FindPython mode, which uses FindPython from 3.12+ forward (3.15+ _highly_ recommended). If you set the minimum or maximum version of CMake to 3.27+, then FindPython is the default (since FindPythonInterp/FindPythonLibs has been removed via policy `CMP0148`).

Starting in pybind11 3.0, the new mode is the default, but we provide backward compatible names (`PYTHON_*` vs. the new `Python_*` names). Set the mode explicitly to avoid the compatibility defines. You can specify this mode explicitly by setting `PYBIND11_FINDPYTHON` to `COMPAT`, but if you are changing your CMakeLists anyway, please just use the `ON` mode. A future release may default to `ON`.

New FindPython mode

To activate this mode, either call `find_package(Python COMPONENTS Interpreter Development)` before finding this package, or set the `PYBIND11_FINDPYTHON` variable to ON. In this mode, you can either use the basic targets, or use the FindPython tools:

```
find_package(Python COMPONENTS Interpreter Development)
find_package(pybind11 CONFIG)

# pybind11 method:
pybind11_add_module(MyModule1 src1.cpp)
```

(continues on next page)

(continued from previous page)

```
# Python method:
Python_add_library(MyModule2 src2.cpp)
target_link_libraries(MyModule2 PUBLIC pybind11::headers)
set_target_properties(MyModule2 PROPERTIES
                                INTERPROCEDURAL_OPTIMIZATION ON
                                CXX_VISIBILITY_PRESET hidden
                                VISIBILITY_INLINES_HIDDEN ON)
```

If you build targets yourself, you may be interested in stripping the output for reduced size; this is the one other feature that the helper function gives you.

Classic mode

Set PythonLibsNew variables to influence python detection and CMAKE_CXX_STANDARD to influence standard setting.

```
find_package(pybind11 CONFIG REQUIRED)

# Create an extension module
add_library(mylib MODULE main.cpp)
target_link_libraries(mylib PUBLIC pybind11::module)

# Or embed the Python interpreter into an executable
add_executable(myexe main.cpp)
target_link_libraries(myexe PUBLIC pybind11::embed)
```

20.1.4 Hints

The following variables can be set to guide the search for this package:

pybind11_DIR
: CMake variable, set to directory containing this Config file.

CMAKE_PREFIX_PATH
: CMake variable, set to root directory of this package.

PATH
: Environment variable, set to bin directory of this package.

CMAKE_DISABLE_FIND_PACKAGE_pybind11
: CMake variable, disables `find_package(pybind11)` when not `REQUIRED`, perhaps to force internal build.

20.1.5 Commands

pybind11_add_module

This module defines the following commands to assist with creating Python modules:

```
pybind11_add_module(<target>
  [STATIC|SHARED|MODULE]
  [THIN_LTO] [OPT_SIZE] [NO_EXTRAS] [WITHOUT_SOABI]
```

(continues on next page)

(continued from previous page)

```
  [PRECOMPILE|NO_PRECOMPILE]
  <files>...
  )
```

Add a module and setup all helpers. You can select the type of the library; the default is `MODULE`. There are several options:

OPT_SIZE
: Optimize for size, even if the `CMAKE_BUILD_TYPE` is not `MinSizeRel`.

THIN_LTO
: Use thin LTO instead of regular if there's a choice (pybind11's selection is disabled if `CMAKE_INTERPROCEDURAL_OPTIMIZATIONS` is set).

WITHOUT_SOABI
: Disable the SOABI component (`PYBIND11_FINDPYTHON` mode only).

NO_EXTRAS
: Disable all extras, exit immediately after making the module.

PRECOMPILE
: Link the target against the `pybind11::precompiled` static library (created on first use); `NO_PRECOMPILE` opts a target out when the `PYBIND11_PRECOMPILE` variable enables it globally.

pybind11_precompile

```
pybind11_precompile()
```

Create the `pybind11::precompiled` static library from the shipped sources (once per build tree). `pybind11_add_module(... PRECOMPILE)` calls this for you; call it directly to link `pybind11::precompiled` into your own targets.

pybind11_strip

```
pybind11_strip(<target>)
```

Strip a target after building it (linux/macOS), called by `pybind11_add_module`.

pybind11_extension

```
pybind11_extension(<target>)
```

Sets the Python extension name correctly for Python on your platform, called by `pybind11_add_module`.

pybind11_find_import(module)

```
pybind11_find_import(<module> [VERSION <number>] [REQUIRED] [QUIET])
```

See if a module is installed. Use the registered name (the one on PyPI). You can specify a `VERSION`, and you can specify `REQUIRED` or `QUIET`. Only available if `NOPYTHON` mode is not active. Sets `module_VERSION` and `module_FOUND`. Caches the result once a valid install is found.

20.1.6 Suggested usage

Using `find_package` with version info is not recommended except for release versions.

```
find_package(pybind11 CONFIG)
find_package(pybind11 2.9 EXACT CONFIG REQUIRED)
```

BIBLIOGRAPHY

[scikit_build_example] https://github.com/pybind/scikit_build_example

[cmake_example] https://github.com/pybind/cmake_example

[python_example] https://github.com/pybind/python_example

[Ccache] https://ccache.dev

[setuptools_scm] https://github.com/pypa/setuptools_scm

[cppimport] https://github.com/tbenthompson/cppimport

[binder] http://cppbinder.readthedocs.io/en/latest/about.html

[AutoWIG] https://github.com/StatisKit/AutoWIG

[semiwrap] https://semiwrap.readthedocs.io

[litgen] https://pthom.github.io/litgen

INDEX

A

add_ostream_redirect (*C++ function*), 217
anyset (*C++ class*), 208
arg (*C++ struct*), 213
arg::arg (*C++ function*), 213
arg::flag_noconvert (*C++ member*), 214
arg::flag_none (*C++ member*), 214
arg::name (*C++ member*), 214
arg::noconvert (*C++ function*), 213
arg::none (*C++ function*), 213
arg::operator= (*C++ function*), 213
arg_v (*C++ struct*), 214
arg_v::arg_v (*C++ function*), 214
arg_v::descr (*C++ member*), 214
arg_v::noconvert (*C++ function*), 214
arg_v::none (*C++ function*), 214
arg_v::type (*C++ member*), 214
arg_v::value (*C++ member*), 214
Args (*C++ class*), 208
args (*C++ class*), 208
args_are_all_keyword_or_ds (*C++ function*), 206
arithmetic (*C++ struct*), 212
as_unsigned (*C++ function*), 206

B

base (*C++ struct*), 212
bool_ (*C++ class*), 208
buffer (*C++ class*), 209
buffer_protocol (*C++ struct*), 212
builtin_exception (*C++ class*), 220
builtin_exception::set_error (*C++ function*), 221
bytearray (*C++ class*), 207
bytes (*C++ class*), 207

C

call_guard (*C++ struct*), 213
call_guard::type (*C++ struct*), 213
call_guard<> (*C++ struct*), 213
capsule (*C++ class*), 208
custom_type_setup (*C++ struct*), 212

D

delattr (*C++ function*), 218
dict (*C++ class*), 208
doc (*C++ struct*), 211
dynamic_attr (*C++ struct*), 212

E

ellipsis (*C++ class*), 208
error_already_set (*C++ class*), 220
error_already_set::discard_as_unraisable (*C++ function*), 220
error_already_set::error_already_set (*C++ function*), 220
error_already_set::matches (*C++ function*), 220
error_already_set::restore (*C++ function*), 220
error_already_set::what (*C++ function*), 220

F

finalize_interpreter (*C++ function*), 215
float_ (*C++ class*), 208
frozenset (*C++ class*), 209
function (*C++ class*), 209

G

get_override (*C++ function*), 219
getattr (*C++ function*), 218
globals (*C++ function*), 218

H

handle (*C++ class*), 203
handle::cast (*C++ function*), 204
handle::dec_ref (*C++ function*), 204
handle::handle (*C++ function*), 204
handle::inc_ref (*C++ function*), 204
handle::operator bool (*C++ function*), 204
handle::operator== (*C++ function*), 204
handle::ptr (*C++ function*), 204
hasattr (*C++ function*), 218
hash (*C++ function*), 218

I

int_ (*C++ class*), 208

is_final (C++ struct), 211
is_method (C++ struct), 211
is_operator (C++ struct), 211
is_setter (C++ struct), 211
isinstance (C++ function), 218
iter (C++ function), 218
iterable (C++ class), 207
iterator (C++ class), 206
iterator::sentinel (C++ function), 207

K

keep_alive (C++ struct), 212
kw_only (C++ struct), 214
KWArgs (C++ class), 208
kwargs (C++ class), 208

L

len (C++ function), 218
len_hint (C++ function), 218
list (C++ class), 208
literals (C++ type), 221

M

make_iterator (C++ function), 210
make_key_iterator (C++ function), 210
make_tuple (C++ function), 210
make_value_iterator (C++ function), 210
memoryview (C++ class), 209
memoryview::from_buffer (C++ function), 209
memoryview::from_memory (C++ function), 209
memoryview::memoryview (C++ function), 209
metaclass (C++ struct), 212
metaclass::metaclass (C++ function), 212
module_ (C++ class), 205
module_::add_object (C++ function), 206
module_::create_extension_module (C++ function), 206
module_::def (C++ function), 205
module_::def_submodule (C++ function), 205
module_::import (C++ function), 206
module_::module_ (C++ function), 205
module_::reload (C++ function), 206
module_local (C++ struct), 212
multiple_inheritance (C++ struct), 212

N

name (C++ struct), 211
none (C++ class), 207

O

object (C++ class), 204
object::~object (C++ function), 205
object::object (C++ function), 205
object::release (C++ function), 205
object_api (C++ class), 202
object_api::annotations (C++ function), 203
object_api::attr (C++ function), 202
object_api::attr_with_type_hint (C++ function), 202
object_api::begin (C++ function), 202
object_api::contains (C++ function), 203
object_api::doc (C++ function), 203
object_api::end (C++ function), 202
object_api::equal (C++ function), 203
object_api::is (C++ function), 203
object_api::is_none (C++ function), 203
object_api::operator() (C++ function), 203
object_api::operator* (C++ function), 202
object_api::operator[] (C++ function), 202
object_api::ref_count (C++ function), 203

P

pos_only (C++ struct), 214
prepend (C++ struct), 212
PYBIND11_EMBEDDED_MODULE (C macro), 215
PYBIND11_MODULE (C macro), 201
PYBIND11_OVERRIDE (C macro), 218
PYBIND11_OVERRIDE_NAME (C macro), 219
PYBIND11_OVERRIDE_PURE (C macro), 219
PYBIND11_OVERRIDE_PURE_NAME (C macro), 219

R

reinterpret_borrow (C++ function), 205
reinterpret_steal (C++ function), 205
release_gil_before_calling_cpp_dtor (C++ struct), 212
repr (C++ function), 218

S

scope (C++ struct), 211
scoped_estream_redirect (C++ class), 217
scoped_interpreter (C++ class), 216
scoped_ostream_redirect (C++ class), 216
sequence (C++ class), 208
set (C++ class), 208
setattr (C++ function), 218
sibling (C++ struct), 211
slice (C++ class), 208
staticmethod (C++ class), 209
str (C++ class), 207
str::str (C++ function), 207

T

tuple (C++ class), 208
type (C++ class), 207
type::handle_of (C++ function), 207

type::of (*C++ function*), 207

W

weakref (*C++ class*), 208