nanobind Documentation

Wenzel Jakob

Dec 08, 2024

CONTENTS

nanobind is a small binding library that exposes C++ types in Python and vice versa. It is reminiscent of Boost.Python and pybind11 and uses near-identical syntax. In contrast to these existing tools, nanobind is *more efficient*: bindings compile in a shorter amount of time, produce smaller binaries, and have better runtime performance.

More concretely, *benchmarks* show up to **~4× faster** compile time, **~5× smaller** binaries, and **~10× lower** runtime overheads compared to pybind11. nanobind also outperforms Cython in important metrics (**3-12×** binary size reduction, **1.6-4×** compilation time reduction, similar runtime performance).

Dependencies

nanobinds depends on

- **Python 3.8+** or **PyPy 7.3.10+** (the *3.8* and *3.9* PyPy flavors are supported, though there are some limitations).
- **CMake 3.15+**.
- **A C++17 compiler**: Clang 8+, GCC 8+, MSVC2019+, and the CUDA NVCC compiler are officially supported. Others (MinGW, Cygwin, Intel, ..) may work as well but will not receive support.

How to cite this project?

Please use the following BibTeX template to cite nanobind in scientific discourse:

```
@misc{nanobind,
   author = {Wenzel Jakob},
   year = {2022},
   note = {https://github.com/wjakob/nanobind},
   title = {nanobind: tiny and efficient C++/Python bindings}
}
```

The nanobind logo was designed by AndoTwin Studio. High-resolution version are available here (light) and here (dark).

CHAPTER

ONE

CHANGELOG

nanobind uses a semantic versioning policy for its API. It also has a separate ABI version that is *not* subject to semantic versioning.

The ABI version is relevant whenever a type binding from one extension module should be visible in another nanobind-based extension module. In this case, both modules must use the same nanobind ABI version, or they will be isolated from each other. Releases that don't explicitly mention an ABI version below inherit that of the preceding release.

1.1 Version 2.4.0 (Dec 6, 2024)

- Added a function annotation *nb::call_policy<Policy>()* which supports custom function wrapping logic, calling `Policy::precall()` before the bound function and `Policy::postcall()` after. This is a low-level interface intended for advanced users. The precall and postcall hooks are able to observe the Python objects forming the function arguments and return value, and the precall hook can change the arguments. See the linked documentation for more details, important caveats, and an example policy. (PR #767)
- *nb::make_iterator* now accepts its iterator arguments by value, rather than by forwarding reference, in order to eliminate the hazard of storing a dangling C++ iterator reference in the returned Python iterator object. (PR #788)
- The `std::variant` type_caster now does two passes when converting from Python. The first pass is done without implicit conversions. This fixes an issue where `std::variant<U, T>` might cast a Python object wrapping a `T` to a `U` if there is an implicit conversion available from `T` to `U`. (issue #769)
- Restored support for constructing types with an overloaded `__new__` that takes no arguments, which regressed with the constructor vector call acceleration that was added in nanobind 2.2.0. (issue #786)
- Bindings for augmented assignment operators (as generated, for example, by `.def(nb::self += nb::self)`) now return the same object in Python in the typical case where the C++ operator returns a reference to `*this`. Previously, after `a += b`, `a` would be replaced with a copy. (PR #803)
- Added an overload to *nb::isinstance* which tests if a Python object is an instance of a Python class. This is in addition to the existing overload, which tests if a Python object is an instance of a bound C++ class. (PR #805).
- Added support for overriding static properties, such as those defined using `def_prop_ro_static`, in subclasses. Previously this would fail with an error. (PR #806).
- Other minor fixes and improvements. (PRs #771, #772, #748, and #753)

1.2 Version 2.3.0

There is no version 2.3.0 due to a deployment mishap.

1.3 Version 2.2.0 (October 3, 2024)

- nanobind can now target free-threaded Python, which replaces the Global Interpreter Lock (GIL) with a fine-grained locking scheme (see PEP 703) to better leverage multi-core parallelism. A *separate documentation page* explains this in detail (PRs #695, #720)
- nanobind has always used PEP 590 vector calls to efficiently dispatch calls to function and method bindings, but it lacked the ability to do so for constructors (e.g., `MyType(arg1, arg2, ...)`).

  Version 2.2.0 adds this missing part, which accelerates object construction by up to a factor of 2×. The difference is especially pronounced when passing keyword arguments to constructors. Note that this improvement only applies to Python version 3.9 and newer (PR #706, commits #e24d7f, #0acecb, #77f910, #2c96d5).
- A new *nb::is_flag()* annotation in *nb::enum_<T>()* produces enumeration bindings deriving from `enum.Flag`, which enables bit-wise combination using compatible operators (&, |, ^, and ~). Further combining the annotation with *nb::is_arithmetic()* creates enumerations deriving from `enum.IntFlag`. (PRs #599, #688, #688, #727, #732)
- A refactor of *nb::ndarray<...>* was an opportunity to realize three usability improvements:

  1. The constructor used to return new nd-arrays from C++ now considers all template arguments:

     - **Memory order**: *c_contig*, *f_contig*.
     - **Shape**: *nb::shape<3, 4, 5>*, etc.
     - **Device type**: *nb::device::cpu*, *nb::device::cuda*, etc.
     - **Framework**: *nb::numpy*, *nb::pytorch*, etc.
     - **Data type**: `uint64_t`, `std::complex<double>`, etc.

     Previously, only the **framework** and **data type** annotations were taken into account when returning nd-arrays, while all of them were examined when *accepting* arrays during overload resolution. This inconsistency was a repeated source of confusion among users.

     To give an example, the following now works out of the box without the need to redundantly specify the shape and strides to the `Array` constructor below:

     ```
     using Array = nb::ndarray<float, nb::numpy, nb::shape<4, 4>, nb::f_contig>;

     struct Matrix4f {
         float m[4][4];
         Array data() { return Array(m); }
     };

     nb::class_<Matrix4f>(m, "Matrix4f")
         .def("data", &Matrix4f::data, nb::rv_policy::reference_internal);
     ```

  2. A new nd-array *.cast()* method forces the immediate creation of a Python object with the specified target framework and return value policy, while preserving the type signature in return values. This is useful to *return temporaries (e.g. stack-allocated memory)* from functions.
  3. Added a new and more general mechanism `nanobind::detail::dtype_traits<T>` to declare custom ndarray data types like `float16` or `bfloat16`. The old interface (`nanobind::ndarray_traits<T>`) still exists but is deprecated and will be removed in the next major release. See the *documentation* for details.

  There are two minor but potentially breaking changes:

1. The nd-array type caster now interprets the *nb::rv_policy::automatic_reference* return value policy analogously to the *nb::rv_policy::automatic*, which means that it references a memory region when the user specifies an `owner`, and it otherwise copies. This makes it safe to use the *nb::cast()* and *nb::ndarray::cast()* functions that use this policy as a default.
2. The *nb::any_contig* memory order annotation, which previously did nothing, now accepts C- or F-contiguous arrays and rejects non-contiguous ones.

For further details on the nd-array changes, see PR #721, For further details on the nd-array changes, see PR #742, and commit 4647ef.

- The NVIDIA CUDA compiler (`nvcc`) is now explicitly supported and included in nanobind's CI test suite (PR #710).
- Added support for return value policy customization to the type casters of `Eigen::Ref<...>` and `Eigen::Map<...>` (commit 67316e).
- Added the *bytearray* wrapper type. (PR #654)
- The *nb::ellipsis* type now renders as `...` when used in *nb::typed<...>* (PR #705).
- The *nb::sig("...")* annotation now supports inline type parameter lists such as `def first[T](l: Sequence[T]) -> T` (PR #704).
- Fixed implicit conversion of complex nd-arrays. (issue #709)
- Casting via *nb::cast* can now specify an owner object for use with the *nb::rv_policy::reference_internal* return value policy (PR #667).
- The `std::optional<T>` type caster is now implemented in such a way that it can also accommodate non-STL frameworks, such as Boost, Abseil, etc. (PR #675)
- ABI version 15.
- Minor fixes and improvements (PRs #703, #724, #723, #722, #715, #696, #693, commit 75d259).

1.4 Version 2.1.0 (Aug 11, 2024)

- Temporary workaround for a internal compiler error in version 17.10 of the MSVC compiler. This workaround will be removed once fixed versions are deployed on GitHub actions. (issue #613, commit f2438b).
- nanobind no longer prevents casting to a C++ container of pointers `T*` where `T` is a type with a user-defined type caster if the caster seems to operate by extracting a `T*` from the Python object rather than a `T`. This change was prompted by discussion #605.
- Switched nanobind wheel generation from setuptools to scikit-build-core (PR #618).
- Improved handling of `const`-ness in *nb::ndarray* (PR #491).
- Keyword argument annotations are now properly supported with *nb::new_*, passed in the same way they would be with *nb::init*. (issue #668)
- Ability to use *nb::cast* to create object with the *nb::rv_policy::reference_internal* return value policy (PR #667).
- Enable `char` type caster to produce `'\0'` (PR #661).
- Added `.def_static()` member to *nb::enum_*, which had been lost in a redesign of the enumeration implementation in nanobind version 2.0.0. (commit 38990e).
- Fixes for two minor sources of memory leaks (PR #595, #647).
- The nd-array wrapper *nb::ndarray* now properly handles CuPy arrays (#594).
- Added *nb::hash()*, a wrapper for the Python `hash()` function (commit 91fafa5).
- Various minor `stubgen` fixes (PRs #667, #658, #632, #620, #592).

1.5 Version 2.0.0 (May 23, 2024)

The 2.0.0 release of nanobind is entirely dedicated to *types*[1]! The project has always advertised seamless Python ↔ C++ interoperability, and this release tries to bring a similar level of interoperability to static type checkers like MyPy, PyRight, PyType, and editors with interactive autocompletion like Visual Studio Code, PyCharm, and many other LSP-compatible IDEs.

This required work on three fronts:

1. **Stub generation**: the above tools all analyze Python code statically without running it. Because the import mechanism of compiled extensions depends the Python interpreter, these tools weren't able to inspect the contents of nanobind-based extensions.

   The usual solution involves writing stubs that expose the module contents to static analysis tools. However, writing stubs by hand is tedious and error-prone.

   This release adds tooling to automatically extract stubs from existing extensions. The process is fully integrated into the CMake-based build system and explained in a *new documentation section*.

2. **Better default annotations**: once stubs were available, this revealed the next problem: the default nanobind-provided function and class signatures were too rudimentary, and this led to a user poor experience.

   The release therefore improves many builtin type caster so that they produce more accurate type signatures. For example, the STL `std::vector<T>` caster now renders as `collections.abc.Sequence[T]` in stubs when it is used as an *input*, and `list[T]` when it is used as part of a return value. The *`nb::make_*_iterator()`* family of functions return typed iterators, etc.

3. **Advanced customization**: a subset of the type signatures in larger binding projects will generally require further customization. The features listed below aim to enable precisely this:

   - In Python, many built-in types are *generic* and can be *parameterized* (e.g., `list[int]`). The *`nb::typed<T, Ts...>`* wrapper enables such parameterization within C++ (for example, the `int`-specialized list would be written as `nb::typed<nb::list, int>`). *Read more*.
   - The opposite is also possible: passing *`nb::is_generic()`* to the class binding constructor

     ```
     nb::class_<MyType>(m, "MyType", nb::is_generic())
     ```

     produces a *generic* type that can be parameterized in Python (e.g. `MyType[int]`). *Read more*.
   - The *`nb::sig`* annotation overrides the signature of a function or method, e.g.:

     ```
     m.def("f", &f, nb::sig("def f(x: Foo = Foo(0)) -> None"), "docstring");
     ```

     Each binding of an overloaded function can be customized separately. This feature can be used to add decorators or control how default arguments are rendered. *Read more*.
   - The *`nb::sig`* annotation can also override *class signatures* in generated stubs. Stubs often take certain liberties in deviating somewhat from the precise type signature of the underlying implementation. For example, the following annotation adds an abstract base class advertising that the class implements a typed iterator.

     ```
     using IntVec = std::vector<int>;

     nb::class_<IntVec>(m, "IntVec",
                        nb::sig("class IntVec(collections.abc.Iterable[int])"));
     ```

     Nanobind can't subclass Python types, hence this declaration is technically untrue. On the flipside, such a declaration can assist static checkers and improve auto-completion in visual IDEs. This is fine since these tools only perform a static analysis and never import the actual extension. *Read more*.

[1] The author of this library had somewhat of a revelation after switching to a new editor and experiencing the benefits of interactive Python code completion and type checking for the first time. This experience also showed how nanobind-based extension were previously a second-class citizen in this typed world, prompting the changes in this release.

- The *nb::for_setter* and *nb::for_getter* annotations enable passing function binding annotations (e.g., signature overrides) specifically to the setter or the getter part of a property.
- The *nb::arg("name")* argument annotation (and `"name"_a` shorthand) now have a *.sig("signature")* member to control how a default value is rendered in the stubs and docstrings. This provides more targeted control compared to overriding the entire function signature.
- Finally, nanobind's stub generator supports *pattern files* containing custom stub replacement rules. This catch-all solution addresses the needs of advanced binding projects, for which the above list of features may still not be sufficient.

Most importantly, it was possible to support these improvements with minimal changes to the core parts of nanobind.

These release breaks API and ABI compatibility, requiring a new major version according to SemVer. The following changes are noteworthy:

- The *nb::enum_<T>()* binding declaration is now a wrapper that creates either a `enum.Enum` or `enum.IntEnum`-derived type. Previously, nanobind relied on a custom enumeration base class that was a frequent source of friction for users.

  This change may break code that casts entries to integers, which now only works for arithmetic (`enum.IntEnum`-derived) enumerations. Replace `int(my_enum_entry)` with `my_enum_entry.value` to work around the issue.

- The *nb::bind_vector<T>()* and *nb::bind_map<T>()* interfaces were found to be severely flawed since element access (`__getitem__`) created views into the internal state of the STL type that were not stable across subsequent modifications.

  This could lead to unexpected changes to array elements and undefined behavior when the underlying storage was reallocated (i.e., use-after-free).

  nanobind 2.0.0 improves these types so that they are safe to use, but this means that element access must now copy by default, potentially making them less convenient. The documentation of *nb::bind_vector<T>()* discusses the issue at length and presents alternative solutions.

- The functions *nb::make_iterator()*, *nb::make_value_iterator()* and *nb::make_key_iterator()* suffer from the same issue as *nb::bind_vector()* explained above.

  nanobind 2.0.0 improves these operations so that they are safe to use, but this means that iterator access must now copy by default, potentially making them less convenient. The documentation of *nb::make_iterator()* discusses the issue and presents alternative solutions.

- The `nb::raw_doc` annotation was found to be too inflexible and was removed in this version.

- The `nb::typed` wrapper listed above actually already existed in previous nanobind versions but was awkward to use, as it required the user to provide a custom type formatter. This release makes the interface more convenient.

- The `nb::any` placeholder to specify an unconstrained *nb::ndarray* axis was removed. This name was given to a new wrapper type `nb::any` indicating `typing.Any`-typed values.

  All use of `nb::any` in existing code must be replaced with `-1` (for example, `nb::shape<3, nb::any, 4>` → `nb::shape<3, -1, 4>`).

- *Keyword-only arguments* are now supported, and can be indicated using the new *nb::kw_only()* function annotation. (PR #448).

- nanobind classes now permit overriding `__new__`, in order to support C++ singletons, caches, and other types that expose factory functions rather than ordinary constructors. Read the section on *customizing Python object creation* for more details. (PR #473).

- When binding methods on a class `T`, nanobind will now produce a Python function that expects a self argument of type `T`. Previously, it would use the type of the member pointer to determine the Python function signature, which could be a base of `T`, which would create problems if nanobind did not know about that base. (PR #471).

- nanobind can now handle keyword arguments that are not interned, which avoids spurious `TypeError` exceptions in constructs like `fn(**pickle.loads(...))`. The speed of normal function calls (which generally do have interned keyword arguments) should be unaffected. (PR #469).
- The `owner=nb::handle()` default value of the *nb::ndarray* constructor was removed since it was bug-prone. You now have to specify the owner explicitly. The previous default (`nb::handle()`) continues to be a valid argument.
- There have been some changes to the API for type casters in order to avoid undefined behavior in certain cases. (PR #549).
  - Type casters that implement custom cast operators must now define a member function template `can_cast<T>()`, which returns false if `operator cast_t<T>()` would raise an exception and true otherwise. `can_cast<T>()` will be called only after a successful call to `from_python()`, and might not be called at all if the caller of `operator cast_t<T>()` can cope with a raised exception. (Users of the `NB_TYPE_CASTER()` convenience macro need not worry about this; it produces cast operators that never raise exceptions, and therefore provides a `can_cast<T>()` that always returns true.)
  - Many type casters for container types (`std::vector<T>`, `std::optional<T>`, etc) implement their `from_python()` methods by delegating to another, "inner" type caster (`T` in these examples) that is allocated on the stack inside `from_python()`. Container casters implemented in this way should make two changes in order to take advantage of the new safety features:
    * Wrap your `flags` (received as an argument of the outer caster's `from_python` method) in `flags_for_local_caster<T>()` before passing them to `inner_caster.from_python()`. This allows nanobind to prevent some casts that would produce dangling pointers or references.
    * If `inner_caster.from_python()` succeeds, then also verify `inner_caster.template can_cast<T>()` before you execute `inner_caster.operator cast_t<T>()`. A failure of `can_cast()` should be treated the same as a failure of `from_python()`. This avoids the possibility of an exception being raised through the noexcept `load_python()` method, which would crash the interpreter.

  The previous `cast_flags::none_disallowed` flag has been removed; it existed to avoid one particular source of exceptions from a cast operator, but `can_cast<T>()` now handles that problem more generally.
- ABI version 14.

Footnote

1.6 Version 1.9.2 (Feb 23, 2024)

- Nanobind instances can now be *made weak-referenceable* by specifying the *nb::is_weak_referenceable* tag in the *nb::class_<..>* constructor. (PR #335, commits fc7709, 3562f6).
- Added a *nb::bool_* wrapper type. (PR #382, commit 90dfba).
- Ensure that the GIL is held when releasing *nb::ndarray*. (issue #377, commit a968e8).
- *nb::try_cast()* no longer crashes the interpreter when attempting to cast a Python `None` to a C++ type that was bound using *nb::class_<...>*. Previously this would raise an exception from the cast operator, which would result in a call to `std::terminate()` because *try_cast()* is declared `noexcept`. (PR #386).
- Fixed memory corruption in a PyPy-specific code path in *nb::module_::def_submodule()* (commit 21eaff).
- Don't implicitly convert complex to non-complex nd-arrays. (issue #364, commit ea2569).
- Support for non-assignable types in the `std::optional<T>` type caster (PR #358, commit 9c9b64).
- nanobind no longer assumes that docstrings provided to function binding (of type `const char *`) have an infinite lifetime and it makes copy. (issue #393, commit b3b6f4).

- Don't pass compiler flags if they may be unsupported by the used compiler. This gets NVCC to work out of the box (that said, this change does not elevate NVCC to being an *officially* supported compiler). (issue #383, commit a307ea).
- Added a CMake install target to the nanobind build system. (PR #356, commit 6bde65, commit 978dbb, commit f5d8de).
- ABI version 13.
- Minor fixes and improvements.

1.7 Version 1.9.0-1.9.1 (Feb 18, 2024)

Releases withdrawn because of a regression. The associated changes are listed above in the 1.9.2 release notes.

1.8 Version 1.8.0 (Nov 2, 2023)

- nanobind now considers two C++ `std::type_info` instances to be equal when their mangled names match. The previously used pointer comparison was fast but fragile and often caused multi-part extensions to not recognize each other's types. This version introduces a two-level caching scheme (search by pointer, then by name) to fix such problems once and for all, while avoiding the cost of constantly comparing very long mangled names. (commit b515b1).
- Fixed casting of complex-valued constant *nb::ndarray<T>* instances. (PR #338, commit ba8c7f).
- Added a type caster for `std::nullopt_t` (PR #350).
- Added the missing C++ → Python portion of the type caster for `Eigen::Ref<..>` (PR #334).
- Minor fixes and improvements.
- ABI version 12.

1.9 Version 1.7.0 (Oct 19, 2023)

1.9.1 New features

- The nd-array class *nb::ndarray<T>* now supports complex-valued `T` (e.g., `std::complex<double>`). For this, the header file `nanobind/stl/complex.h` must be included. (PR #319, commit 6cbd13).
- Added the function *nb::del()*, which takes an arbitrary accessor object as input and tries to delete the associated entry. The C++ statement

  ```
  nb::del(o[key]);
  ```

  is equivalent to `del o[key]` in Python. (commit 4dd745).
- Exposed several convenience functions for raising exceptions as public API: *nb::raise*, *nb::raise_type_error*, and *nb::raise_python_error*. (commit 0b7f3b).
- Added *nb::globals()*. (PR #311, commit f0a9eb).
- The `char*` type caster now accepts `nullptr` and converts it into a Python `None` object. (PR #318, commit 30a6ba).
- Added the function *nb::is_alive()*, which returns `false` when nanobind was destructed by Python (e.g., during interpreter shutdown) making further use of the API illegal. (commit b431d0).
- Minor fixes and improvements.
- ABI version 11.

1.9.2 Bugfixes

- The behavior of the *nb::keep_alive<Nurse, Patient>* function binding annotation was changed as follows: when the function call requires the implicit conversion of an argument, the lifetime constraint now applies to the newly produced argument instead of the original object. The change was rolled into a minor release since the former behavior is arguably undesirable and dangerous. (commit 9d4b2e).
- STL type casters previously raised an exception when casting a Python container containing a `None` element into a C++ container that was not able to represent `nullptr` (e.g., `std::vector<T>` instead of `std::vector<T*>`). However, this exception was raised in a context where exceptions were not allowed, causing the process to be `abort()`-ed, which is very bad. This issue is now fixed, and such conversions are refused. (PR #318, commits d1ad3b and 5f25ae).
- The STL sequence casters (`std::vector<T>`, etc.) now refuse to unpack `str` and `bytes` objects analogous to pybind11. (commit 7e4a88).

1.10 Version 1.6.2 (Oct 3, 2023)

- Added a missing include file used by the new intrusive reference counting sample implementation from v1.6.0. (commit 31d115).

1.11 Version 1.6.1 (Oct 2, 2023)

- Added missing namespace declaration to the *ref* intrusive reference counting RAII helper class added in version 1.6.0. (commit 3ba352).

1.12 Version 1.6.0 (Oct 2, 2023)

1.12.1 New features

- Several *nb::ndarray<..>* improvements:
  1. CPU loops involving nanobind nd-arrays weren't getting properly vectorized. This release of nanobind adds *views*, which provide an efficient abstraction that enables better code generation. See the documentation section on *array views* for details. (commit 8f602e).
  2. Added support for nonstandard arithmetic types (e.g., `__int128` or `__fp16`) in nd-arrays. See the *documentation section* for details. (commit 49eab2).
  3. Shape constraints like *nb::shape<nb::any, nb::any, nb::any>* are tedious to write. Now, there is a shorter form: *nb::ndim<3>*. (commit 1350a5).
  4. Added an explicit constructor that can be used to add or remove nd-array constraints. (commit a1ac207).
- Added the wrapper class *nb::weakref*. (commit 78887f).
- Added the methods *nb::dict::contains()* and *nb::mapping::contains()* to the Python type wrappers. (commit 64d87a).
- Added *nb::exec()* and *nb:eval()*. (PR #299).
- Added a type caster for `std::complex<T>`. (PR #292, commit dcbed4).
- Added an officially supported sample implementation of *intrusive reference counting* via the *intrusive_counter* *intrusive_base*, and *ref* classes. (commit 3fa1af).

1.12.2 Bugfixes

- Fixed a serious issue involving combinations of bound types (e.g., `T`) and type casters (e.g., `std::vector<T>`), where nanobind was too aggressive in its use of *move semantics*. Calling a bound function from Python taking such a list (e.g., `f([t1, t2, ..])`) would destruct `t1, t2, ..` if the type `T` exposed a move constructor, which is highly non-intuitive and no longer happens as of this fix.

  Further investigation also revealed inefficiencies in the previous implementation where moves were actually possible but not done (e.g., for functions taking an STL vector by value). Some binding projects may see speedups as a consequence of this change. (issue #307, commit 122015).

1.13 Version 1.5.2 (Aug 24, 2023)

- Fixed a severe issue with inheritance of the `Py_TPFLAGS_HAVE_GC` flag affecting classes that derive from other classes with a *nb::dynamic_attr* annotation. (issue #279, commit dbedad).
- Implicit conversion of nd-arrays to conform to contiguity constraints such as *c_contig* and *f_contig* previously failed in some cases that are now addressed. (issue #278 commit ed929b).

1.14 Version 1.5.1 (Aug 23, 2023)

- Fixed serious reference counting issue introduced in nanobind version 1.5.0, which affected the functions *python_error::traceback()* and *python_error::what()*, causing undefined behavior via use-after-free. Also addressed an unrelated minor UB sanitizer warning. (issue #277, commits 30d30c and c48b18).
- Extended the internal data structure tag so that it isolates different MSVC versions from each other (they are often not ABI compatible, see pybind11 issue #4779). This means that nanobind 1.5.1 effectively bumps the ABI version to "10.5" when compiling for MSVC, and the internals will be isolated from extensions built with nanobind v1.5.0 or older. (commit c7f3cd).
- Incorporated fixes so that nanobind works with PyPy 3.10. (commits fb5508 and 2ed10a).
- Fixed type caster for `std::vector<bool>`. (PR #256).
- Fixed compilation in debug mode on MSVC. (PR #253).

1.15 Version 1.5.0 (Aug 7, 2023)

- Support for creating *chained exceptions* via the *nb::raise_from()* and *nb::chain_error()* functions. (commits 041520 and beb699).
- Many improvements to the handling of return value policies in *nb::ndarray<..>* to avoid unnecessary copies. (commit ffd22b, a79575, and 6f0c3f).
- The *nb::ndarray<..>* class now has an additional convenience constructor that takes the shape and (optionally) strides using `std::initializer_list`. (commit de1117).
- Added a non-throwing function *nb::try_cast()* as an alternative to *nb::cast()*. (commit 6ca852).
- The `nb::list` and `nb::tuple` default constructors now construct an empty list/tuple instead of an invalid null-initialized handle. (commit 506185)
- New low-level interface for wrapping existing C++ instances via *nb::inst_take_ownership()* *nb::inst_reference()*. Also added convenience functions to replace the contents of an instance with that of another. *nb::inst_replace_copy()* along with *nb::inst_replace_move()* (commit 1c462d).
- Added a low-level abstraction around *nb::type_get_slot()* around `PyType_GetSlot`, but with more consistent behavior across Python versions. (commit d555e9).
- The *nb::list::append()* method now performs perfect forwarding. (commit 2219d0).

- Inference of `automatic*` return value policy was entirely moved to the base C++ class type caster. (commit 1ff9df).
- Switch to the new Python 3.12 error status API if available. (commit 36751c).
- Various minor fixes and improvements.
- ABI version 10.

1.16 Version 1.4.0 (June 8, 2023)

- Improved the efficiency of the function dispatch loop. (PR #227).
- Significant improvements to the Eigen type casters (generalized stride handling to avoid unnecessary copies, support for conversion via `nb::cast()`, many refinements to the `Eigen::Ref<T>` interface). (PR #215).
- Added a `NB_DOMAIN` parameter to *`nanobind_add_module()`* which can isolate extensions from each other to avoid binding clashes. See the associated *FAQ entry* for details. (commit 977119).
- Reduced the severity of nanobind encountering a duplicate type binding (commits f3b0e6, and 2c9124).
- Support for pickling/unpickling nanobind objects. (commit 59843e).
- ABI version 9.

1.17 Version 1.3.2 (June 2, 2023)

- Fixed compilation on 32 bit processors (only `i686` tested so far). (PR #224).
- Fixed compilation on PyPy 3.8. (commit cd8135).
- Reduced binary bloat of musllinux wheels. (commit f52513).

1.18 Version 1.3.1 (May 31, 2023)

- CMake build system improvements for stable ABI wheel generation. (PR #222).

1.19 Version 1.3.0 (May 31, 2023)

This is a big release. The sections below cover added features, efficiency improvements, and miscellaneous fixes and improvements.

1.19.1 New features

- nanobind now supports binding types that inherit from `std::enable_shared_from_this<T>`. See the *advanced section on object ownership* for more details. (PR #212).
- Added a type caster between Python `datetime/timedelta` objects and C++ `std::chrono::duration/std::chrono::time_point`, ported from pybind11. (PR #175).
- The *`nb::ndarray<..>`* class can now use the buffer protocol to receive and return arrays representing read-only memory. (PR #217).
- Added *`nb::python_error::discard_as_unraisable()`* as a wrapper around `PyErr_WriteUnraisable()`. (PR #175).

1.19.2 Efficiency improvements:

- Reduced the per-instance overhead of nanobind by 1 pointer and simplified the internal hash table types to crunch `libnanobind`. (commit de018d).
- Supplemental type data specified via *`nb::supplement<T>()`* is now stored directly within the type object instead of being referenced through an indirection. (commit d82ca9).
- Reduced the number of exception-related exports to further crunch `libnanobind`. (commit 763962).
- Reduced the size of nanobind type objects by 5 pointers. (PR #194, #195, and commit d82ca9).
- Internal nanobind types (`nb_type`, `nb_static_property`, `nb_ndarray`) are now constructed on demand. This reduces the size of the `libnanobind` component in static (`NB_STATIC`) builds when those features are not used. (commits 95e45a, 375083, and e033c8).
- Added a small function cache to improve code generation in limited API builds. (commit f0f4aa).
- Refined compiler and linker flags across platforms to ensure compact binaries especially in `NB_STATIC` builds. (commit 5ead9f)
- nanobind enums now take advantage of *supplemental data* to improve the speed of object and name lookups. Note that this prevents use of `nb::supplement<T>()` with enums for other purposes. (PR #195).

1.19.3 Miscellaneous fixes and improvements

- Use the new PEP-697 interface to access data in type objects when compiling stable ABI3 wheels. This improves forward compatibility (the Python team may at some point significantly refactor the layout and internals of type objects). (PR #211):
- Added introspection attributes `__self__` and `__func__` to nanobind bound methods, to make them more like regular Python bound methods. Fixed a bug where `some_obj.method.__call__()` would behave differently than `some_obj.method()`. (PR #216).
- Updated the implementation of *`nb::enum_`* so it does not take advantage of any private nanobind type details. As a side effect, the construct `nb::class_<T>(..., nb::is_enum(...))` is no longer permitted; use `nb::enum_<T>(...)` instead. (PR #195).
- Added the `nb::type_slots_callback` class binding annotation, similar to `nb::type_slots` but allowing more dynamic choices. (PR #195).
- nanobind type objects now treat attributes specially whose names begin with `@`. These attributes can be set once, but not rebound or deleted. This safeguard allows a borrowed reference to the attribute value to be safely stashed in the type supplement, allowing arbitrary Python data associated with the type to be accessed without a dictionary lookup while keeping this data visible to the garbage collector. (PR #195).
- Fixed surprising behavior in enumeration comparisons and arithmetic (PR #207):
  - Enum equality comparisons (== and !=) now can only be true if both operands have the same enum type, or if one is an enum and the other is an `int`. This resolves some confusing results and ensures that enumerators of different types have a distinct identity, which is important if they're being put into the same set or used as keys in the same dictionary. All of the following were previously true but will now evaluate as false:
    - `FooEnum(1) == BarEnum(1)`
    - `FooEnum(1) == 1.2`
    - `FooEnum(1) == "1"`
  - Enum ordering comparisons (<, <=, >=, >) and arithmetic operations (when using the *`is_arithmetic`* annotation) now require that any non-enum operand be a Python number (an object that defines `__int__`, `__float__`, and/or `__index__`) and will avoid truncating non-integer operands to integers. Note that unlike with equality comparisons, ordering and arithmetic operations *do* still permit two operands that are enums of different types. Some examples of changed behavior:

    * `FooEnum(1) < 1.2` is now true (used to be false)
    * `FooEnum(2) * 1.5` is now 3.0 (used to be 2)
    * `FooEnum(3) - "2"` now raises an exception (used to be 1)
  - Enum comparisons and arithmetic operations with unsupported types now return `NotImplemented` rather than raising an exception. This means equality comparisons such as `some_enum == None` will return unequal rather than failing; order comparisons such as `some_enum < None` will still fail, but now with a more informative error.
- ABI version 8.

1.20 Version 1.2.0 (April 24, 2023)

- Improvements to the internal C++ → Python instance map data structure to improve performance and address type confusion when returning previously registered instances. (commit 716354, discussion 189).
- Added up-to-date nanobind benchmarks on Linux including comparisons to Cython. (commit 834cf3 and 39e163).
- Removed the superfluous `nb_enum` metaclass. (commit 9c1985).
- Fixed a corner case that prevented `nb::cast<char>` from working. (commit 9ae320).

1.21 Version 1.1.1 (April 6, 2023)

- Added documentation on packaging and distributing nanobind modules. (commit 0715b2).
- Made the conversion *`handle::operator bool()`* explicit. (PR #173).
- Support *`nb::typed<..>`* in return values. (PR #174).
- Tweaks to definitions in `nb_types.h` to improve compatibility with further C++ compilers (that said, there is no change about the official set of supported compilers). (commit b8bd10)

1.22 Version 1.1.0 (April 5, 2023)

- Added *`size`*, *`shape_ptr`*, *`stride_ptr`* members to to the *`nb::ndarray<..>`* class. (PR #161).
- Allow macros in *`NB_MODULE(..)`* name parameter. (PR #168).
- The *`nb::ndarray<..>`* interface is more tolerant when converting Python (PyTorch/NumPy/..) arrays with a size-0 dimension that have mismatched strides. (PR #162).
- Removed the `<anonymous>` label from docstrings of anonymous functions, which caused issues in MyPy. (PR #172).
- Fixed an issue in the propagation of return value policies that broke user-provided/custom policies in properties (PR #170).
- The Eigen interface now converts 1x1 matrices to 1x1 NumPy arrays instead of scalars. (commit 445781).
- The `nanobind` package now has a simple command line interface. (commit d5ccc8).

1.23 Version 1.0.0 (March 28, 2023)

- Nanobind now has a logo. (commit b65d31).
- Fixed a subtle issue involving function/method properties and the IPython command line interface. (PR #151).
- Added a boolean type to the *nb::ndarray<..>* interface. (PR #150).
- Minor fixes and improvements.

1.24 Version 0.3.1 (March 8, 2023)

- Added a type caster for `std::filesystem::path`. (PR #138 and commit 0b05cd).
- Fixed technical issues involving implicit conversions (commits 022935 and 5aefe3) and construction of type hierarchies with custom garbage collection hooks (commit 022935).
- Re-enabled the 'chained fixups' linker optimization for recent macOS deployment targets. (commit 2f29ec).

1.25 Version 0.3.0 (March 8, 2023)

- Botched release, replaced by 0.3.1 on the same day.

1.26 Version 0.2.0 (March 3, 2023)

- Nanobind now features documentation on readthedocs.
- The documentation process revealed a number of inconsistencies in the *class_<T>::def** naming scheme. nanobind will from now on use the following shortened and more logical interface:

Type	method
Methods & constructors	*.def()*
Fields	*.def_ro()*, *.def_rw()*
Properties	*.def_prop_ro()*, *.def_prop_rw()*
Static methods	*.def_static()*
Static fields	*.def_ro_static()*, *.def_rw_static()*
Static properties	*.def_prop_ro_static()*, *.def_prop_rw_static()*

  Compatibility wrappers with deprecation warnings were also added to help port existing code. They will be removed when nanobind reaches version 1.0. (commits cb0dc3 and b5ed96)
- The `nb::tensor<..>` class has been renamed to *nb::ndarray<..>*, and it is now located in a different header file (`nanobind/ndarray.h`). A compatibility wrappers with a deprecation warning was retained in the original header file. It will be removed when nanobind reaches version 1.0. (commit a6ab8b).
- Dropped the first two arguments of the *NB_OVERRIDE_*()* macros that turned out to be unnecessary in nanobind. (commit 22bc21).
- Added casters for dense matrix/array types from the Eigen library. (PR #120).
- Added casters for sparse matrix/array types from the Eigen library. (PR #126).
- Implemented *nb::bind_vector<T>()* analogous to similar functionality in pybind11. (commit f2df8a).
- Implemented *nb::bind_map<T>()* analogous to similar functionality in pybind11. (PR #114).

- nanobind now *automatically downcasts* polymorphic objects in return values analogous to pybind11. (commit cab96a).
- nanobind now supports *tag-based polymorphism*. (commit 6ade94).
- Updated tuple/list iterator to satisfy the `std::forward_iterator` concept. (PR #117).
- Fixed issues with non-writeable tensors in NumPy. (commit 25cc3c).
- Removed use of some C++20 features from the codebase. This now makes it possible to use nanobind on Visual Studio 2017 and GCC 7.3.1 (used on RHEL 7). (PR #115).
- Added the *nb::typed<...>* wrapper to override the type signature of an argument in a bound function in the generated docstring. (commit b3404c4).
- Added an *nb::implicit_convertible<A, B>()* function analogous to the one in pybind11. (commit aba4af).
- Updated *nb::make_*_iterator<..>()* so that it returns references of elements, not copies. (commit 8916f5).
- Changed the CMake build system so that the library component (`libnanobind`) is now compiled statically by default. (commit 8418a4).
- Switched shared library linking on macOS back to a two-level namespace. (commit fe4965).
- Various minor fixes and improvements.
- ABI version 7.

1.27 Version 0.1.0 (January 3, 2023)

- Allow nanobind methods on non-nanobind) classes. (PR #104).
- Fix dangling `tp_members` pointer in type initialization. (PR #99).
- Added a runtime setting to suppress leak warnings. (PR #109).
- Added the ability to hash `nb::enum_<..>` instances (PR #106).
- Fixed the signature of `nb::enum_<..>::export_values()`. (commit 714d17).
- Double-check GIL status when performing reference counting operations in debug mode. (commit a1b245).
- Fixed a reference leak that occurred when module initialization fails. (commit adfa9e).
- Improved robustness of `nb::tensor<..>` caster. (commit 633672).
- Upgraded the internally used `tsl::robin_map<>` hash table to address a rare overflow issue discovered in this codebase. (commit 3b81b1).
- Various minor fixes and improvements.
- ABI version 6.

1.28 Version 0.0.9 (Nov 23, 2022)

- PyPy 7.3.10 or newer is now supported subject to certain limitations. (commits f935f93 and b343bbd).
- Three changes that reduce the binary size and improve runtime performance of binding libraries. (commits 07b4e1fc, 9a803796, and cba4d285).
- Fixed a reference leak in `python_error::what()` (commit 61393ad).
- Adopted a new policy for function type annotations. (commit c855c90).

- Improved the effectiveness of link-time-optimization when building extension modules with the `NB_STATIC` flag. This leads to smaller binaries. (commit f64d2b9).
- Nanobind now relies on standard mechanisms to inherit the `tp_traverse` and `tp_clear` type slots instead of trying to reimplement the underlying CPython logic (commit efa09a6b).
- Moved nanobind internal data structures from `builtins` to Python interpreter state dictionary. (issue #96, commit ca23da7).
- Various minor fixes and improvements.

1.29 Version 0.0.8 (Oct 27, 2022)

- Caster for `std::array<..>`. (commit be34b16).
- Caster for `std::set<..>` and `std::unordered_set` (PR #87).
- Ported `nb::make[_key_,_value]_iterator()` from pybind11. (commit 34d0be1).
- Caster for untyped `void *` pointers. (commit 6455fff).
- Exploit move constructors in `nb::class_<T>::def_readwrite()` and `nb::class_<T>::def_readwrite_static()` (PR #94).
- Redesign of the `std::function<>` caster to enable cyclic garbage collector traversal through inter-language callbacks (PR #95).
- New interface for specifying custom type slots during Python type construction. (commit 38ba18a).
- Fixed potential undefined behavior related to `nb_func` garbage collection by Python's cyclic garbage collector. (commit 662e1b9).
- Added a workaround for spurious reference leak warnings caused by other extension modules in conjunction with `typing.py` (commit 5e11e80).
- Various minor fixes and improvements.
- ABI version 5.

1.30 Version 0.0.7 (Oct 14, 2022)

- Fixed a regression involving function docstrings in `pydoc`. (commit 384f4a).

1.31 Version 0.0.6 (Oct 14, 2022)

- Fixed undefined behavior that could lead to crashes when nanobind types were freed. (commit 39266e).
- Refactored nanobind so that it works with `Py_LIMITED_API` (PR #37).
- Dynamic instance attributes (PR #38).
- Intrusive pointer support (PR #43).
- Byte string support (PR #62).
- Casters for `std::variant<..>` and `std::optional<..>` (PR #67).
- Casters for `std::map<..>` and `std::unordered_map<..>` (PR #73).
- Caster for `std::string_view<..>` (PR #68).
- Custom exception support (commit 41b7da).
- Register nanobind functions with Python's cyclic garbage collector (PR #86).

- Various minor fixes and improvements.
- ABI version 3.

1.32 Version 0.0.5 (May 13, 2022)

- Enumeration export.
- Implicit number conversion for numpy scalars.
- Various minor fixes and improvements.

1.33 Version 0.0.4 (May 13, 2022)

- Botched release, replaced by 0.0.5 on the same day.

1.34 Version 0.0.3 (Apr 14, 2022)

- DLPack support.
- Iterators for various Python type wrappers.
- Low-level interface to instance creation.
- Docstring generation improvements.
- Various minor fixes and improvements.

1.35 Version 0.0.2 (Mar 10, 2022)

- Initial release of the nanobind codebase.
- ABI version 1.

1.36 Version 0.0.1 (Feb 21, 2022)

- Placeholder package on PyPI.

CHAPTER

TWO

WHY ANOTHER BINDING LIBRARY?

I started the pybind11 project back in 2015 to generate better C++/Python bindings for a project I had been working on. Thanks to many amazing contributions by others, pybind11 has since become a core dependency of software used across the world including flagship projects like PyTorch and Tensorflow. Every day, it is downloaded over 400'000 times. Hundreds of contributed extensions and generalizations address use cases of this diverse audience. However, all of this success also came with costs: the complexity of the library grew tremendously, which had a negative impact on efficiency.

Curiously, the situation now is reminiscent of 2015: binding generation with existing tools (Boost.Python, pybind11) is slow and produces enormous binaries with overheads on runtime performance. At the same time, key improvements in C++17 and Python 3.8 provide opportunities for drastic simplifications. Therefore, I am starting *another* binding project. This time, the scope is intentionally limited so that this doesn't turn into an endless cycle.

2.1 So what is different?

nanobind is highly related to pybind11 and inherits most of its conventions and syntax. The main difference is a change in philosophy: pybind11 must deal with *all of C++* to bind legacy codebases, while nanobind targets a smaller C++ subset. *The codebase has to adapt to the binding tool and not the other way around*, which allows nanobind to be simpler and faster. Pull requests with extensions and generalizations to handle subtle fringe cases were welcomed in pybind11, but they will likely be rejected in this project.

An overview of removed features is provided in a *separate section*. Besides feature removal, the rewrite was also an opportunity to address *long-standing performance issues* and add a number of *major quality-of-life improvements* and *smaller features*.

2.2 Performance improvements

The *benchmark section* evaluates the impact of the following performance improvements:

- **Compact objects**: C++ objects are now co-located with the Python object whenever possible (less pointer chasing compared to pybind11). The per-instance overhead for wrapping a C++ type into a Python object shrinks by a factor of 2.3x. (pybind11: 56 bytes, nanobind: 24 bytes.)
- **Compact functions**: C++ function binding information is now co-located with the Python function object (less pointer chasing).
- **Compact types**: C++ type binding information is now co-located with the Python type object (less pointer chasing, fewer hashtable lookups).
- **Fast hash table**: nanobind upgrades several important internal associative data structures that previously used `std::unordered_map` to a more efficient alternative (tsl::robin_map, which is included as a git submodule).
- **Vector calls**: function calls from/to Python are realized using PEP 590 vector calls, which gives a nice speed boost. The main function dispatch loop no longer allocates heap memory.

- **Library component**: pybind11 was designed as a header-only library, which is generally a good thing because it simplifies the compilation workflow. However, one major downside of this is that a large amount of redundant code has to be compiled in each binding file (e.g., the function dispatch loop and all of the related internal data structures). nanobind compiles a separate shared or static support library ("*libnanobind*") and links it against the binding code to avoid redundant compilation. The CMake interface *nanobind_add_module()* fully automates these extra steps.
- **Smaller headers**: `#include <pybind11/pybind11.h>` pulls in a large portion of the STL (about 2.1 MiB of headers with Clang and libc++). nanobind minimizes STL usage to avoid this problem. Type casters even for for basic types like `std::string` require an explicit opt-in by including an extra header file (e.g. `#include <nanobind/stl/string.h>`).
- **Simpler compilation**: pybind11 was dependent on *link time optimization* (LTO) to produce reasonably-sized bindings, which makes linking a build time bottleneck. With nanobind's split into a precompiled library and minimal metatemplating, LTO is no longer crucial and can be skipped.
- **Free-threading**: Python 3.13+ supports a free-threaded mode that removes the *Global Interpreter Lock* (GIL). Both pybind11 and nanobind support free-threading as of recently. When comparing the two, nanobind provides better multi-core scaling using a localized locking scheme. In pybind11, lock contention on a central `internals` data structure used in every binding operation becomes a bottleneck in practice.
- **Lifetime management**: nanobind maintains efficient internal data structures for lifetime management (needed for *nb::keep_alive*, *nb::rv_policy::reference_internal*, the `std::shared_ptr` interface, etc.). With these changes, bound types no longer need to be weak-referenceable, which saves a pointer per instance.

2.3 Major additions

nanobind includes a number of quality-of-life improvements for developers:

- **N-dimensional arrays**: nanobind can exchange data with modern array programming frameworks. It uses either DLPack or the buffer protocol to achieve *zero-copy* CPU/GPU array exchange with frameworks like NumPy, PyTorch, TensorFlow, JAX, etc. See the *section on n-dimensional arrays* for details.
- **Stable ABI**: nanobind can target Python's stable ABI interface starting with Python 3.12. This means that extension modules will be compatible with future version of Python without having to compile separate binaries per interpreter. That vision is still relatively far out, however: it will require Python 3.12+ to be widely deployed.
- **Stub generation**: nanobind ships with a custom *stub generator* and CMake integration to automatically create high quality stubs as part of the build process. Stubs make compiled extension code compatible with visual autocomplete in editors like Visual Studio Code and static type checkers like MyPy, PyRight and PyType.
- **Smart pointers, ownership, etc.**: corner cases in pybind11 related to smart/unique pointers and callbacks could lead to undefined behavior. A later pybind11 redesign (`smart_holder`) was able to address these problems, but this came at the cost of further increased runtime overheads. The object ownership model of nanobind avoids this undefined behavior without penalizing runtime performance.
- **Leak warnings**: When the Python interpreter shuts down, nanobind reports instance, type, and function leaks related to bindings, which is useful for tracking down reference counting issues. If these warnings are undesired, call *nb::set_leak_warnings(false)*. nanobind also fully deletes its internal data structures when the Python interpreter terminates, which avoids memory leak reports in tools like *valgrind*.
- **Better docstrings**: pybind11 pre-renders docstrings while the binding code runs. In other words, every call to `.def(...)` to bind a function immediately creates the underlying docstring. When a function takes a C++ type as parameter that is not yet registered in pybind11, the docstring will include a C++ type name (e.g. `std::vector<int, std::allocator<int>>`), which can look rather ugly. pybind11 binding declarations must be carefully arranged to work around this issue.

  nanobind avoids the issue altogether by not pre-rendering docstrings: they are created on the fly when queried. nanobind also has improved out-of-the-box compatibility with documentation generation tools like

Sphinx.

- **Low-level API**: nanobind exposes an optional low-level API to provide fine-grained control over diverse aspects including *instance creation*, *type creation*, and it can store *supplemental data* in types. The low-level API provides a useful escape hatch to pursue advanced projects that were not foreseen in the design of this library.

2.4 Minor additions

The following lists minor-but-useful additions relative to pybind11.

- **Finding Python objects associated with a C++ instance**: In addition to all of the return value policies supported by pybind11, nanobind provides one additional policy named `nb::rv_policy::none` that *only* succeeds when the return value is already a known/registered Python object. In other words, this policy will never attempt to move, copy, or reference a C++ instance by constructing a new Python object.

  The new `nb::find()` function encapsulates this behavior. It resembles `nb::cast()` in the sense that it returns the Python object associated with a C++ instance. But while `nb::cast()` will create that Python object if it doesn't yet exist, `nb::find()` will return a `nullptr` object. This function is useful to interface with Python's *cyclic garbage collector*.

- **Parameterized wrappers**: The `nb::handle_t<T>` type behaves just like the `nb::handle` class and wraps a `PyObject *` pointer. However, when binding a function that takes such an argument, nanobind will only call the associated function overload when the underlying Python object wraps a C++ instance of type `T`.

  Similarly, the `nb::type_object_t<T>` type behaves just like the `nb::type_object` class and wraps a `PyTypeObject *` pointer. However, when binding a function that takes such an argument, nanobind will only call the associated function overload when the underlying Python type object is a subtype of the C++ type `T`.

  Finally, the `nb::typed<T, Ts...>` annotation can parameterize any other type. The feature exists to improve the expressiveness of type signatures (e.g., to turn `list` into `list[int]`). Note, however, that nanobind does not perform additional runtime checks in this case. Please see the section on *parameterizing generics* for further details.

- **Signature overrides**: it may sometimes be necessary to tweak the type signature of a class or function to provide richer type information to static type checkers like MyPy or PyRight. In such cases, specify the `nb::sig` attribute to override the default nanobind-provided signature.

  For example, the following function signature annotation creates an overload that should only be called with an 1-valued integer literal. While the function also includes a runtime check, a static type checker can now ensure that this error condition cannot possibly be triggered by a given piece of code.

```
m.def("f",
      [](int arg) {
          if (arg != 1)
             nb::raise("invalid input");
          return arg;
      },
      nb::sig("def f(arg: typing.Literal[1], /) -> int"));
```

  Please see the section on *customizing function signatures* and *class signatures* for further details.

2.5 TLDR

My recommendation is that current pybind11 users look into migrating to nanobind. Fixing all the long-standing issues in pybind11 (see above list) would require a substantial redesign and years of careful work by a team of C++ metaprogramming experts. At the same time, changing anything in pybind11 is extremely hard because of the large number of downstream users and their requirements on API/ABI stability. I personally don't have the time and energy to fix pybind11 and have moved my focus to this project.

CHAPTER

THREE

BENCHMARKS

Note: **TL;DR**: nanobind bindings compile up to **~4× faster** and produce **~5× smaller** binaries with **~10× lower** runtime overheads compared to pybind11.

nanobind also outperforms Cython in important metrics (**3-12×** binary size reduction, **1.6-4×** compilation time reduction, similar runtime performance).

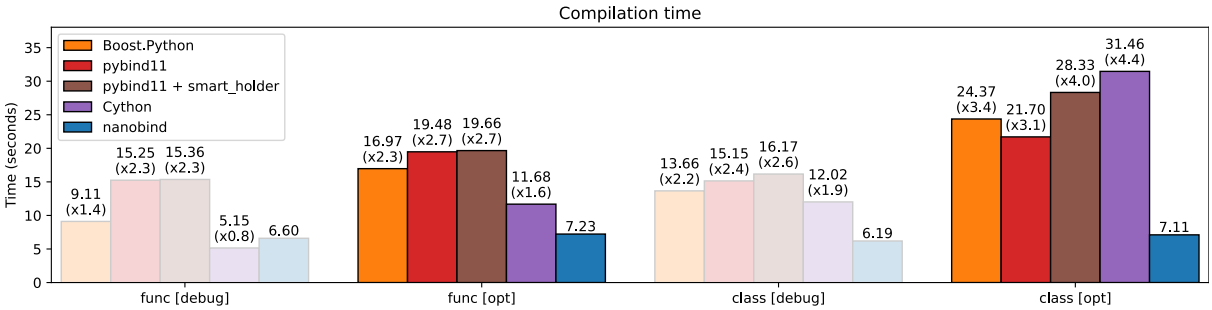
The following experiments analyze the performance of a large function-heavy (`func`) and class-heavy (`class`) binding microbenchmark compiled using Boost.Python, Cython, pybind11. The `pybind11 + smart_holder` results below refer to a special branch that addresses long-standing issues related to holder types in pybind11.

Each experiment is shown twice: light gray `[debug]` columns provide data for a debug build, and `[opt]` shows a size-optimized build that is representative of a deployment scenario. The former is included to show that nanobind performance is also good during a typical development workflow.

A comparison with cppyy, which uses dynamic compilation, is also shown later. Details on the experimental setup can be found *below*.

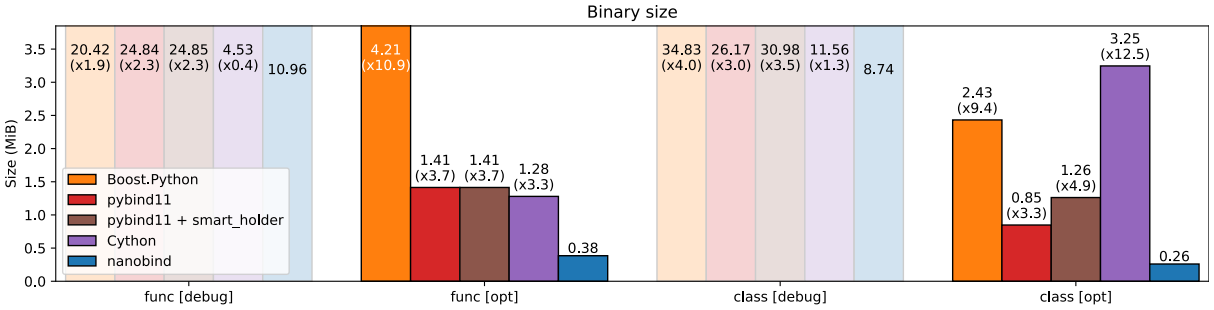
3.1 Compilation time

The first plot contrasts the compilation time, where "*number* ×" annotations denote the amount of time spent relative to nanobind. As shown below, nanobind achieves a **~2.7-4.4× improvement** compared to pybind11 and a **1.6-4.4x improvement** compared to Cython.



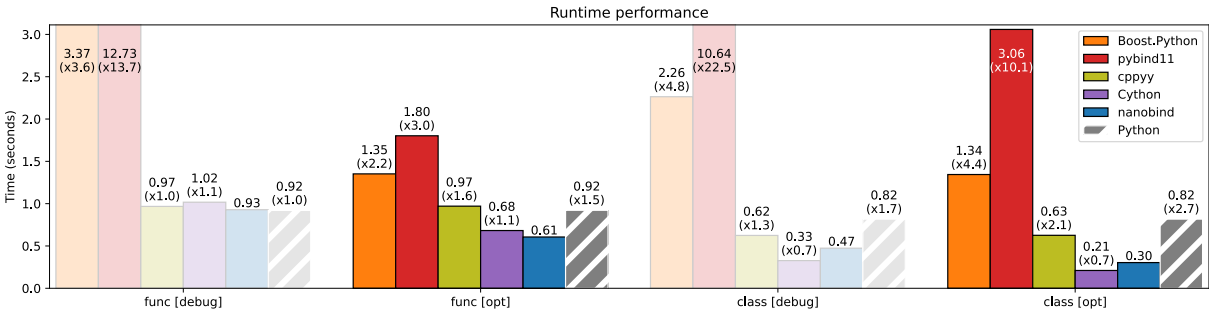
3.2 Binary size

The extremely large size of generated binaries has been a persistent problem of many prior binding libraries. nanobind significantly improves this metric in size-optimized builds. There is a **~11× improvement** compared to Boost.Python, a **3-5× improvement** compared to pybind11, and a **3-12× improvement** compared to Cython.



3.3 Performance

The last experiment compares the runtime performance overheads by calling a bound function many times in a loop. Here, it is also interesting to additionally compare against cppyy (green bar) and a pure Python implementation that runs bytecode without binding overheads (hatched gray bar). The `smart_holder` branch of pybind11 is not explicitly listed since its runtime performance matches the base version.



This data shows that the overhead of calling a nanobind function is lower than that of an equivalent function call done within CPython. The functions benchmarked here don't perform CPU-intensive work, so this this mainly measures the overheads of performing a function call, boxing/unboxing arguments and return values, etc.

The difference to pybind11 is **significant**: a **~3× improvement** for simple functions, and an **~10× improvement** when classes are being passed around. Complexities in pybind11 related to overload resolution, multiple inheritance, and holders are the main reasons for this difference. Those features were either simplified or completely removed in nanobind.

The runtime performance of Cython and nanobind are similar (Cython leads in one experiment and trails in another one). Cython generates specialized binding code for every function and class, which is highly redundant (long compile times, large binaries) but can also be beneficial for performance.

Finally, there is a **~1.6-2.1× improvement** in both experiments compared to cppyy (please ignore the two `[debug]` columns—I did not feel comfortable adjusting the JIT compilation flags; all cppyy bindings are therefore optimized.)

3.4 Discussion

Performance improvements compared to pybind11 are the result of optimizations discussed in the *previous section*.

cppyy also achieves good performance in the comparison above. It is based on dynamic parsing of C++ code and *just-in-time* (JIT) compilation of bindings via the LLVM compiler infrastructure. The authors of cppyy report that their tool produces bindings with much lower overheads compared to pybind11, and the above plots show that this is indeed true.

While nanobind retakes the performance lead, there are other qualitative factors make these two tools appropriate to different audiences: cppyy has its origin in CERN's ROOT mega-project and must be highly dynamic to work with that codebase: it can parse header files to generate bindings as needed. cppyy works particularly well together with PyPy and can avoid boxing/unboxing overheads with this combination. The main downside of cppyy is that it depends on Cling/Clang/LLVM that must be deployed on the user's side and then run there. There isn't a way of pre-generating bindings and then shipping just the output of this process.

nanobind is relatively static in comparison: you must tell it which functions to expose via binding declarations. These declarations offer a high degree of flexibility that users will typically use to create bindings that feel *pythonic*. At compile-time, those declarations turn into a sequence of CPython API calls, which produces self-contained bindings that are easy to redistribute via PyPI or elsewhere. Tools like cibuildwheel and scikit-build can fully automate the process of generating *Python wheels* for each target platform. A minimal example project shows how to do this automatically via GitHub Actions.

3.5 Details

The microbenchmark wraps a *large* number of trivial functions that only perform a few additions. The objective of this is to quantify the overhead of bindings on compilation time, binary size, and runtime performance. The function-heavy benchmark (`func_*`) consists of 720 declarations of the form (with permuted integer types)

```
m.def("test_0050", [](uint16_t a, int64_t b, int32_t c, uint64_t d, uint32_t e, float␣
 ↪f) {
    return a+b+c+d+e+f;
});
```

while the latter (`class_*`) does exactly the same computation but packaged up in `structs` with bindings.

```
struct Struct50 {
    uint16_t a; int64_t b; int32_t c; uint64_t d; uint32_t e; float f;
    Struct50(uint16_t a, int64_t b, int32_t c, uint64_t d, uint32_t e, float f)
        : a(a), b(b), c(c), d(d), e(e), f(f) { }
    float sum() const { return a+b+c+d+e+f; }
};

py::class_<Struct50>(m, "Struct50")
    .def(py::init<uint16_t, int64_t, int32_t, uint64_t, uint32_t, float>())
    .def("sum", &Struct50::sum);
```

The code to generate the plots shown above is available here.

Each test was compiled in debug mode (`debug`) and with optimizations (`opt`) that minimize size (i.e., `-Os`). Benchmarking was performed on a AMD Ryzen 9 7950X workstation running Ubuntu 22.04.2 LTS. CPU boost was disabled, and all core clock frequencies were pinned. Reported timings are the median of five runs. Compilation used clang++ 15.0.7 with consistent compilation flags for all experiments (see the referenced notebook file for detail). The used package versions were Python 3.10.6, cppyy 1.12.13, Cython 0.29.28, and nanobind 1.2.0.

CHAPTER

FOUR

PORTING GUIDE

The API of nanobind is *extremely* similar to that of pybind11, which makes porting existing projects easy. Parts of the nanobind documentation are almost verbatim copies pybind11's documentation.

A number of noteworthy changes are documented below.

4.1 Namespace

nanobind types and functions are located in the `nanobind` namespace. The following shorthand alias is recommended and used throughout the documentation:

```
namespace nb = nanobind;
```

4.2 Name changes

The following macros, types, and functions were renamed:

pybind11	nanobind
`PYBIND11_MODULE(..)`	*NB_MODULE(..)*
`PYBIND11_OVERRIDE_*(..)`	*NB_OVERRIDE_*(..)*
`error_already_set`	*python_error*
`type::of<T>()`	*type<T>()*
`type`	*type_object*
`reinterpret_borrow<T>(x)`	*borrow<T>(x)*
`reinterpret_steal<T>(x)`	*steal<T>(x)*
`.def_readwrite(..)`	*.def_rw(..)*
`.def_readonly(..)`	*.def_ro(..)*
`.def_property(..)`	*.def_prop_rw(..)*
`.def_property_readonly(..)`	*.def_prop_ro(..)*
`.def_readwrite_static(..)`	*.def_rw_static(..)*
`.def_readonly_static(..)`	*.def_ro_static(..)*
`.def_property_static(..)`	*.def_prop_rw_static(..)*
`.def_property_readonly_static(..)`	*.def_prop_ro_static(..)*
`register_exception<T>`	*exception<T>*

4.3 None/null arguments

In contrast to pybind11, nanobind does *not* permit `None`-valued function arguments by default. You must enable them explicitly using the *`arg::none()`* argument annotation, e.g.:

```
m.def("func", &func, "arg"_a.none());
```

It is also possible to set a `None` default value, in which case the *`.none()`* annotation can be omitted.

```
m.def("func", &func, "arg"_a = nb::none());
```

`None`-valued arguments are only supported by two of the three parameter passing styles described in the section on *information exchange*. In particular, they are supported by *bindings* and *wrappers*, *but not* by *type casters*.

4.4 Shared pointers and holders

When nanobind instantiates a C++ type within Python, the resulting instance data is stored *within* the created Python object ("`PyObject`"). Alternatively, when an already existing C++ instance is transferred to Python via a function return value and *`rv_policy::reference`*, *`rv_policy::reference_internal`*, or *`rv_policy::take_ownership`*, nanobind creates a smaller wrapper `PyObject` that only stores a pointer to the instance data.

This is *very different* from pybind11, where the instance `PyObject` contained a *holder type* (typically `std::unique_ptr<T>`) storing a pointer to the instance data. Dealing with holders caused inefficiencies and introduced complexity; they were therefore removed in nanobind. This has implications on object ownership, shared ownership, and interactions with C++ shared/unique pointers. The *intermediate* and *advanced* sections on object ownership provide further detail.

The gist is that it is no longer necessary to specify holder types in the type declaration:

pybind11:

```
py::class_<MyType, std::shared_ptr<MyType>>(m, "MyType")
```

nanobind:

```
nb::class_<MyType>(m, "MyType")
```

To bind functions that exchange shared/unique pointers, you must add one or both of the following include directives to your code:

```
#include <nanobind/stl/unique_ptr.h>
#include <nanobind/stl/shared_ptr.h>
```

Binding functions that take `std::unique_ptr<T>` arguments involves some limitations that can be avoided by changing their signatures to `std::unique_ptr<T, nb::deleter<T>>` (*details*).

Use of `std::enable_shared_from_this<T>` is permitted, but since nanobind does not use holder types, an object constructed in Python will typically not have any associated `std::shared_ptr<T>` until it is passed to a C++ function that accepts `std::shared_ptr<T>`. That means a C++ function that accepts a raw `T*` and calls `shared_from_this()` on it might stop working when ported from pybind11 to nanobind. You can solve this problem by always passing such objects across the Python/C++ boundary as `std::shared_ptr<T>` rather than as `T*`. See the *advanced section on object ownership* for more details.

4.5 Custom constructors

In pybind11, custom constructors (i.e. ones that do not already exist in the C++ class) could be specified as a lambda function returning an instance of the desired type.

```
py::class_<MyType>(m, "MyType")
    .def(py::init([](int) { return MyType(...); }));
```

Unfortunately, the implementation of this feature was quite complex and often required further internal calls to the move or copy constructor. nanobind instead reverts to how pybind11 originally implemented this feature using in-place construction (placement new):

```
nb::class_<MyType>(m, "MyType")
    .def("__init__", [](MyType *t) { new (t) MyType(...); });
```

The provided lambda function will be called with a pointer to uninitialized memory that has already been allocated (this memory region is co-located with the Python object for reasons of efficiency). The lambda function can then either run an in-place constructor and return normally (in which case the instance is assumed to be correctly constructed) or fail by raising an exception.

To turn an existing factory function into a constructor, you will need to combine the above pattern with an invocation of the move/copy-constructor, e.g.:

```
nb::class_<MyType>(m, "MyType")
    .def("__init__", [](MyType *t) { new (t) MyType(MyType::create()); });
```

4.6 Implicit conversions

In pybind11, implicit conversions were specified using a follow-up function call. This also works in nanobind, but it is recommended that you already specify them within the constructor declaration:

pybind11:

```
py::class_<MyType>(m, "MyType")
    .def(py::init<MyOtherType>());

py::implicitly_convertible<MyOtherType, MyType>();
```

nanobind:

```
nb::class_<MyType>(m, "MyType")
    .def(nb::init_implicit<MyOtherType>());
```

4.7 Trampoline classes

Trampolines, i.e., polymorphic class implementations that forward virtual function calls to Python, now require an extra *NB_TRAMPOLINE(parent, size)* declaration, where `parent` refers to the parent class and `size` is at least as big as the number of *NB_OVERRIDE_*()* calls. nanobind caches information to enable efficient function dispatch, for which it must know the number of trampoline "slots".

The macro `PYBIND11_OVERRIDE_*(..)` required the base type and return value as the first two arguments. This information is no longer needed in nanobind, and the arguments should be removed in *NB_OVERRIDE_*()*:

An example:

```
struct PyAnimal : Animal {
    NB_TRAMPOLINE(Animal, 1);

    std::string name() const override {
        NB_OVERRIDE(name);
    }
};
```

Trampoline declarations with an insufficient size may eventually trigger a Python `RuntimeError` exception with a descriptive label, e.g.:

```
nanobind::detail::get_trampoline('PyAnimal::what()'): the trampoline ran out of
slots (you will need to increase the value provided to the NB_TRAMPOLINE() macro)
```

4.8 Iterator bindings

Use of the *nb::make_iterator()*, *nb::make_key_iterator()*, and *nb::make_value_iterator()* functions requires including the additional header file `nanobind/make_iterator.h`. The interface of these functions has also slightly changed: all take a Python scope and a name as first and second arguments, which are used to permanently "install" the iterator type (which is created on demand). See the test suite for a worked out example.

4.9 Type casters

The API of custom type casters has changed *significantly*. The following changes are needed:

- `load()` was renamed to `from_python()`. The function now takes an extra `uint8_t flags` parameter (instead `bool convert`, which is now represented by the flag `nb::detail::cast_flags::convert`). A `cleanup_list *` pointer keeps track of Python temporaries that are created by the conversion, and which need to be deallocated after a function call has taken place.

  `flags` and `cleanup` should be passed to any recursive usage of `type_caster::from_python()`. If casting fails due to a Python exception, the function should clear it (`PyErr_Clear()`) and return `false`. If a severe error condition arises that should be reported, use Python warning API calls for this, e.g. `PyErr_WarnFormat()`.

- `cast()` was renamed to `from_cpp()`. The function takes a return value policy (as before) and a `cleanup_list *` pointer. If casting fails due to a Python exception, the function should *leave the error set* (note the asymmetry compared to `from_python()`) and return `nullptr`.

Note that the cleanup list is only available when `from_python()` or `from_cpp()` are called as part of function dispatch, while usage by *nb::cast()* may set `cleanup` to `nullptr` if implicit conversions are not enabled. This case should be handled gracefully by refusing the conversion if the cleanup list is absolutely required.

Type casters may not raise C++ exceptions. Both `from_python()` and `from_cpp()` must be annotated with `noexcept`. Exceptions or failure conditions caused by a conversion should instead be caught *within* the function body and handled as follows:

- `from_python()`: return `false`. That's it. (Failed Python to C++ conversion occur all the time while dispatching calls, causing nanobind to simply move to the next function overload. Dedicated error reporting would add undesirable overheads). In case of a severe internal error, use the CPython warning API (e.g., `PyErr_Warn()`) to notify the user.

- `from_cpp()`: a failure here is more serious, since a return value of a successfully evaluated cannot be converted, causing the call to fail. To provide further detail, use the CPython error API (e.g., `PyErr_Format()`) and return an invalid handle (`return nb::handle();`).

The `std::pair<T1, T2>` type caster (link) may be useful as a starting point of custom implementations.

4.10 Removed features

A number of pybind11 features are unavailable in nanobind. The list below uses the following symbols:

- ○: This removal is a design choice. Use pybind11 if you need this feature.
- •: Unclear, to be discussed.

Removed features include:

- ○ **Multiple inheritance**: this feature was a persistent source of complexity in pybind11 and it is one of the main casualties in creating nanobind.
- ○ **Holders**: nanobind instances co-locate instance data with a Python object instead of accessing it via a holder type. This is a major difference compared to pybind11 and will require changes to binding code that used custom holders (e.g. unique or shared pointers). The *intermediate* and *advanced* sections on object ownership provide further detail.
- ○ **Multi-intepreter, Embedding**: The ability to embed Python in an executable or run several independent Python interpreters in the same process is unsupported. Nanobind caters to bindings only. Multi-interpreter support would require TLS lookups for nanobind data structures, which is undesirable.
- ○ **Function binding annotations**: The `pos_only` argument annotation was removed. However, the same behavior can be achieved by creating unnamed arguments; see the discussion in the section on *keyword-only arguments*.
- ○ **Metaclasses**: creating types with custom metaclasses is unsupported.
- ○ **Module-local bindings**: support was removed (both for types and exceptions).
- ○ **Custom allocation**: C++ classes with an overloaded or deleted `operator new` / `operator delete` are not supported.
- ○ **Compilation**: workarounds for buggy or non-standard-compliant compilers were removed and will not be reintroduced.
- ○ The `options` class for customizing docstring generation was removed.
- ○ The NumPy array class (`py::array`) was removed in exchange for a more powerful alternative (*nb::ndarray<..>*) that additionally supports CPU/GPU tensors produced by various frameworks (NumPy, PyTorch, TensorFlow, JAX, etc.). Its API is not compatible with pybind11, however.
- • Buffer protocol binding (`.def_buffer()`) was removed in favor of *nb::ndarray<..>*.
- • Support for evaluating Python files was removed.

Bullet points marked with • may be reintroduced eventually, but this will need to be done in a careful opt-in manner that does not affect code complexity, binary size, and compilation/runtime performance of basic bindings that don't depend on these features.

CHAPTER

FIVE

FREQUENTLY ASKED QUESTIONS

5.1 Importing my module fails with an `ImportError`

If importing the module fails as shown below, you have not specified a matching module name in *nanobind_add_module()* and *NB_MODULE()*.

```
>>> import my_ext
ImportError: dynamic module does not define module export function (PyInit_my_ext)
```

5.2 Importing fails due to missing `[lib]nanobind.{dylib,so,dll}`

If importing the module fails as shown below, the extension cannot find the `nanobind` shared library component.

```
>>> import my_ext
ImportError: dlopen(my_ext.cpython-311-darwin.so, 0x0002):
Library not loaded: '@rpath/libnanobind.dylib'
```

This is really more of a general C++/CMake/build system issue than one of nanobind specifically. There are two solutions:

1. Build the library component statically by specifying the `NB_STATIC` flag in *nanobind_add_module()* (this is the default starting with nanobind 0.2.0).

2. Ensure that the various shared libraries are installed in the right destination, and that their rpath is set so that they can find each other.

   You can control the build output directory of the shared library component using the following CMake command:

   ```
   set_target_properties(nanobind
     PROPERTIES
     LIBRARY_OUTPUT_DIRECTORY                <path>
     LIBRARY_OUTPUT_DIRECTORY_RELEASE        <path>
     LIBRARY_OUTPUT_DIRECTORY_DEBUG          <path>
     LIBRARY_OUTPUT_DIRECTORY_RELWITHDEBINFO <path>
     LIBRARY_OUTPUT_DIRECTORY_MINSIZEREL     <path>
   )
   ```

   Depending on the flags provided to *nanobind_add_module()*, the shared library component may have a different name following the pattern `nanobind[-abi3][-lto]`.

   The following CMake commands may be useful to adjust the build and install rpath of the extension:

```
set_property(TARGET my_ext APPEND PROPERTY BUILD_RPATH "$<TARGET_FILE_
 ↪DIR:nanobind>")
set_property(TARGET my_ext APPEND PROPERTY INSTALL_RPATH ".. ?? ..")
```

5.3 Why are reference arguments not updated?

Functions like the following example can be exposed in Python, but they won't propagate updates to mutable reference arguments.

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

This isn't specific to builtin types but also applies to STL collections and other types when they are handled using *type casters*. Please read the full section on *information exchange between C++ and Python* to understand the issue and alternatives.

5.4 Why am I getting errors about leaked functions and types?

When the Python interpreter shuts down, it informs nanobind about this using a `Py_AtExit()` callback. If any nanobind-created instances, functions, or types are still alive at this point, then *something went wrong* because they should have been deleted by the garbage collector. Although this does not always indicate a serious problem, the decision was made to have nanobind complain rather noisily about the presence of such leaks.

Other binding tools (e.g., pybind11) are on the opposite of the spectrum: because they never report leaks, it is quite easy to accidentally introduce many of them until a developer eventually realizes that something is very wrong.

Leaks mainly occur for four reasons:

- **Reference counting bugs**. If you write raw Python C API code or use the nanobind wrappers including functions like `Py_[X]INCREF()`, `Py_[X]DECREF()`, *`nb::steal()`*, *`nb::borrow()`*, *`.dec_ref()`*, *`.inc_ref()`* , etc., then incorrect use of such calls can cause a reference to leak that prevents the associated object from being deleted.
- **Reference cycles**. Python's garbage collector frees unused objects that are part of a circular reference chains (e.g., `A->B->C->A`). This requires all types in the cycle to implement the `tp_traverse` *type slot*, and at least one of them to implement the `tp_clear` type slot. See the section on *cyclic garbage collection* for details on how to do this with nanobind.
- **Interactions with other tools that leak references**. Python extension libraries—especially *huge* ones with C library components like PyTorch, Tensorflow, etc., have been observed to leak references to nanobind objects.

  Some of these frameworks cache JIT-compiled functions based on the arguments with which they were called, and such caching schemes could leak references to nanobind types if they aren't cleaned up by the responsible extensions (this is a hypothesis). In this case, the leak would be benign—even so, it should be fixed in the responsible framework so that leak warnings aren't cluttered with flukes and can be more broadly useful.
- **Older Python versions**: Very old Python versions (e.g., 3.8) don't do a good job cleaning up global references when the interpreter shuts down. The following code may leak a reference if it is a top-level statement in a Python file or the REPL.

  ```
  a = my_ext.MyObject()
  ```

  Such a warning is benign and does not indicate an actual leak. It simply highlights a flaws in the interpreter shutdown logic of old Python versions. Wrap your code into a function to address this issue even on such versions:

```
def run():
    a = my_ext.MyObject()
    # ...

if __name__ == '__main__':
    run()
```

- **Exceptions**. Some exceptions such as `AttributeError` have been observed to hold references, e.g. to the object which lacked the desired attribute. If the last exception raised by the program references a nanobind instance, then this may be reported as a leak since Python finalization appears not to release the exception object. See issue #376 for a discussion.

If you find leak warnings to be a nuisance, then you can disable them in the C++ binding code via the *nb::set_leak_warnings()* function.

```
nb::set_leak_warnings(false);
```

This is a *global flag* shared by all nanobind extension libraries in the same ABI domain. If you do so, then please isolate your extension from others by passing the `NB_DOMAIN` parameter to *nanobind_add_module()*.

5.5 Compilation fails with a static assertion mentioning NB_MAKE_OPAQUE()

If your compiler generates an error of the following sort, you are mixing type casters and bindings in a way that has them competing for the same types:

```
nanobind/include/nanobind/nb_class.h:207:40: error: static assertion failed: ↙
Attempted to create a constructor for a type that won't be  handled by the nanobind's␣
↪↙
class type caster. Is it possible that you forgot to add NB_MAKE_OPAQUE() somewhere?
```

For example, the following won't work:

```
#include <nanobind/stl/vector.h>
#include <nanobind/stl/bind_vector.h>

namespace nb = nanobind;

NB_MODULE(my_ext, m) {
    // The following line cannot be compiled
    nb::bind_vector<std::vector<int>>(m, "VectorInt");

    // This doesn't work either
    nb::class_<std::vector<int>>(m, "VectorInt");
}
```

This is not specific to STL vectors and will happen whenever casters and bindings target overlapping types.

Type casters employ a pattern matching technique known as partial template specialization. For example, `nanobind/stl/vector.h` installs a pattern that detects *any* use of `std::vector<T, Allocator>`, which overlaps with the above binding of a specific vector type.

The deeper reason for this conflict is that type casters enable a *compile-time* transformation of nanobind code, which can conflict with binding declarations that are a *runtime* construct.

To fix the conflict in this example, add the line *NB_MAKE_OPAQUE(T)*, which adds another partial template specialization pattern for `T` that says: "ignore `T` and don't use a type caster to handle it".

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

Warning: If your extension consists of multiple source code files that involve overlapping use of type casters and bindings, you are *treading on thin ice*. It is easy to violate the *One Definition Rule* (ODR) [details] in such a case, which may lead to undefined behavior (miscompilations, etc.).

Here is a hypothetical example of an ODR violation: an extension contains two source code files: `src_1.cpp` and `src_2.cpp`.

- `src_1.cpp` binds a function that returns an `std::vector<int>` using a *type caster* (`nanobind/stl/vector.h`).
- `src_2.cpp` binds a function that returns an `std::vector<int>` using a *binding* (`nanobind/stl/bind_vector.h`), and it also installs the needed type binding.

The problem is that a partially specialized class in the nanobind implementation namespace (specifically, `nanobind::detail::type_caster<std::vector<int>>`) now resolves to *two different implementations* in the two compilation units. It is unclear how such a conflict should be resolved at the linking stage, and you should consider code using such constructions broken.

To avoid this issue altogether, we recommend that you create a single include file (e.g., `binding_core.h`) containing all of the nanobind include files (binding, type casters), your own custom type casters (if present), and *NB_MAKE_OPAQUE(T)* declarations. Include this header consistently in all binding compilation units. The construction shown in the example (mixing type casters and bindings for the same type) is not allowed, and cannot occur when following the recommendation.

5.6 How can I preserve the `const`-ness of values in bindings?

This is a limitation of nanobind, which casts away `const` in function arguments and return values. This is in line with the Python language, which has no concept of const values. Additional care is therefore needed to avoid bugs that would be caught by the type checker in a traditional C++ program.

5.7 How can I reduce build time?

Large binding projects should be partitioned into multiple files, as shown in the following example:

`example.cpp`:

```
void init_ex1(nb::module_ &);
void init_ex2(nb::module_ &);
/* ... */

NB_MODULE(my_ext, m) {
    init_ex1(m);
    init_ex2(m);
    /* ... */
}
```

`ex1.cpp`:

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

`ex2.cpp`:

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

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.

5.8 How can I avoid conflicts with other projects using nanobind?

Suppose that a type binding in your project conflicts with another extension, for example because both expose a common type (e.g., `std::latch`). nanobind will warn whenever it detects such a conflict:

```
RuntimeWarning: nanobind: type 'latch' was already registered!
```

In the worst case, this could actually break both packages (especially if the bindings of the two packages expose an inconsistent/incompatible API).

The higher-level issue here is that nanobind will by default try to make type bindings visible across extensions because this is helpful to partition large binding projects into smaller parts. Such information exchange requires that the extensions:

- use the same nanobind *ABI version* (see the *Changelog* for details).
- use the same compiler (extensions built with GCC and Clang are isolated from each other).
- use ABI-compatible versions of the C++ library.
- use the stable ABI interface consistently (stable and unstable builds are isolated from each other).
- use debug/release mode consistently (debug and release builds are isolated from each other).

In addition, nanobind provides a feature to intentionally scope extensions to a named domain to avoid conflicts with other extensions. To do so, specify the `NB_DOMAIN` parameter in CMake:

```
nanobind_add_module(my_ext
                    NB_DOMAIN my_project
                    my_ext.cpp)
```

In this case, inter-extension type visibility is furthermore restricted to extensions in the `"my_project"` domain.

5.9 I'd like to use this project, but with $BUILD_SYSTEM instead of CMake

A difficult aspect of C++ software development is the sheer number of competing build systems, including

- CMake,
- Meson,
- xmake,
- Premake,

- Bazel,
- Conan,
- Autotools,
- and many others.

The author of this project has some familiarity with CMake but lacks expertise with this large space of alternative tools. Maintaining and shipping support for other build systems is therefore considered beyond the scope of this *nano* project (see also the *why?* part of the documentation that explains the rationale for being somewhat restrictive towards external contributions).

If you wish to create and maintain an alternative interface to nanobind, then my request would be that you create and maintain separate repository (see, e.g., pybind11_bazel as an example how how this was handled in the case of pybind11). Please carefully review the file nanobind-config.cmake. Besides getting things to compile, it specifies a number of platform-dependent compiler and linker options that are needed to produce *optimal* (small and efficient) binaries. Nanobind uses a complicated and non-standard set of linker parameters on macOS, which is the result of a lengthy investigation. Other parameters like linker-level dead code elimination and size-based optimization were similarly added following careful analysis. The CMake build system provides the ability to compile `libnanobind` into either a shared or a static library, to optionally target the stable ABI, and to isolate it from other extensions via the `NB_DOMAIN` parameter. All of these are features that would be nice to retain in an alternative build system. If you've made a build system compatible with another tool that is sufficiently feature-complete, then please file an issue and I am happy to reference it in the documentation.

5.10 Are there tools to generate nanobind bindings automatically?

litgen is an automatic Python bindings generator compatible with both pybind11 and nanobind, designed to create documented and easily discoverable bindings. It reproduces header documentation directly in the bindings, making the generated API intuitive and well-documented for Python users. Powered by srcML (srcml.org), a high-performance, multi-language parsing tool, litgen takes a developer-centric approach. The C++ API to be exposed to Python must be C++14 compatible, although the implementation can leverage more modern C++ features.

5.11 How to cite this project?

Please use the following BibTeX template to cite nanobind in scientific discourse:

```
@misc{nanobind,
   author = {Wenzel Jakob},
   year = {2022},
   note = {https://github.com/wjakob/nanobind},
   title = {nanobind: tiny and efficient C++/Python bindings}
}
```

CHAPTER

SIX

INSTALLING THE LIBRARY

The *nanobind* project is hosted at wjakob/nanobind on GitHub. To use the library in your own projects, it is usually easiest to install it using one of the following three methods:

6.1 Install via Pip (recommended)

Run the following command in your terminal to install a package containing both C++ and CMake source code needed to compile extension modules.

```
python -m pip install nanobind
```

6.2 Install via Conda

The following alternative installs an equivalent package through Conda. It is provided for users that develop Conda-based extensions with a build-time dependency on *nanobind*, in which case the PyPI package cannot be used.

```
conda install -c conda-forge nanobind
```

6.3 Install as a Git submodule

If you prefer not to involve external package managers, and if your project uses the Git control system, you may also directly reference *nanobind* as a Git submodule. In the main directory of your repository, run the following commands:

```
git submodule add https://github.com/wjakob/nanobind ext/nanobind
git submodule update --init --recursive
```

This assumes you are placing your dependencies in `ext/`.

The *next section* will explain how to set up a basic build system that you can use to build your first extension module.

CHAPTER

SEVEN

SETTING UP A BUILD SYSTEM

This section assumes that you have followed the instructions to *install* nanobind. The easiest way to compile a nanobind-based extension involves a CMake-based build system. Other build systems can likely be used as well, but they are not officially supported. (The first section of the *CMake API reference* mentions some alternatives.)

Here, we will create a new package from scratch. If you already have an existing CMake build system, it should be straightforward to merge some of the following snippets into it.

7.1 Preliminaries

Begin by creating a new file named `CMakeLists.txt` in the root directory of your project. It should start with the following lines that declare a project name and tested CMake version range. The third line line searches for Python >= 3.8 including the `Development.Module` component required by nanobind. The name of this module changed across CMake versions, hence the additional conditional check.

```
cmake_minimum_required(VERSION 3.15...3.27)
project(my_project) # Replace 'my_project' with the name of your project

if (CMAKE_VERSION VERSION_LESS 3.18)
  set(DEV_MODULE Development)
else()
  set(DEV_MODULE Development.Module)
endif()

find_package(Python 3.8 COMPONENTS Interpreter ${DEV_MODULE} REQUIRED)
```

Add the following lines below. They configure CMake to perform an optimized *release* build by default unless another build type is specified. Without this addition, binding code may run slowly and produce large binaries.

```
if (NOT CMAKE_BUILD_TYPE AND NOT CMAKE_CONFIGURATION_TYPES)
  set(CMAKE_BUILD_TYPE Release CACHE STRING "Choose the type of build." FORCE)
  set_property(CACHE CMAKE_BUILD_TYPE PROPERTY STRINGS "Debug" "Release" "MinSizeRel"
 ↪"RelWithDebInfo")
endif()
```

7.2 Finding nanobind

Next, we must inform CMake about the presence of nanobind so that it can load the functionality needed to compile extension modules. The details of this step depend on *how you installed* nanobind, in the *previous section*.

1. If you installed nanobind as a Pip or Conda package, append the following lines at the end of `CMakeLists.txt`. They query the package to determine its installation path and then import it.

```
# Detect the installed nanobind package and import it into CMake
execute_process(
  COMMAND "${Python_EXECUTABLE}" -m nanobind --cmake_dir
  OUTPUT_STRIP_TRAILING_WHITESPACE OUTPUT_VARIABLE nanobind_ROOT)
find_package(nanobind CONFIG REQUIRED)
```

2. If you installed nanobind as a Git submodule, append the following lines at the end of `CMakeLists.txt` to point CMake to the directory where nanobind is checked out.

```
add_subdirectory(${CMAKE_CURRENT_SOURCE_DIR}/ext/nanobind)
```

7.3 Building an extension

Finally, we are ready to build an extension! Append the following line at the end of `CMakeLists.txt`. It will compile a new extension named `my_ext` from the source code contained in the file `my_ext.cpp`.

```
nanobind_add_module(my_ext my_ext.cpp)
```

nanobind_add_module() resembles standard CMake commands like `add_executable()` and `add_library()`. Any number of source code and header files can be declared when the extension is more complex and spread out over multiple files.

Note: One opinionated choice of *nanobind_add_module()* is that it optimizes the *size* of the extension by default (i.e., `-Os` is passed to the compiler regardless of the project-wide settings). You must specify the `NOMINSIZE` parameter to the command to disable this behavior and, e.g., optimize extension code for speed (i.e., `-O3`):

```
nanobind_add_module(my_ext NOMINSIZE my_ext.cpp)
```

The default is chosen this way since extension code usually wraps existing C++ libraries, in which the main computation takes place. Optimizing the bindings for speed does not measurably improve performance, but it does make the bindings *significantly* larger.

If you observe slowdowns when porting a pybind11 extension, or if your extension performs significant amounts of work within the binding layer, then you may want to experiment with passing the `NOMINSIZE` parameter.

The *next section* will review the contents of example module implementation in `my_ext.cpp`.

CHAPTER

EIGHT

CREATING YOUR FIRST EXTENSION

This section assumes that you have followed the instructions to *install* nanobind and set up a basic *build system*.

We are now ready to define a first basic extension that wraps a function to add two numbers. Create a new file `my_ext.cpp` with the following contents (the meaning of this code will be explained shortly):

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

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

NB_MODULE(my_ext, m) {
    m.def("add", &add);
}
```

Afterwards, you should be able to compile and run the extension.

8.1 Building using CMake

Launch `cmake` in the project directory to set up a build system that will write all output into a separate `build` subdirectory.

```
cmake -S . -B build
```

Note: If this step fails with an error message saying that Python cannot be found, you will need to install a suitable Python 3 development package.

For example, on Ubuntu you would run:

```
apt install libpython3-dev
```

On Windows, you we recommend downloading and running one of the installers provided by the Python foundation.

Note: If you have multiple versions of Python on your system, the CMake build system may not find the specific version you had in mind. This is problematic: extension built for one version of Python usually won't run on another version. You can provide a hint to the build system to help it find a specific version.

In this case, delete the `build` folder (if you already created one) and re-run `cmake` while specifying the command line parameter `-DPython_EXECUTABLE=<path to python executable>`.

```
rm -Rf build
cmake -S . -B build -DPython_EXECUTABLE=<path to python executable>
```

Assuming the `cmake` ran without issues, you can now compile the extension using the following command:

```
cmake --build build
```

Finally, navigate into the `build` directory and launch an interactive Python session:

```
cd build
python3
```

(The default build output directory is different on Windows: use `cd build\Debug` and `python` instead of the above.)

You should be able to import the extension and call the newly defined function `my_ext.add()`.

```
Python 3.11.1 (main, Dec 23 2022, 09:28:24) [Clang 14.0.0 (clang-1400.0.29.202)] on␣
 ↪darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import my_ext
>>> my_ext.add(1, 2)
3
```

8.2 Binding functions

Let's step through the example binding code to understand what each line does. The directive on the first line includes the core parts of nanobind:

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

nanobind also provides many optional add-on components that are aren't included by default. They are discussed throughout this documentation along with pointers to the header files that must be included when using them.

Next is the function to be exposed in Python, followed by the mysterious-looking *NB_MODULE* macro.

```
int add(int a, int b) { return a + b; }

NB_MODULE(my_ext, m) {
    m.def("add", &add);
}
```

NB_MODULE(my_ext, m) declares the extension with the name `my_ext`. This name **must** match the extension name provided to the `nanobind_add_module()` function in the CMake build system—otherwise, importing the extension will fail with an obscure error about a missing symbol. The second argument (`m`) names a variable of type *nanobind::module_* that represents the created module.

The part within curly braces (`{`, `}`) consists of a sequence of statements that initialize the desired function and class bindings. It is best thought of as the `main()` function that will run when a user imports the extension into a running Python session.

In this case, there is only one binding declaration that wraps the `add` referenced using the ampersand (`&`) operator. nanobind determines the function's type signature and generates the necessary binding code. All of this happens automatically at compile time.

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 go back to Boost.Python, though the implementation in nanobind is very different.

8.3 Keyword and default arguments

There are limits to what nanobind can determine at compile time. For example, the argument names were lost and calling `add()` in Python using keyword arguments fails:

```
>>> my_ext.add(a=1, b=2)
TypeError: add(): incompatible function arguments. The following argument types are␣
 ↪supported:
    1. add(arg0: int, arg1: int, /) -> int

Invoked with types: kwargs = { a: int, b: int }
```

Let's improve the bindings to fix this. We will also add a docstring and a default `b` argument so that `add()` increments when only one value is provided. The modified binding code looks as follows:

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

namespace nb = nanobind;
using namespace nb::literals;

int add(int a, int b = 1) { return a + b; }

NB_MODULE(my_ext, m) {
    m.def("add", &add, "a"_a, "b"_a = 1,
          "This function adds two numbers and increments if only one is provided.");
}
```

Let's go through all of the changed lines. The first sets up a short namespace alias named `nb`:

```
namespace nb = nanobind;
```

This is convenient because binding code usually ends up referencing many classes and functions from this namespace. The subsequent `using` declaration is optional and enables a convenient syntax for annotating function arguments:

```
using namespace nb::literals;
```

Without it, you would have to change every occurrence of the pattern `"..."_a` to the more verbose `nb::arg("...")`.

The function binding declaration includes several changes. It is common to pile on a few attributes and modifiers in *`.def(...)`* binding declarations, which can be specified in any order.

```
m.def("add", &add, "a"_a, "b"_a = 1,
      "This function adds two numbers and increments if only one is provided.");
```

The string at the end is a docstring that will later show up in generated documentation. The argument annotations (`"a"_a, "b"_a`) associate parameters with names for keyword-based argument passing.

Besides argument names, nanobind also cannot infer *default arguments*—you *must repeat them* in the binding declaration. In the above snippet, the `"b"_a = 1` annotation informs nanobind about the value of the default argument.

8.4 Exporting values

To export a value, use the *attr()* function to register it in the module as shown below. Bound classes and built-in types are automatically converted when they are assigned in this way.

```
m.attr("the_answer") = 42;
```

8.5 Docstrings

Let's add one more bit of flourish by assigning a docstring to the extension module itself. Include the following line anywhere in the body of the `NB_MODULE() {...}` declaration:

```
m.doc() = "A simple example python extension";
```

After recompiling the extension, you should be able to view the associated documentation using the `help()` builtin or the ? operator in IPython.

```
>>> import my_ext
>>> help(my_ext)

Help on module my_ext:

NAME
    my_ext - A simple example python extension

DATA
    add = <nanobind.nb_func object>
        add(a: int, b: int = 1) -> int

        This function adds two numbers and increments if only one is provided

    the_answer = 42

FILE
    /Users/wjakob/my_ext/my_ext.cpython-311-darwin.so
```

The automatically generated documentation covers functions, classes, parameter and return value type information, argument names, and default arguments.

8.6 Binding a custom type

Let's now turn to an object oriented example. We will create bindings for a simple C++ type named `Dog` defined as follows:

```
#include <string>

struct Dog {
    std::string name;

    std::string bark() const {
        return name + ": woof!";
    }
};
```

The Dog bindings look as follows:

```
#include <nanobind/nanobind.h>
#include <nanobind/stl/string.h>

namespace nb = nanobind;

NB_MODULE(my_ext, m) {
    nb::class_<Dog>(m, "Dog")
        .def(nb::init<>())
        .def(nb::init<const std::string &>())
        .def("bark", &Dog::bark)
        .def_rw("name", &Dog::name);
}
```

Let's look at selected lines of this example, starting with the added include directive:

```
#include <nanobind/stl/string.h>
```

nanobind has a minimal core and initially doesn't know how to deal with STL types like `std::string`. This line imports a *type caster* that realizes a bidirectional conversion (C++ `std::string` ↔ Python `str`) to make the example usable. An upcoming *documentation section* will provide more detail on type casters and other alternatives.

The class binding declaration *nb::class_<T>()* supports both `class` and `struct`-style data structures.

```
nb::class_<Dog>(m, "Dog")
```

Here, it associates the C++ type `Dog` with a new Python type named `"Dog"` and installs it in the *nb::module_* `m`.

Initially, this type is completely empty—it has no members and cannot be instantiated. The subsequent chain of binding declarations binds two constructor overloads (via *nb::init<...>()*), a method, and the mutable `name` field (via *.def_rw(..)*, where `rw` stands for read/write access).

```
.def(nb::init<>())
.def(nb::init<const std::string &>())
.def("bark", &Dog::bark)
.def_rw("name", &Dog::name);
```

An interactive Python session demonstrating this example is shown below:

```
Python 3.11.1 (main, Dec 23 2022, 09:28:24) [Clang 14.0.0 (clang-1400.0.29.202)] on␣
 ↪darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import my_ext
>>> d = my_ext.Dog('Max')
>>> print(d)
<my_ext.Dog object at 0x1044540f0>
>>> d.name
'Max'
>>> d.name = 'Charlie'
>>> d.bark()
'Charlie: woof!'
```

The example showed how to bind constructors, methods, and mutable fields. Many other things can be bound using analogous *nb::class_<...>* methods:

Type	Method
Methods & constructors	*.def()*
Fields	*.def_ro()*, *.def_rw()*
Properties	*.def_prop_ro()*, *.def_prop_rw()*
Static methods	*.def_static()*
Static fields	*.def_ro_static()*, *.def_rw_static()*
Static properties	*.def_prop_ro_static()*, *.def_prop_rw_static()*

Note: All of these binding declarations support *docstrings*, *keyword, and default argument* annotations as before.

8.7 Binding lambda functions

Note how `print(d)` produced a rather useless summary in the example above:

```
>>> print(d)
<my_ext.Dog object at 0x1044540f0>
```

To address this, we can add a special Python method named `__repr__` that returns a human-readable summary. Unfortunately, a corresponding function with such functionality does not currently exist in the C++ type, and it would be nice if we did not have to modify it. We can bind a *lambda function* to achieve both goals:

```
nb::class_<Dog>(m, "Dog")
    // ... skipped ...
    .def("__repr__",
         [](const Dog &p) { return "<my_ext.Dog named '" + p.name + "'>"; });
```

nanobind supports both stateless[1] and stateful lambda closures.

8.8 Higher order functions

nanobind's support for higher-order functions[2] further blurs the language boundary. The snippet below extends the `Dog` class with higher-order function `bark_later()` that calls *nb::cpp_function()* to convert and return a *stateful* C++ lambda function (`callback`) as a Python function object.

```
nb::class_<Dog>(m, "Dog")
    // ... skipped ...
    .def("bark_later", [](const Dog &p) {
        auto callback = [name = p.name] {
            nb::print(nb::str("{}: woof!").format(name));
        };
        return nb::cpp_function(callback);
    });
```

The lambda function captures the `Dog::name()` property (a C++ `std::string`) and in turn calls Python functions (*nb::print()*, *nb::str::format()*) to print onto the console. Here is an example use of the binding in Python:

[1] Stateless closures are those with an empty pair of brackets `[]` as the capture object.
[2] Higher-order functions are functions that take functions as arguments and/or return them.

```
>>> f = d.bark_later()
>>> f
<nanobind.nb_func object at 0x10537c140>
>>> f()
Charlie: woof!
```

8.9 Wrap-up

This concludes the basic part of the documentation, which provided a first taste of nanobind and typical steps needed to create a custom extension.

The upcoming intermediate-level material covers performance and safety-critical points:

- C++ and Python can exchange information in various different ways.

  Which one is best for a particular task?

- A bound object can simultaneously exist in both C++ and Python.

  Who owns it? When is it safe to delete it?

Following these topics, the documentation revisits function and class bindings in full detail.

CHAPTER

NINE

BUILDING EXTENSIONS USING BAZEL

If you prefer the Bazel build system to CMake, you can build extensions using the nanobind-bazel project.

Note: This project is a community contribution maintained by Nicholas Junge, please report issues directly in the nanobind-bazel repository linked above.

9.1 Adding nanobind-bazel to your Bazel project

To use nanobind-bazel in your project, you need to add it to your project's dependency graph. Using bzlmod, the de-facto dependency management system in Bazel starting with version 7.0, you can simply specify it as a `bazel_dep` in your MODULE.bazel file:

```
# Place this in your MODULE.bazel file.
# The major version of nanobind-bazel is equal to the version
# of the internally used nanobind.
# In this case, we are building bindings with nanobind v2.2.0.
bazel_dep(name = "nanobind_bazel", version = "2.2.0")
```

To instead use a development version from GitHub, you can declare the dependency as a `git_override()` in your MODULE.bazel:

```
# MODULE.bazel
bazel_dep(name = "nanobind_bazel", version = "")
git_override(
    module_name = "nanobind_bazel",
    commit = COMMIT_SHA, # replace this with the actual commit you want.
    remote = "https://github.com/nicholasjng/nanobind-bazel",
)
```

In local development scenarios, you can clone nanobind-bazel to your machine, and then declare it as a `local_path_override()` dependency:

```
# MODULE.bazel
bazel_dep(name = "nanobind_bazel", version = "")
local_path_override(
    module_name = "nanobind_bazel",
    path = "/path/to/nanobind-bazel/", # replace this with the actual path.
)
```

Note: At minimum, Bazel version 6.4.0 is required to use nanobind-bazel.

9.2 Declaring and building nanobind extension targets

The main tool to build nanobind C++ extensions for your Python bindings is the *nanobind_extension()* rule.

Like all public nanobind-bazel APIs, it resides in the `build_defs` submodule. To import it into a BUILD file, use the builtin `load` command:

```
# In a BUILD file, e.g. my_project/BUILD
load("@nanobind_bazel//:build_defs.bzl", "nanobind_extension")

nanobind_extension(
    name = "my_ext",
    srcs = ["my_ext.cpp"],
)
```

In this short snippet, a nanobind Python module called `my_ext` is declared, with its contents coming from the C++ source file of the same name. Conveniently, only the actual module name must be declared - its place in your Python project hierarchy is automatically determined by the location of your build file.

For a comprehensive list of all available build rules in nanobind-bazel, refer to the rules section in the *nanobind-bazel API reference*.

9.3 Building against the stable ABI

As in nanobind's CMake config, you can build bindings targeting Python's stable ABI, starting from version 3.12. To do this, specify the target version using the `@nanobind_bazel//:py-limited-api` flag. For example, to build extensions against the CPython 3.12 stable ABI, pass the option `@nanobind_bazel//:py-limited-api="cp312"` to your `bazel build` command.

For more information about available flags, refer to the flags section in the *nanobind-bazel API reference*.

9.4 Generating stubs for built extensions

You can also use Bazel to generate stubs for an extension directly at build time with the `nanobind_stubgen` macro. Here is an example of a nanobind extension with a stub file generation target declared directly alongside it:

```
# Same as before in a BUILD file
load(
    "@nanobind_bazel//:build_defs.bzl",
    "nanobind_extension",
    "nanobind_stubgen",
)

nanobind_extension(
    name = "my_ext",
    srcs = ["my_ext.cpp"],
)

nanobind_stubgen(
    name = "my_ext_stubgen",
    module = ":my_ext",
)
```

You can then generate stubs on an extension by invoking `bazel run //my_project:my_ext_stubgen`. Note that this requires actually running the target instead of only building it via `bazel build`, since a Python script needs to be executed for stub generation.

Naturally, since stub generation relies on the given shared object files, the actual extensions are built in the process before invocation of the stub generation script.

9.5 nanobind-bazel and Python packaging

Unlike CMake, which has a variety of projects supporting PEP517-style Python package builds, Bazel does not currently have a fully featured PEP517-compliant packaging backend available.

To produce Python wheels containing bindings built with nanobind-bazel, you have various options, with two of the most prominent strategies being

1. Using a wheel builder script with the facilities provided by a Bazel support package for Python, such as `py_binary` or `py_wheel` from rules_python. This is a lower-level, more complex workflow, but it provides more granular control of how your Python wheel is built.

2. Building all extensions with Bazel through a subprocess, by extending a Python build backend such as `setuptools`. This allows you to stick to those well-established build tools, like `setuptools`, at the expense of more boilerplate Python code and slower build times, since Bazel is only invoked to build the bindings extensions (and their dependencies).

In general, while the latter method requires less setup and customization, its drawbacks weigh more severely for large projects with more extensions.

Note: An example of packaging with the mentioned setuptools customization method can be found in the nanobind_example repository, specifically, on the `bazel` branch. It also contains an example of how to customize flag names and set default build options across platforms with a `.bazelrc` file.

CHAPTER

TEN

BUILDING EXTENSIONS USING MESON

If you prefer the Meson build system to CMake, you can build extensions using the Meson WrapDB package.

Note: This package is a community contribution maintained by Will Ayd, please report issues directly in the Meson WrapDB repository.

10.1 Adding nanobind to your Meson project

To use Meson as the build generator in your Python project, you will want to install the meson-python project as a build dependency. To do so, simply add the following to your pyproject.toml file:

```
[project]
name = "my_project_name"
dynamic = ['version']

[build-system]
requires = ['meson-python']

build-backend = 'mesonpy'
```

In your project root, you will also want to create the subprojects folder that Meson can install into. Then you will need to install the wrap packages for both nanobind and robin-map:

```
mkdir -p subprojects
meson wrap install robin-map
meson wrap install nanobind
```

The `meson.build` definition in your project root should look like:

```
project(
  'my_project_name',
  'cpp',
  version: '0.0.1',
)

py = import('python').find_installation()
nanobind_dep = dependency('nanobind', static: true)
py.extension_module(
  'my_module_name',
  sources: ['path_to_module.cc'],
  dependencies: [nanobind_dep],
  install: true,
)
```

With this configuration, you may then call:

```
meson setup builddir
meson compile -C builddir
```

To compile the extension in the `builddir` folder.

Alternatively, if you don't care to have a local build folder, you can use the Python build frontend of your choosing to install the package as an editable install. With pip, this would look like:

```
python -m pip install -e .
```

10.2 Building against the stable ABI

As in nanobind's CMake config, you can build bindings targeting Python's stable ABI, starting from version 3.12. To do this, specify the target version using the `limited_api` argument in your configuration. For example, to build extensions against the CPython 3.12 stable ABI, you would use:

```
py.extension_module(
  'my_module_name',
  sources: ['path_to_module.cc'],
  dependencies: [nanobind_dep],
  install: true,
  limited_api: '3.12',
)
```

In your `meson.build` file.

CHAPTER

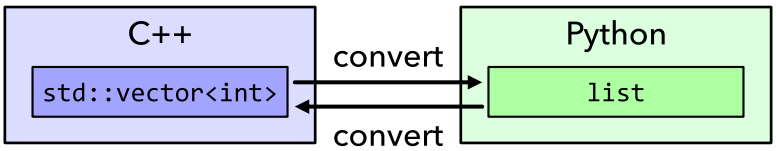
ELEVEN

EXCHANGING INFORMATION

nanobind offers three fundamentally different ways of exchanging information between Python and C++. Depending on the task at hand, one will usually be preferable over the others, hence it is important to be aware of their advantages and disadvantages.

11.1 Option 1: Type Casters

A *type caster* translates C++ objects into equivalent Python objects and vice versa. The illustration below shows a translation between C++ (blue) and Python (green) worlds, where a `std::vector<int>` instance converts from/to a Python `list` containing `int` objects.



Example: The following function doubles the entries of an STL vector and returns the result.

```
using IntVector = std::vector<int>;

IntVector double_it(const IntVector &in) {
    IntVector out(in.size());
    for (size_t i = 0; i < in.size(); ++i)
        out[i] = in[i] * 2;
    return out;
}
```

To expose it in Python, we can use the `std::vector<...>` type caster that is located in an optional header file named `nanobind/stl/vector.h`:

```
#include <nanobind/stl/vector.h>

NB_MODULE(my_ext, m) {
    m.def("double_it", &double_it);
}
```

That's all there is to it. The Python version of the function features an automatically generated docstring, type checks, and (if needed) error reporting.

```
>>> import my_ext
>>> my_ext.double_it([1, 2, 3])
[2, 4, 6]

>>> my_ext.double_it([1, 2, 'foo'])
```

(continues on next page)

(continued from previous page)

```
TypeError: double_it(): incompatible function arguments. The following argument types␣
↪are supported:
    1. double_it(arg: list[int], /) -> list[int]
```

What are the implications of using type casters?

Pro: this approach is simple and convenient, especially when standard (STL) types are involved. Usually, all that is needed is an `#include` directive to pull in the right header file. Complex nested types (e.g. vectors of hash tables of strings) work automatically by combining type casters recursively.

The following table lists the currently available type casters along with links to external projects that provide further casters:

Type	Type caster header
`char`, `char*`, `void*`, `nullptr_t`, `bool`, `int`, `unsigned int`, `long`, `unsigned long`, …	Built-in (no include file needed)
`std::array<..>`	`#include <nanobind/stl/array.h>`
`std::chrono::duration<..>`, `std::chrono::time_point<..>` (*more details*)	`#include <nanobind/stl/chrono.h>`
`std::complex<..>`	`#include <nanobind/stl/complex.h>`
`std::filesystem::path`	`#include <nanobind/stl/filesystem.h>`
`std::function<..>`	`#include <nanobind/stl/function.h>`
`std::list<..>`	`#include <nanobind/stl/list.h>`
`std::map<..>`	`#include <nanobind/stl/map.h>`
`std::optional<..>`	`#include <nanobind/stl/optional.h>`
`std::pair<..>`	`#include <nanobind/stl/pair.h>`
`std::set<..>`	`#include <nanobind/stl/set.h>`
`std::string`	`#include <nanobind/stl/string.h>`
`std::string_view`	`#include <nanobind/stl/string_view.h>`
`std::wstring`	`#include <nanobind/stl/wstring.h>`
`std::tuple<..>`	`#include <nanobind/stl/tuple.h>`
`std::shared_ptr<..>`	`#include <nanobind/stl/shared_ptr.h>`
`std::unique_ptr<..>`	`#include <nanobind/stl/unique_ptr.h>`
`std::unordered_set<..>`	`#include <nanobind/stl/unordered_set.h>`
`std::unordered_map<..>`	`#include <nanobind/stl/unordered_map.h>`
`std::variant<..>`	`#include <nanobind/stl/variant.h>`
`std::vector<..>`	`#include <nanobind/stl/vector.h>`
`nb::ndarray<..>`	`#include <nanobind/ndarray.h>`
`Eigen::Matrix<..>`, `Eigen::Array<..>`, `Eigen::Ref<..>`, `Eigen::Map<..>`	`#include <nanobind/eigen/dense.h>`
`Eigen::SparseMatrix<..>`	`#include <nanobind/eigen/sparse.h>`
Apache Arrow types	https://github.com/maximiliank/nanobind_pyarrow
…	Please reach out if you have additions to this list.

Con: Every transition between the Python and C++ side will generally require a conversion step (in this case, to re-create all list elements). This can be wasteful when the other side only needs to access a small part of the data. Conversely, the overhead should not be a problem when the data is fully "consumed" following conversion.

A select few type casters (`std::unique_ptr<..>`, `std::shared_ptr<..>`, *`nb::ndarray`*, and `Eigen::*`) are special in the sense that they can perform a type conversion *without* copying the underlying data. Besides those few exceptions type casting always implies that a copy is made.

11.1.1 Mutable reference issue

Another subtle limitation of type casters is that they don't propagate updates through mutable references. Consider the following alternative implementation of the `double_it` function:

```
void double_it(IntVector &in) {
    for (int &value : in)
        value *= 2;
}
```

nanobind can wrap this function without problems, but it won't behave as expected:

```
>>> x = [1, 2, 3]
>>> my_ext.double_it(x)
>>> x
[1, 2, 3]  # <-- oops, unchanged!
```

How could this happen? The reason is that type casters convert function arguments and return values once, but further changes will not automatically propagate across the language barrier because the representations are not intrinsically linked to each other. This problem is not specific to STL types—for example, the following function will similarly not update its argument once exposed in Python.

```
void double_it(int &in) { in *= 2; }
```

This is because builtin types like `int`, `str`, `bool`, etc., are all handled by type casters.

A simple alternative to propagate updates while retaining the convenience of type casters is to bind a small wrapper lambda function that returns a tuple with all output arguments. An example:

```
int foo(int &in) { in *= 2; return std::sqrt(in); }
```

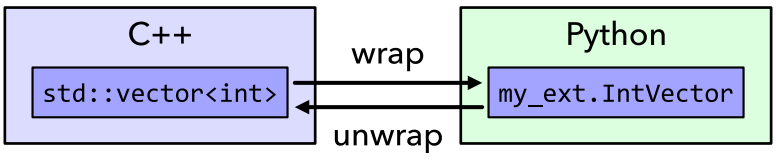
And the binding code

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

In this case, a type caster (`#include <nanobind/stl/tuple.h`) must be included to handle the `std::tuple<int, int>` return value.

11.2 Option 2: Bindings

Bindings expose C++ types in Python; the ability to create them is the main feature of nanobind. In the list-of-integer example, they cause Python to interpret `std::vector<int>` as a new Python type called `my_ext.IntVector`.



Example: to switch the previous example to bindings, we first replace the type caster header (nanobind/stl/vector.h) by its binding variant (nanobind/stl/bind_vector.h) and then invoke the *nb::bind_vector<T>()* function to create a *new Python type named* `IntVector` within the module `m`.

```
#include <nanobind/stl/bind_vector.h>

using IntVector = std::vector<int>;
```

(continues on next page)

(continued from previous page)

```
IntVector double_it(const IntVector &in) { /* .. omitted .. */ }

namespace nb = nanobind;

NB_MODULE(my_ext, m) {
    nb::bind_vector<IntVector>(m, "IntVector");
    m.def("double_it", &double_it);
}
```

Any function taking or returning integer vectors will now use the type binding. In the Python session below, nanobind performs an implicit conversion from the Python list `[1, 2, 3]` to a `my_ext.IntVector` before calling the `double_it` function.

```
>>> import my_ext
>>> my_ext.double_it([1, 2, 3])
my_ext.IntVector([2, 4, 6])

>>> my_ext.double_it.__doc__
'double_it(arg: my_ext.IntVector, /) -> my_ext.IntVector'
```

Let's go through the implications of using bindings:

Pro: bindings don't require the costly conversion step when crossing the language boundary. They also support mutable references, so the *issue discussed in the context of type casters* does not arise. Sometimes, binding is the only available option: when a C++ type does not have an equivalent Python type, casting simply does not make sense.

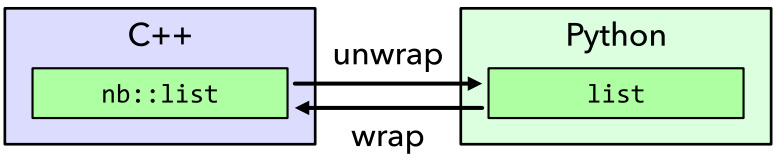
Con: Creating good bindings that feel natural in Python requires some additional work. We cheated in this example by relying on the *nb::bind_vector<T>()* helper function that did all the heavy lifting. Such helpers are currently only available for a few special cases (vectors, ordered/unordered maps, iterators):

Type	Binding helper header
`std::vector<..>`	`#include <nanobind/stl/bind_vector.h>` (*docs*)
`std::map<..>`	`#include <nanobind/stl/bind_map.h>` (*docs*)
`std::unordered_map<..>`	`#include <nanobind/stl/bind_map.h>` (*docs*)
Forward iterators	`#include <nanobind/make_iterator.h>` (*docs*)
Other types	See the previous example on *binding custom types*.

In general, you will need to write the binding code yourself. The previous section on *binding custom types* showed an example of such a type binding.

11.3 Option 3: Wrappers

The last option is only rarely used, but it can be powerful alternative in some cases. nanobind provides *wrapper* classes to use Python types within C++. You can think of this as a kind of *reverse binding*. For example, a Python list can be accessed through the *nb::list* type:



This is what the example looks like when expressed using *nb::list* and *nb::int_*.

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

namespace nb = nanobind;

nb::list double_it(nb::list l) {
    nb::list result;
    for (nb::handle h: l)
        result.append(h * nb::int_(2));
    return result;
}

NB_MODULE(my_ext, m) {
    m.def("double_it", &double_it);
}
```

The implications of using wrappers are:

Pro: Wrappers require no copying or type conversion. With them, C++ begins to resemble dynamically typed Python code and can perform highly general operations on Python objects. Wrappers are useful to tap into the powerful Python software ecosystem (NumPy, Matplotlib, PyTorch, etc).

Con: Functions based on wrappers cannot run without Python. In contrast to option 1 (*type casters*) and 2 (*bindings*), we can no longer reuse an existing function and process its arguments and return value to interface the Python and C++ worlds: the entire function must be rewritten using nanobind-specific wrapper types. Every operation will translate into a corresponding Python C API call, which means that wrappers aren't suitable for performance-critical loops or multithreaded computations.

The following wrappers are available and require no additional include directives: *any*, *bytearray*, *bytes*, *callable*, *capsule*, *dict*, *ellipsis*, *handle*, *handle_t<T>*, *bool_*, *int_*, *float_*, *iterable*, *iterator*, *list*, *mapping*, *module_*, *object*, *set*, *sequence*, *slice*, *str*, *tuple*, *weakref*, *type_object*, *type_object_t<T>*, *args*, and *kwargs*.

11.4 Discussion

The choices outlined above are more fine-grained than they may appear. For example, it is possible to use type casters, bindings, and wrappers to handle multiple arguments of *a single function*.

They can also be combined *within* a single function argument. For example, you can type cast a `std::vector<T>` containing bindings or wrappers.

In general, we recommend that you use

1. type casters for STL containers, and
2. bindings for other custom types.

If the former turn out to be a performance bottleneck, it is easy to replace them with bindings or wrappers later on. Wrappers are only rarely useful; you will usually know it when you need them.

CHAPTER

TWELVE

OBJECT OWNERSHIP

Python and C++ don't manage the lifetime and storage of objects in the same way. Consequently, two questions arise whenever an object crosses the language barrier:

- Who actually *owns* this object? C++? Python? Both?!
- Can we safely determine when it is no longer needed?

This is important: we *must* exclude the possibility that Python destroys an object that is still being used by C++ (or vice versa).

The *previous section* introduced three ways of exchanging information between C++ and Python: *type casters*, *bindings*, and *wrappers*. It is specifically *bindings* for which these two questions must be answered.

12.1 A problematic example

Consider the following problematic example to see what can go wrong:

```
#include <nanobind/nanobind.h>
namespace nb = nanobind;

struct Data { };
Data data; // Data global variable & function returning a pointer to it
Data *get_data() { return &data; }

NB_MODULE(my_ext, m) {
    nb::class_<Data>(m, "Data");

    // KABOOM, calling this function will crash the Python interpreter
    m.def("get_data", &get_data);
}
```

The bound function `my_ext.get_data()` returns a Python object of type `my_ext.Data` that wraps the pointer `&data` and takes ownership of it.

When Python eventually garbage collects the object, nanobind will try to free the (non-heap-allocated) C++ instance via `operator delete`, causing a segmentation fault.

To avoid this problem, we can

1. **Provide more information**: the problem was that nanobind *incorrectly* transferred ownership of a C++ instance to the Python side. To fix this, we can add add a *return value policy* annotation that clarifies what to do with the return value.
2. **Make ownership transfer explicit**: C++ types passed via *unique pointers* (`std::unique_ptr<T>`) make the ownership transfer explicit in the type system, which would have revealed the problem in this example.

3. **Switch to shared ownership**: C++ types passed via *shared pointers* (`std::shared_ptr<T>`), or which use *intrusive reference counting* can be shared by C++ and Python. The whole issue disappears because ownership transfer is no longer needed.

The remainder of this section goes through each of these options.

12.2 Return value policies

nanobind provides several *return value policy* annotations that can be passed to `module_::def()`, `class_::def()`, and `cpp_function()`. The default policy is `rv_policy::automatic`, which is usually a reasonable default (but not in this case!).

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

```
m.def("get_data", &get_data, nb::rv_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 this library, it is important that you familiarize yourself with the different options below. In particular, the following policies are available:

- `rv_policy::take_ownership`: Create a thin Python object wrapper around the returned C++ instance without making a copy and transfer ownership to Python. When the Python wrapper is eventually garbage collected, nanobind will call the C++ `delete` operator to free the C++ instance.

  In the example below, a function uses this policy to transfer ownership of a heap-allocated C++ instance to Python:

  ```
  m.def("make_data", []{ return new Data(); }, nb::rv_policy::take_ownership);
  ```

  The return value policy declaration could actually have been omitted here because `take_ownership` is the default for *pointer return values* (see `automatic`).

- `rv_policy::copy`: Copy-construct a new Python object from the C++ instance. The copy will be owned by Python, while C++ retains ownership of the original.

  In the example below, a function uses this policy to return a reference to a C++ instance. The owner and lifetime of such a reference may not be clear, so the safest route is to make a copy.

  ```
  struct A {
     B &b() { /* .. unknown code .. */ }
  };

  nb::class_<A>(m, "A")
     .def("b", &A::b, nb::rv_policy::copy);
  ```

  The return value policy declaration could actually have been omitted here because `copy` is the default for *lvalue reference* return values (see `automatic`).

- `rv_policy::move`: Move-construct a new Python object from the C++ instance. The new object will be owned by Python, while C++ retains ownership of the original (whose contents were likely invalidated by the move operation).

  In the example below, a function uses this policy to return a C++ instance by value. The `copy` operation mentioned above would also be safe to use, but move construction has the potential of being significantly more efficient.

  ```
  struct A {
     B b() { return B(...); }
  };
  ```

(continues on next page)

(continued from previous page)

```
nb::class_<A>(m, "A")
    .def("b", &A::b, nb::rv_policy::move);
```

The return value policy declaration could actually have been omitted here because *move* is the default for *functions that return by value* (see *automatic*).

- *rv_policy::reference*: Create a thin Python object wrapper around the returned C++ instance without making a copy, but *do not transfer ownership to Python*. nanobind will never call the C++ `delete` operator, even when the wrapper expires. The C++ side is responsible for destructing the C++ instance.

  This return value policy is *dangerous* and should be used cautiously. Undefined behavior will ensue when the C++ side deletes the instance while it is still being used by Python. If you need to use this policy, combine it with a *keep_alive* function binding annotation to manage the lifetime. Or use the simple and safe *reference_internal* alternative described next.

  Below is an example use of this return value policy to reference a global variable that does not need ownership and lifetime management.

```
Data data; // This is a global variable

m.def("get_data", []{ return &data; }, nb::rv_policy::reference)
```

- *rv_policy::reference_internal*: A policy for *methods* that expose an internal field. The lifetime of the field must match that of the parent object.

  The policy resembles *reference* in that it creates creates a thin Python object wrapper around the returned C++ field without making a copy, and without transferring ownership to Python.

  Furthermore, it ensures that the instance owning the field (implicit `this`/`self` argument) cannot be garbage collected while an object representing the field is alive.

  The example below uses this policy to implement a *getter* that permits mutable access to an internal field.

```
struct MyClass {
public:
    MyField &field() { return m_field; }

private:
    MyField m_field;
};

nb::class_<MyClass>(m, "MyClass")
    .def("field", &MyClass::field, nb::rv_policy::reference_internal);
```

  More advanced variations of this scheme are also possible using combinations of *reference* and the *keep_alive* function binding annotation.

- *rv_policy::none*: This is the most conservative policy: it simply refuses the cast unless the C++ instance already has a corresponding Python object, in which case the question of ownership becomes moot.

- *rv_policy::automatic*: This is the default return value policy, which falls back to *take_ownership* when the return value is a pointer, *move* when it is a rvalue reference, and *copy* when it is a lvalue reference.

- *rv_policy::automatic_reference*: This policy matches *automatic* but falls back to `reference` when the return value is a pointer. It is the default for function arguments when calling Python functions from C++ code via *detail::api::operator()()*. You probably won't need to use this policy in your own code.

12.3 Unique pointers

Passing a STL unique pointer embodies an ownership transfer—a return value policy annotation is therefore not needed. To bind functions that receive or return `std::unique_ptr<..>`, add the extra include directive

```
#include <nanobind/stl/unique_ptr.h>
```

Note: While this this header file technically contains a *type caster*, it is *not* affected by their usual limitations (mandatory copy/conversion, inability to mutate function arguments).

Example: The following example binds two functions that create and consume instances of a C++ type `Data` via unique pointers.

```
#include <nanobind/stl/unique_ptr.h>

namespace nb = nanobind;

NB_MODULE(my_ext, m) {
    struct Data { };
    nb::class_<Data>(m, "Data");
    m.def("create", []() { return std::make_unique<Data>(); });
    m.def("consume", [](std::unique_ptr<Data> x) { /* no-op */ });
}
```

Calling a function taking a unique pointer from Python invalidates the passed Python object. nanobind will refuse further use of it:

```
Python 3.11.1 (main, Dec 23 2022, 09:28:24) [Clang 14.0.0 (clang-1400.0.29.202)] on␣
 ↪darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import my_ext

>>> x = my_ext.create()
>>> my_ext.consume(x)

>>> my_ext.consume(x)
<stdin>:1: RuntimeWarning: nanobind: attempted to access an uninitialized instance of␣
 ↪type 'my_ext.Data'!

TypeError: consume(): incompatible function arguments. The following argument types␣
 ↪are supported:
    1. consume(arg: my_ext.Data, /) -> None

Invoked with types: my_ext.Data
```

We strongly recommend that you replace all use of `std::unique_ptr<T>` by `std::unique_ptr<T, nb::deleter<T>>` in your code. Without the latter type declaration, which references a custom nanobind-provided deleter *nb::deleter<T>*, nanobind cannot transfer ownership of objects constructed using *nb::init<...>* to C++ and will refuse to do so with an error message. Further detail on this special case can be found in the *advanced section* on object ownership.

12.4 Shared ownership

In a *shared ownership* model, an object can have multiple owners that each register their claim by holding a *reference*. The system keeps track of the total number of references and destroys the object once the count reaches zero. Passing such an object in a function call shares ownership between the caller and callee. nanobind makes this behavior seamless so that everything works regardless of whether caller/callee are written in C++ or Python.

12.4.1 Shared pointers

STL shared pointers (`std::shared_ptr<T>`) allocate a separate control block to keep track of the reference count, which makes them very general but also slightly less efficient than other alternatives.

nanobind's support for shared pointers requires an extra include directive:

```
#include <nanobind/stl/shared_ptr.h>
```

Note: While this this header file technically contains a *type caster*, it is *not* affected by their usual limitations (mandatory copy/conversion, inability to mutate function arguments).

You don't need to specify a return value policy annotation when a function returns a shared pointer.

nanobind's implementation of `std::shared_ptr` support typically allocates a new `shared_ptr` control block each time a Python object must be converted to `std::shared_ptr<T>`. The new `shared_ptr` "owns" a reference to the Python object, and its deleter drops that reference. This has the advantage that the Python portion of the object will be kept alive by its C++-side references (which is important when implementing C++ virtual methods in Python), but it can be inefficient when passing the same object back and forth between Python and C++ many times, and it means that the `use_count()` method of `std::shared_ptr` will return a value that does not capture all uses. Some of these problems can be mitigated by modifying `T` so that it inherits from `std::enable_shared_from_this<T>`. See the *advanced section* on object ownership for more details on the implementation.

nanobind has limited support for objects that inherit from `std::enable_shared_from_this<T>` to allow safe conversion of raw pointers to shared pointers. The safest way to deal with these objects is to always use `std::make_shared<T>(...)` when constructing them in C++, and always pass them across the Python/C++ boundary wrapped in an explicit `std::shared_ptr<T>`. If you do this, then there shouldn't be any surprises. If you will be passing raw `T*` pointers around, then read the *advanced section on object ownership* for additional caveats.

12.4.2 Intrusive reference counting

Intrusive reference counting is the most flexible and efficient way of handling shared ownership. The main downside is that you must adapt the base class of your object hierarchy to the needs of nanobind.

The core idea is to define base class (e.g. `Object`) common to all bound types requiring shared ownership. That class contains a builtin atomic counter (e.g., `m_ref_count`) and a Python object pointer (e.g., `m_py_object`).

```
class Object {
...
private:
    mutable std::atomic<size_t> m_ref_count { 0 };
    PyObject *m_py_object = nullptr;
};
```

The core idea is that such `Object` instances can either be managed by C++ or Python. In the former case, the `m_ref_count` field keeps track of the number of outstanding references. In the latter case, reference counting is handled by Python, and the `m_ref_count` field remains unused.

This is actually little wasteful—nanobind therefore ships with a more efficient reference counter sample implementation that supports both use cases while requiring only `sizeof(void*)` bytes of storage:

```
#include <nanobind/intrusive/counter.h>

class Object {
...
private:
    intrusive_counter m_ref_count;
};
```

Please read the dedicated *section on intrusive reference counting* for more details on how to set this up.

CHAPTER

THIRTEEN

FUNCTIONS

13.1 Binding annotations

Besides *keyword and default arguments*, *docstrings*, and *return value policies*, other function binding annotations can be specified to achieve different goals as described below.

13.2 Default arguments revisited

A noteworthy point about the previously discussed way of specifying *default arguments* is that nanobind immediately converts them into Python objects. Consider the following example:

```
nb::class_<MyClass>(m, "MyClass")
    .def("f", &MyClass::f, "value"_a = SomeType(123));
```

nanobind must be set up to deal with values of the type `SomeType` (via a prior instantiation of `nb::class_<SomeType>`), or an exception will be thrown.

The "preview" of the default argument in the function signature is generated using the object's `__str__` method. If not available, the signature may not be very helpful, e.g.:

```
>> help(my_ext.MyClass)

class MyClass(builtins.object)
 |  Methods defined here:
 ....
 |  f(...)
 |      f(self, value: my_ext.SomeType = <my_ext.SomeType object at 0x1004d7230>) ->␣
 ↪None
```

In such cases, you can either refine the implementation of the type in question or manually override how nanobind renders the default value using the *`.sig("string") method`*:

```
nb::class_<MyClass>(m, "MyClass")
    .def("f", &MyClass::f, "value"_a.sig("SomeType(123)") = SomeType(123));
```

13.3 Implicit conversions, and how to suppress them

Consider the following function taking a floating point value as input:

```
m.def("double", [](float x) { return 2.f * x; });
```

We can call this function using a Python `float`, but an `int` works just as well:

```
>>> my_ext.double(2)
4.0
```

nanobind performed a so-called *implicit conversion* for convenience. The same mechanism generalizes to custom types defining a *nb::init_implicit<T>()*-style constructor:

```
nb::class_<A>(m, "A")
    // Following this line, nanobind will automatically convert 'B' -> 'A' if needed
    .def(nb::init_implicit<B>());
```

This behavior is not always desirable—sometimes, it is better to give up or try another function overload. To achieve this behavior, use the *.noconvert()* method of the *nb::arg* annotation to mark the argument as *non-converting*. An example:

```
m.def("double", [](float x) { return 2.f * x; }, nb::arg("x").noconvert());
```

The same experiment now fails with a `TypeError`:

```
>>> my_ext.double(2)
TypeError: double(): incompatible function arguments. The following ↙
argument types are supported:
    1. double(x: float) -> float

Invoked with types: int
```

You may, of course, combine this with the `_a` shorthand notation (see the section on *keyword arguments*) or specify *unnamed* non-converting arguments using *nb::arg().noconvert()*.

Note: The number of *nb::arg* annotations must match the argument count of the function. To enable no-convert behaviour for just one of several arguments, you will need to specify *nb::arg().noconvert()* for that argument, and *nb::arg()* for the remaining ones.

13.4 None arguments

A common design pattern in C/C++ entails passing `nullptr` to pointer-typed arguments to indicate a missing value. Since nanobind cannot know whether a function uses such a convention, it refuses conversions from `None` to `nullptr` by default. For example, consider the following binding code:

```
struct Dog { };
const char *bark(Dog *dog) {
    return dog != nullptr ? "woof!" : "(no dog)";
}

NB_MODULE(my_ext, m) {
    nb::class_<Dog>(m, "Dog")
        .def(nb::init<>());
```

(continues on next page)

(continued from previous page)

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

Calling the function with `None` raises an exception:

```
>>> my_ext.bark(my_ext.Dog())
'woof!'
>>> my_ext.bark(None)
TypeError: bark(): incompatible function arguments. The following ↙
argument types are supported:
    1. bark(arg: my_ext.Dog, /) -> str
```

To switch to a more permissive behavior, call the *.none()* method of the *nb::arg* annotation:

```
m.def("bark", &bark, nb::arg("dog").none());
```

With this change, the function accepts `None`, and its signature also changes to reflect this fact.

```
>>> my_ext.bark(None)
'(no dog)'

>>> my_ext.bark.__doc__
'bark(dog: Optional[my_ext.Dog]) -> str'
```

You may also specify a `None` default argument value, in which case the annotation can be omitted:

```
m.def("bark", &bark, nb::arg("dog") = nb::none());
```

Note that passing values *by pointer* (including null pointers) is only supported for *bound* types. *Type casters* and *wrappers* cannot be used in such cases and will produce compile-time errors.

Alternatively, you can also use `std::optional<T>` to pass an optional argument *by value*. To use it, you must include the header file associated needed by its type caster:

```
#include <nanobind/stl/optional.h>

NB_MODULE(my_ext, m) {
    m.def("bark", [](std::optional<Dog> d) { ... }, nb::arg("dog") = nb::none());
}
```

13.5 Overload resolution order

nanobind relies on a two-pass scheme to determine the right implementation when a bound function or method with multiple overloads is called from Python.

The first pass attempts to call each overload while disabling implicit argument conversion—it's as if every argument had a matching *nb::arg().noconvert()* annotation as described *above*. The process terminates successfully when nanobind finds an overload that is compatible with the provided arguments.

If the first pass fails, a second pass retries all overloads while enabling implicit argument conversion. If the second pass also fails, the function dispatcher raises a `TypeError`.

Within each pass, nanobind tries overloads in the order in which they were registered. Consequently, it prefers an overload that does not require implicit conversion to one that does, but otherwise prefers earlier-defined overloads to later-defined ones. Within the second pass, the precise number of implicit conversions needed does not influence the order.

The special exception *nb::next_overload* can also influence overload resolution. Raising this exception from an overloaded function causes it to be skipped, and overload resolution resumes. This can be helpful in complex situations where the value of a parameter must be inspected to see if a particular overload is eligible.

13.6 Accepting *args and **kwargs

Python supports functions that accept an arbitrary number of positional and keyword arguments:

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

Such functions can also be created using nanobind:

```
void generic(nb::args args, nb::kwargs kwargs) {
    for (auto v: args)
        nb::print(nb::str("Positional: {}").format(v));
    for (auto kv: kwargs)
        nb::print(nb::str("Keyword: {} -> {}").format(kv.first, kv.second));
}

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

The class *nb::args* derives from *nb::tuple* and *nb::kwargs* derives from *nb::dict*.

You may also use them individually or even combine them with ordinary parameters. Note that *nb::kwargs* must be the last parameter if it is specified, and any parameters after *nb::args* are implicitly *keyword-only*, just like in regular Python.

13.7 Expanding *args and **kwargs

Conversely, nanobind can also expand standard containers to add positional and keyword arguments to a Python call. The example below shows how to do this using the wrapper types *nb::object*, *nb::callable*, *nb::list*, *nb::dict*

```
nb::object my_call(nb::callable callable) {
    nb::list list;
    nb::dict dict;

    list.append("positional");
    dict["keyword"] = "value";

    return callable(1, *list, **dict);
}

NB_MODULE(my_ext, m) {
    m.def("my_call", &my_call);
}
```

Here is an example use of the above extension in Python:

```
>>> def x(*args, **kwargs):
...     print(args)
...     print(kwargs)
...
```

(continues on next page)

(continued from previous page)

```
>>> import my_ext
>>> my_ext.my_call(x)
(1, 'positional')
{'keyword': 'value'}
```

13.8 Keyword-only parameters

Python supports keyword-only parameters; these can't be filled positionally, thus requiring the caller to specify their name. They can be used to enforce more clarity at call sites if a function has multiple paramaters that could be confused with each other, or to accept named options alongside variadic `*args`.

```
def example(val: int, *, check: bool) -> None:
    # val can be passed either way; check must be given as a keyword arg
    pass

example(val=42, check=True)   # good
example(check=False, val=5)   # good
example(100, check=True)      # good
example(200, False)           # TypeError:
    # example() takes 1 positional argument but 2 were given

def munge(*args: int, invert: bool = False) -> int:
    return sum(args) * (-1 if invert else 1)

munge(1, 2, 3)                # 6
munge(4, 5, 6, invert=True)   # -15
```

nanobind provides a *nb::kw_only()* annotation that allows you to produce bindings that behave like these examples. It must be placed before the *nb::arg()* annotation for the first keyword-only parameter; you can think of it as equivalent to the bare `*,` in a Python function signature. For example, the above examples could be written in C++ as:

```
void example(int val, bool check);
int munge(nb::args args, bool invert);

m.def("example", &example,
      nb::arg("val"), nb::kw_only(), nb::arg("check"));

// Parameters after *args are implicitly keyword-only:
m.def("munge", &munge,
      nb::arg("args"), nb::arg("invert"));

// But you can be explicit about it too, as long as you put the
// kw_only annotation in the correct position:
m.def("munge", &munge,
      nb::arg("args"), nb::kw_only(), nb::arg("invert"));
```

Note: nanobind does *not* support the `pos_only()` argument annotation provided by pybind11, which marks the parameters before it as positional-only. However, a parameter can be made effectively positional-only by giving it no name (using an empty *nb::arg()* specifier).

13.9 Function templates

Consider the following function signature with a *template parameter*:

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

A template must be instantiated with concrete types to be usable, which is a compile-time operation. The generic version version therefore cannot be used in bindings:

```
m.def("process", &process); // <-- this will not compile
```

You must bind each instantiation separately, either as a single function with overloads, or as separately named functions.

```
// Option 1:
m.def("process", &process<int>);
m.def("process", &process<std::string>);

// Option 2:
m.def("process_int", &process<int>);
m.def("process_string", &process<std::string>);
```

13.10 Lifetime annotations

The *nb::keep_alive<Nurse, Patient>()* annotation 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.

The example below applies the annotation to a hypothetical operation that appends an entry to a log data structure.

```
nb::class_<Log>(m, "Log")
    .def("append",
         [](Log &log, Entry *entry) -> void { ... },
         nb::keep_alive<1, 2>());
```

Here, `Nurse = 1` refers to the `log` argument, while `Patient = 2` refers to `entry`. Setting `Nurse/Patient = 0` would select the function return value (here, the function doesn't return anything, so `0` is not a valid choice).

The example uses the annotation to tie the lifetime of the `entry` to that of `log`. Without it, Python could potentially delete `entry` *before* `log`, which would be problematic if the `log.append()` operation causes `log` to reference `entry` through a pointer address instead of making a copy. Whether or not this is a good design is another question (for example, shared ownership via `std::shared_ptr<T>` or intrusive reference counting would avoid the problem altogether).

See the definition of *nb::keep_alive* for further discussion and limitations of this method.

13.11 Call guards

The *nb::call_guard<T>()* annotation allows any scope guard `T` to be placed around the function call. For example, this definition:

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

is equivalent to the following pseudocode:

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

The only requirement is that `T` is default-constructible, but otherwise any scope guard will work. This feature is often combined with *nb::gil_scoped_release* to release the Python *global interpreter lock* (GIL) during a long-running C++ routine to permit parallel execution.

Multiple guards should be specified as *nb::call_guard<T1, T2, T3...>*. Construction occurs left to right, while destruction occurs in reverse.

If your wrapping needs are more complex than *nb::call_guard<T>()* can handle, it is also possible to define a custom "call policy", which can observe or modify the Python object arguments and observe the return value. See the documentation of *nb::call_policy<Policy>* for details.

13.12 Higher-order functions

The C++11 standard introduced lambda functions and the generic polymorphic function wrapper `std::function<>`, 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 use of the *nb::cpp_function* conversion function. Usage is similar to defining methods of classes:

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

After including the extra header file `nanobind/stl/function.h`, it is almost trivial to generate binding code for all of these functions.

```
#include <nanobind/stl/function.h>

NB_MODULE(my_ext, m) {
    m.def("func_arg", &func_arg);
    m.def("func_ret", &func_ret);
```

(continues on next page)

(continued from previous page)

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

The following interactive session shows how to call them from Python.

```
Python 3.11.1 (main, Dec 23 2022, 09:28:24) [Clang 14.0.0 (clang-1400.0.29.202)] on␣
 ↪darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import my_ext
>>> def square(i):
...     return i*i
...
>>> my_ext.func_arg(square)
100
>>> square_plus_1 = my_ext.func_ret(square)
>>> square_plus_1(4)
17
>>> plus_1 = my_ext.func_cpp()
>>> plus_1.__doc__
'<anonymous>(number: int) -> int'
>>> plus_1(number=43)
44
```

Note: This functionality is very useful when generating bindings for callbacks in C++ libraries (e.g. GUI libraries, asynchronous networking libraries, etc.).

13.13 Minimizing binding overheads

The code that dispatches function calls from Python to C++ is in general *highly optimized*. When it is important to further reduce binding overheads to an absolute minimum, consider removing annotations for *keyword and default arguments* along with other advanced binding annotations.

In the snippet below, `f1` has lower binding overheads compared to `f2`.

```
NB_MODULE(my_ext, m) {
    m.def("f1", [](int) { /* no-op */ });
    m.def("f2", [](int) { /* no-op */ }, "arg"_a);
}
```

This is because `f1`:

1. Does *not* use any of the following advanced argument annotations features:
   - **Named function arguments**, e.g., *`nb::arg("name")`* or `"name"_a`.
   - **Default argument values**, e.g., *`nb::arg() = 0`* or `"name"_a = false`.
   - **Nullability** or **implicit conversion** flags, e.g., *`nb::arg().none()`* or *`"name"_a.noconvert()`*.
2. Has no *`nb::keep_alive<Nurse, Patient>()`* annotations.
3. Takes no variable-length positional (*`nb::args`*) or keyword (*`nb::kwargs`*) arguments.
4. Has a to total of **8 or fewer** function arguments.

If all of the above conditions are satisfied, nanobind switches to a specialized dispatcher that is optimized to handle a small number of positional arguments. Otherwise, it uses the default dispatcher that works in any situation. It

is also worth noting that functions with many overloads generally execute more slowly, since nanobind must first select a suitable one.

These differences are mainly of interest when a function that does *very little* is called at a *very high rate*, in which case binding overheads can become noticeable.

Regarding point 1 of the above list, note that **locking** is okay, as long as the annotation does not provide an argument name. In other words, a function binding with a `nb::arg().lock()` for some of its arguments stays on the fast path. This is mainly of interest for *free-threaded* extensions.

CHAPTER

FOURTEEN

CLASSES

The material below builds on the section on *binding custom types* and reviews advanced scenarios involving object-oriented code.

14.1 Frequently used

Click on the following *nb::class_<..>::def_** members for examples on how to bind various different kinds of methods, fields, etc.

Type	method
Methods & constructors	*.def()*
Fields	*.def_ro()*, *.def_rw()*
Properties	*.def_prop_ro()*, *.def_prop_rw()*
Static methods	*.def_static()*
Static fields	*.def_ro_static()*, *.def_rw_static()*
Static properties	*.def_prop_ro_static()*, *.def_prop_rw_static()*

14.2 Subclasses

Consider the following two data structures with an inheritance relationship:

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

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

To indicate the inheritance relationship to nanobind, specify the C++ base class as an extra template parameter of *nb::class_<..>*:

```
#include <nanobind/stl/string.h>

NB_MODULE(my_ext, m) {
    nb::class_<Pet>(m, "Pet")
       .def(nb::init<const std::string &>())
       .def_rw("name", &Pet::name);

    nb::class_<Dog, Pet /* <- C++ parent type */>(m, "Dog")
        .def(nb::init<const std::string &>())
```

(continues on next page)

(continued from previous page)

```
        .def("bark", &Dog::bark);
}
```

Alternatively, you can also pass the type object as an ordinary parameter.

```
auto pet = nb::class_<Pet>(m, "Pet")
   .def(nb::init<const std::string &>())
   .def_rw("name", &Pet::name);

nb::class_<Dog>(m, "Dog", pet /* <- Parent type object */)
    .def(nb::init<const std::string &>())
    .def("bark", &Dog::bark);
```

Instances expose fields and methods of both types as expected:

```
>>> d = my_ext.Dog("Molly")
>>> d.name
'Molly'
>>> d.bark()
'Molly: woof!'
```

14.3 Automatic downcasting

nanobind obeys signatures when returning regular non-polymorphic C++ objects from functions: building on the *previous example*, consider the following function that returns a `Dog` object as a `Pet` base pointer.

```
m.def("pet_store", []() { return (Pet *) new Dog{"Molly"}; });
```

nanobind cannot safely determine that this is in fact an instance of the `Dog` subclass. Consequently, only fields and methods of the base type remain accessible:

```
>>> p = my_ext.pet_store()
>>> type(p)
<class 'my_ext.Pet'>
>>> p.bark()
AttributeError: 'Pet' object has no attribute 'bark'
```

In C++, a type is only considered polymorphic if it (or one of its base classes) has at least one *virtual function*. Let's add a virtual default destructor to make `Pet` and its subtypes polymorphic.

```
struct Pet {
    virtual ~Pet() = default;
    std::string name;
};
```

With this change, nanobind is able to inspect the returned C++ instance's virtual table and infer that it can be represented by a more specialized Python object of type `my_ext.Dog`.

```
>>> p = my_ext.pet_store()
>>> type(p)
<class 'my_ext.Dog'>
>>> p.bark()
'Molly: woof!'
```

Note: Automatic downcasting of polymorphic instances is only supported when the subtype has been registered using *nb::class_<..>*. Otherwise, the return type listed in the function signature takes precedence.

14.4 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_; }

    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.

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

Here, `nb::overload_cast` only requires the parameter types to be specified, and it deduces the return type.

Note: In cases where a function overloads by `const`-ness, an additional `nb::const_` parameter is needed to select the right overload, e.g., `nb::overload_cast<int>(&Pet::get, nb::const_)`.

To define overloaded constructors, simply declare one after the other using the normal *.def(nb::init<...>())* syntax.

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

```
>>> help(my_ext.Pet)
class Pet(builtins.object)
 |  Methods defined here:
 |
 |  __init__(...)
 |      __init__(self, arg0: str, arg1: int, /) -> None
 |
 |  set(...)
 |      set(self, arg: int, /) -> None
 |      set(self, arg: str, /) -> None
 |
 |      Overloaded function.
 |
 |      1. ``set(self, arg: int, /) -> None``
 |
 |      Set the pet's age
```

(continues on next page)

(continued from previous page)

```
 |
 |      2. ``set(self, arg: str, /) -> None``
 |
 |      Set the pet's name
```

The format of the docstring with a leading overload list followed by a repeated list with details is designed to be compatible with the Sphinx documentation generator.

14.5 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:

```
nb::class_<Pet> pet(m, "Pet");

pet.def(nb::init<const std::string &, Pet::Kind>())
    .def_rw("name", &Pet::name)
    .def_rw("type", &Pet::type)
    .def_rw("attr", &Pet::attr);

nb::enum_<Pet::Kind>(pet, "Kind")
    .value("Dog", Pet::Kind::Dog)
    .value("Cat", Pet::Kind::Cat)
    .export_values();

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

To ensure that the nested types `Kind` and `Attributes` are created within the scope of `Pet`, the `pet` type object is passed as the `scope` argument of the subsequent *nb::enum_<T>* and *nb::class_<T>* binding declarations. The *.export_values()* function exports the enumeration entries into the parent scope, which should be skipped for newer C++11-style strongly typed enumerations.

```
>>> from my_ext import Pet
>>> p = Pet("Lucy", Pet.Cat)
>>> p.attr.age = 3
```

(continues on next page)

(continued from previous page)

```
>>> p.type
my_ext.Kind.Cat
>>> p.type.__name__
'Cat'
>>> int(p.type)
1
```

Note: When the annotation *nb::is_arithmetic()* is passed to *nb::enum_<T>*, the resulting Python type will support arithmetic and bit-level operations (and, or, xor, negation). The operands of these operations may be either enumerators. When the annotation *nb::is_flag()* is passed to *nb::enum_<T>*, the resulting Python type will be a class derived from `enum.Flag`, meaning its enumerators can be combined using bit-wise operators in a type-safe way: the result will have the same enumeration type as the operands, and only enumerators of the same type can be combined. When passing both `is_arithmetic` and `is_flag`, the resulting Python type will be `enum.IntFlag`, supporting both arithmetic and bit-wise operations.

```
nb::enum_<Pet::Kind>(pet, "Kind", nb::is_arithmetic())
   ...
```

By default, these are omitted.

14.6 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_rw()* or *class_::def_prop_rw()*.

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

Trying to set any other attribute results in an error:

```
>>> p = my_ext.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 *nb::dynamic_attr* tag must be added to the *nb::class_* constructor:

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

Now everything works as expected:

```
>>> p = my_ext.Pet()
>>> p.name = "Charly"  # OK, overwrite value in C++
>>> p.age = 2  # OK, dynamically add a new attribute
```

Note that there is a small runtime cost for a class with dynamic attributes. Not only because of the addition of an instance dictionary, 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, nanobind classes are more efficient than native Python classes. Enabling dynamic attributes just brings them on par.

14.7 Weak references

By default, nanobind instances cannot be referenced via Python's `weakref` class, and attempting to do so will raise an exception.

To support this, add the *nb::is_weak_referenceable* tag to the *nb::class_* constructor. Note that this will increase the size of every instance by `sizeof(void*)` due to the need to store a weak reference list.

```
nb::class_<Pet>(m, "Pet", nb::is_weak_referenceable());
```

14.8 Extending C++ classes in Python

Bound C++ types can be extended within Python, which is helpful to dynamically extend compiled code with further fields and other functionality. Bind classes with the *is_final* annotation to forbid subclassing.

Consider the following example bindings of a `Dog` and `DogHouse` class.

```
#include <nanobind/stl/string.h>

namespace nb = nanobind;

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

struct DogHouse {
    Dog dog;
};

NB_MODULE(my_ext, m) {
    nb::class_<Dog>(m, "Dog")
        .def(nb::init<const std::string &>())
        .def("bark", &Dog::bark)
        .def_rw("name", &Dog::name);

    nb::class_<DogHouse>(m, "DogHouse")
        .def(nb::init<Dog>())
        .def_rw("dog", &DogHouse::dog);
}
```

The following Python snippet creates a new `GuardDog` type that extends `Dog` with an `.alarm()` method.

```
>>> import my_ext
>>> class GuardDog(my_ext.Dog):
...     def alarm(self, count = 3):
...         for i in range(count):
...             print(self.bark())
...
>>> gd = GuardDog("Max")
>>> gd.alarm()
Max: woof!
Max: woof!
Max: woof!
```

This Python subclass is best thought of as a "rich wrapper" around an existing C++ base object. By default, that wrapper will disappear when nanobind makes a copy or transfers ownership to C++.

```
>>> d = my_ext.DogHouse()
>>> d.dog = gd
>>> d.dog.alarm()
AttributeError: 'Dog' object has no attribute 'alarm'
```

To preserve it, adopt a shared ownership model using *shared pointers* or *intrusive reference counting*. For example, updating the code as follows fixes the problem:

```
#include <nanobind/stl/shared_ptr.h>

struct DogHouse {
    std::shared_ptr<Dog> dog;
};
```

```
>>> d = my_ext.DogHouse()
>>> d.dog = gd
>>> d.dog.alarm()
Max: woof!
Max: woof!
Max: woof!
```

14.9 Overriding virtual functions in Python

Building on the previous example on *inheriting C++ types in Python*, let's investigate how a C++ *virtual function* can be overridden in Python. In the code below, the virtual method `bark()` is called by a global `alarm()` function (now written in C++).

```
#include <iostream>

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

void alarm(Dog *dog, size_t count = 3) {
    for (size_t i = 0; i < count; ++i)
        std::cout << dog->bark() << std::endl;
}
```

Normally, the binding code would look as follows:

```
#include <nanobind/stl/string.h>

namespace nb = nanobind;
using namespace nb::literals;

NB_MODULE(my_ext, m) {
    nb::class_<Dog>(m, "Dog")
        .def(nb::init<const std::string &>())
        .def("bark", &Dog::bark)
        .def_rw("name", &Dog::name);

    m.def("alarm", &alarm, "dog"_a, "count"_a = 3);
}
```

However, this don't work as expected. We can subclass and override without problems, but virtual function calls originating from C++ aren't being propagated to Python:

```
>>> class ShihTzu(my_ext.Dog):
...     def bark(self):
...         return self.name + ": yip!"
...

>>> dog = ShihTzu("Mr. Fluffles")

>>> dog.bark()
Mr. Fluffles: yip!

>>> my_ext.alarm(dog)
Mr. Fluffles: woof!     # <-- oops, alarm() is calling the base implementation
Mr. Fluffles: woof!
Mr. Fluffles: woof!
```

To fix this behavior, you must implement a *trampoline class*. A trampoline has the sole purpose of capturing virtual function calls in C++ and forwarding them to Python.

```
#include <nanobind/trampoline.h>

struct PyDog : Dog {
    NB_TRAMPOLINE(Dog, 1);

    std::string bark() const override {
        NB_OVERRIDE(bark);
    }
};
```

This involves an additional include directive and the line *NB_TRAMPOLINE(Dog, 1)* to mark the class as a trampoline for the `Dog` base type. The count (1) denotes to the total number of virtual method slots that can be overridden within Python.

Note: The number of virtual method slots is used to preallocate memory. Trampoline declarations with an insufficient size may eventually trigger a Python `RuntimeError` exception with a descriptive label, e.g.:

```
nanobind::detail::get_trampoline('PyDog::bark()'): the trampoline ran out of
slots (you will need to increase the value provided to the NB_TRAMPOLINE() macro)
```

The macro *NB_OVERRIDE(bark)* intercepts the virtual function call, checks if a Python override exists, and forwards the call in that case. If no override was found, it falls back to the base class implementation. You will need to replicate this pattern for every method that should support overriding in Python.

The macro accepts an variable argument list to pass additional parameters. For example, suppose that the virtual function `bark()` had an additional `int volume` parameter—in that case, the syntax would need to be adapted as follows:

```
std::string bark(int volume) const override {
    NB_OVERRIDE(bark, volume);
}
```

The macro *NB_OVERRIDE_PURE()* should be used for pure virtual functions, and *NB_OVERRIDE()* should be used for functions which have a default implementation. There are also two alternate macros *NB_OVERRIDE_PURE_NAME()* and *NB_OVERRIDE_NAME()* which take a string as first argument to specify the name of function in Python. This is useful when the C++ and Python versions of the function have different names (e.g., `operator+` vs `__add__`).

The binding code needs a tiny adaptation (highlighted) to inform nanobind of the trampoline that will be used whenever Python code extends the C++ class.

```
nb::class_<Dog, PyDog /* <-- trampoline */>(m, "Dog")
```

If the *nb::class_<..>* declaration also specifies a base class, you may specify it and the trampoline in either order. Also, note that binding declarations should be made against the actual class, not the trampoline:

```
nb::class_<Dog, PyDog>(m, "Dog")
   .def(nb::init<const std::string &>())
   .def("bark", &PyDog::bark); /* <--- THIS IS WRONG, use &Dog::bark */
```

With the trampoline in place, our example works as expected:

```
>>> my_ext.alarm(dog)
Mr. Fluffles: yip!
Mr. Fluffles: yip!
Mr. Fluffles: yip!
```

The following special case needs to be mentioned: you *may not* implement a Python trampoline for a method that returns a reference or pointer to a type requiring *type casting*. For example, attempting to expose a hypothetical virtual method `const std::string &get_name() const` as follows

```
const std::string &get_name() const override {
    NB_OVERRIDE(get_name);
}
```

will fail with a static assertion failure:

```
include/nanobind/nb_cast.h:352:13: error: static_assert failed due to requirement '...
 ↪'
"nanobind::cast(): cannot return a reference to a temporary."
```

This is not a fluke. The Python would return a `str` object that nanobind can easily type-cast into a temporary `std::string` instance. However, when the virtual function call returns on the C++ side, that temporary will already have expired. There isn't a good solution to this problem, and nanobind therefore simply refuses to do it. You will need to change your approach by either using *bindings* instead of *type casters* or changing your virtual method interfaces to return by value.

14.10 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 { return Vector2(-x, -y); }
    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 to_string() 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 <nanobind/operators.h>

NB_MODULE(my_ext, m) {
    nb::class_<Vector2>(m, "Vector2")
        .def(nb::init<float, float>())
        .def(nb::self + nb::self)
        .def(nb::self += nb::self)
        .def(nb::self *= float())
        .def(float() * nb::self)
        .def(nb::self * float())
        .def(-nb::self)
        .def("__repr__", &Vector2::to_string);
}
```

Note that a line involving *nb::self* like

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

is really just short hand notation for

```
.def("__mul__", [](const Vector2 &a, float b) {
    return a * b;
}, nb::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 *nb::is_operator()* flag marker is needed to inform nanobind that this is an operator, which returns `NotImplemented` when invoked with incompatible arguments rather than throwing a type error.

When binding *in-place* operators such as `operator+=`, and when their implementation is guaranteed to end with `return *this`, it is recommended that you set a return value policy of *rv_policy::none*, i.e.,

```
.def(nb::self += nb::self, nb::rv_policy::none)
```

Otherwise, the function binding will return a new copy of the object, which is usually not desired.

14.11 Binding protected member functions

It's normally not possible to expose `protected` member functions to Python:

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

nb::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
};

nb::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:
    NB_TRAMPOLINE(A, 1);
    int foo() const override { NB_OVERRIDE(foo); }
};

class Publicist : public A {
```

(continues on next page)

(continued from previous page)

```
public:
    using A::foo;
};

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

14.12 Binding classes with template parameters

nanobind can also wrap classes that have template parameters. Consider these classes:

```
struct Cat {};
struct Dog {};

template <typename PetType> struct PetHouse {
    PetHouse(PetType& pet);
    PetType& get();
};
```

C++ templates may only be instantiated at compile time, so nanobind can only wrap instantiated templated classes. You cannot wrap a non-instantiated template:

```
// BROKEN (this will not compile)
nb::class_<PetHouse>(m, "PetHouse");
    .def("get", &PetHouse::get);
```

You must explicitly specify each template/type combination that you want to wrap separately.

```
// ok
nb::class_<PetHouse<Cat>>(m, "CatHouse")
    .def("get", &PetHouse<Cat>::get);

// ok
nb::class_<PetHouse<Dog>>(m, "DogHouse")
    .def("get", &PetHouse<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);
};

nb::class_<MyClass<int>>(m, "MyClassT")
    .def("fn", &MyClass<int>::fn<std::string>);
```

14.13 Tag-based polymorphism

The section on *automatic downcasting* explained how nanobind can infer the type of polymorphic C++ objects at runtime. It can be desirable to extend this automatic downcasting behavior to non-polymorphic classes, for example to support *tag-based polymorphism*. In this case, instances expose a method or field to identify their type.

For example, consider the following class hierarchy where `Pet::kind` serves this purpose:

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

namespace nb = nanobind;

enum class PetKind { Cat, Dog };

struct Pet { const PetKind kind; };
struct Dog : Pet { Dog() : Pet{PetKind::Dog} { } };
struct Cat : Pet { Cat() : Pet{PetKind::Cat} { } };

namespace nb = nanobind;

NB_MODULE(my_ext, m) {
    nb::class_<Pet>(m, "Pet");
    nb::class_<Dog>(m, "Dog");
    nb::class_<Cat>(m, "Cat");

    nb::enum_<PetKind>(m, "PetKind")
        .value("Cat", PetKind::Cat)
        .value("Dog", PetKind::Dog);

    m.def("make_pet", [](PetKind kind) -> Pet* {
        switch (kind) {
            case PetKind::Dog: return new Dog();
            case PetKind::Cat: return new Cat();
        }
    });
}
```

This code initially doesn't work as expected (the `make_pet` function binding always creates instances of the `Pet` base class).

```
>>> my_ext.make_pet(my_ext.PetKind.Cat)
<my_ext.Pet object at 0x10305ee10>

>>> my_ext.make_pet(my_ext.PetKind.Dog)
<my_ext.Pet object at 0x10328e530>
```

To fix this, partially specialize the `type_hook` class to provide the `type_hook<T>::get()` method:

```
namespace nanobind::detail {
    template <> struct type_hook<Pet> {
        static const std::type_info *get(Pet *p) {
            if (p) {
                switch (p->kind) {
                    case PetKind::Dog: return &typeid(Dog);
                    case PetKind::Cat: return &typeid(Cat);
                }
            }
            return &typeid(Pet);
```

(continues on next page)

(continued from previous page)

```
        }
    };
} // namespace nanobind::detail
```

The method will be invoked whenever nanobind needs to convert a C++ pointer of type `T*` to a Python object. It should inspect the instance and return a pointer to a suitable RTTI record. With this override, downcasting works as expected:

```
>>> my_ext.make_pet(my_ext.PetKind.Cat)
<my_ext.Cat object at 0x104da6e10>

>>> my_ext.make_pet(my_ext.PetKind.Dog)
<my_ext.Dog object at 0x104da6ef0>
```

14.14 Binding unions

nb::class_<..> can also be used to provide bindings for unions. A basic and useless example:

```
union Example {
    int ival;
    double dval;

    std::string to_string(size_t active_idx) const {
        return active_idx == 1 ? std::to_string(dval) : std::to_string(ival);
    }
};
static_assert(sizeof(Example) == sizeof(double));

nb::class_<Example>(m, "Example")
    .def_rw("ival", &Example::ival)
    .def_rw("dval", &Example::dval)
    .def("to_string", &Example::to_string);
```

```
>>> u = my_ext.Example()
>>> u.ival = 42
>>> u.to_string(0)
'42'
>>> u.dval = 1.25
>>> u.to_string(1)
'1.250000'
```

Direct binding of union variant members is only safe if all members of the union are trivially copyable types (as in this example), but more complex unions can also be supported by binding lambdas or member functions that enforce the necessary invariants.

This is a low-level feature and should be used with care; even when all members are trivially copyable, reading from a union member other than the most recently written one produces undefined behavior in C++. Unless you need to bind an existing API that uses union types, you're probably better off using `std::variant<..>`, which knows what member is active and can thus enforce all the ncessary invariants for you.

14.15 Pickling

To pickle and unpickle objects bound using nanobind, expose the `__getstate__` and `__setstate__` methods. They should return and retrieve the internal instance state using representations that themselves support pickling. The example below, e.g., does this using a tuple.

The `__setstate__` method should construct the object in-place analogous to custom `__init__`-style constructors.

```
#include <nanobind/stl/tuple.h>

struct Pet {
     std::string name;
     int age;
     Pet(const std::string &name, int age) : name(name), age(age) { }
};

NB_MODULE(my_ext, m) {
    nb::class_<Pet>(m, "Pet")
       // ...
       .def("__getstate__", [](const Pet &pet) { return std::make_tuple(pet.name, pet.
 ↪age); })
       .def("__setstate__", [](Pet &pet, const std::tuple<std::string, int> &state) {
            new (&pet) Pet(
                std::get<0>(state),
                std::get<1>(state)
            );
       });
 }
```

14.16 Customizing Python object creation

Sometimes you might need to bind a class that can't be constructed in the usual way:

```
class Pet {
  private:
    Pet(/* ... */);
  public:
    static std::unique_ptr<Pet> make(std::string name, int age);
    void speak();
};
```

You can use *.def_static()* to produce bindings that let you write `Pet.make("Fido", 2)` in Python, just like you would write `Pet::make("Fido", 2)` in C++. But sometimes it's nice to provide a more Pythonic interface than that, like `Pet("Fido", 2)`. To do that, nanobind lets you override `__new__`.

Since this is a rarely-used feature in Python, let's recap. Object initialization in Python occurs in two phases:

- the *constructor*, `__new__`, allocates memory for the object;
- the *initializer*, `__init__`, sets up the object's initial state.

So far, all the ways we've seen of binding C++ constructors (*nb::init<..>()*, `.def("__init__", ...)`) produce Python object *initializers*. nanobind augments these with its own Python object constructor, which allocates a Python object that has space in its memory layout for the C++ object to slot in. The `__init__` method then fills in that space by calling a C++ constructor.

This split between `__new__` and `__init__` has a lot of benefits, including a reduction in unnecessary allocations, but it does mean that anything created from Python must be able to control where its C++ innards are stored.

Sometimes, as with the example of `Pet` above, that's not feasible. In such cases, you can go down one level and override `__new__` directly:

```
nb::class_<Pet>(m, "Pet")
    .def(nb::new_(&Pet::make), "name"_a, "age"_a)
    .def("speak", &Pet::speak);
```

Passing *nb::new_* to *.def()* here creates two magic methods on `Pet`:

- A `__new__` that uses the given function to produce a new `Pet`. It is converted to a Python object in the same way as the return value of any other function you might write bindings for. In particular, you can pass a *nb::rv_policy* as an additional argument to *.def()* to control how this conversion occurs.
- A `__init__` that takes the same arguments as `__new__` but performs no operation. This is necessary because Python automatically calls `__init__` on the object returned by `__new__` in most cases.

You can provide a lambda as the argument of *nb::new_*. This is most useful when the lambda returns a pointer or smart pointer; if it's returning a value, then `.def("__init__", ...)` will have better performance. Additionally, you can chain multiple calls to `.def(nb::new_(...))` in order to create an overload set. The following example demonstrates both of these capabilities:

```
nb::class_<Pet>(m, "Pet")
    .def(nb::new_([]() { return Pet::make(getRandomName(), 0); }))
    .def(nb::new_(&Pet::make), "name"_a, "age"_a)
    .def("speak", &Pet::speak);
```

If you need even more control, perhaps because you need to access the type object that Python passes as the first argument of `__new__` (which *nb::new_* discards), you can write a `.def_static("__new__", ...)` and matching `.def("__init__", ...)` yourself.

Note: Unpickling an object of type `Foo` normally requires that `Foo.__new__(Foo)` produce something that `__setstate__` can be called on. Any custom *nb::new_* methods will not satisfy this requirement, because they return a fully-constructed object. In order to maintain pickle compatibility, nanobind by default will add an additional `__new__` overload that takes no extra arguments and calls the nanobind built-in *inst_alloc()*. This won't make your class constructible with no arguments, because there's no corresponding `__init__`; it just helps unpickling work. If your first *nb::new_* method is one that takes no arguments, then nanobind won't add its own, and you'll have to deal with unpickling some other way.

CHAPTER

FIFTEEN

EXCEPTIONS

15.1 Automatic conversion of C++ exceptions

When Python calls a C++ function, that function might raise an exception instead of returning a result. In such a case, nanobind will capture the C++ exception and then raise an equivalent exception within Python. This automatic conversion supports `std::exception`, common subclasses, and several classes that convert to specific Python exceptions as shown below:

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`
`nb::stop_iteration`	`StopIteration` (used to implement custom iterator)
`nb::index_error`	`IndexError` (used to indicate out of bounds access in `__getitem__`, `__setitem__`, etc.)
`nb::key_error`	`KeyError` (used to indicate an invalid access in `__getitem__`, `__setitem__`, etc.)
`nb::value_error`	`ValueError` (used to indicate an invalid value in operations like `container.remove(...)`)
`nb::type_error`	`TypeError`
`nb::buffer_error`	`BufferError`
`nb::import_error`	`ImportError`
`nb::attribute_error`	`AttributeError`
Any other exception	`SystemError`

Exception translation is not bidirectional. A C++ `catch (nb::key_error)` block will not catch a Python `KeyError`. Use *`nb::python_error`* for this purpose (see the *example below* for details).

The is also a special exception *`nb::cast_error`* that may be raised by the call operator *`nb::handle::operator()`* and *`nb::cast()`* when argument(s) cannot be converted to Python objects.

15.2 Handling custom exceptions

nanobind can also expose custom exception types. The *nb::exception<T>* helper resembles *nb::class_<T>* and registers a new exception type within the provided scope.

```
NB_MODULE(my_ext, m) {
    nb::exception<CppExp>(m, "PyExp");
}
```

Here, it creates `my_ext.PyExp`. Subsequently, any C++ exception of type `CppExp` crossing the language barrier will automatically convert to `my_ext.PyExp`.

A Python exception base class can optionally be specified. For example, the snippet below causes `PyExp` to inherit from `RuntimeError` (the default is `Exception`). The built-in Python exception classes are listed here.

```
nb::exception<CppExp>(module, "PyExp", PyExc_RuntimeError);
```

In more complex cases, *nb::register_exception_translator()* can be called to register a custom exception translation routine. It takes a stateless callable (e.g. a function pointer or a lambda function without captured variables) with the call signature `void(const std::exception_ptr &, void*)` and an optional payload pointer value that will be passed to the second parameter of the callable.

When a C++ exception is captured by nanobind, all registered exception translators are tried in reverse order of registration (i.e. the last registered translator has the first chance of handling the exception).

Inside the translator, call `std::rethrow_exception()` within a `try-catch` block to re-throw the exception and capture supported exception types. The `catch` block should call `PyErr_SetString` or `PyErr_Format` (1, 2) to set a suitable Python error status. The following example demonstrates this pattern to convert `MyCustomException` into a Python `IndexError`.

```
nb::register_exception_translator(
    [](const std::exception_ptr &p, void * /* unused */) {
        try {
            std::rethrow_exception(p);
        } catch (const MyCustomException &e) {
            PyErr_SetString(PyExc_IndexError, e.what());
        }
    });
```

Multiple exceptions can be handled by a single translator. nanobind captures unhandled exceptions and forwards them to the preceding translator. If none of the exception translators succeeds, it will convert according to the previously discussed default rules.

Note: When the exception translator returns normally, it must have set a Python error status. Otherwise, Python will crash with the message `SystemError: error return without exception set`.

Unsupported exception types should not be caught, or may be explicitly (re-)thrown to delegate them to the other exception translators.

15.3 Capturing Python exceptions within C++

When nanobind-based C++ code calls a Python function that raises an exception, it will automatically convert into a *nb::python_error* raised on the C++ side. This exception type can be caught and handled in C++ or propagate back into Python, where it will undergo reverse conversion.

Exception raised in Python	Translated to C++ exception type
Any Python `Exception`	*nb::python_error*

The class exposes various members to obtain further information about the exception. The *.type()* and *.value()* methods provide information about the exception type and value, while *.what()* generates a human-readable representation including a backtrace.

A use of the *.matches()* method to distinguish different exception types is shown below:

```
try {
    nb::object file = nb::module_::import_("io").attr("open")("file.txt", "r");
    nb::object text = file.attr("read")();
    file.attr("close")();
} catch (const nb::python_error &e) {
    if (e.matches(PyExc_FileNotFoundError)) {
        nb::print("file.txt not found");
    } else if (e.matches(PyExc_PermissionError)) {
        nb::print("file.txt found but not accessible");
    } else {
        throw;
    }
}
```

Note that the previously discussed *automatic conversion* of C++ exception does not apply here. Errors raised from Python *always* convert to *nb::python_error*.

15.4 Handling errors from the Python C API

Whenever possible, use *nanobind wrappers* instead of calling the Python C API directly. Otherwise, you must carefully manage reference counts and adhere to the nanobind error protocol outlined below.

When a Python C API call fails with an error status, you must immediately `throw nb::python_error();` to capture the error and handle it using appropriate C++ mechanisms. This includes calls to error setting functions such as `PyErr_SetString` (*custom exception translators* are excluded from this rule).

```
PyErr_SetString(PyExc_TypeError, "C API type error demo");
throw nb::python_error();

// But it would be easier to simply...
throw nb::type_error("nanobind wrapper type error");
```

Alternately, to ignore the error, call PyErr_Clear(). Any Python error must be thrown or cleared, or nanobind will be left in an invalid state.

15.5 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 nanobind, you can use the *nb::raise_from* function, which requires a *nb::python_error* and re-raises it with a chained exception object.

```
nb::callable f = ...;
int arg = 123;
try {
    f(arg);
} catch (nb::python_error &e) {
    nb::raise_from(e, PyExc_RuntimeError, "Could not call 'f' with %i", arg);
}
```

The function is internally based on the Python function `PyErr_FormatV` and takes `printf`-style arguments following the format descriptor.

An even lower-level interface is available via *nb::chain_error*.

15.6 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 are logged by Python as an unraisable error. In Python 3.8+, a system hook is triggered and an auditing event is logged.

Any noexcept function should have a try-catch block that traps *nb::python_error* (or any other exception that can occur). 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 (nb::python_error &e) {
        // 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.
        e.discard_as_unraisable(__func__);
    } catch (const std::exception &e) {
        // Log and discard C++ exceptions.
        third_party::log(e);
    }
}
```

CHAPTER

SIXTEEN

N-DIMENSIONAL ARRAYS

nanobind provides two alternative interfaces to exchange array data between Python and C++.

16.1 The nb::ndarray<..> class

nanobind can exchange n-dimensional arrays (henceforth "**nd-arrays**") with popular array programming frameworks including NumPy, PyTorch, TensorFlow, JAX, and CuPy. It supports *zero-copy* exchange using two protocols:

- The classic buffer protocol.
- DLPack, a GPU-compatible generalization of the buffer protocol.

nanobind knows how to talk to each framework and takes care of all the nitty-gritty details.

To use this feature, you must add the include directive

```
#include <nanobind/ndarray.h>
```

to your code. Following this, you can bind functions with *nb::ndarray<...>*-typed parameters and return values.

16.1.1 Array input arguments

A function that accepts an *nb::ndarray<>*-typed parameter (i.e., *without* template parameters) can be called with *any* writable array from any framework regardless of the device on which it is stored. The following example binding declaration uses this functionality to inspect the properties of an arbitrary input array:

```
m.def("inspect", [](const nb::ndarray<>& a) {
    printf("Array data pointer : %p\n", a.data());
    printf("Array dimension : %zu\n", a.ndim());
    for (size_t i = 0; i < a.ndim(); ++i) {
        printf("Array dimension [%zu] : %zu\n", i, a.shape(i));
        printf("Array stride    [%zu] : %zd\n", i, a.stride(i));
    }
    printf("Device ID = %u (cpu=%i, cuda=%i)\n", a.device_id(),
        int(a.device_type() == nb::device::cpu::value),
        int(a.device_type() == nb::device::cuda::value)
    );
    printf("Array dtype: int16=%i, uint32=%i, float32=%i\n",
        a.dtype() == nb::dtype<int16_t>(),
        a.dtype() == nb::dtype<uint32_t>(),
        a.dtype() == nb::dtype<float>()
    );
});
```

Below is an example of what this function does when called with a NumPy array:

```
>>> my_module.inspect(np.array([[1,2,3], [3,4,5]], dtype=np.float32))
Array data pointer : 0x1c30f60
Array dimension : 2
Array dimension [0] : 2
Array stride    [0] : 3
Array dimension [1] : 3
Array stride    [1] : 1
Device ID = 0 (cpu=1, cuda=0)
Array dtype: int16=0, uint32=0, float32=1
```

16.1.2 Array constraints

In practice, it can often be useful to *constrain* what kinds of arrays constitute valid inputs to a function. For example, a function expecting CPU storage would likely crash if given a pointer to GPU memory, and nanobind should therefore prevent such undefined behavior. The *nb::ndarray<...>* class accepts template arguments to specify such constraints. For example the binding below guarantees that the implementation can only be called with CPU-resident arrays with shape (·,·,3) containing 8-bit unsigned integers.

```
using RGBImage = nb::ndarray<uint8_t, nb::shape<-1, -1, 3>, nb::device::cpu>;

m.def("process", [](RGBImage data) {
    // Double brightness of the MxNx3 RGB image
    for (size_t y = 0; y < data.shape(0); ++y)
        for (size_t x = 0; x < data.shape(1); ++x)
            for (size_t ch = 0; ch < 3; ++ch)
                data(y, x, ch) = (uint8_t) std::min(255, data(y, x, ch) * 2);
});
```

The above example also demonstrates the use of *operator()*, which provides direct read/write access to the array contents assuming that they are reachable through the CPU's virtual address space.

Overview

Overall, the following kinds of constraints are available:

- **Data type**: a type annotation like `float`, `uint8_t`, etc., constrain the numerical representation of the ndarray. Complex arrays (i.e., `std::complex<float>` or `std::complex<double>`) are also supported.
- **Constant arrays**: further annotating the data type with `const` makes it possible to call the function with constant arrays that do not permit write access. Without the annotation, calling the binding would fail with a `TypeError`.

  You can alternatively accept constant arrays of *any type* by not specifying a data type at all and instead passing the *nb::ro* annotation.
- **Shape**: The *nb::shape* annotation (as in `nb::shape<-1, 3>`) simultaneously constrains the number of array dimensions and the size per dimension. A value of `-1` leaves the size of the associated dimension unconstrained.

  nb::ndim<N> is shorter when only the dimension should be constrained. For example, `nb::ndim<3>` is equivalent to `nb::shape<-1, -1, -1>`.
- **Device tags**: annotations like *nb::device::cpu* or *nb::device::cuda* constrain the source device and address space.
- **Memory order**: two ordering tags *nb::c_contig* and *nb::f_contig* enforce contiguous storage in either C or Fortran style.

In the case of matrices, C-contiguous implies row-major and F-contiguous implies column-major storage. Without this tag, arbitrary non-contiguous representations (e.g. produced by slicing operations) and other unusual layouts are permitted.

This tag is mainly useful when your code directly accesses the array contents via *nb::ndarray<...>::data()*, while assuming a particular layout.

A third order tag named *nb::any_contig* accepts *both* `F`- and `C`-contiguous arrays while rejecting non-contiguous ones.

Type signatures

nanobind displays array constraints in docstrings and error messages. For example, suppose that we now call the `process()` function with an invalid input. This produces the following error message:

```
>>> my_module.process(np.zeros(1))

TypeError: process(): incompatible function arguments. The following argument types␣
 ↪are supported:
1. process(arg: ndarray[dtype=uint8, shape=(*, *, 3), device='cpu'], /) -> None

Invoked with types: numpy.ndarray
```

Note that these type annotations are intended for humans–they will not currently work with automatic type checking tools like MyPy (which at least for the time being don't provide a portable or sufficiently flexible annotation of n-dimensional arrays).

Overload resolution

A function binding can declare multiple overloads with different nd-array constraints (e.g., a CPU and a GPU implementation), in which case nanobind will call the first matching overload. When no perfect match can be found, nanobind will try each overload once more while performing basic implicit conversions: it will convert strided arrays into C- or F-contiguous arrays (if requested) and perform type conversion. This, e.g., makes it possible to call a function expecting a `float32` array with `float64` data. Implicit conversions create temporary nd-arrays containing a copy of the data, which can be undesirable. To suppress them, add an *nb::arg("my_array_arg").noconvert()* or *"my_array_arg"_a.noconvert()* argument annotation.

16.1.3 Passing arrays within C++ code

You can think of the *nb::ndarray* class as a reference-counted pointer resembling `std::shared_ptr<T>` that can be freely moved or copied. This means that there isn't a big difference between a function taking `ndarray` by value versus taking a constant reference `const ndarray &` (i.e., the former does not create an additional copy of the underlying data).

Copies of the *nb::ndarray* wrapper will point to the same underlying buffer and increase the reference count until they go out of scope. You may call freely call *nb::ndarray<...>* methods from multithreaded code even when the GIL is not held, for example to examine the layout of an array and access the underlying storage.

There are two exceptions to this: creating a *new* nd-array object from C++ (discussed *later*) and casting it to Python via the *ndarray::cast()* function both involve Python API calls that require that the GIL is held.

16.1.4 Returning arrays from C++ to Python

Passing an nd-array across the C++ → Python language barrier is a two-step process:

1. Creating an *nb::ndarray<...>* instance, which only stores *metadata*, e.g.:
   - Where is the data located in memory? (pointer address and device)
   - What is its type and shape?
   - Who owns this data?

   An actual Python object is not yet constructed at this stage.
2. Converting the *nb::ndarray<...>* into a Python object of the desired type (e.g. `numpy.ndarray`).

Normally, step 1 is your responsibility, while step 2 is taken care of by the binding layer. To understand this separation, let's look at an example. The `.view()` function binding below creates a 4×4 column-major NumPy array view into a `Matrix4f` instance.

```
struct Matrix4f { float m[4][4] { }; };

using Array = nb::ndarray<float, nb::numpy, nb::shape<4, 4>, nb::f_contig>;

nb::class_<Matrix4f>(m, "Matrix4f")
    .def(nb::init<>())
    .def("view",
         [](Matrix4f &m){ return Array(data); },
         nb::rv_policy::reference_internal);
```

In this case:

- step 1 is the `Array(data)` call in the lambda function.
- step 2 occurs outside of the lambda function when the nd-array *nb::ndarray<...> type caster* constructs a NumPy array from the metadata.

Data *ownership* is an important aspect of this two-step process: because the NumPy array points directly into the storage of another object, nanobind must keep the `Matrix4f` instance alive as long as the NumPy array exists, which the *reference_internal* return value policy signals to nanobind. More generally, wrapping an existing memory region without copying requires that that this memory region remains valid throughout the lifetime of the created array (more on this point *shortly*).

Recall the discussion of the *nd-array constraint* template parameters. For the return path, you will generally want to add a *framework* template parameter to the nd-array parameters that indicates the desired Python type.

- *nb::numpy*: create a `numpy.ndarray`.
- *nb::pytorch*: create a `torch.Tensor`.
- *nb::tensorflow*: create a `tensorflow.python.framework.ops.EagerTensor`.
- *nb::jax*: create a `jaxlib.xla_extension.DeviceArray`.
- *nb::cupy*: create a `cupy.ndarray`.
- No framework annotation. In this case, nanobind will create a raw Python `dltensor` capsule representing the DLPack metadata.

This annotation also affects the auto-generated docstring of the function, which in this case becomes:

```
view(self) -> numpy.ndarray[float32, shape=(4, 4), order='F']
```

Note that the framework annotation only plays a role when passing arrays from C++ to Python. It does not constrain the reverse direction (for example, a PyTorch array would still be accepted by a function taking the `Array` alias defined above as input. For this reason, you may want to add a *nb::device::cpu* device annotation).

Dynamic array configurations

The previous example was rather simple because all the array configuration was fully known at compile time and specified via the *nb::ndarray<...>* template parameters. In general, there are often dynamic aspects of the configuration that must be explicitly passed to the constructor. Its signature (with some simplifications) is given below. See the *ndarray::ndarray()* documentation for a more detailed specification and another variant of the constructor.

```
ndarray(void *data,
        std::initializer_list<size_t> shape = { },
        handle owner = { },
        std::initializer_list<int64_t> strides = { },
        dlpack::dtype dtype = ...,
        int device_type = ...,
        int device_id = 0,
        char order = ...) { .. }
```

The parameters have the following role:

- `data`: CPU/GPU/.. memory address of the data.
- `shape`: number of dimensions and size along each axis.
- `owner`: a Python object owning the storage, which must be kept alive while the array object exists.
- `strides`: specifies the data layout in memory. You only need to specify this parameter if it has a non-standard `order` (e.g., if it is non-contiguous). Note that the `strides` count elements, not bytes.
- `dtype` data type (floating point, signed/unsigned integer), bit depth.
- `device_type` and `device_id`: device type and number, e.g., for multi-GPU setups.
- `order`: coefficient memory order. Default: `'C'` (C-style) ordering, specify `'F'` for Fortran-style ordering.

The parameters generally have inferred defaults based on the array's compile-time template parameters. Passing them explicitly overrides these defaults with information available at runtime.

Data ownership

Let's look at a fancier example that uses the constructor arguments explained above to return a dynamically sized 2D array. This example also shows another mechanism to express *data ownership*:

```
m.def("create_2d",
      [](size_t rows, size_t cols) {
          // Allocate a memory region an initialize it
          float *data = new float[rows * cols];
          for (size_t i = 0; i < rows * cols; ++i)
              data[i] = (float) i;

          // Delete 'data' when the 'owner' capsule expires
          nb::capsule owner(data, [](void *p) noexcept {
             delete[] (float *) p;
          });

          return nb::ndarray<nb::numpy, float, nb::ndim<2>>(
              /* data = */ data,
              /* shape = */ { rows, cols },
              /* owner = */ owner
          );
});
```

The `owner` parameter should specify a Python object, whose continued existence keeps the underlying memory region alive. Nanobind will temporarily increase the `owner` reference count in the *`ndarray::ndarray()`* constructor and then decrease it again when the created NumPy array expires.

The above example binding returns a *new* memory region that should be deleted when it is no longer in use. This is done by creating a `nb::capsule`, an opaque pointer with a destructor callback that runs at this point and takes care of cleaning things up.

If there is already an existing Python object, whose existence guarantees that it is safe to access the provided storage region, then you may alternatively pass this object as the `owner`—nanobind will make sure that this object isn't deleted as long as the created array exists. If the owner is a C++ object with an associated Python instance, you may use *`nb::find()`* to look up the associated Python object. When binding methods, you can use the *`reference_internal`* return value policy to specify the implicit `self` argument as the `owner` upon return, which was done in the earlier `Matrix4f` *example*.

> **Warning:** If you do not specify an owner and use a return value policy like *`rv_policy::reference`* (see also the the section on *nd-array return value policies*), nanobind will assume that the array storage **remains valid forever**.
>
> This is one of the most frequent issues reported on the nanobind GitHub repository: users forget to think about data ownership and run into data corruption.
>
> If there isn't anything keeping the array storage alive, it will likely be released and reused at some point, while stale arrays still point to the associated memory region (i.e., a classic "use-after-free" bug).

In more advanced situations, it may be helpful to have a capsule that manages the lifetime of data structures containing *multiple* storage regions. The same capsule can be referenced from different nd-arrays and will call the deleter when all of them have expired:

```
m.def("return_multiple", []() {
    struct Temp {
        std::vector<float> vec_1;
        std::vector<float> vec_2;
    };

    Temp *temp = new Temp();
    temp->vec_1 = std::move(...);
    temp->vec_2 = std::move(...);

    nb::capsule deleter(temp, [](void *p) noexcept {
        delete (Temp *) p;
    });

    size_t size_1 = temp->vec_1.size();
    size_t size_2 = temp->vec_2.size();

    return std::make_pair(
        nb::ndarray<nb::pytorch, float>(temp->vec_1.data(), { size_1 }, deleter),
        nb::ndarray<nb::pytorch, float>(temp->vec_2.data(), { size_2 }, deleter)
    );
});
```

Return value policies

Function bindings that return nd-arrays can specify return value policy annotations to determine whether or not a copy should be made. They are interpreted as follows:

- The default *rv_policy::automatic* and *rv_policy::automatic_reference* policies cause the array to be copied when it has no owner and when it is not already associated with a Python object.
- The policy *rv_policy::reference* references an existing memory region and never copies.
- *rv_policy::copy* always copies.
- *rv_policy::none* refuses the cast unless the array is already associated with an existing Python object (e.g. a NumPy array), in which case that object is returned.
- *rv_policy::reference_internal* retroactively sets the nd-array's `owner` field to a method's `self` argument. It fails with an error if there is already a different owner.
- *rv_policy::move* is unsupported and demoted to *rv_policy::copy*.

Returning temporaries

Returning nd-arrays from temporaries (e.g. stack-allocated memory) requires extra precautions.

```
using Vector3f = nb::ndarray<float, nb::numpy, nb::shape<3>>;
m.def("return_vec3", []{
    float data[] { 1, 2, 3 };
    // !!! BAD don't do this !!!
    return Vector3f(data);
});
```

Recall the discussion at the *beginning* of this subsection. The *nb::ndarray<...>* constructor only creates *metadata* describing this array, with the actual array creation happening *after* of the function call. That isn't safe in this case because `data` is a temporary on the stack that is no longer valid once the function has returned. To fix this, we could use the *nb::cast()* method to *force* the array creation in the body of the function:

```
using Vector3f = nb::ndarray<float, nb::numpy, nb::shape<3>>;
m.def("return_vec3", []{
    float data[] { 1, 2, 3 };
    // OK.
    return nb::cast(Vector3f(data));
});
```

While safe, one unfortunate aspect of this change is that the function now has a rather non-informative docstring `return_vec3() -> object`, which is a consequence of *nb::cast()* returning a generic *nb::object*.

To fix this, you can use the nd-array *.cast()* method, which is like *nb::cast()* except that it preserves the type signature:

```
using Vector3f = nb::ndarray<float, nb::numpy, nb::shape<3>>;
m.def("return_vec3", []{
    float data[] { 1, 2, 3 };
    // Perfect.
    return Vector3f(data).cast();
});
```

16.1.5 Nonstandard arithmetic types

Low or extended-precision arithmetic types (e.g., `int128`, `float16`, `bfloat16`) are sometimes used but don't have standardized C++ equivalents. If you wish to exchange arrays based on such types, you must register a partial overload of `nanobind::detail::dtype_traits` to inform nanobind about it.

You are expressively allowed to create partial overloads of this class despite it being in the `nanobind::detail` namespace.

For example, the following snippet makes `__fp16` (half-precision type on `aarch64`) available by providing

1. `value`, a DLPack `nanobind::dlpack::dtype` type descriptor, and
2. `name`, a type name for use in docstrings and error messages.

```
namespace nanobind::detail {
    template <> struct dtype_traits<__fp16> {
        static constexpr dlpack::dtype value {
            (uint8_t) dlpack::dtype_code::Float, // type code
            16, // size in bits
            1   // lanes (simd), usually set to 1
        };
        static constexpr auto name = const_name("float16");
    };
}
```

16.1.6 Fast array views

The following advice applies to performance-sensitive CPU code that reads and writes arrays using loops that invoke *nb::ndarray<...>::operator()*. It does not apply to GPU arrays because they are usually not accessed in this way.

Consider the following snippet, which fills a 2D array with data:

```
void fill(nb::ndarray<float, nb::ndim<2>, nb::c_contig, nb::device::cpu> arg) {
    for (size_t i = 0; i < arg.shape(0); ++i)
        for (size_t j = 0; j < arg.shape(1); ++j)
            arg(i, j) = /* ... */;
}
```

While functional, this code is not perfect. The problem is that to compute the address of an entry, `operator()` accesses the DLPack array descriptor. This indirection can break certain compiler optimizations.

nanobind provides the method *ndarray<...>::view()* to fix this. It creates a tiny data structure that provides all information needed to access the array contents, and which can be held within CPU registers. All relevant compile-time information (*nb::ndim*, *nb::shape*, *nb::c_contig*, *nb::f_contig*) is materialized in this view, which enables constant propagation, auto-vectorization, and loop unrolling.

An improved version of the example using such a view is shown below:

```
void fill(nb::ndarray<float, nb::ndim<2>, nb::c_contig, nb::device::cpu> arg) {
    auto v = arg.view(); // <-- new!

    for (size_t i = 0; i < v.shape(0); ++i) // Important; use 'v' instead of 'arg'␣
 ↪everywhere in loop
        for (size_t j = 0; j < v.shape(1); ++j)
            v(i, j) = /* ... */;
}
```

Note that the view performs no reference counting. You may not store it in a way that exceeds the lifetime of the original array.

When using OpenMP to parallelize expensive array operations, pass the `firstprivate(view_1, view_2, ...)` so that each worker thread can copy the view into its register file.

```
auto v = arg.view();
#pragma omp parallel for schedule(static) firstprivate(v)
for (...) { /* parallel loop */ }
```

Specializing views at runtime

As mentioned earlier, element access via `operator()` only works when both the array's scalar type and its dimension are specified within the type (i.e., when they are known at compile time); the same is also true for array views. However, sometimes, it is useful that a function can be called with different array types.

You may use the *ndarray<...>::view()* method to create *specialized* views if a run-time check determines that it is safe to do so. For example, the function below accepts contiguous CPU arrays and performs a loop over a specialized 2D `float` view when the array is of this type.

```
void fill(nb::ndarray<nb::c_contig, nb::device::cpu> arg) {
    if (arg.dtype() == nb::dtype<float>() && arg.ndim() == 2) {
        auto v = arg.view<float, nb::ndim<2>>(); // <-- new!

        for (size_t i = 0; i < v.shape(0); ++i) {
            for (size_t j = 0; j < v.shape(1); ++j) {
                v(i, j) = /* ... */;
            }
        }
    } else { /* ... */ }
}
```

16.1.7 Array libraries

The Python array API standard defines a common interface and interchange protocol for nd-array libraries. In particular, to support inter-framework data exchange, custom array types should implement the

- __dlpack__ and
- __dlpack_device__

methods. This is easy thanks to the nd-array integration in nanobind. An example is shown below:

```
nb::class_<MyArray>(m, "MyArray")
   // ...
   .def("__dlpack__", [](nb::kwargs kwargs) {
       return nb::ndarray<>( /* ... */);
   })
   .def("__dlpack_device__", []() {
       return std::make_pair(nb::device::cpu::value, 0);
   });
```

Returning a raw *nb::ndarray* without framework annotation will produce a DLPack capsule, which is what the interface expects.

The `kwargs` argument can be used to provide additional parameters (for example to request a copy), please see the DLPack documentation for details. Note that nanobind does not yet implement the versioned DLPack protocol. The version number should be ignored for now.

16.1.8 Frequently asked questions

Why does my returned nd-array contain corrupt data?

If your nd-array bindings lead to undefined behavior (data corruption or crashes), then this is usually an ownership issue. Please review the section on *data ownership* for details.

Why does nanobind not accept my NumPy array?

When binding a function that takes an `nb::ndarray<T, ...>` as input, nanobind will by default require that array to be writable. This means that the function cannot be called using NumPy arrays that are marked as constant.

If you wish your function to be callable with constant input, either change the parameter to `nb::ndarray<const T, ...>` (if the array is parameterized by type), or write `nb::ndarray<nb::ro>` to accept a read-only array of any type.

Limitations related to `dtypes`

Libraries like NumPy support arrays with flexible internal representations (*dtypes*), including

- Floating point and integer arrays with various bit depths
- Null-terminated strings
- Arbitrary Python objects
- Heterogeneous data structures composed of multiple fields

nanobind's *`nb::ndarray<...>`* is based on the DLPack array exchange protocol, which causes it to be more restrictive. Presently supported dtypes include signed/unsigned integers, floating point values, complex numbers, and boolean values. Some *nonstandard arithmetic types* can be supported as well.

Nanobind can receive and return *read-only* arrays via the buffer protocol when exhanging data with NumPy. The DLPack interface currently ignores this annotation.

16.2 The *Eigen* linear algebra library

Eigen is a header-only C++ library for linear algebra that offers dense and sparse matrix types along with a host of algorithms that operate on them. Owing to its widespread use in many scientific projects, nanobind includes custom type casters that enable bidirectional conversion between Eigen and Python array programming libraries.

These casters build on the previously discussed *n-dimensional array* class. You can therefore think of this section as an easier interface to the same features that is preferable if your project uses Eigen.

16.2.1 Dense matrices and vectors

Add the following include directive to your binding code to exchange dense Eigen types:

```
#include <nanobind/eigen/dense.h>
```

Following this, you should be able to bind functions that accept and return values of type `Eigen::Matrix<..>`, `Eigen::Array<..>`, `Eigen::Vector<..>`, `Eigen::Ref<..>`, `Eigen::Map<..>`, and their various specializations. Unevaluated expression templates are also supported.

nanobind may need to evaluate or copy the matrix/vector contents during type casting, which is sometimes undesirable. The following cases explain when copying is needed, and how it can be avoided.

C++ → Python

Consider the following C++ function returning a dense Eigen type (`Eigen::MatrixXf` in this example). The bound Python version of `f()` returns this data in the form of a `numpy.ndarray`.

```
Eigen::MatrixXf f() { ... }
```

If the C++ function returns *by value*, and when the Eigen type represents an evaluated expression, nanobind will capture and wrap it in a NumPy array without making a copy. All other cases (returning by reference, returning an unevaluated expression template) either evaluate or copy the array.

Python → C++

The reverse direction is more tricky. Consider the following 3 functions taking variations of a dense `Eigen::MatrixXf`:

```
void f1(const Eigen::MatrixXf &x) { ... }
void f2(const Eigen::Ref<Eigen::MatrixXf> &x) { ... }
void f3(const nb::DRef<Eigen::MatrixXf> &x) { ... }
```

The Python bindings of these three functions can be called using any of a number of different CPU-resident 2D array types (NumPy arrays, PyTorch/Tensorflow/JAX tensors, etc.). However, the following limitations apply:

- `f1()` will always perform a copy of the array contents when called from Python. This is because `Eigen::MatrixXf` is designed to *own* the underlying storage, which is sadly incompatible with the idea of creating a view of an existing Python array.
- `f2()` very likely copies as well! This may seem non-intuitive, since `Eigen::Ref<..>` exists to avoid this exact problem.

  The problem is that Eigen normally expects a very specific memory layout (Fortran/column-major layout), while Python array frameworks actually use the *opposite* by default (C/row-major layout). Array slices are even more problematic and always require a copy.
- `f3()` uses *nb::DRef* to support *any* memory layout (row-major, column-major, slices) without copying. It may still perform an implicit conversion when called with the *wrong data type*—for example, the function expects a single precision array, but NumPy matrices often use double precision.

  If that is undesirable, you may bind the function as follows, in which case nanobind will report a `TypeError` if an implicit conversion would be needed.

  ```
  m.def("f1", &f1, nb::arg("x").noconvert());
  ```

  This parameter passing convention can also be used to mutate function parameters, e.g.:

  ```
  void f4(nb::DRef<Eigen::MatrixXf> x) { x *= 2; }
  ```

16.2.2 Sparse matrices

Add the following include directive to your binding code to exchange sparse Eigen types:

```
#include <nanobind/eigen/sparse.h>
```

The `Eigen::SparseMatrix<..>` type maps to either `scipy.sparse.csr_matrix` or `scipy.sparse.csc_matrix` depending on whether row- or column-major storage is used.

There is no support for Eigen sparse vectors because an equivalent type does not exist as part of `scipy.sparse`.

CHAPTER

SEVENTEEN

PACKAGING

A Python *wheel* is a self-contained binary file that bundles Python code and extension libraries along with metadata such as versioned package dependencies. Wheels are easy to download and install, and they are the recommended mechanism for distributing extensions created using nanobind.

This section walks through the recommended sequence of steps to build wheels and optionally automate this process to simultaneously target many platforms (Linux, Windows, macOS) and processors (i386, x86_64, arm64) using the GitHub Actions CI service.

Note that all of the recommended practices have already been implemented in the nanobind_example repository, which is a minimal C++ project with nanobind-based bindings. You may therefore prefer to clone this repository and modify its contents.

17.1 Step 1: Overview

The example project has a simple directory structure:

```
├── README.md
├── CMakeLists.txt
├── pyproject.toml
└── src/
    ├── my_ext.cpp
    └── my_ext/
        └── __init__.py
```

The file `CMakeLists.txt` contains the C++-specifics part of the build system, while `pyproject.toml` explains how to turn the example into a wheel. The file `README.md` should ideally explain how to use the project in more detail. Its contents are arbitrary, but the file must exist for the following build system to work.

All source code is located in a `src` directory containing the Python package as a subdirectory.

Compilation will turn `my_ext.cpp` into a shared library in the package directory, which has an underscored platform-dependent name (e.g., `_my_ext_impl.cpython-311-darwin.so`) to indicate that it is an implementation detail. The `src/my_ext/__init__.py` imports the extension and exposes relevant functionality. In this small example project, it only contains a single line:

```
from ._my_ext_impl import hello
```

The file `src/my_ext.cpp` contains minimal bindings for an example function:

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

NB_MODULE(_my_ext_impl, m) {
    m.def("hello", []() { return "Hello world!"; });
}
```

The next two steps will set up the infrastructure needed for wheel generation.

17.2 Step 2: Specify build dependencies and metadata

In the root directory of the project, create a file named `pyproject.toml` listing *build-time dependencies*. Note that runtime dependencies *do not* need to be added here. The following core dependencies are required by nanobind:

```
[build-system]
requires = ["scikit-build-core >=0.4.3", "nanobind >=1.3.2"]
build-backend = "scikit_build_core.build"
```

You may need to increase the minimum nanobind version in the above snippet if you are using features from versions newer than 1.3.2.

Just below the list of build-time requirements, specify project metadata including:

- The project's name (which must be a valid package name)
- The version number
- A brief (1-line) description of the project
- The name of a more detailed README file
- The list of authors with email addresses.
- The software license
- The project web page
- Runtime dependencies, if applicable

An example is shown below:

```
[project]
name = "my_ext"
version = "0.0.1"
description = "A brief description of what this project does"
readme = "README.md"
requires-python = ">=3.8"
authors = [
    { name = "Your Name", email = "your.email@address.com" },
]
classifiers = [
    "License :: BSD",
]
# Optional: runtime dependency specification
# dependencies = [ "cryptography >=41.0" ]

[project.urls]
Homepage = "https://github.com/your/project"
```

We will use scikit-build-core to build wheels, and this tool also has its own configuration block in `pyproject.toml`. The following defaults are recommended:

```
[tool.scikit-build]
# Protect the configuration against future changes in scikit-build-core
minimum-version = "0.4"

# Setuptools-style build caching in a local directory
build-dir = "build/{wheel_tag}"

# Build stable ABI wheels for CPython 3.12+
wheel.py-api = "cp312"
```

17.3 Step 3: Set up a CMake build system

Next, we will set up a suitable `CMakeLists.txt` file in the root directory. Since this build system is designed to be invoked through `scikit-build-core`, it does not make sense to perform a standalone CMake build. The message at the top warns users attempting to do this.

```
# Set the minimum CMake version and policies for highest tested version
cmake_minimum_required(VERSION 3.15...3.27)

# Set up the project and ensure there is a working C++ compiler
project(my_ext LANGUAGES CXX)

# Warn if the user invokes CMake directly
if (NOT SKBUILD)
  message(WARNING "\
  This CMake file is meant to be executed using 'scikit-build-core'.
  Running it directly will almost certainly not produce the desired
  result. If you are a user trying to install this package, use the
  command below, which will install all necessary build dependencies,
  compile the package in an isolated environment, and then install it.
  =====================================================================
   $ pip install .
  =====================================================================
  If you are a software developer, and this is your own package, then
  it is usually much more efficient to install the build dependencies
  in your environment once and use the following command that avoids
  a costly creation of a new virtual environment at every compilation:
  =====================================================================
   $ pip install nanobind scikit-build-core[pyproject]
   $ pip install --no-build-isolation -ve .
  =====================================================================
  You may optionally add -Ceditable.rebuild=true to auto-rebuild when
  the package is imported. Otherwise, you need to rerun the above
  after editing C++ files.")
endif()
```

Next, import Python and nanobind including the `Development.SABIModule` component that can be used to create stable ABI builds.

```
# Try to import all Python components potentially needed by nanobind
find_package(Python 3.8
  REQUIRED COMPONENTS Interpreter Development.Module
  OPTIONAL_COMPONENTS Development.SABIModule)

# Import nanobind through CMake's find_package mechanism
find_package(nanobind CONFIG REQUIRED)
```

The last two steps build and install the actual extension

```
# We are now ready to compile the actual extension module
nanobind_add_module(
  # Name of the extension
  _my_ext_impl

  # Target the stable ABI for Python 3.12+, which reduces
  # the number of binary wheels that must be built. This
  # does nothing on older Python versions
```

(continues on next page)

(continued from previous page)

```
  STABLE_ABI

  # Source code goes here
  src/my_ext.cpp
)

# Install directive for scikit-build-core
install(TARGETS _my_ext_impl LIBRARY DESTINATION my_ext)
```

17.4 Step 4: Install the package locally

To install the package, run

```
$ cd <project-directory>
$ pip install .
```

`pip` will parse the `pyproject.toml` file and create a fresh environment containing all needed dependencies. Following this, you should be able to install and access the extension.

```
>>> import my_ext
>>> my_ext.hello()
'Hello world!'
```

Alternatively, you can use the following command to generate a `.whl` file instead of installing the package.

```
$ pip wheel .
```

17.5 Step 5: Incremental rebuilds

The `pip install` and `pip wheel` commands are extremely conservative to ensure reproducible builds. They create a pristine virtual environment and install build-time dependencies before compiling the extension *from scratch*.

It can be frustrating to wait for this lengthy sequence of steps after every small change to a source file during the active development phase of a project. To avoid this, first install the project's build dependencies, e.g.:

```
$ pip install nanobind scikit-build-core[pyproject]
```

Next, install the project without build isolation to enable incremental builds:

```
$ pip install --no-build-isolation -ve .
```

This command will need to be run after every change to reinstall the updated package. For an even more interactive experience, use

```
$ pip install --no-build-isolation -Ceditable.rebuild=true -ve .
```

This will automatically rebuild any code (if needed) whenever the `my_ext` package is imported into a Python session.

17.6 Step 6: Build wheels in the cloud

On my machine, the `pip wheel` command produces a file named `my_ext-0.0.1-cp311-cp311-macosx_13_0_arm64.whl` that is specific to Python 3.11 running on an arm64 macOS machine. Other Python versions and operating systems each require their own wheels, which leads to a dauntingly large build matrix (though nanobind's stable ABI support will help to significantly reduce the size of this matrix once Python 3.12 is more widespread).

Rather than building these wheels manually on different machines, it is far more efficient to use GitHub actions along with the powerful cibuildwheel package to fully automate the process.

To do so, create a file named `.github/workflows/wheels.yml` containing the contents of the following file. You may want to remove the `on: push:` lines, otherwise, the action will run after every commit, which is perhaps a bit excessive. In this case, you can still trigger the action manually on the *Actions* tab of the GitHub project page.

Furthermore, append the following `cibuildwheel`-specific configuration to `pyproject.toml`:

```
[tool.cibuildwheel]
# Necessary to see build output from the actual compilation
build-verbosity = 1

# Optional: run pytest to ensure that the package was correctly built
# test-command = "pytest {project}/tests"
# test-requires = "pytest"

# Needed for full C++17 support on macOS
[tool.cibuildwheel.macos.environment]
MACOSX_DEPLOYMENT_TARGET = "10.14"
```

Following each run, the action provides a downloadable *build artifact*, which is a ZIP file containing all the individual wheel files for each platform.

By default, `cibuildwheel` will launch a very large build matrix, and it is possible that your extension is not compatible with every single configuration. For example, suppose that the project depends on Python 3.9+ and a 64 bit processor. In this case, add further entries to the `[tool.cibuildwheel]` block to remove incompatible configurations from the matrix:

```
skip = ["cp38-*", "pp38-*"] # Skip CPython and PyPy 3.8
archs = ["auto64"]          # Only target 64 bit architectures
```

The cibuildwheel documentation explains the possible options.

If you set up a GitHub actions secret named `pypi_password` containing a PyPI authentication token, the action will automatically upload the generated wheels to the Python Package Index (PyPI) when the action is triggered by a software release event.

CHAPTER

EIGHTEEN

TYPING

This section covers three broad typing-related topics:

1. How to create rich type annotation in C++ bindings so that projects using them can be effectively type-checked.
2. How to *automatically generate stub files* that are needed to enable static type checking and autocompletion in Python IDEs.
3. How to write *pattern files* to handle advanced use cases requiring significant stub customization.

18.1 Signature customization

In larger binding projects, some customization of function or class signatures is often needed so that static type checkers can effectively use the generated stubs.

18.1.1 Functions

Nanobind generates typed function signatures automatically, but these are not always satisfactory. For example, the following function binding

```
nb::class_<Int>(m, "Int")
    .def(nb::self == nb::self);
```

is likely to be rejected because the default `__eq__` function signature

```
__eq__(self, arg: Int, /) -> bool
```

is more specific than that of the parent class `object`:

```
__eq__(self, arg: object, /) -> bool
```

In this case, a static type checker like MyPy will report a failure:

```
error: Argument 1 of "__eq__" is incompatible with supertype "object"; supertype␣
↪defines the argument type as "object"  [override]
```

To handle such cases, you can use the *`nb::sig`* attribute to overrides the function signature with a custom string.

```
nb::class_<Int>(m, "Int")
    .def(nb::self == nb::self,
         nb::sig("def __eq__(self, arg: object, /) -> bool"));
```

The argument must be a valid Python function signature of the form `def name(...) -> ...` without trailing colon (`":"`) and newlines, where `name` must furthermore match the name given to the binding declaration. In this

case, the name is implicitly given by the operator. It must match `"name"` in the case of `.def("name", ...)`-style bindings with an explicit name. The signature can span multiple lines, e.g., to prefix one or more decorators.

The modified signature is shown in generated stubs, docstrings, and error messages (e.g., when a function receives incompatible arguments).

In cases where a custom signature is only needed to tweak how nanobind renders the signature of a default argument, the more targeted *nb::arg("name").sig("signature")* annotation is preferable to *nb::sig*.

18.1.2 Classes

Signature customization is also available for class bindings, though only stubs are affected in this case.

Consider the example below, which defines an iterable vector type storing integers. Suppose that `GeneralIterator` iterates over arbitrary data and does not provide a useful `int`-typed signature.

```
using IntVec = std::vector<int>;

nb::class_<IntVec>(m, "IntVec")
   .def("__iter__",
        [](const IntVec &v) -> GeneralIterator { ... })
```

It may be useful to inherit from `collections.abc.Iterable[int]` to communicate more information to static type checkers, but such a Python → C++ inheritance chain is not permitted by nanobind.

Stubs often take certain liberties in deviating somewhat from the precise type signature of the underlying implementation, which is fine as long as this improves the capabilities of the type checker (the stubs are only used by the static type checking phase, which never imports the actual extension).

Here, we could specify

```
nb::class_<IntVec>(m, "IntVec",
                   nb::sig("class IntVec(collections.abc.Iterable[int])"));
```

This is technically a lie. Such shenanigans are worthwhile because they can greatly improve the development experience (e.g. VS Code autocomplete) involving compiled extensions.

The supplied signature string must be a valid Python class signature of the form `class ClassName(...)` excluding trailing colon (`":"`) and newline, where `ClassName` must furthermore match the name provided in the main class binding declaration. The signature can span multiple lines, e.g., to prefix one or more decorators.

18.2 Generic types

18.2.1 Parameterizing generic types

Various standard Python types are generic can be parameterized to improve the effectiveness of static type checkers such as MyPy. In the presence of such a specialization, a type checker can, e.g., infer that the variable `a` below is of type `int`.

```
def f() -> list[int]: ...

a = f()[0]
```

This is even supported for *abstract types*—for example, `collections.abc.Mapping[str, int]` indicates an abstract mapping from strings to integers.

nanobind provides the template class *nb::typed<T, Ts...>* to generate parameterized type annotations in C++ bindings. For example, the argument and return value of the following function binding reproduces the exact list and mapping types mentioned above.

```
m.def("f", [](nb::typed<nb::mapping, nb::str, int> arg)
           -> nb::typed<nb::list, int> { ... });
```

(Usually, *nb::typed<T, Ts...>* would be applied to *wrapper* types, though this is not a strict limitation.)

An important limitation of this feature is that it *only* affects function signatures. Nanobind will (as always) ensure that `f` can only be called with a `nb::mapping`, but it will *not* insert additional runtime checks to verify that `arg` indeed maps strings to integers. It is the responsibility of the function to perform these checks and, if needed, to raise a *nb::type_error*.

The parameterized C++ type *nb::typed<T, Ts...>* subclasses the type `T` and can be used interchangeably with `T`. The other arguments (`Ts...`) are used to generate a Python type signature but have no other effect (for example, parameterizing by `str` on the Python end can alternatively be achieved by passing `nb::str`, `std::string`, or `const char*` as part of the `Ts..` parameter pack).

18.2.2 Creating generic types

Python types inheriting from types.Generic can be *parameterized* by other types including generic type variables that act as placeholders. Such constructions enable more effective static type checking. In the snippet below, tools like MyPy or PyRight can infer that `x` and `y` have types `Wrapper[int]` and `int`, respectively.

```
import typing

# 1. Instantiate a placeholder type ("type variable") used below
T = typing.TypeVar("T")

# 2. Create a generic type by inheriting from typing.Generic
class Wrapper(typing.Generic[T]):
    # The constructor references the placeholder type
    def __init__(self, value: T):
        self.value = value

    # .. this type is then preserved in the getter
    def get(self) -> T:
        return self.value

# Based on the typed constructor, MyPy knows that 'x' has type 'Wrapper[int]'
x = Wrapper(3)

# Based on the typed 'Wrapped.get' method, 'y' is inferred to have type 'int'
y = x.get()
```

Note that parameterization of a generic type doesn't generate new code or modify its functionality. It is not to be confused with C++ template instantiation. The feature only exists to propagate fine-grained type information and thereby aid static type checking.

Similar functionality can also be supported in nanobind-based binding projects. This looks as follows:

```
#include <nanobind/typing.h> // needed by nb::type_var below

struct Wrapper {
    nb::object value;
};

NB_MODULE(my_ext, m) {
    // 1. Instantiate a placeholder type ("type variable") used below
    m.attr("T") = nb::type_var("T");
```

(continues on next page)

(continued from previous page)

```
    // 2. Create a generic type, and indicate in generated stubs
    //    that it derives from Generic[T]
    nb::class_<Wrapper> wrapper(m, "Wrapper", nb::is_generic(),
                                nb::sig("class Wrapper(typing.Generic[T])"))
        .def(nb::init<nb::object>(),
             nb::sig("def __init__(self, arg: T, /) -> None"))
        .def("get", [](Wrapper &w) { return w.value; },
             nb::sig("def get(self, /) -> T"));
}
```

This involves the following steps:

- The *nb::type_var* constructor generates a type variable analogous to the previous Python snippet and assigns it to the name `"T"` within the module.
- If we were to follow the previous Python example, the next step would require defining `Wrapper` as a subclass of `typing.Generic[T]`. However, this isn't possible because nanobind-based classes cannot derive from Python types.
- The solution to this problem takes the following *liberties*:
  - It passes the *nb::is_generic* annotation to the *nb::class_<...>* constructor, causing the addition of a `__class_getattr__` member that enables type parameterization. Following this step, an expression like `Wrapper[int]` becomes valid and returns a `typing.TypeAlias` (in other words, the behavior is *as if* we had derived from `typing.Generic[T]`).

    However, MyPy and similar tools don't quite know what to do with custom types overriding `__class_getattr__` themselves, since the official parameterization mechanism is to subclass `typing.Generic`.
  - Therefore, we *lie* about this in the stub and declare `typing.Generic[T]` as a base class. Only static type checkers will see this information, and it helps them to interpret how the type works.
  - That's it!

You may also extend parameterized forms of such generic types:

```
nb::class_<Subclass>(m, "Subclass", wrapper[nb::type<Foo>()]);
```

nanobind's stub generator will render this as `class Subclass(Wrapper[Foo]):`.

18.2.3 Any-typed return values

The return value of a function can sometimes be unclear (dynamic), in which case it can be helpful to declare `typing.Any` as a pragmatic return type (this effectively disables analysis of the return value in static type checkers). nanobind provides a `nb::any` wrapper type that is equivalent to `nb::object` except that its type signature renders as `typing.Any` to facilitate this.

18.3 Stub generation

A *stub file* provides a *typed* and potentially documented summary of a module's class, function, and variable declarations. Stub files have the extension `.pyi` and are often shipped along with Python extensions. They are needed to enable autocompletion and static type checking in tools like Visual Studio Code, MyPy, PyRight and PyType.

Take for example the following function:

```
def square(x: int) -> int:
    '''Return the square of the input'''
    return x*x
```

The associated default stub removes the body, while retaining the docstring:

```
def square(x: int) -> int:
    '''Return the square of the input'''
```

An undocumented stub replaces the entire body with the Python ellipsis object (`...`).

```
def square(x: int) -> int: ...
```

Complex default arguments are often also abbreviated with `...` to improve the readability of signatures. You can read more about stub files in the typing documentation and the MyPy documentation.

nanobind's `stubgen` tool automates the process of stub generation to turn modules containing a mixture of ordinary Python code and C++ bindings into an associated `.pyi` file.

The main challenge here is that C++ bindings are unlike ordinary Python objects, which causes standard mechanisms to extract their signature to fail. Existing tools like MyPy's stubgen and pybind11-stubgen must therefore parse docstrings to infer function signatures, which is brittle and does not always produce high-quality output.

nanobind functions expose a `__nb_signature__` property, which provides structured information about typed function signatures, overload chains, and default arguments. nanobind's `stubgen` leverages this information to reliably generate high-quality stubs that are usable by static type checkers.

There are three ways to interface with the stub generator described in the following subsections.

18.3.1 CMake interface

nanobind's CMake interface provides the *nanobind_add_stub()* command for stub generation at build or install time. It generates a single stub at a time–more complex cases involving large numbers of stubs are easily handled using standard CMake constructs (e.g. a `foreach()` loop).

The command requires a target name (e.g., `my_ext_stub`) that must be unique but has no other significance. Once all dependencies (`DEPENDS` parameter) are met, it will invoke `stubgen` to turn a single module (`MODULE` parameter) into a stub file (`OUTPUT` parameter).

For this to work, the module must be importable. `stubgen` will add all paths specified as part of the `PYTHON_PATH` parameter and then execute `import my_ext`, raising an error if this fails.

```
nanobind_add_stub(
  my_ext_stub
  MODULE my_ext
  OUTPUT my_ext.pyi
  PYTHON_PATH $<TARGET_FILE_DIR:my_ext>
  DEPENDS my_ext
)
```

Typed extensions normally identify themselves via the presence of an empty file named `py.typed` in each module directory. *nanobind_add_stub()* can optionally generate this file as well.

```
nanobind_add_stub(
  ...
  MARKER_FILE py.typed
  ...
)
```

CMake tracks the generated outputs in its dependency graph. The combination of compiled extension module, stub, and marker file can subsequently be installed by subsequent `install()` directives.

```
install(TARGETS my_ext DESTINATION ".")
install(FILES py.typed my_ext.pyi DESTINATION ".")
```

In certain situations, it may be tricky to import an extension that is built but not yet installed to its final destination. To handle such cases, specify the `INSTALL_TIME` parameter to *nanobind_add_stub()* to delay stub generation to the installation phase.

```
install(TARGETS my_ext DESTINATION ".")

nanobind_add_stub(
  my_ext_stub
  INSTALL_TIME
  MODULE my_ext
  OUTPUT my_ext.pyi
  PYTHON_PATH "."
)
```

This requires several changes:

1. `PYTHON_PATH` must be adjusted so that it references a location relative to `CMAKE_INSTALL_PREFIX` from which the installed module is importable.
2. The *nanobind_add_stub()* command should be preceded by `install(TARGETS my_ext)` and `install(FILES` commands that place all data (compiled extension files, plain Python code, etc.) needed to bring the module into an importable state.

   Place all relevant `install()` directives within the same `CMakeLists.txt` file to ensure that these steps are executed sequentially.
3. Dependencies (`DEPENDS`) no longer need to be listed. These are build-time constraints that do not apply in the installation phase.
4. The output file path (`OUTPUT`) is relative to `CMAKE_INSTALL_PREFIX` and may need adjustments as well.

The *nanobind_add_stub()* command has a few other options, please refer to its documentation for details.

18.3.2 Command line interface

Alternatively, you can invoke `stubgen` on the command line. The nanobind package must be installed for this to work, e.g., via `pip install nanobind`. The command line interface is also able to generate multiple stubs at once (simply specify `-m MODULE` several times).

```
$ python -m nanobind.stubgen -m my_ext -M py.typed
Module "my_ext" ..
  - importing ..
  - analyzing ..
  - writing stub "my_ext.pyi" ..
  - writing marker file "py.typed" ..
```

Unless an output file (`-o`) or output directory (`-O`) is specified, this places the `.pyi` files directly into the module. Existing stubs are overwritten without warning.

The program has the following command line options:

```
usage: python -m nanobind.stubgen [-h] [-o FILE] [-O PATH] [-i PATH] [-m MODULE]
                                  [-r] [-M FILE] [-P] [-D] [-q]
```

(continues on next page)

(continued from previous page)

```
Generate stubs for nanobind-based extensions.

options:
  -h, --help                    show this help message and exit
  -o FILE, --output-file FILE   write generated stubs to the specified file
  -O PATH, --output-dir PATH    write generated stubs to the specified directory
  -i PATH, --import PATH        add the directory to the Python import path (can
                                specify multiple times)
  -m MODULE, --module MODULE    generate a stub for the specified module (can
                                specify multiple times)
  -r, --recursive               recursively process submodules
  -M FILE, --marker-file FILE   generate a marker file (usually named 'py.typed')
  -p FILE, --pattern-file FILE  apply the given patterns to the generated stub
                                (see the docs for syntax)
  -P, --include-private         include private members (with single leading or
                                trailing underscore)
  -D, --exclude-docstrings      exclude docstrings from the generated stub
  -q, --quiet                   do not generate any output in the absence of failures
```

18.3.3 Python interface

Finally, you can import `stubgen` into your own Python programs and use it to programmatically generate stubs with a finer degree of control.

To do so, construct an instance of the `StubGen` class and repeatedly call `.put()` to register modules or contents within the modules (specific methods, classes, etc.). Afterwards, the `.get()` method returns a string containing the stub declarations.

```
from nanobind.stubgen import StubGen
import my_module

sg = StubGen()
sg.put(my_module)
print(sg.get())
```

Note that for now, the `nanobind.stubgen.StubGen` API is considered experimental and not subject to the semantic versioning policy used by the nanobind project.

18.4 Pattern files

In complex binding projects requiring static type checking, the previously discussed mechanisms for controlling typed signatures (*nb::sig*, *nb::typed*) may be insufficient. Two common reasons are as follows:

- the `@typing.overload` chain associated with a function may sometimes require significant deviations from the actual overloads present on the C++ side.
- Some members of a module could be inherited from existing Python packages or extension libraries, in which case patching their signature via *nb::sig* is not even an option.

`stubgen` supports *pattern files* as a last-resort solution to handle such advanced needs. These are files written in a *domain-specific language* (DSL) that specifies replacement patterns to dynamically rewrite stubs during generation. To use one, simply add it to the *nanobind_add_stub()* command.

```
nanobind_add_stub(
  ...
```

(continues on next page)

(continued from previous page)

```
  PATTERN_FILE  <PATH>
  ...
)
```

A pattern file contains sequence of patterns. Each pattern consists of a query and an indented replacement block to be applied when the query matches.

```
# This is the first pattern
query 1:
    replacement 1

# And this is the second one
query 2:
    replacement 2
```

Empty lines and lines beginning with # are ignored. The amount of indentation is arbitrary: `stubgen` will re-indent the replacement as needed based on where the query matched.

When the stub generator traverses the module, it computes the *fully qualified name* of every type, function, property, etc. (for example: `"my_ext.MyClass.my_function"`). The queries in a pattern file are checked against these qualified names one by one until the first one matches.

For example, suppose that we had the following lackluster stub entry:

```
class MyClass:
    def my_function(arg: object) -> object: ...
```

The pattern below matches this function stub and inserts an alternative with two typed overloads.

```
my_ext.MyClass.my_function:
    @overload
    def my_function(arg: int) -> int:
        """A helpful docstring"""

    @overload
    def my_function(arg: str) -> str: ...
```

Patterns can also *remove* entries, by simply not specifying a replacement block. Also, queries don't have to match the entire qualified name. For example, the following pattern deletes all occurrences of anything containing the string `secret` somewhere in its name

```
secret:
```

In fact (you may have guessed it), the queries are *regular expressions*! The query supports all features of Python's builtin re library.

When the query uses *groups*, the replacement block may access the contents of each numbered group using using the syntax `\1`, `\2`, etc. This permits writing generic patterns that can be applied to a number of stub entries at once:

```
__(eq|ne)__:
    def __\1__(self, arg, /) -> bool: ...
```

Named groups are also supported:

```
__(?P<op>eq|ne)__:
    def __\op__(self, arg, /) -> bool : ...
```

Finally, sometimes, it is desirable to rewrite only the signature of a function in a stub but to keep its docstring so that it doesn't have to be copied into the pattern file. The special escape code `\doc` references the previously

existing docstring.

```
my_ext.lookup:
    def lookup(array: Array[T], index: int) -> T:
        \doc
```

If your replacement rule requires additional types to work (e.g., from `typing.*`), you may use the special `\from` escape code to import them:

```
@overload
my_ext.lookup:
    \from typing import Optional as _Opt, Literal
    def lookup(array: Array[T], index: Literal[0] = 0) -> _Opt[T]:
        \doc
```

You may also add free-form text the beginning or the end of the generated stub. To do so, add an entry that matches on `module_name.__prefix__` or `module_name.__suffix__`.

CHAPTER

NINETEEN

UTILITIES

19.1 Evaluating Python expressions from strings

nanobind provides the *eval()* and *exec()* functions to evaluate Python expressions and statements. The following example illustrates how they can be used.

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

...

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

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

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

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

Note: *eval()* accepts 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` and *exec()* is just a shortcut for `eval<eval_statements>`.

CHAPTER

TWENTY

FREE-THREADED PYTHON

Free-threading is an experimental new Python feature that replaces the Global Interpreter Lock (GIL) with a fine-grained locking scheme to better leverage multi-core parallelism. The resulting benefits do not come for free: extensions must explicitly opt-in and generally require careful modifications to ensure correctness.

Nanobind can target free-threaded Python since version 2.2.0. This page explains how to do so and discusses a few caveats. Besides this page, make sure to review py-free-threading.github.io for a more comprehensive discussion of free-threaded Python. PEP 703 explains the nitty gritty details.

20.1 Opting in

To opt into free-threaded Python, pass the `FREE_THREADED` parameter to the *nanobind_add_module()* CMake target command. For other build systems, refer to their respective documentation pages.

```
nanobind_add_module(
  my_ext                  # Target name
  FREE_THREADED           # Opt into free-threading
  my_ext.h                # Source code files below
  my_ext.cpp)
```

nanobind ignores the `FREE_THREADED` parameter when the registered Python version does not support free-threading.

Note: **Stable ABI**: Note that there currently is no stable ABI for free-threaded Python, hence the `STABLE_ABI` parameter will be ignored in free-threaded extensions builds. It is valid to combine the `STABLE_ABI` and `FREE_THREADED` arguments: the build system will choose between the two depending on the detected Python version.

Warning: Loading an Python extension that does not support free-threading disables free-threading globally. In larger binding projects with multiple extensions, all of them must be adapted.

If free-threading was requested and is available, the build system will set the `NB_FREE_THREADED` preprocessor flag. This can be helpful to specialize binding code with `#ifdef` blocks, e.g.:

```
#if !defined(NB_FREE_THREADED)
... // simple GIL-protected code
#else
... // more complex thread-aware code
#endif
```

20.2 Caveats

Free-threading can violate implicit assumptions made by extension developers when previously serial operations suddenly run concurrently, producing undefined behavior (race conditions, crashes, etc.).

Let's consider a concrete example: the binding code below defines a `Counter` class with an increment operation.

```
struct Counter {
    int value = 0;
    void inc() { value++; }
};

nb::class_<Counter>(m, "Counter")
    .def("inc", &Counter::inc)
    .def_ro("value", &Counter::value);
```

If multiple threads call the `inc()` method of a single `Counter`, the final count will generally be incorrect, as the increment operation `value++` does not execute atomically.

To fix this, we could modify the C++ type so that it protects its `value` member from concurrent modification, for example using an atomic number type (e.g., `std::atomic<int>`) or a critical section (e.g., based on `std::mutex`).

The race condition in the above example is relatively benign. However, in more complex projects, combinations of concurrency and unsafe memory accesses could introduce non-deterministic data corruption and crashes.

Another common source of problems are *global variables* undergoing concurrent modification when no longer protected by the GIL. They will likewise require supplemental locking. The *next section* explains a Python-specific locking primitive that can be used in binding code besides the solutions mentioned above.

20.3 Python locks

Nanobind provides convenience functionality encapsulating the mutex implementation that is part of Python ("`PyMutex`"). It is slightly more efficient than OS/language-provided synchronization primitives and generally preferable within Python extensions.

The class *`ft_mutex`* is analogous to `std::mutex`, and *`ft_lock_guard`* is analogous to `std::lock_guard`. Note that they only exist to add *supplemental* critical sections needed in free-threaded Python, while becoming inactive (no-ops) when targeting regular GIL-protected Python.

With these abstractions, the previous `Counter` implementation could be rewritten as:

```
struct Counter {
    int value = 0;
    nb::ft_mutex mutex;

    void inc() {
        nb::ft_lock_guard guard(mutex);
        value++;
    }
};
```

These locks are very compact (`sizeof(nb::ft_mutex) == 1`), though this is a Python implementation detail that could change in the future.

20.4 Argument locking

Modifying class and function definitions as shown above may not always be possible. As an alternative, nanobind also provides a way to *retrofit* supplemental locking onto existing code. The idea is to lock individual arguments of a function *before* being allowed to invoke it. A built-in mutex present in every Python object enables this.

To do so, call the *.lock()* member of *nb::arg()* annotations to indicate that an argument must be locked, e.g.:

- *nb::arg("my_parameter").lock()*
- *"my_parameter"_a.lock()* (short-hand form)

In methods bindings, pass *nb::lock_self()* to lock the implicit `self` argument. Note that at most 2 arguments can be locked per function, which is a limitation of the Python locking API.

The example below shows how this functionality can be used to protect `inc()` and a new `merge()` function that acquires two simultaneous locks.

```
struct Counter {
    int value = 0;

    void inc() { value++; }
    void merge(Counter &other) {
        value += other.value;
        other.value = 0;
    }
};

nb::class_<Counter>(m, "Counter")
    .def("inc", &Counter::inc, nb::lock_self())
    .def("merge", &Counter::merge, nb::lock_self(), "other"_a.lock())
    .def_ro("value", &Counter::value);
```

The above solution has an obvious drawback: it only protects *bindings* (i.e., transitions from Python to C++). For example, if some other part of a C++ codebase calls `merge()` directly, the binding layer won't be involved, and no locking takes place. If such behavior can introduce race conditions, a larger-scale redesign of your project may be in order.

Note: Adding locking annotations indiscriminately is inadvisable because locked calls are more costly than unlocked ones. The *.lock()* and *nb::lock_self()* annotations are ignored in GIL-protected builds, hence this added cost only applies to free-threaded extensions.

Furthermore, when adding locking annotations to a function, consider keeping the arguments *unnamed* (i.e., *nb::arg().lock()* instead of *nb::arg("name").lock()*) if the function will never be called with keyword arguments. Processing named arguments causes small *binding overheads* that may be undesirable if a function that does very little is called at a very high rate.

Note: **Python API and locking**: When the lock-protected function performs Python API calls (e.g., using *wrappers* like *nb::dict*), Python may temporarily release locks to avoid deadlocks. Here, even basic reference counting such as a *nb::object* variable expiring at the end of a scope counts as an API call.

These locks will be reacquired following the Python API call. This behavior resembles ordinary (GIL-protected) Python code, where operations like Py_DECREF() can cause cause arbitrary Python code to execute. The semantics of this kind of relaxed critical section are described in the Python documentation.

20.5 Miscellaneous notes

20.6 API

The following API specific to free-threading has been added:

- *nb::ft_mutex*
- *nb::ft_lock_guard*
- *nb::ft_object_guard*
- *nb::ft_object2_guard*
- *nb::arg::lock()*

20.6.1 API stability

The interface explained in this is excluded from the project's semantic versioning policy. Free-threading is still experimental, and API breaks may be necessary based on future experience and changes in Python itself.

20.6.2 Wrappers

Wrapper types like *nb::list* may be used in multi-threaded code. Operations like *nb::list::append()* internally acquire locks and behave just like their ordinary Python counterparts. This means that race conditions can still occur without larger-scale synchronization, but such races won't jeopardize the memory safety of the program.

20.6.3 GIL scope guards

Prior to free-threaded Python, the nanobind scope guards *gil_scoped_acquire* and *gil_scoped_release* would normally be used to acquire/release the GIL and enable parallel regions.

These remain useful and should not be removed from existing code: while no longer blocking operations, they set and unset the current Python thread context and inform the garbage collector.

The *gil_scoped_release* RAII scope guard class plays a special role in free-threaded builds, since it releases all *argument locks* held by the current thread.

20.6.4 Immortalization

Python relies on a technique called *reference counting* to determine when an object is no longer needed. This approach can become a bottleneck in multi-threaded programs, since increasing and decreasing reference counts requires coordination among multiple processor cores. Python type and function objects are especially sensitive, since their reference counts change at a very high rate.

Similar to free-threaded Python itself, nanobind avoids this bottleneck by *immortalizing* functions (`nanobind.nb_func`, `nanobind.nb_method`) and type bindings. Immortal objects don't require reference counting and therefore cannot cause the bottleneck mentioned above. The main downside of this approach is that these objects leak when the interpreter shuts down. Free-threaded nanobind extensions disable the internal *leak checker*, since it would produce many warning messages caused by immortal objects.

20.6.5 Internal data structures

Nanobind maintains various internal data structures that store information about instances and function/type bindings. These data structures also play an important role to exchange type/instance data in larger projects that are split across several independent extension modules.

The layout of these data structures differs between ordinary and free-threaded extensions, therefore nanobind isolates them from each other by assigning a different ABI version tag. This means that multi-module projects will need to consistently compile either free-threaded or non-free-threaded modules.

Free-threaded nanobind uses thread-local and sharded data structures to avoid lock and atomic contention on the internal data structures, which would otherwise become a bottleneck in multi-threaded Python programs.

20.6.6 Thread sanitizers

The thread sanitizer (TSAN) offers an effective way of tracking down undefined behavior in multithreaded application.

To use TSAN with nanonbind extensions, you *must* also create a custom Python build that has TSAN enabled. This is because nanobind internally builds on Python locks. If the implementation of the locks is not instrumented by TSAN, the tool will detect a large volume of false positives.

To make a TSAN-instrumented Python build, download a Python source release and to pass the following options to its `configure` script:

```
$ ./configure --disable-gil --with-thread-sanitizer <.. other options ..>
```

CHAPTER

TWENTYONE

OBJECT OWNERSHIP, CONTINUED

This section covers intrusive reference counting as an alternative to shared pointers, and it explains the nitty-gritty details of how shared and unique pointer conversion is implemented in nanobind.

21.1 Intrusive reference counting

nanobind provides a custom intrusive reference counting solution that completely solves the issue of shared C++/Python object ownership, while avoiding the overheads and complexities of traditional C++ shared pointers (`std::shared_ptr<T>`).

The main limitation is that it requires adapting the base class of an object hierarchy according to the needs of nanobind, which may not always be possible.

21.1.1 Motivation

Consider the following simple class with intrusive reference counting:

```
class Object {
public:
    void inc_ref() const noexcept { ++m_ref_count; }

    void dec_ref() const noexcept {
        if (--m_ref_count == 0)
            delete this;
    }

private:
    mutable std::atomic<size_t> m_ref_count { 0 };
};
```

It contains an atomic counter that stores the number of references. When the counter reaches zero, the object deallocates itself. Easy and efficient.

The advantage of over `std::shared_ptr<T>` is that no separate control block must be allocated. Technical band-aids like `std::enable_shared_from_this<T>` can also be avoided, since the reference count is always found in the object itself.

However, one issue that tends to arise when a type like `Object` is wrapped using nanobind is that there are now *two* separate reference counts referring to the same object: one in Python's `PyObject`, and one in `Object`. This can lead to a problematic reference cycle:

- Python's `PyObject` needs to keep the `Object` instance alive so that it can be safely passed to C++ functions.
- The C++ `Object` may in turn need to keep the `PyObject` alive. This is the case when a subclass uses *trampolines* (*NB_TRAMPOLINE*, *NB_OVERRIDE*) to catch C++ virtual function calls and potentially dispatch

them to an overridden implementation in Python. In this case, the C++ instance needs to be able to perform a function call on its own Python object identity, which requires a reference.

The source of the problem is that there are *two* separate counters that try to reason about the reference count of *one* instance, which leads to an uncollectable inter-language reference cycle.

21.1.2 The solution

We can solve the problem by using just one counter:

- if an instance lives purely on the C++ side, the `m_ref_count` field is used to reason about the number of references.
- The first time that an instance is exposed to Python (by being created from Python, or by being returned from a bound C++ function), lifetime management switches over to Python.

The file nanobind/intrusive/counter.h includes an official sample implementation of this functionality. It contains an extra optimization to pack *either* a reference counter or a pointer to a `PyObject*` into a single `sizeof(void*)`-sized field.

The most basic interface, *intrusive_counter* represents an atomic counter that can be increased (via *intrusive_counter::inc_ref()*) or decreased (via *intrusive_counter::dec_ref()*). When the counter reaches zero, the object should be deleted, which `dec_ref()` indicates by returning `true`.

In addition to this simple counting mechanism, ownership of the object can also be transferred to Python (via *intrusive_counter::set_self_py()*). In this case, subsequent calls to `inc_ref()` and `dec_ref()` modify the reference count of the underlying Python object.

To incorporate intrusive reference counting into your own project, you would usually add an *intrusive_counter*-typed member to the base class of an object hierarchy and expose it as follows:

```
#include <nanobind/intrusive/counter.h>

class Object {
public:
    void inc_ref() noexcept { m_ref_count.inc_ref(); }
    bool dec_ref() noexcept { return m_ref_count.dec_ref(); }

    // Important: must declare virtual destructor
    virtual ~Object() = default;

    void set_self_py(PyObject *self) noexcept {
        m_ref_count.set_self_py(self);
    }

private:
    nb::intrusive_counter m_ref_count;
};

// Convenience function for increasing the reference count of an instance
inline void inc_ref(Object *o) noexcept {
    if (o)
       o->inc_ref();
}

// Convenience function for decreasing the reference count of an instance
// and potentially deleting it when the count reaches zero
inline void dec_ref(Object *o) noexcept {
    if (o && o->dec_ref())
```

(continues on next page)

(continued from previous page)

```
        delete o;
}
```

Alternatively, you could also inherit from *intrusive_base*, which obviates the need for all of the above declarations:

```
class Object : public nb::intrusive_base {
public:
    // ...
};
```

The main change in the bindings is that the base class must specify a *nb::intrusive_ptr* annotation to inform an instance that lifetime management has been taken over by Python. This annotation is automatically inherited by all subclasses. In the linked example, this is done via the `Object::set_self_py()` method that we can now call from the class binding annotation:

```
nb::class_<Object>(
  m, "Object",
  nb::intrusive_ptr<Object>(
      [](Object *o, PyObject *po) noexcept { o->set_self_py(po); }));
```

Also, somewhere in your binding initialization code, you must register Python reference counting hooks with the intrusive reference counter class. This allows its implementation of the code in `nanobind/intrusive/counter.h` to *not* depend on Python (this means that it can be used in projects where Python bindings are an optional component).

```
nb::intrusive_init(
    [](PyObject *o) noexcept {
        nb::gil_scoped_acquire guard;
        Py_INCREF(o);
    },
    [](PyObject *o) noexcept {
        nb::gil_scoped_acquire guard;
        Py_DECREF(o);
    });
```

These `counter.h` include file references several functions that must be compiled somewhere inside the project, which can be accomplished by including the following file from a single `.cpp` file.

```
#include <nanobind/intrusive/counter.inl>
```

Having to call *inc_ref()* and *dec_ref()* many times to perform manual reference counting in project code can quickly become tedious. Nanobind also ships with a *ref<T>* RAII helper class to help with this.

```
#include <nanobind/intrusive/ref.h>

void foo() {
    /// Assignment to ref<T> automatically increases the object's reference count
    ref<MyObject> x = new MyObject();

    // ref<T> can be used like a normal pointer
    x->func();

} // <-- ref::~ref() calls dec_ref(), which deletes the now-unreferenced instance
```

When the file `nanobind/intrusive/ref.h` is included following `nanobind/nanobind.h`, it also exposes a custom type caster to bind functions taking or returning `ref<T>`-typed values.

That's it. If you use this approach, any potential issues involving shared pointers, return value policies, reference leaks with trampolines, etc., can be avoided from the beginning.

21.2 Shared pointers, continued

The following continues the *discussion of shared pointers* in the introductory section on object ownership and provides detail on how shared pointer conversion is *implemented* by nanobind.

When the user calls a C++ function taking an argument of type `std::shared_ptr<T>` from Python, ownership of that object must be shared between C++ to Python. nanobind does this by increasing the reference count of the `PyObject` and then creating a `std::shared_ptr<T>` with a new control block containing a custom deleter that will in turn reduce the Python reference count upon destruction of the shared pointer.

When a C++ function returns a `std::shared_ptr<T>`, nanobind checks if the instance already has a `PyObject` counterpart (nothing needs to be done in this case). Otherwise, it indicates shared ownership by creating a temporary `std::shared_ptr<T>` on the heap that will be destructed when the `PyObject` is garbage collected.

The approach in nanobind was chosen following on discussions with Ralf Grosse-Kunstleve; it is unusual in that multiple `shared_ptr` control blocks are potentially allocated for the same object, which means that `std::shared_ptr<T>::use_count()` generally won't show the true global reference count.

21.2.1 enable_shared_from_this

The C++ standard library class `std::enable_shared_from_this<T>` allows an object that inherits from it to locate an existing `std::shared_ptr<T>` that manages that object. nanobind supports types that inherit from `enable_shared_from_this`, with some caveats described in this section.

Background (not nanobind-specific): Suppose a type `ST` inherits from `std::enable_shared_from_this<ST>`. When a raw pointer `ST *obj` or `std::unique_ptr<ST> obj` is wrapped in a shared pointer using a constructor of the form `std::shared_ptr<ST>(obj, ...)`, a reference to the new `shared_ptr`'s control block is saved (as `std::weak_ptr<ST>`) inside the object. This allows new `shared_ptr`s that share ownership with the existing one to be obtained for the same object using `obj->shared_from_this()` or `obj->weak_from_this()`.

nanobind's support for `std::enable_shared_from_this` consists of three behaviors:

- If a raw pointer `ST *obj` is returned from C++ to Python, and there already exists an associated `std::shared_ptr<ST>` which `obj->shared_from_this()` can locate, then nanobind will produce a Python instance that shares ownership with it. The behavior is identical to what would happen if the C++ code did `return obj->shared_from_this();` (returning an explicit `std::shared_ptr<ST>` to Python) rather than `return obj;`. The return value policy has limited effect in this case; you will get shared ownership on the Python side regardless of whether you used *`rv_policy::take_ownership`* or *`rv_policy::reference`*. (*`rv_policy::copy`* and *`rv_policy::move`* will still create a new object that has no ongoing relationship to the returned pointer.)
  - Note that this behavior occurs only if such a `std::shared_ptr<ST>` already exists! If not, then nanobind behaves as it would without `enable_shared_from_this`: a raw pointer will transfer exclusive ownership to Python by default, or will create a non-owning reference if you use *`rv_policy::reference`*.
- If a Python object is passed to C++ as `std::shared_ptr<ST> obj`, and there already exists an associated `std::shared_ptr<ST>` which `obj->shared_from_this()` can locate, then nanobind will produce a `std::shared_ptr<ST>` that shares ownership with it: an additional reference to the same control block, rather than a new control block (as would occur without `enable_shared_from_this`). This improves performance and makes the result of `shared_ptr::use_count()` more accurate.
- If a Python object is passed to C++ as `std::shared_ptr<ST> obj`, and there is no associated `std::shared_ptr<ST>` that `obj->shared_from_this()` can locate, then nanobind will produce a `std::shared_ptr<ST>` as usual (with a new control block whose deleter drops a Python object reference), *and* will do so in a way that enables future calls to `obj->shared_from_this()` to find it as long as any `shared_ptr` that shares this control block is still alive on the C++ side.

(Once all of the `std::shared_ptr<ST>`s that share this control block have been destroyed, the underlying PyObject reference being managed by the `shared_ptr` deleter will be dropped, and `shared_from_this()` will stop working. It can be reenabled by passing the Python object back to C++ as `std::shared_ptr<ST>` once more, which will create another control block.)

Bindings for a class that supports `enable_shared_from_this` will be slightly larger than bindings for a class that doesn't, as nanobind must produce type-specific code to implement the above behaviors.

Warning: The `shared_from_this()` method will only work when there is actually a `std::shared_ptr` managing the object. A nanobind instance constructed from Python will not have an associated `std::shared_ptr` yet, so `shared_from_this()` will throw an exception if you pass such an instance to C++ using a reference or raw pointer. `shared_from_this()` will only work when there exists a corresponding live `std::shared_ptr` on the C++ side.

The only situation where nanobind will create the first `std::shared_ptr` for an object (thus enabling `shared_from_this()`), even with `enable_shared_from_this`, is when a Python instance is passed to C++ as the explicit type `std::shared_ptr<T>`. If you don't do this, or if no such `std::shared_ptr` is still alive, then `shared_from_this()` will throw an exception. It also works to create the `std::shared_ptr` on the C++ side, such as by using a factory function which always uses `std::make_shared<T>(...)` to construct the object, and returns the resulting `std::shared_ptr<T>` to Python.

If you need to enable `shared_from_this` immediately upon regular Python-side object construction (i.e., `SomeType(*args)` rather than `SomeType.some_fn(*args)`), you can bind a C++ function that returns `std::shared_ptr<T>` as your class's `__new__` method. See the documentation on *customizing object creation*.

Warning: C++ code that receives a raw pointer `T *obj` *must not* assume that it has exclusive ownership of `obj`, or even that `obj` is allocated on the C++ heap (via `operator new`); `obj` might instead be a subobject of a nanobind instance allocated from Python. This applies even if `T` supports `shared_from_this()` and there is no associated `std::shared_ptr`. Lack of a `shared_ptr` does *not* imply exclusive ownership; it just means there's no way to share ownership with whoever the current owner is.

21.3 Unique pointers

The following continues the *discussion of unique pointers* in the introductory section on object ownership and provides detail on how unique pointer conversion is *implemented* by nanobind.

Whereas `std::shared_ptr<..>` could abstract over details concerning storage and the deletion mechanism, this is not possible in the simpler `std::unique_ptr<..>`, which means that some of those details leak into the type signature.

When the user calls a C++ function taking an argument of type `std::unique_ptr<T, Deleter>` from Python, ownership of that object must be transferred from C++ to Python.

- When `Deleter` is `std::default_delete<T>` (i.e., the default when no `Deleter` is specified), this ownership transfer is only possible when the instance was originally created by a *new expression* within C++ and nanobind has taken over ownership (i.e., it was created by a function returning a raw pointer `T *value` with *`rv_policy::take_ownership`*, or a function returning a `std::unique_ptr<T>`). This limitation exists because the `Deleter` will execute the statement `delete value` when the unique pointer expires, causing undefined behavior when the object was allocated within Python (the problem here is that nanobind uses the Python memory allocator and furthermore co-locates Python and C++ object storage. A *delete expression* cannot be used in such a case). nanobind detects this, refuses unsafe conversions with a `TypeError` and emits a separate warning.
- To enable ownership transfer under all conditions, nanobind provides a custom `Deleter` named *`nb::deleter<T>`* that uses reference counting to keep the underlying `PyObject` alive during the lifetime of the unique pointer. Following this route requires changing function signatures so that they use

`std::unique_ptr<T, nb::deleter<T>>` instead of `std::unique_ptr<T>`. This custom deleter supports ownership by both C++ and Python and can be used in all situations.

In both cases, a Python object may continue to exist after ownership was transferred to C++ side. nanobind marks this object as *invalid*: any operations involving it will fail with a `TypeError`. Reverse ownership transfer at a later point will make it usable again.

Binding functions that return a `std::unique_ptr<T, Deleter>` always works: nanobind will then acquire or reacquire ownership of the object.

Deleters other than `std::default_delete<T>` or `nb::deleter<T>` are *not supported*.

CHAPTER

TWENTYTWO

LOW-LEVEL INTERFACE

nanobind exposes a low-level interface to provide fine-grained control over the sequence of steps that instantiates a Python object wrapping a C++ instance. This is useful when writing generic binding code that manipulates nanobind-based objects of various types.

Given a previous *nb::class_<...>* binding declaration, the *nb::type<T>()* template function can be used to look up the Python type object associated with a C++ class named `MyClass`.

```
nb::handle py_type = nb::type<MyClass>();
```

In the case of failure, this line will return a `nullptr` pointer, which can be checked via `py_type.is_valid()`. We can verify that the type lookup succeeded, and that the returned instance indeed represents a nanobind-owned type (via *nb::type_check()*, which is redundant in this case):

```
assert(py_type.is_valid() && nb::type_check(py_type));
```

nanobind knows the size, alignment, and C++ RTTI `std::type_info` record of all bound types. They can be queried on the fly via *nb::type_size()*, *nb::type_align()*, and *nb::type_info()* in situations where this is useful.

```
assert(nb::type_size(py_type) == sizeof(MyClass) &&
       nb::type_align(py_type) == alignof(MyClass) &&
       nb::type_info(py_type) == typeid(MyClass));
```

Given a type object representing a C++ type, we can create an uninitialized instance via *nb::inst_alloc()*. This is an ordinary Python object that can, however, not (yet) be passed to bound C++ functions to prevent undefined behavior. It must first be initialized.

```
nb::object py_inst = nb::inst_alloc(py_type);
```

We can confirm via *nb::inst_check()* that this newly created instance is managed by nanobind, that it has the correct type in Python. Calling *nb::inst_ready()* reveals that the *ready* flag of the instance is set to `false` (i.e., it is still uninitialized).

```
assert(nb::inst_check(py_inst) &&
       py_inst.type().is(py_type) &&
       !nb::inst_ready(py_inst));
```

For simple *plain old data* (POD) types, the *nb::inst_zero()* function can be used to *zero-initialize* the object and mark it as ready.

```
nb::inst_zero(py_inst);
assert(nb::inst_ready(py_inst));
```

We can destruct this default instance via *nb::inst_destruct()* and convert it back to non-ready status. This memory region can then be reinitialized once more.

```
nb::inst_destruct(py_inst);
assert(!nb::inst_ready(py_inst));
```

What follows is a more interesting example, where we use a lesser-known feature of C++ (the "placement new" operator) to construct an instance *in-place* into the memory region allocated by nanobind.

```
// Get a C++ pointer to the uninitialized instance data
MyClass *ptr = nb::inst_ptr<MyClass>(py_inst);

// Perform an in-place construction of the C++ object at address 'ptr'
new (ptr) MyClass(/* constructor arguments go here */);
```

Following this constructor call, we must inform nanobind that the instance object is now fully constructed via *nb::inst_mark_ready()*. When its reference count reaches zero, nanobind will then automatically call the in-place destructor (`MyClass::~MyClass`).

```
nb::inst_mark_ready(py_inst);
assert(nb::inst_ready(py_inst));
```

Let's destroy this instance once more manually (which will, again, call the C++ destructor and mark the Python object as non-ready).

```
nb::inst_destruct(py_inst);
```

Another useful feature is that nanobind can copy- or move-construct `py_inst` from another instance of the same type via *nb::inst_copy()* and *nb::inst_move()*. These functions call the C++ copy or move constructor and transition `py_inst` back to `ready` status. This is equivalent to calling an in-place version of these constructors followed by a call to *nb::inst_mark_ready()* but compiles to more compact code (the *nb::class_<MyClass>* declaration had already created bindings for both constructors, and this simply calls those bindings).

```
if (copy_instance)
    nb::inst_copy(/* dst = */ py_inst, /* src = */ some_other_instance);
else
    nb::inst_move(/* dst = */ py_inst, /* src = */ some_other_instance);
```

Both functions assume that the destination object is uninitialized. Two alternative versions *nb::inst_replace_copy()* and *nb::inst_replace_move()* destruct an initialized instance and replace it with the contents of another by either copying or moving.

```
if (copy_instance)
    nb::inst_replace_copy(/* dst = */ py_inst, /* src = */ some_other_instance);
else
    nb::inst_replace_move(/* dst = */ py_inst, /* src = */ some_other_instance);
```

Note that these functions are all *unsafe* in the sense that they do not verify that their input arguments are valid. This is done for performance reasons, and such checks (if needed) are therefore the responsibility of the caller. Functions labeled `nb::type_*` should only be called with nanobind type objects, and functions labeled `nb::inst_*` should only be called with nanobind instance objects.

The functions *nb::type_check()* and *nb::inst_check()* are exceptions to this rule: they accept any Python object and test whether something is a nanobind type or instance object.

Two further functions *nb::type_name()* and *nb::inst_name()* determine the type name associated with a type or instance thereof. These also accept non-nanobind types and instances.

22.1 Even lower-level interface

Every nanobind object has two important flags that control its behavior:

1. `ready`: is the object fully constructed? If set to `false`, nanobind will raise an exception when the object is passed to a bound C++ function.
2. `destruct`: Should nanobind call the C++ destructor when the instance is garbage collected?

The functions *nb::inst_zero()*, *nb::inst_mark_ready()*, *nb::inst_move()*, and *nb::inst_copy()* set both of these flags to `true`, and *nb::inst_destruct()* sets both of them to `false`.

In rare situations, the destructor should *not* be invoked when the instance is garbage collected, for example when working with a nanobind instance representing a field of a parent instance created using the *nb::rv_policy::reference_internal* return value policy. The library therefore exposes two more functions *nb::inst_state()* and *nb::inst_set_state()* that can be used to access them individually.

22.2 Referencing existing instances

The above examples used the function *nb::inst_alloc()* to allocate a Python object along with space to hold a C++ instance associated with the binding `py_type`.

```
nb::object py_inst = nb::inst_alloc(py_type);

// Next, perform a C++ in-place construction into the
// address given by nb::inst_ptr<MyClass>(py_inst)
... omitted, see the previous examples ...
```

What if the C++ instance already exists? nanobind also supports this case via the *nb::inst_reference()* and *nb::inst_take_ownership()* functions—in this case, the Python object references the existing memory region, which is potentially (slightly) less efficient due to the need for an extra indirection.

```
MyClass *inst = new MyClass();

// Transfer ownership of 'inst' to Python (which will use a delete
// expression to free it when the Python instance is garbage collected)
nb::object py_inst = nb::inst_take_ownership(py_type, inst);

// We can also wrap C++ instances that should not be destructed since
// they represent offsets into another data structure. In this case,
// the optional 'parent' parameter ensures that 'py_inst' remains alive
// while 'py_subinst' exists to prevent undefined behavior.
nb::object py_subinst = nb::inst_reference(
    py_field_type, &inst->field, /* parent = */ py_inst);
```

22.3 Supplemental type data

nanobind can stash supplemental data *inside* the type object of bound types. This involves the *nb::supplement<T>()* class binding annotation to reserve space and *nb::type_supplement<T>()* to access the reserved memory region.

An example use of this fairly advanced feature are libraries that register large numbers of different types (e.g. flavors of tensors). A single generically implemented function can then query the supplemental data block to handle each tensor type slightly differently.

Here is what this might look like in an implementation:

```
struct MyTensorMetadata {
    bool stored_on_gpu;
    // ..
    // should be a POD (plain old data) type
};

// Register a new type MyTensor, and reserve space for sizeof(MyTensorMedadata)
nb::class_<MyTensor> cls(m, "MyTensor", nb::supplement<MyTensorMedadata>())

/// Mutable reference to 'MyTensorMedadata' portion in Python type object
MyTensorMedadata &supplement = nb::type_supplement<MyTensorMedadata>(cls);
supplement.stored_on_gpu = true;
```

The *nb::supplement<T>()* annotation implicitly also passes *nb::is_final()* to ensure that type objects with supplemental data cannot be subclassed in Python.

nanobind requires that the specified type `T` be trivially default constructible. It zero-initializes the supplement when the type is first created but does not perform any further custom initialization or destruction. You can fill the supplement with different contents following the type creation, e.g., using the placement new operator.

The contents of the supplemental data are not directly visible to Python's cyclic garbage collector, which creates challenges if you want to reference Python objects. The recommended workaround is to store the Python objects as attributes of the type object (in its `__dict__`) and store a borrowed `PyObject*` reference in the supplemental data. If you use an attribute name that begins with the symbol `@`, then nanobind will prevent Python code from rebinding or deleting the attribute after it has been set, making the borrowed reference reasonably safe.

CHAPTER

TWENTYTHREE

CUSTOMIZING TYPE CREATION

nanobind exposes a low-level interface to install custom *type slots* (`PyType_Slot` in the CPython API) in newly constructed types. This provides an escape hatch to realize features that were not foreseen in the design of this library.

To use this feature, specify the *nb::type_slots()* annotation when creating the type.

```
nb::class_<MyClass>(m, "MyClass", nb::type_slots(slots));
```

Here, `slots` should refer to an array of function pointers that are tagged with a corresponding slot identifier. For example, here is an example function that overrides the addition operator so that it behaves like a multiplication.

```
PyObject *myclass_tp_add(PyObject *a, PyObject *b) {
    return PyNumber_Multiply(a, b);
}

PyType_Slot slots[] = {
    { Py_nb_add, (void *) myclass_tp_add },
    { 0, nullptr }
};
```

The `slots` array specified in the previous *nb::class_<MyClass>()* declaration references the function `myclass_tp_add` and is followed by a mandatory null terminator. Information on type slots can be found in the CPython documentation sections covering type objects and type construction.

This example is contrived because it could have been accomplished using builtin features:

```
nb::class_<MyClass>(m, "MyClass")
    .def("__add__",
         [](const MyClass &a, const MyClass &b) { return a * b; },
         nb::is_operator())
```

The next section introduces a more interesting use case.

23.1 Cyclic garbage collection

Python tracks the lifetime of objects using an approach known as *reference counting*. An object can be safely deconstructed once it is no longer referenced from elsewhere, which happens when its reference count reaches zero.

This mechanism is simple and efficient, but it breaks down when objects form *reference cycles*. For example, consider the following data structure

```
struct Wrapper {
    std::shared_ptr<Wrapper> value;
};
```

with associated bindings

```
nb::class_<Wrapper>(m, "Wrapper")
    .def(nb::init<>())
    .def_rw("value", &Wrapper::value);
```

If we instantiate this class with a cycle, it can never be reclaimed (even when Python shuts down and is supposed to free up all memory):

```
>>> a = my_ext.Wrapper()
>>> a.value = a
>>> del a
```

nanobind will loudly complain about this when the Python interpreter shuts down:

```
>>> exit()
nanobind: leaked 1 instances!
nanobind: leaked 1 types!
 - leaked type "my_ext.Wrapper"
nanobind: leaked 3 functions!
 - leaked function "<anonymous>"
 - leaked function "__init__"
 - leaked function "<anonymous>"
nanobind: this is likely caused by a reference counting issue in the binding code.
```

The leaked `Wrapper` instance `a` references the `Wrapper` type, which in turn references function definitions, causing a longer sequence of warnings.

Python provides a *cyclic garbage collector* that can in principle solve this problem. To operate correctly, it requires information about how objects are connected to each other.

We can provide a `tp_traverse` type slot that walks through the object graph to inform the cyclic GC, and a `tp_clear` slot to break any detected reference cycles:

```
int wrapper_tp_traverse(PyObject *self, visitproc visit, void *arg) {
    // Retrieve a pointer to the C++ instance associated with 'self' (never fails)
    Wrapper *w = nb::inst_ptr<Wrapper>(self);

    // If w->value has an associated CPython object, return it.
    // If not, value.ptr() will equal NULL, which is also fine.
    nb::handle value = nb::find(w->value);

    // Inform the Python GC about the instance (if non-NULL)
    Py_VISIT(value.ptr());

    return 0;
}

int wrapper_tp_clear(PyObject *self) {
    // Retrieve a pointer to the C++ instance associated with 'self' (never fails)
    Wrapper *w = nb::inst_ptr<Wrapper>(self);

    // Clear the cycle!
    w->value.reset();

    return 0;
}

// Slot data structure referencing the above two functions
```

(continues on next page)

(continued from previous page)

```
PyType_Slot slots[] = {
    { Py_tp_traverse, (void *) wrapper_tp_traverse },
    { Py_tp_clear, (void *) wrapper_tp_clear },
    { 0, nullptr }
};
```

The type `visitproc` and macro `Py_VISIT()` are part of the Python C API.

Note: When targeting free-threaded Python, it is important that the `tp_traverse` callback does not hold additional references to the objects being traversed.

A previous version of this documentation page suggested the following

```
nb::object value = nb::find(w->value);
Py_VISIT(value.ptr());
```

However, these now have to change to

```
nb::handle value = nb::find(w->value);
Py_VISIT(value.ptr());
```

The expression *nb::inst_ptr<Wrapper>(self)* efficiently returns the C++ instance associated with a Python object and is explained in the documentation about nanobind's *low level interface*.

Note the use of the *nb::find()* function, which behaves like *nb::cast()* by returning the Python object associated with a C++ instance. The main difference is that *nb::cast()* will create the Python object if it doesn't exist, while *nb::find()* returns a `nullptr` object in that case.

To activate this machinery, the `Wrapper` type bindings must be made aware of these extra type slots:

```
nb::class_<Wrapper>(m, "Wrapper", nb::type_slots(slots))
```

With this change, the cycle can be garbage-collected, and the leak warnings disappear.

23.2 Reference cycles involving functions

What if our wrapper class from the previous example instead stored a function object?

```
struct Wrapper {
    std::function<void(void)> value;
};
```

It may not be immediately obvious, but functions are one of the main sources of reference cycles! For example, in Python we could write

```
>>> a = my_ext.Wrapper()
>>> a.value = lambda: print(a)
```

This function is actually a function closure because it references external variable state (its body accesses `a`). This creates an inter-language cycle `Wrapper` → `function` (itself wrapped in `std::function<void(void)>`) → `Wrapper`.

Such cycles are extremely common when Python-based callbacks can be installed in C++ classes. An example would be a callback handler triggered by a button press in a GUI framework. It is important to detect and handle such cycles.

When given a `std::function<>` instance, *nb::find()* retrieves the associated Python `function` object (if present), which means that the previous `wrapper_tp_traverse()` traversal function continues to work without changes. The `tp_clear` slot requires small touch-ups:

```
int wrapper_tp_clear(PyObject *self) {
    Wrapper *w = nb::inst_ptr<Wrapper>(self);
    w->value = nullptr;
    return 0;
}
```

That's it!

CHAPTER

TWENTYFOUR

C++ API REFERENCE (CORE)

24.1 Macros

NB_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. It **must** match the module name given to the *nanobind_add_module()* function in the CMake build system.

The second macro argument defines a variable of type *module_*. The body of the declaration typically contains a sequence of operations that populate the module variable with contents.

```
NB_MODULE(example, m) {
    m.doc() = "Example module";

    // Add bindings here
    m.def("add", []() {
        return "Hello, World!";
    });
}
```

NB_MAKE_OPAQUE(T)

The macro registers a partial template specialization pattern for the type `T` that marks it as *opaque*, meaning that nanobind won't try to run its type casting template machinery on it.

This is useful when trying to register a binding for `T` that is simultaneously also covered by an existing type caster.

This macro should be used at the top level (outside of namespaces and program code).

24.2 Python object API

Nanobind ships with a wide range of Python wrapper classes like *object*, *list*, etc. Besides class-specific operations (e.g., *list::append()*), these classes also implement core operations that can be performed on *any* Python object. Since it would be tedious to implement this functionality over and over again, it is realized by the following mixin class that lives in the `nanobind::detail` namespace.

template<typename **Derived**>
class **api**

This mixin class adds common functionality to various nanobind types using the curiously recurring template pattern (CRTP). The only requirement for the *Derived* template parameter is that it implements the member function `PyObject *ptr() const` that gives access to the underlying Python object pointer.

Derived &**derived**()

Obtain a mutable reference to the derived class.

const *Derived* &**derived**() const

Obtain a const reference to the derived class.

handle **inc_ref**() const

Increases the reference count and returns a reference to the Python object.

handle **dec_ref**() const

Decreases the reference count and returns a reference to the Python object.

iterator **begin**() const

Return a forward iterator analogous to `iter()` in Python. The object must be a collection that supports the iteration protocol. This interface provides a generic iterator that works any type of Python object. The *tuple*, *list*, and *dict* wrappers provide more efficient specialized alternatives.

iterator **end**() const

Return a sentinel that ends the iteration.

handle **type**() const

Return a *handle* to the underlying Python type object.

operator *handle*() const

Return a *handle* wrapping the underlying `PyObject*` pointer.

detail::*accessor*<obj_attr> **attr**(*handle* key) const

Analogous to `self.key` in Python, where `key` is a Python object. The result is wrapped in an *accessor* so that it can be read and written.

detail::*accessor*<str_attr> **attr**(const char *key) const

Analogous to `self.key` in Python, where `key` is a C-style string. The result is wrapped in an *accessor* so that it can be read and written.

detail::*accessor*<str_attr> **doc**() const

Analogous to `self.__doc__`. The result is wrapped in an *accessor* so that it can be read and written.

detail::*accessor*<obj_item> **operator[]**(*handle* key) const

Analogous to `self[key]` in Python, where `key` is a Python object. The result is wrapped in an *accessor* so that it can be read and written.

detail::*accessor*<str_item> **operator[]**(const char *key) const

Analogous to `self[key]` in Python, where `key` is a C-style string. The result is wrapped in an *accessor* so that it can be read and written.

template<typename **T**, enable_if_t<std::is_arithmetic_v<*T*>> = 1>
detail::*accessor*<num_item> **operator[]**(*T* key) const

Analogous to `self[key]` in Python, where `key` is an arithmetic type (e.g., an integer). The result is wrapped in an *accessor* so that it can be read and written.

template<*rv_policy* **policy** = *rv_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, while expanding any detected variable length argument and keyword argument packs. The result is returned as an *object* and may need to be converted back into a Python object using *cast()*.

Type conversion is performed using the return value policy *policy*

When type conversion of arguments or return value fails, the function raises a *cast_error*. When the Python function call fails, it instead raises a *python_error*.

args_proxy **operator***() const

Given a a tuple or list, this helper function performs variable argument list unpacking in function calls resembling the * operator in Python. Applying *operator*()* twice yields ** keyword argument unpacking for dictionaries.

bool **is**(*handle* value) const

Analogous to `self is value` in Python.

bool **is_none**() const

Analogous to `self is None` in Python.

bool **is_type**() const

Analogous to `isinstance(self, type)` in Python.

bool **is_valid**() const

Checks if this wrapper contains a valid Python object (in the sense that the `PyObject *` pointer is non-null).

template<typename **T**>
bool **equal**(const *api*<*T*> &other)

Equivalent to `self == other` in Python.

template<typename **T**>
bool **not_equal**(const *api*<*T*> &other)

Equivalent to `self != other` in Python.

template<typename **T**>
bool **operator<**(const *api*<*T*> &other)

Equivalent to `self < other` in Python.

template<typename **T**>
bool **operator<=**(const *api*<*T*> &other)

Equivalent to `self <= other` in Python.

template<typename **T**>
bool **operator>**(const *api*<*T*> &other)

Equivalent to `self > other` in Python.

template<typename **T**>
bool **operator>=**(const *api*<*T*> &other)

Equivalent to `self >= other` in Python.

object **operator-**()

Equivalent to `-self` in Python.

object **operator~**()

Equivalent to `~self` in Python.

template<typename **T**>
object **operator+**(const *api*<*T*> &other)

Equivalent to `self + other` in Python.

template<typename **T**>
object **operator-**(const *api*<*T*> &other)

Equivalent to `self - other` in Python.

template<typename **T**>
object **operator***(const *api*<*T*> &other)

Equivalent to `self * other` in Python.

template<typename **T**>
object **operator/**(const *api*<*T*> &other)

Equivalent to `self / other` in Python.

template<typename **T**>

object **floor_div**(const *api*<*T*> &other)

Equivalent to `self // other` in Python.

template<typename **T**>
object **operator|**(const *api*<*T*> &other)

Equivalent to `self | other` in Python.

template<typename **T**>
object **operator&**(const *api*<*T*> &other)

Equivalent to `self & other` in Python.

template<typename **T**>
object **operator^**(const *api*<*T*> &other)

Equivalent to `self ^ other` in Python.

template<typename **T**>
object **operator<<**(const *api*<*T*> &other)

Equivalent to `self << other` in Python.

template<typename **T**>
object **operator>>**(const *api*<*T*> &other)

Equivalent to `self >> other` in Python.

template<typename **T**>
object **operator+=**(const *api*<*T*> &other)

Equivalent to `self += other` in Python. Note that the *api* version of the in-place operator does not update the `self` reference, which may lead to unexpected results when working with immutable types that return their result instead of updating `self`.

The *object* class and subclasses override the in-place operators to achieve more intuitive behavior.

template<typename **T**>
object **operator-=**(const *api*<*T*> &other)

Equivalent to `self -= other` in Python. See *operator+=()* for limitations.

template<typename **T**>
object **operator*=**(const *api*<*T*> &other)

Equivalent to `self *= other` in Python. See *operator+=()* for limitations.

template<typename **T**>
object **operator/=**(const *api*<*T*> &other)

Equivalent to `self /= other` in Python. See *operator+=()* for limitations.

template<typename **T**>
object **operator|=**(const *api*<*T*> &other)

Equivalent to `self |= other` in Python. See *operator+=()* for limitations.

template<typename **T**>
object **operator&=**(const *api*<*T*> &other)

Equivalent to `self &= other` in Python. See *operator+=()* for limitations.

template<typename **T**>
object **operator^=**(const *api*<*T*> &other)

Equivalent to `self ^= other` in Python. See *operator+=()* for limitations.

template<typename **T**>
object **operator<<=**(const *api*<*T*> &other)

Equivalent to `self <<= other` in Python. See *operator+=()* for limitations.

template<typename **T**>

object **operator>>=**(const *api*<*T*> &other)

Equivalent to `self >>= other` in Python. See *operator+=()* for limitations.

template<typename **Impl**>
class **accessor**

This helper class facilitates attribute and item access. Casting an *accessor* to a *handle* or *object* subclass causes a corresponding call to `__getitem__` or `__getattr__` depending on the template argument *Impl*. Assigning a *handle* or *object* subclass causes a call to `__setitem__` or `__setattr__`.

24.3 Handles and objects

nanobind provides two styles of Python object wrappers: classes without reference counting deriving from *handle*, and reference-counted wrappers deriving from *object*. Reference counting bugs can be really tricky to track down, hence it is recommended that you always prefer *object*-style wrappers unless there are specific reasons that warrant the use of raw handles.

24.3.1 Without reference counting

class **handle** : public detail::*api*<*handle*>

This class provides a thin wrapper around a raw `PyObject *` pointer. Its main purpose is to intercept various C++ operations and convert them into Python C API calls. It does *not* do any reference counting and can be somewhat unsafe to use.

handle() = default

Default constructor. Creates an invalid handle wrapping a null pointer. (*detail::api::is_valid()* is `false`)

handle(const *handle*&) = default

Default copy constructor.

handle(*handle*&&) = default

Default move constructor.

handle(const PyObject *o)

Initialize a handle from a Python object pointer. Does not change the reference count of *o*.

handle(const PyTypeObject *o)

Initialize a handle from a Python type object pointer. Does not change the reference count of *o*.

handle &**operator=**(const *handle*&) = default

Default copy assignment operator.

handle &**operator=**(*handle*&&) = default

Default move assignment operator.

explicit **operator bool**() const

Check if the handle refers to a valid Python object. Equivalent to *detail::api::is_valid()*

handle **inc_ref**() const noexcept

Increases the reference count and returns a reference to the Python object. Never raises an exception.

handle **dec_ref**() const noexcept

Decreases the reference count and returns a reference to the Python object. Never raises an exception.

PyObject ***ptr**() const

Return the underlying `PyObject*` pointer.

24.3.2 With reference counting

class **object** : public *handle*

This class provides a convenient RAII wrapper around a `PyObject*` pointer. Like *handle*, it intercepts various C++ operations and converts them into Python C API calls.

The main difference to *handle* is that it uses reference counting to keep the underlying Python object alive.

Use the *borrow()* and *steal()* functions to create an *object* from a *handle* or `PyObject*` pointer.

object() = default

Default constructor. Creates an invalid object wrapping a null pointer. (*detail::api::is_valid()* is `false`)

object(*object* &&o)

Move constructor. Steals the object from *o* without changing its reference count.

object(const *object* &o)

Copy constructor. Acquires a new reference to *o* (if valid).

~object()

Decrease the reference count of the referenced Python object (if valid).

object &**operator=**(*object* &&o)

Move assignment operator. Decreases the reference count of the currently held object (if valid) and steals the object from *o* without changing its reference count.

object &**operator=**(const *object* &o)

Copy assignment operator. Decreases the reference count of the currently held object (if valid) and acquires a new reference to the object *o* (if valid).

void **reset**()

Decreases the reference count of the currently held object (if valid) and resets the internal pointer to `nullptr`.

handle **release**()

Resets the internal pointer to `nullptr` and returns its previous contents as a *handle*. This operation does not change the object's reference count and should be used carefully.

template<typename **T**>
object &**operator+=**(const api<*T*> &other)

Equivalent to `self += other` in Python.

template<typename **T**>
object &**operator-=**(const api<*T*> &other)

Equivalent to `self -= other` in Python.

template<typename **T**>
object &**operator*=**(const api<*T*> &other)

Equivalent to `self *= other` in Python.

template<typename **T**>
object &**operator/=**(const api<*T*> &other)

Equivalent to `self /= other` in Python.

template<typename **T**>
object &**operator|=**(const api<*T*> &other)

Equivalent to `self |= other` in Python.

template<typename **T**>

object &**operator&=**(const api<*T*> &other)

Equivalent to `self &= other` in Python.

template<typename **T**>
object &**operator^=**(const api<*T*> &other)

Equivalent to `self ^= other` in Python.

template<typename **T**>
object &**operator<<=**(const api<*T*> &other)

Equivalent to `self <<= other` in Python.

template<typename **T**>
object &**operator>>=**(const api<*T*> &other)

Equivalent to `self >>= other` in Python.

template<typename **T** = *object*>
T **borrow**(*handle* h)

Create a reference-counted Python object wrapper of type *T* from a raw handle or `PyObject *` pointer. The target type *T* must be *object* (the default) or one of its derived classes. The function does not perform any conversions or checks—it is up to the user to make sure that the target type is correct.

The function *borrows* a reference, which means that it will increase the reference count while constructing `T`.

For example, consider the Python C API function PyList_GetItem(), whose documentation states that it returns a borrowed reference. An interface between this API and nanobind could look as follows:

```
PyObject* list = ...;
Py_ssize_t index = ...;
nb::object o = nb::borrow(PyList_GetItem(obj, index));
```

Using *steal()* in this setting is incorrect and would lead to a reference underflow.

template<typename **T** = *object*>
T **steal**(*handle* h)

Create a reference-counted Python object wrapper of type *T* from a raw handle or `PyObject *` pointer. The target type *T* must be *object* (the default) or one of its derived classes. The function does not perform any conversions or checks—it is up to the user to make sure that the target type is correct.

The function *steals* a reference, which means that constructing `T` leaves the object's reference count unchanged.

For example, consider the Python C API function PyObject_Str(), whose documentation states that it returns a *new reference*. An interface between this API and nanobind could look as follows:

```
PyObject* value = ...;
nb::object o = nb::steal(PyObject_Str(value));
```

Using *borrow()* in this setting is incorrect and would lead to a reference leak.

24.4 Attribute access

bool **hasattr**(*handle* h, const char *key) noexcept

Check if the given object has an attribute string `key`. The function never raises an exception and returns `false` in case of an internal error.

Equivalent to `hasattr(h, key)` in Python.

bool **hasattr**(*handle* h, *handle* key) noexcept

Check if the given object has a attribute represented by the Python object `key`. The function never raises an exception and returns `false` in case of an internal error.

Equivalent to `hasattr(h, key)` in Python.

object **getattr**(*handle* h, const char *key)

Equivalent to `h.key` and `getattr(h, key)` in Python. Raises *python_error* if the operation fails.

object **getattr**(*handle* h, *handle* key)

Equivalent to `h.key` and `getattr(h, key)` in Python. Raises *python_error* if the operation fails.

object **getattr**(*handle* h, const char *key, *handle* def) noexcept

Equivalent to `getattr(h, key, def)` in Python. Never raises an exception and returns `def` when the operation fails, or when the desired attribute could not be found.

object **getattr**(*handle* h, *handle* key, *handle* def) noexcept

Equivalent to `getattr(h, key, def)` in Python. Never raises an exception and returns `def` when the operation fails, or when the desired attribute could not be found.

void **setattr**(*handle* h, const char *key, *handle* value)

Equivalent to `h.key = value` and `setattr(h, key, value)` in Python. Raises *python_error* if the operation fails.

void **setattr**(*handle* h, *handle* key, *handle* value)

Equivalent to `h.key = value` and `setattr(h, key, value)` in Python. Raises *python_error* if the operation fails.

void **delattr**(*handle* h, const char *key)

Equivalent to `del h.key` and `delattr(h, key)` in Python. Raises *python_error* if the operation fails.

void **delattr**(*handle* h, *handle* key)

Equivalent to `del h.key` and `delattr(h, key)` in Python. Raises *python_error* if the operation fails.

template<typename **T**>
void **del**(detail::*accessor*<*T*>&)

Remove an element from a sequence or mapping. The C++ statement

```
nb::del(o[key]);
```

is equivalent to `del o[key]` in Python.

When the element cannot be removed, the function will raise *python_error* wrapping either a Python `IndexError` (for sequence types) or a `KeyError` (for mapping types).

template<typename **T**>
void **del**(detail::*accessor*<*T*>&&)

Rvalue equivalent of the above expression.

24.5 Size queries

size_t **len**(*handle* h)

Equivalent to `len(h)` in Python. Raises *python_error* if the operation fails.

size_t **len**(const *tuple* &t)

Equivalent to `len(t)` in Python. Optimized variant for tuples.

size_t **len**(const *list* &l)

Equivalent to `len(l)` in Python. Optimized variant for lists.

size_t **len**(const *dict* &d)

Equivalent to `len(d)` in Python. Optimized variant for dictionaries.

size_t **len**(const *set* &d)

Equivalent to `len(d)` in Python. Optimized variant for sets.

size_t **len_hint**(*handle* h)

Equivalent to `operator.length_hint(h)` in Python. Raises *python_error* if the operation fails.

24.6 Type queries

template<typename **T**>
isinstance(*handle* h)

Checks if the Python object *h* represents a valid instance of the C++ type *T*. This works for bound C++ classes, basic types (`int`, `bool`, etc.), and Python type wrappers (*list*, *dict*, *module_*, etc.).

Note: the check even works when *T* involves a type caster (e.g., an STL types like `std::vector<float>`). However, this involve a wasteful attempt to convert the object to C++. It may be more efficient to just perform the conversion using *cast()* and catch potential raised exceptions.

isinstance(*handle* inst, *handle* cls)

Checks if the Python object *inst* is an instance of the Python type *cls*.

template<typename **T**>
handle **type**() noexcept

Returns the Python type object associated with the C++ type *T*. When the type not been bound via nanobind, the function returns an invalid handle (*detail::api::is_valid()* is `false`).

Note: in contrast to the *isinstance()* function above, builtin types, type wrappers, and types handled using type casters, are *not* supported.

24.7 Wrapper classes

class **tuple** : public *object*

Wrapper class representing Python `tuple` instances.

Use the standard `operator[]` C++ operator with an integer argument to read tuple elements (the bindings for this operator are provided by the parent class and not listed here). Once created, the set is immutable and its elements cannot be replaced.

Use the `make_tuple()` function to create new tuples.

tuple()

Create an empty tuple

tuple(*handle* h)

Attempt to convert a given Python object into a tuple. Analogous to the expression `tuple(h)` in Python.

size_t **size**() const

Return the number of tuple elements.

detail::fast_iterator **begin**() const

Return a forward iterator analogous to `iter()` in Python. The function overrides a generic version in *detail::api* and is more efficient for tuples.

detail::fast_iterator **end**() const

Return a sentinel that ends the iteration.

template<typename **T**, enable_if_t<std::is_arithmetic_v<*T*>> = 1>
detail::*accessor*<num_item_tuple> **operator[]**(*T* key) const

Analogous to `self[key]` in Python, where `key` is an arithmetic type (e.g., an integer). The result is wrapped in an *accessor* so that it can be read and converted. Write access is not possible.

The function overrides the generic version in *detail::api* and is more efficient for tuples.

class **list** : public *object*

Wrapper class representing Python `list` instances.

Use the standard `operator[]` C++ operator with an integer argument to read and write list elements (the bindings for this operator are provided by the parent class and not listed here).

Use the *nb::del* function to remove elements.

list()

Create an empty list

list(*handle* h)

Attempt to convert a given Python object into a list. Analogous to the expression `list(h)` in Python.

size_t **size**() const

Return the number of list elements.

template<typename **T**>
void **append**(*T* &&value)

Append an element to the list. When *T* does not already represent a wrapped Python object, the function performs a cast.

template<typename **T**>
void **insert**(Py_ssize_t index, *T* &&value)

Insert an element to the list (at index `index`, which may also be negative). When *T* does not already represent a wrapped Python object, the function performs a cast.

void **clear**()

Clear the list entries.

void **extend**(*handle* h)

Analogous to the `.extend(h)` method of `list` in Python.

void **sort**()

Analogous to the `.sort()` method of `list` in Python.

void **reverse**()

Analogous to the `.reverse()` method of `list` in Python.

template<typename **T**, enable_if_t<std::is_arithmetic_v<*T*>> = 1>
detail::*accessor*<num_item_list> **operator[]**(*T* key) const

Analogous to `self[key]` in Python, where `key` is an arithmetic type (e.g., an integer). The result is wrapped in an *accessor* so that it can be read and written.

The function overrides the generic version in *detail::api* and is more efficient for lists.

detail::fast_iterator **begin**() const

Return a forward iterator analogous to `iter()` in Python. The operator provided here overrides the generic version in *detail::api* and is more efficient for lists.

detail::fast_iterator **end**() const

Return a sentinel that ends the iteration.

class **dict** : public *object*

Wrapper class representing Python `dict` instances.

Use the standard `operator[]` C++ operator to read and write dictionary elements (the bindings for this operator are provided by the parent class and not listed here).

Use the *nb::del* function to remove elements.

dict()

Create an empty dictionary

size_t **size**() const

Return the number of dictionary elements.

template<typename **T**>
bool **contains**(*T* &&key) const

Check whether the dictionary contains a particular key. When *T* does not already represent a wrapped Python object, the function performs a cast.

detail::dict_iterator **begin**() const

Return an item iterator that returns `std::pair<handle, handle>` key-value pairs analogous to `iter(dict.items())` in Python.

In free-threaded Python, the :cpp:class:`detail::dict_iterator` class acquires a lock to the underlying dictionary to enable the use of the efficient but thread-unsafe `PyDict_Next()` Python C traversal routine.

detail::dict_iterator **end**() const

Return a sentinel that ends the iteration.

list **keys**() const

Return a list containing all dictionary keys.

list **values**() const

Return a list containing all dictionary values.

list **items**() const

Return a list containing all dictionary items as `(key, value)` pairs.

void **clear**()

Clear the contents of the dictionary.

void **update**(*handle* h)

Analogous to the `.update(h)` method of `dict` in Python.

class **set** : public *object*

Wrapper class representing Python `set` instances.

set()

Create an empty set

set(*handle* h)

Attempt to convert a given Python object into a set. Analogous to the expression `set(h)` in Python.

size_t **size**() const

Return the number of set elements.

template<typename **T**>
void **add**(*T* &&key)

Add a key to the set. When *T* does not already represent a wrapped Python object, the function performs a cast.

template<typename **T**>

bool **contains**(*T* &&key) const

Check whether the set contains a particular key. When *T* does not already represent a wrapped Python object, the function performs a cast.

void **clear**()

Clear the contents of the set.

template<typename **T**>
bool **discard**(*T* &&key)

Analogous to the `.discard(h)` method of the `set` type in Python. Returns `true` if the item was deleted successfully, and `false` if the value was not present. When *T* does not already represent a wrapped Python object, the function performs a cast.

class **module_** : public *object*

Wrapper class representing Python `module` instances. The underscore at the end disambiguates the class name from the C++20 `module` declaration.

template<typename **Func**, typename ...**Extra**>
*module_* &**def**(const char *name, *Func* &&f, const *Extra*&... extra)

Bind the function *f* to the identifier *name* within the module. Returns a reference to `*this` so that longer sequences of binding declarations can be chained, as in `m.def(...).def(...);`. The variable length *extra* parameter can be used to pass docstrings and other *function binding annotations*.

Example syntax:

```
void test() { printf("Hello world!"); }

NB_MODULE(example, m) {
    // here, "m" is variable of type 'module_'.
    m.def("test", &test, "A test function")
     .def(...); // more binding declarations
}
```

*module_* **import_**(const char *name)

Import the Python module with the specified name and return a reference to it. The underscore at the end disambiguates the function name from the C++20 `import` statement.

Example usage:

```
nb::module_ np = nb::module_::import_("numpy");
nb::object np_array = np.attr("array");
```

*module_* **import_**(*handle* name)

Import the Python module with the specified name and return a reference to it. In contrast to the version above, this function expects a Python object as key.

*module_* **def_submodule**(const char *name, const char *doc = nullptr)

Create a Python submodule within an existing module and return a reference to it. Can be chained recursively.

Example usage:

```
NB_MODULE(example, m) {
    nb::module_ m2 = m.def_submodule("sub", "A submodule of 'example'");
    nb::module_ m3 = m2.def_submodule("subsub", "A submodule of 'example.sub'
 ↪");
}
```

class **capsule** : public *object*

Capsules are small opaque Python objects that wrap a C or C++ pointer and a cleanup routine.

capsule(const void *ptr, void (*cleanup)(void*) noexcept = nullptr)

Construct an *unnamed* capsule wrapping the pointer `p`. When the capsule is garbage collected, Python will call the destructor *cleanup* (if provided) with the value of `p`.

capsule(const void *ptr, const char *name, void (*cleanup)(void*) noexcept = nullptr)

Construct a *named* capsule with name *name* wrapping the pointer `p`. When the capsule is garbage collected, Python will call the destructor *cleanup* (if provided) with the value of `p`.

const char ***name**() const

Return the capsule name (or `nullptr` when the capsule is unnamed)

void ***data**() const

Return the pointer wrapped by the capsule.

class **bool_** : public *object*

This wrapper class represents Python `bool` instances.

bool_(*handle* h)

Performs a boolean cast within Python. This is equivalent to the Python expression `bool(h)`.

explicit **bool_**(bool value)

Convert an C++ boolean instance into a Python `bool`.

explicit **operator bool**() const

Extract the boolean value underlying this object.

class **int_** : public *object*

This wrapper class represents Python `int` instances. It can handle large numbers requiring more than 64 bits of storage.

int_(*handle* h)

Performs an integer cast within Python. This is equivalent to the Python expression `int(h)`.

template<typename **T**, detail::enable_if_t<std::is_arithmetic_v<*T*>> = 0>
explicit **int_**(*T* value)

Convert an C++ arithmetic type into a Python integer.

template<typename **T**, detail::enable_if_t<std::is_arithmetic_v<*T*>> = 0>
explicit **operator** *T*() const

Convert a Python integer into a C++ arithmetic type.

class **float_** : public *object*

This wrapper class represents Python `float` instances.

float_(*handle* h)

Performs an floating point cast within Python. This is equivalent to the Python expression `float(h)`.

explicit **float_**(double value)

Convert an C++ double value into a Python float objecct

explicit **operator double**() const

Convert a Python float object into a C++ double value

class **str** : public *object*

This wrapper class represents Python unicode `str` instances.

str(*handle* h)

Performs a string cast within Python. This is equivalent equivalent to the Python expression `str(h)`.

str(const char *s)

Convert a null-terminated C-style string in UTF-8 encoding into a Python string.

str(const char *s, size_t n)

Convert a C-style string in UTF-8 encoding of length `n` bytes into a Python string.

const char ***c_str**() const

Convert a Python string into a null-terminated C-style string with UTF-8 encoding.

Note: The C string will be deleted when the *str* instance is garbage collected.

template<typename ...**Args**>
str **format**(*Args*&&... args)

C++ analog of the Python routine `str.format`. Can be called with positional and keyword arguments.

class **bytes** : public *object*

This wrapper class represents Python unicode `bytes` instances.

bytes(*handle* h)

Performs a cast within Python. This is equivalent to the Python expression `bytes(h)`.

bytes(const char *s)

Convert a null-terminated C-style string encoding into a Python `bytes` object.

bytes(const void *buf, size_t n)

Convert a byte buffer `buf` of length `n` bytes into a Python `bytes` object. The buffer can contain embedded null bytes.

const char ***c_str**() const

Convert a Python bytes object into a null-terminated C-style string.

size_t **size**() const

Return the size in bytes.

const void ***data**() const

Convert a Python `bytes` object into a byte buffer of length *bytes::size()* bytes.

class **bytearray** : public *object*

This wrapper class represents Python `bytearray` instances.

bytearray()

Create an empty `bytearray`.

bytearray(*handle* h)

Performs a cast within Python. This is equivalent to the Python expression `bytearray(h)`.

bytearray(const void *buf, size_t n)

Convert a byte buffer `buf` of length `n` bytes into a Python `bytearray` object. The buffer can contain embedded null bytes.

const char ***c_str**() const

Convert a Python `bytearray` object into a null-terminated C-style string.

size_t **size**() const

Return the size in bytes.

void ***data**()

Convert a Python `bytearray` object into a byte buffer of length *bytearray::size()* bytes.

const void ***data**() const

Convert a Python `bytearray` object into a byte buffer of length *bytearray::size()* bytes.

void **resize**(size_t n)

Resize the internal buffer of a Python `bytearray` object to `n`. Any space added by this method, which calls `PyByteArray_Resize`, will not be initialized and may contain random data.

class **type_object** : public *object*

Wrapper class representing Python `type` instances.

class **sequence** : public *object*

Wrapper class representing arbitrary Python sequence types.

class **mapping** : public *object*

Wrapper class representing arbitrary Python mapping types.

template<typename **T**>
bool **contains**(*T* &&key) const

Check whether the map contains a particular key. When *T* does not already represent a wrapped Python object, the function performs a cast.

list **keys**() const

Return a list containing all of the map's keys.

list **values**() const

Return a list containing all of the map's values.

list **items**() const

Return a list containing all of the map's items as `(key, value)` pairs.

class **iterator** : public *object*

Wrapper class representing a Python iterator.

iterator &**operator++**()

Advance to the next element (pre-increment form).

iterator &**operator++**(int)

Advance to the next element (post-increment form).

handle **operator***() const

Return the item at the current position.

handle **operator->**() const

Convenience routine for pointer-style access.

friend bool **operator==**(const *iterator* &a, const *iterator* &b);

Iterator equality comparison operator.

friend bool **operator!=**(const *iterator* &a, const *iterator* &b);

Iterator inequality comparison operator.

class **iterable** : public *object*

Wrapper class representing an object that can be iterated upon (in the sense that calling *iter()* is valid).

class **slice** : public *object*

Wrapper class representing a Python slice object.

slice(*handle* start, *handle* stop, *handle* step)

Create the slice object given by `slice(start, stop, step)` in Python.

template<typename **T**, detail::enable_if_t<std::is_arithmetic_v<*T*>> = 0>
slice(*T* stop)

Create the slice object `slice(stop)`, where *stop* is represented by a C++ integer type.

template<typename **T**, detail::enable_if_t<std::is_arithmetic_v<*T*>> = 0>
slice(*T* start, *T* stop)

Create the slice object `slice(start, stop)`, where *start* and *stop* are represented by a C++ integer type.

template<typename **T**, detail::enable_if_t<std::is_arithmetic_v<*T*>> = 0>

slice(*T* start, *T* stop, *T* step)

Create the slice object `slice(start, stop, step)`, where *start*, *stop*, and *step* are represented by a C++ integer type.

detail::tuple<Py_ssize_t, Py_ssize_t, Py_ssize_t, size_t> **compute**(size_t size) const

Adjust the slice to the *size* value of a given container. Returns a tuple containing `(start, stop, step, slice_length)`.

class **ellipsis** : public *object*

Wrapper class representing a Python ellipsis (`...`) object.

ellipsis()

Create a wrapper referencing the unique Python `Ellipsis` object.

class **not_implemented** : public *object*

Wrapper class representing a Python `NotImplemented` object.

not_implemented()

Create a wrapper referencing the unique Python `NotImplemented` object.

class **callable** : public *object*

Wrapper class representing a callable Python object.

class **weakref** : public *object*

Wrapper class representing a Python weak reference object.

explicit **weakref**(*handle* obj, *handle* callback = {})

Construct a new weak reference that points to *obj*. If provided, Python will invoke the callable *callback* when *obj* expires.

class **args** : public *tuple*

Variable argument keyword list for use in function argument declarations.

class **kwargs** : public *dict*

Variable keyword argument keyword list for use in function argument declarations.

class **any** : public *object*

This wrapper class represents Python `typing.Any`-typed values. On the C++ end, this type is interchangeable with `object`. The only difference is the type signature when used in function arguments and return values.

24.8 Parameterized wrapper classes

template<typename **T**>
class **handle_t** : public *handle*

Wrapper class representing a handle to a subclass of the C++ type *T*. It can be used to bind functions that take the associated Python object in its wrapped form, while rejecting objects with a different type (i.e., it is more discerning than *handle*, which accepts *any* Python object).

```
// Bind the class A
class A { int value; };
nb::class_<A>(m, "A");

// Bind a function that takes a Python object representing a 'A' instance
m.def("process_a", [](nb::handle_t<A> h) {
   PyObject * a_py = h.ptr();   // PyObject* pointer to wrapper
   A &a_cpp = nb::cast<A &>(h); // Reference to C++ instance
});
```

template<typename **T**>
class **type_object_t** : public *type_object*

Wrapper class representing a Python type object that is a subtype of the C++ type *T*. It can be used to bind functions that only accept type objects satisfying this criterion (i.e., it is more discerning than *type_object*, which accepts *any* Python type object).

24.9 Error management

nanobind provides a range of functionality to convert C++ exceptions into equivalent Python exceptions and raise captured Python error state in C++. The *exception* class is also relevant in this context, but is listed in the reference section on *class binding*.

struct **error_scope**

RAII helper class that temporarily stashes any existing Python error status. This is important when running Python code in the context of an existing failure that must be processed (e.g., to generate an error message).

error_scope()

Stash the current error status (if any)

~error_scope()

Restore the stashed error status (if any)

struct **python_error** : public std::exception

Exception that represents a detected Python error status.

python_error()

This constructor may only be called when a Python error has occurred (`PyErr_Occurred()` must be `true`). It creates a C++ exception object that represents this error and clears the Python error status.

python_error(const *python_error*&)

Copy constructor

python_error(*python_error*&&) noexcept

Move constructor

const char ***what**() noexcept

Return a stringified version of the exception. nanobind internally normalizes the exception and generates a traceback that is included as part of this string. This can be a relatively costly operation and should only be used if all of this detail is actually needed.

bool **matches**(*handle* exc) noexcept

Checks whether the exception has the same type as *exc*.

The argument to this function is usually one of the Standard Exceptions.

void **restore**() noexcept

Restore the error status in Python and clear the *python_error* contents. This may only be called once, and you should not reraise the *python_error* in C++ afterward.

void **discard_as_unraisable**(*handle* context) noexcept

Pass the error to Python's `sys.unraisablehook()`, which prints a traceback to `sys.stderr` by default but may be overridden. Like *restore()*, this consumes the error and you should not reraise the exception in C++ afterward.

The *context* argument should be some object whose `repr()` helps identify the location of the error. The default `sys.unraisablehook()` prints a traceback that begins with the text `Exception ignored in:` followed by the result of `repr(context)`.

Example use case: handling a Python error that occurs in a C++ destructor where you cannot raise a C++ exception.

void **discard_as_unraisable**(const char *context) noexcept

Convenience wrapper around the above function, which takes a C-style string for the `context` argument.

handle **type**() const

Returns a handle to the exception type

handle **value**() const

Returns a handle to the exception value

object **traceback**() const

Returns a handle to the exception's traceback object

class **cast_error**

The function *cast()* raises this exception to indicate that a cast was unsuccessful.

cast_error()

Constructor

class **next_overload**

Raising this special exception from a bound function informs nanobind that the function overload detected incompatible inputs. nanobind will then try other overloads before reporting a `TypeError`.

This feature is useful when a multiple overloads of a function accept overlapping or identical input types (e.g. *object*) and must run code at runtime to select the right overload.

You should probably write a thorough docstring that explicitly mentions the expected inputs in this case, since the behavior won't be obvious from the auto-generated function signature. It can be frustrating when a function call fails with an error message stating that the provided arguments aren't compatible with any overload, when the associated error message suggests otherwise.

next_overload()

Constructor

class **builtin_exception** : public std::runtime_error

General-purpose class to propagate builtin Python exceptions from C++. A number of convenience functions (see below) instantiate it.

builtin_exception **stop_iteration**(const char *what = nullptr)

Convenience wrapper to create a *builtin_exception* C++ exception instance that nanobind will re-raise as a Python `StopIteration` exception when it crosses the C++ ↔ Python interface.

builtin_exception **index_error**(const char *what = nullptr)

Convenience wrapper to create a *builtin_exception* C++ exception instance that nanobind will re-raise as a Python `IndexError` exception when it crosses the C++ ↔ Python interface.

builtin_exception **key_error**(const char *what = nullptr)

Convenience wrapper to create a *builtin_exception* C++ exception instance that nanobind will re-raise as a Python `KeyError` exception when it crosses the C++ ↔ Python interface.

builtin_exception **value_error**(const char *what = nullptr)

Convenience wrapper to create a *builtin_exception* C++ exception instance that nanobind will re-raise as a Python `ValueError` exception when it crosses the C++ ↔ Python interface.

builtin_exception **type_error**(const char *what = nullptr)

Convenience wrapper to create a *builtin_exception* C++ exception instance that nanobind will re-raise as a Python `TypeError` exception when it crosses the C++ ↔ Python interface.

builtin_exception **buffer_error**(const char *what = nullptr)

Convenience wrapper to create a *builtin_exception* C++ exception instance that nanobind will re-raise as a Python `BufferError` exception when it crosses the C++ ↔ Python interface.

builtin_exception **import_error**(const char *what = nullptr)

Convenience wrapper to create a *builtin_exception* C++ exception instance that nanobind will re-raise as a Python `ImportError` exception when it crosses the C++ ↔ Python interface.

builtin_exception **attribute_error**(const char *what = nullptr)

Convenience wrapper to create a *builtin_exception* C++ exception instance that nanobind will re-raise as a Python `AttributeError` exception when it crosses the C++ ↔ Python interface.

void **register_exception_translator**(void (*exception_translator)(const std::exception_ptr&, void*), void *payload = nullptr)

Install an exception translator callback that will be invoked whenever nanobind's function call dispatcher catches a previously unknown C++ exception. This exception translator should follow a standard structure of re-throwing an exception, catching a specific type, and converting this into a Python error status upon "success".

Here is an example for a hypothetical `ZeroDivisionException`.

```
register_exception_translator(
    [](const std::exception_ptr &p, void * /*payload*/) {
        try {
            std::rethrow_exception(p);
        } catch (const ZeroDivisionException &e) {
            PyErr_SetString(PyExc_ZeroDivisionError, e.what());
        }
    }, nullptr /*payload*/);
```

Generally, you will want to use the more convenient exception binding interface provided by *exception* class. This function provides an escape hatch for more specialized use cases.

void **chain_error**(*handle* type, const char *fmt, ...) noexcept

Raise a Python error of type `type` using the format string `fmt` interpreted by `PyErr_FormatV`.

If a Python error state was already set prior to calling this method, then the new error is *chained* on top of the existing one. Otherwise, the function creates a new error without initializing its `__cause__` field.

void **raise_from**(*python_error* &e, *handle* type, const char *fmt, ...)

Convenience wrapper around *chain_error*. It takes an existing Python error (e.g. caught in a `catch` block) and creates an additional Python exception with the current error as cause. It then re-raises *python_error*. The argument `fmt` is a `printf`-style format string interpreted by `PyErr_FormatV`.

Usage of this function is explained in the documentation section on *exception chaining*.

void **raise**(const char *fmt, ...)

This function takes a `printf`-style format string with arguments and then raises a `std::runtime_error` with the formatted string. The function has no dependence on Python, and nanobind merely includes it for convenience.

void **raise_type_error**(const char *fmt, ...)

This function is analogous to *raise()*, except that it raises a *builtin_exception* that will convert into a Python `TypeError` when crossing the language interface.

void **raise_python_error**()

This function should only be called if a Python error status was set by a prior operation, which should now be raised as a C++ exception. The function is analogous to the statement `throw python_error();` but compiles into more compact code.

24.10 Casting

template<typename **T**, typename **Derived**>
T **cast**(const detail::*api*<*Derived*> &value, bool convert = true)

Convert the Python object `value` (typically a `handle` or a `object` subclass) into a C++ object of type `T`.

When the `convert` argument is set to `true` (the default), the implementation may also attempt *implicit conversions* to perform the cast.

The function raises a `cast_error` when the conversion fails. See `try_cast()` for an alternative that never raises.

template<typename **T**, typename **Derived**>
bool **try_cast**(const detail::*api*<*Derived*> &value, *T* &out, bool convert = true) noexcept

Convert the Python object `value` (typically a `handle` or a `object` subclass) into a C++ object of type `T`, and store it in the output parameter `out`.

When the `convert` argument is set to `true` (the default), the implementation may also attempt *implicit conversions* to perform the cast.

The function returns `false` when the conversion fails. In this case, the `out` parameter is left untouched. See `cast()` for an alternative that instead raises an exception in this case.

template<typename **T**>
object **cast**(*T* &&value, *rv_policy* policy = *rv_policy*::*automatic_reference*)

Convert the C++ object `value` into a Python object. The return value policy `policy` is used to handle ownership-related questions when a new Python object must be created.

The function raises a `cast_error` when the conversion fails.

template<typename **T**>
object **cast**(*T* &&value, *rv_policy* policy, *handle* parent)

Convert the C++ object `value` into a Python object. The return value policy `policy` is used to handle ownership-related questions when a new Python object must be created. A valid `parent` object is required when specifying a `reference_internal` return value policy.

The function raises a `cast_error` when the conversion fails.

template<typename **T**>
object **find**(const *T* &value) noexcept

Return the Python object associated with the C++ instance `value`. When no such object can be found, the function it returns an invalid object (`detail::api::is_valid()` is `false`).

template<*rv_policy* **policy** = *rv_policy*::*automatic*, typename ...**Args**>
tuple **make_tuple**(*Args*&&... args)

Create a Python tuple from a sequence of C++ objects `args...`. The return value policy `policy` is used to handle ownership-related questions when a new Python objects must be created.

The function raises a `cast_error` when the conversion fails.

24.11 Common binding annotations

The following annotations can be specified in both function and class bindings.

struct **scope**

scope(*handle* value)

Captures the Python scope (e.g., a `module_` or `type_object`) in which the function or class should be registered.

24.12 Function binding annotations

The following annotations can be specified using the variable-length `Extra` parameter of *module_::def()*, *class_::def()*, *cpp_function()*, etc.

struct **name**

name(const char *value)

Specify this annotation to override the name of the function.

nanobind will internally copy the string when creating a function binding, hence dynamically generated arguments with a limited lifetime are legal.

struct **arg**

Function argument annotation to enable keyword-based calling, default arguments, passing `None`, and implicit conversion hints. Note that when a function argument should be annotated, you *must* specify annotations for all arguments of that function.

Example use:

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

It is usually convenient to add the following `using` declaration to your binding code.

```
using namespace nb::literals;
```

In this case, the argument annotations can be shortened:

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

explicit **arg**(const char *name = nullptr)

Create a function argument annotation. The name is optional.

template<typename **T**>
arg_v **operator=**(*T* &&value) const

Return an argument annotation that is like this one but also assigns a default value to the argument. The default will be converted into a Python object immediately, so its bindings must have already been defined.

arg &**none**(bool value = true)

Set a flag noting that the function argument accepts `None`. Can only be used for python wrapper types (e.g. *handle*, *int_*) and types that have been bound using *class_*. You cannot use this to implement functions that accept null pointers to builtin C++ types like `int *i = nullptr`.

arg &**noconvert**(bool value = true)

Set a flag noting that implicit conversion should never be performed for this function argument.

arg &**sig**(const char *sig)

Override the signature of the default argument value. This is useful when the argument value is unusually complex so that the default method to explain it in docstrings and stubs (`str(value)`) does not produce acceptable output.

arg_locked **lock**()

Return an argument annotation that is like this one but also requests that this argument be locked when dispatching a function call in free-threaded Python extensions. It does nothing in regular GIL-protected extensions.

struct **is_method**

Indicate that the bound function is a method.

struct **is_operator**

Indicate that the bound operator represents a special double underscore method (`__add__`, `__radd__`, etc.) that implements an arithmetic operation.

When a bound functions with this annotation is called with incompatible arguments, it will return `NotImplemented` rather than raising a `TypeError`.

struct **is_implicit**

Indicate that the bound constructor can be used to perform implicit conversions.

struct **lock_self**

Indicate that the implicit `self` argument of a method should be locked when dispatching a call in a free-threaded extension. This annotation does nothing in regular GIL-protected extensions.

template<typename ...**Ts**>
struct **call_guard**

Invoke the call guard(s) *Ts* when the bound function executes. The RAII helper *gil_scoped_release* is often combined with this feature.

template<size_t **Nurse**, size_t **Patient**>
struct **keep_alive**

Following evaluation of the bound function, keep the object referenced by index `Patient` alive *as long as* the object with index `Nurse` exists. This uses the following indexing convention:

- Index `0` refers to the return value of methods. It should not be used in constructors or functions that do not return a result.
- Index `1` refers to the first argument. In methods and constructors, index `1` refers to the implicit `this` pointer, while regular arguments begin at index `2`.

The annotation has the following runtime characteristics:

- It does nothing when the nurse or patient object are `None`.
- It raises an exception when the nurse object is neither weak-referenceable nor an instance of a binding created via *nb::class_<..>*.

Two additional caveats regarding *keep_alive* are noteworthy:

- It *usually* doesn't make sense to specify a `Nurse` or `Patient` for an argument or return value handled by a *type caster* (e.g., a STL vector handled via the include directive `#include <nanobind/stl/vector.h>`). That's because type casters copy-convert the Python object into an equivalent C++ object, whose lifetime is decoupled from the original Python object. However, the *keep_alive* annotation *only* affects the lifetime of Python objects *and not their C++ copy*.
- Dispatching a Python → C++ function call may require the *implicit conversion* of function arguments. In this case, the objects passed to the C++ function differ from the originally specified arguments. The `Nurse` and `Patient` annotation always refer to the *final* object following implicit conversion.

struct **sig**

sig(const char *value)

This is *both* a class and a function binding annotation.

1. When used in functions bindings, it provides complete control over the function's type signature by replacing the automatically generated version with `value`. You can use it to add or change arguments and return values, tweak how default values are rendered, and add custom decorators.

   Here is an example:

```
nb::def("function_name", &function_name,
        nb::sig(
            "@decorator(decorator_args..)\n"
            "def function_name(arg_1: type_1 = def_1, ...) -> ret"
        ));
```

2. When used in class bindings, the annotation enables complete control over how the class is rendered by nanobind's `stubgen` program. You can use it add decorators, specify `typing.TypeVar`-parameterized base classes, metaclasses, etc.

   Here is an example:

```
nb::class_<Class>(m, "Class",
                  nb::sig(
                      "@decorator(decorator_args..)\n"
                      "class Class(Base1[T], Base2, meta=Meta)"
                  ));
```

   Deviating significantly from the nanobind-generated signature likely means that the class or function declaration is a *lie*, but such lies can be useful to type-check complex binding projects.

   Specifying decorators isn't required—the above are just examples to show that this is possible.

   nanobind will internally copy the signature during function/type creation, hence dynamically generated strings with a limited lifetime are legal.

   The provided string should be valid Python signature, but *without* a trailing colon (`":"`) or trailing newline. Furthermore, nanobind analyzes the string and expects to find the name of the function or class on the *last line* between the `"def"` / `"class"` prefix and the opening parenthesis.

   For function bindings, this name must match the specified function name in `.def("name", ..)`-style binding declarations, and for class bindings, the specified name must match the `name` argument of *nb::class_*.

enum class **rv_policy**

A return value policy determines the question of *ownership* when a bound function returns a previously unknown C++ instance that must now be converted into a Python object.

Return value policies apply to functions that return values handled using *class bindings*, which means that their Python equivalent was registered using *class_<...>*. They are ignored in most other cases. One exception are STL types handled using *type casters* (e.g. `std::vector<T>`), which contain a nested type `T` handled using class bindings. In this case, the return value policy also applies recursively.

A return value policy is unnecessary when the type itself clarifies ownership (e.g., `std::unique_ptr<T>`, `std::shared_ptr<T>`, a type with *intrusive reference counting*).

The following policies are available (where *automatic* is the default). Please refer to the *return value policy section* of the main documentation, which clarifies the list below using concrete examples.

enumerator **take_ownership**

Create a Python object that wraps the existing C++ instance and takes full ownership of it. No copies are made. Python will call the C++ destructor and `delete` operator when the Python wrapper is garbage collected at some later point. The C++ side *must* relinquish ownership and is not allowed to destruct the instance, or undefined behavior will ensue.

enumerator **copy**

Copy-construct a new Python object from the C++ instance. The new copy will be owned by Python, while C++ retains ownership of the original.

enumerator **move**

Move-construct a new Python object from the C++ instance. The new object will be owned by Python, while C++ retains ownership of the original (whose contents were likely invalidated by the move operation).

enumerator **reference**

Create a Python object that wraps the existing C++ instance *without taking ownership* of it. No copies are made. Python will never call the destructor or `delete` operator, even when the Python wrapper is garbage collected.

enumerator **reference_internal**

A safe extension of the *reference* policy for methods that implement some form of attribute access. It creates a Python object that wraps the existing C++ instance *without taking ownership* of it. Additionally, it adjusts reference counts to keeps the method's implicit `self` argument alive until the newly created object has been garbage collected.

enumerator **none**

This is the most conservative policy: it simply refuses the cast unless the C++ instance already has a corresponding Python object, in which case the question of ownership becomes moot.

enumerator **automatic**

This is the default return value policy, which falls back to *take_ownership* when the return value is a pointer, *move* when it is a rvalue reference, and *copy* when it is a lvalue reference.

enumerator **automatic_reference**

This policy matches *automatic* but falls back to *reference* when the return value is a pointer.

struct **kw_only**

Indicate that all following function parameters are keyword-only. This may only be used if you supply an *arg* annotation for each parameters, because keyword-only parameters are useless if they don't have names. For example, if you write

```
int some_func(int one, const char* two);

m.def("some_func", &some_func,
      nb::arg("one"), nb::kw_only(), nb::arg("two"));
```

then in Python you can write `some_func(42, two="hi")`, or `some_func(one=42, two="hi")`, but not `some_func(42, "hi")`.

Just like in Python, any parameters appearing after variadic **args* are implicitly keyword-only. You don't need to include the *kw_only* annotation in this case, but if you do include it, it must be in the correct position: immediately after the *arg* annotation for the variadic **args* parameter.

template<typename **T**>
struct **for_getter**

When defining a property with a getter and a setter, you can use this to only pass a function binding attribute to the getter part. An example is shown below.

```
nb::class_<MyClass>(m, "MyClass")
  .def_prop_rw("value", &MyClass::value,
          nb::for_getter(nb::sig("def value(self, /) -> int")),
          nb::for_setter(nb::sig("def value(self, value: int, /) -> None")),
          nb::for_getter("docstring for getter"),
          nb::for_setter("docstring for setter"));
```

template<typename **T**>
struct **for_setter**

Analogous to *for_getter*, but for setters.

template<typename **Policy**>
struct **call_policy**

Request that custom logic be inserted around each call to the bound function, by calling `Policy::precall(args, nargs, cleanup)` before Python-to-C++ argument conversion, and `Policy::postcall(args, nargs, ret)` after C++-to-Python return value conversion.

If multiple call policy annotations are provided for the same function, then their precall and postcall hooks will both execute left-to-right according to the order in which the annotations were specified when binding the function.

The *nb::call_guard<T>()* annotation should be preferred over `call_policy` unless the wrapper logic depends on the function arguments or return value. If both annotations are combined, then *nb::call_guard<T>()* always executes on the "inside" (closest to the bound function, after argument conversions and before return value conversion) regardless of its position in the function annotations list.

Your `Policy` class must define two static member functions:

static void **precall**(PyObject **args, size_t nargs, detail::cleanup_list *cleanup);

A hook that will be invoked before calling the bound function. More precisely, it is called after any *argument locks* have been obtained, but before the Python arguments are converted to C++ objects for the function call.

This hook may access or modify the function arguments using the *args* array, which holds borrowed references in one-to-one correspondence with the C++ arguments of the bound function. If the bound function is a method, then `args[0]` is its *self* argument. *nargs* is the number of function arguments. It is actually passed as `std::integral_constant<size_t, N>()`, so you can match on that type if you want to do compile-time checks with it.

The *cleanup* list may be used as it is used in type casters, to cause some Python object references to be released at some point after the bound function completes. (If the bound function is part of an overload set, the cleanup list isn't released until all overloads have been tried.)

`precall()` may choose to throw a C++ exception. If it does, it will preempt execution of the bound function, and the exception will be treated as if the bound function had thrown it.

static void **postcall**(PyObject **args, size_t nargs, *handle* ret);

A hook that will be invoked after calling the bound function and converting its return value to a Python object, but only if the bound function returned normally.

args stores the Python object arguments, with the same semantics as in `precall()`, except that arguments that participated in implicit conversions will have had their `args[i]` pointer updated to reflect the new Python object that the implicit conversion produced. *nargs* is the number of arguments, passed as a `std::integral_constant` in the same way as for `precall()`.

ret is the bound function's return value. If the bound function returned normally but its C++ return value could not be converted to a Python object, then `postcall()` will execute with *ret* set to null, and the Python error indicator might or might not be set to explain why.

If the bound function did not return normally – either because its Python object arguments couldn't be converted to the appropriate C++ types, or because the C++ function threw an exception – then `postcall()` **will not execute**. If you need some cleanup logic to run even in such cases, your `precall()` can add a capsule object to the cleanup list; its destructor will run eventually, but with no promises as to when. A *nb::call_guard* might be a better choice.

`postcall()` may choose to throw a C++ exception. If it does, the result of the wrapped function will be destroyed, and the exception will be raised in its place, as if the bound function had thrown it just before returning.

Here is an example policy to demonstrate. `nb::call_policy<returns_references_to<I>>()` behaves like *nb::keep_alive<0, I>()*, except that the return value is a treated as a list of objects rather than a single one.

```
template <size_t I>
struct returns_references_to {
    static void precall(PyObject **, size_t, nb::detail::cleanup_list *) {}

    template <size_t N>
    static void postcall(PyObject **args,
                         std::integral_constant<size_t, N>,
                         nb::handle ret) {
        static_assert(I > 0 && I < N,
                      "I in returns_references_to<I> must be in the "
```

(continues on next page)

(continued from previous page)

```
                    "range [1, number of C++ function arguments]");
        if (!nb::isinstance<nb::sequence>(ret)) {
            throw std::runtime_error("return value should be a sequence");
        }
        for (nb::handle nurse : ret) {
            nb::detail::keep_alive(nurse.ptr(), args[I]);
        }
    }
};
```

For a more complex example (binding an object that uses trivially-copyable callbacks), see `tests/test_callbacks.cpp` in the nanobind source distribution.

24.13 Class binding annotations

The following annotations can be specified using the variable-length `Extra` parameter of the constructor *class_::class_()*.

Besides the below options, also refer to the *sig* which is usable in both function and class bindings. It can be used to override class declarations in generated *stubs*,

struct **is_final**

Indicate that a type cannot be subclassed.

struct **dynamic_attr**

Indicate that instances of a type require a Python dictionary to support the dynamic addition of attributes.

struct **is_weak_referenceable**

Indicate that instances of a type require a weak reference list so that they can be referenced by the Python `weakref.*` types.

struct **is_generic**

If present, nanobind will add a `__class_getitem__` function to the newly created type that permits constructing *parameterized* versions (e.g., `MyType[int]`). The implementation of this function is equivalent to

```
def __class_getitem__(cls, value):
    import types
    return types.GenericAlias(cls, value)
```

See the section on *creating generic types* for an example.

This feature is only supported on Python 3.9+. Nanobind will ignore the attribute in Python 3.8 builds.

template<typename **T**>
struct **supplement**

Indicate that `sizeof(T)` bytes of memory should be set aside to store supplemental data in the type object. See *Supplemental type data* for more information.

struct **type_slots**

type_slots(PyType_Slot *value)

nanobind uses the `PyType_FromSpec` Python C API interface to construct types. In certain advanced use cases, it may be helpful to append additional type slots during type construction. This class binding annotation can be used to accomplish this. The provided list should be followed by a zero-initialized `PyType_Slot` element. See *Customizing type creation* for more information about this feature.

template<typename **T**>

struct **intrusive_ptr**

nanobind provides a custom interface for intrusive reference-counted C++ types that nicely integrate with Python reference counting. See the *separate section* on this topic. This annotation marks a type as compatible with this interface.

intrusive_ptr(void (*set_self_py)(*T**, PyObject*) noexcept)

Declares a callback that will be invoked when a C++ instance is first cast into a Python object.

24.14 Enum binding annotations

The following annotations can be specified using the variable-length `Extra` parameter of the constructor *enum_::enum_()*.

struct **is_arithmetic**

Indicate that the enumeration supports arithmetic operations. This enables both unary (`-`, `~`, `abs()`) and binary (`+`, `-`, `*`, `//`, `&`, `|`, `^`, `<<`, `>>`) operations with operands of either enumeration or numeric types.

The result will be as if the operands were first converted to integers. (So `Shape(2) + Shape(1) == 3` and `Shape(2) * 1.5 == 3.0`.) It is unspecified whether operations on mixed enum types (such as `Shape.Circle + Color.Red`) are permissible.

Passing this annotation changes the Python enumeration parent class to either `enum.IntEnum` or `enum.IntFlag`, depending on whether or not the flag enumeration attribute is also specified (see *is_flag*).

struct **is_flag**

Indicate that the enumeration supports bit-wise operations. This enables the operators (`|`, `&`, `^`, and `~`) with two enumerators as operands.

The result has the same type as the operands, i.e., `Shape(2) | Shape(1)` will be equivalent to `Shape(3)`.

Passing this annotation changes the Python enumeration parent class to either `enum.IntFlag` or `enum.Flag`, depending on whether or not the enumeration is also marked to support arithmetic operations (see *is_arithmetic*).

24.15 Function binding

object **cpp_function**(Func &&f, const Extra&... extra)

Convert the function *f* into a Python callable. This function has a few overloads (not shown here) to separately deal with function/method pointers and lambda functions.

The variable length *extra* parameter can be used to pass a docstring and other *function binding annotations*.

24.16 Class binding

template<typename **T**, typename ...**Ts**>
class **class_** : public *object*

Binding helper class to expose a custom C++ type *T* (declared using either the `class` or `struct` keyword) in Python.

The variable length parameter *Ts* is optional and can be used to specify the base class of *T* and/or an alias needed to realize *trampoline classes*.

When the type `T` was previously already registered (either within the same extension or another extension), the `class_<..>` declaration is redundant. nanobind will print a warning message in this case:

```
RuntimeWarning: nanobind: type 'MyType' was already registered!
```

The `class_<..>` instance will subsequently wrap the original type object instead of creating a new one.

template<typename ...**Extra**>
class_(*handle* scope, const char *name, const *Extra*&... extra)

Bind the type *T* to the identifier *name* within the scope *scope*. The variable length *extra* parameter can be used to pass a docstring and other *class binding annotations*.

template<typename **Func**, typename ...**Extra**>
*class_* &**def**(const char *name, *Func* &&f, const *Extra*&... extra)

Bind the function *f* and assign it to the class member *name*. The variable length *extra* parameter can be used to pass a docstring and other *function binding annotations*.

This function has two overloads (listed just below) to handle constructor binding declarations.

Example:

```
struct A {
    void f() { /*...*/ }
};

nb::class_<A>(m, "A")
    .def(nb::init<>()) // Bind the default constructor
    .def("f", &A::f);  // Bind the method A::f
```

template<typename ...**Args**, typename ...**Extra**>
*class_* &**def**(*init*<*Args*...> arg, const *Extra*&... extra)

Bind a constructor. The variable length *extra* parameter can be used to pass a docstring and other *function binding annotations*.

template<typename **Arg**, typename ...**Extra**>
*class_* &**def**(*init_implicit*<*Arg*> arg, const *Extra*&... extra)

Bind a constructor that may be used for implicit type conversions. The constructor must take a single argument of an unspecified type *Arg*.

When nanobind later tries to dispatch a function call requiring an argument of type *T* while *Arg* was actually provided, it will run this constructor to perform the necessary conversion.

The variable length *extra* parameter can be used to pass a docstring and other *function binding annotations*.

This constructor generates more compact code than a separate call to *implicitly_convertible()*, but is otherwise equivalent.

template<typename **Func**, typename ...**Extra**>
*class_* &**def**(*new_*<*Func*> arg, const *Extra*&... extra)

Bind a C++ factory function as a Python object constructor (`__new__`). This is an advanced feature; prefer *nb::init<..>* where possible. See the discussion of *customizing object creation* for more details.

template<typename **C**, typename **D**, typename ...**Extra**>
*class_* &**def_rw**(const char *name, *D C*::* p, const *Extra*&... extra)

Bind the field *p* and assign it to the class member *name*. nanobind constructs a `property` object with *read-write* access (hence the `rw` suffix) to do so.

Every access from Python will read from or write to the C++ field while performing a suitable conversion (using *type casters*, *bindings*, or *wrappers*) as determined by its type.

The variable length *extra* parameter can be used to pass a docstring and other *function binding annotations* that are forwarded to the anonymous functions used to construct the property. Use the *nb::for_getter* and *nb::for_setter* to pass annotations specifically to the setter or getter part.

Example:

```
struct A { int value; };

nb::class_<A>(m, "A")
    .def_rw("value", &A::value); // Enable mutable access to the field␣
↪A::value
```

template<typename **C**, typename **D**, typename ...**Extra**>
*class_* &**def_ro**(const char *name, *D C*::* p, const *Extra*&... extra)

Bind the field *p* and assign it to the class member *name*. nanobind constructs a `property` object with *read only* access (hence the `ro` suffix) to do so.

Every access from Python will read the C++ field while performing a suitable conversion (using *type casters*, *bindings*, or *wrappers*) as determined by its type.

The variable length *extra* parameter can be used to pass a docstring and other *function binding annotations* that are forwarded to the anonymous functions used to construct the property.

Example:

```
struct A { int value; };

nb::class_<A>(m, "A")
    .def_ro("value", &A::value);  // Enable read-only access to the field␣
↪A::value
```

template<typename **Getter**, typename **Setter**, typename ...**Extra**>
*class_* &**def_prop_rw**(const char *name, *Getter* &&getter, *Setter* &&setter, const *Extra*&... extra)

Construct a *mutable* (hence the `rw` suffix) Python `property` and assign it to the class member *name*. Every read access will call the function `getter` with the *T* instance, and every write access will call the `setter` with the *T* instance and value to be assigned.

The variable length *extra* parameter can be used to pass a docstring and other *function binding annotations*. Use the *nb::for_getter* and *nb::for_setter* to pass annotations specifically to the setter or getter part.

Note that this function implicitly assigns the *rv_policy::reference_internal* return value policy to *getter* (as opposed to the usual *rv_policy::automatic*). Provide an explicit return value policy as part of the *extra* argument to override this.

Example: the example below uses *def_prop_rw()* to expose a C++ setter/getter pair as a more "Pythonic" property:

```
class A {
public:
    A(int value) : m_value(value) { }
    void set_value(int value) { m_value = value; }
    int value() const { return m_value; }
private:
    int m_value;
};

nb::class_<A>(m, "A")
    .def(nb::init<int>())
    .def_prop_rw("value",
        [](A &t) { return t.value() ; },
        [](A &t, int value) { t.set_value(value); });
```

template<typename **Getter**, typename ...**Extra**>

*class_* &**def_prop_ro**(const char *name, *Getter* &&getter, const *Extra*&... extra)

Construct a *read-only* (hence the `ro` suffix) Python `property` and assign it to the class member *name*. Every read access will call the function `getter` with the *T* instance.

The variable length *extra* parameter can be used to pass a docstring and other *function binding annotations*.

Note that this function implicitly assigns the *rv_policy::reference_internal* return value policy to *getter* (as opposed to the usual *rv_policy::automatic*). Provide an explicit return value policy as part of the *extra* argument to override this.

Example: the example below uses *def_prop_ro()* to expose a C++ getter as a more "Pythonic" property:

```
class A {
public:
    A(int value) : m_value(value) { }
    int value() const { return m_value; }
private:
    int m_value;
};

nb::class_<A>(m, "A")
    .def(nb::init<int>())
    .def_prop_ro("value",
        [](A &t) { return t.value() ; });
```

template<typename **Func**, typename ...**Extra**>
*class_* &**def_static**(const char *name, *Func* &&f, const *Extra*&... extra)

Bind the *static* function *f* and assign it to the class member *name*. The variable length *extra* parameter can be used to pass a docstring and other *function binding annotations*.

Example:

```
struct A {
    static void f() { /*...*/ }
};

nb::class_<A>(m, "A")
    .def_static("f", &A::f);  // Bind the static method A::f
```

template<typename **D**, typename ...**Extra**>
*class_* &**def_rw_static**(const char *name, *D* *p, const *Extra*&... extra)

Bind the *static* field *p* and assign it to the class member *name*. nanobind constructs a class `property` object with *read-write* access (hence the `rw` suffix) to do so.

Every access from Python will read from or write to the static C++ field while performing a suitable conversion (using *type casters*, *bindings*, or *wrappers*) as determined by its type.

The variable length *extra* parameter can be used to pass a docstring and other *function binding annotations* that are forwarded to the anonymous functions used to construct the property Use the *nb::for_getter* and *nb::for_setter* to pass annotations specifically to the setter or getter part.

Example:

```
struct A { inline static int value = 5; };

nb::class_<A>(m, "A")
    // Enable mutable access to the static field A::value
    .def_rw_static("value", &A::value);
```

template<typename **D**, typename ...**Extra**>
*class_* &**def_ro_static**(const char *name, *D* *p, const *Extra*&... extra)

Bind the *static* field *p* and assign it to the class member *name*. nanobind constructs a class `property` object with *read-only* access (hence the `ro` suffix) to do so.

Every access from Python will read the static C++ field while performing a suitable conversion (using *type casters*, *bindings*, or *wrappers*) as determined by its type.

The variable length *extra* parameter can be used to pass a docstring and other *function binding annotations* that are forwarded to the anonymous functions used to construct the property

Example:

```
struct A { inline static int value = 5; };

nb::class_<A>(m, "A")
    // Enable read-only access to the static field A::value
    .def_ro_static("value", &A::value);
```

template<typename **Getter**, typename **Setter**, typename ...**Extra**>
*class_* &**def_prop_rw_static**(const char *name, *Getter* &&getter, *Setter* &&setter, const *Extra*&... extra)

Construct a *mutable* (hence the `rw` suffix) Python `property` and assign it to the class member *name*. Every read access will call the function `getter` with *T*'s Python type object, and every write access will call the `setter` with *T*'s Python type object and value to be assigned.

The variable length *extra* parameter can be used to pass a docstring and other *function binding annotations*. Use the *nb::for_getter* and *nb::for_setter* to pass annotations specifically to the setter or getter part.

Note that this function implicitly assigns the *rv_policy::reference* return value policy to *getter* (as opposed to the usual *rv_policy::automatic*). Provide an explicit return value policy as part of the *extra* argument to override this.

Example: the example below uses *def_prop_rw_static()* to expose a static C++ setter/getter pair as a more "Pythonic" property:

```
class A {
public:
   static void set_value(int value) { s_value = value; }
   static int value() { return s_value; }
private:
   inline static int s_value = 5;
};

nb::class_<A>(m, "A")
    .def_prop_rw_static("value",
        [](nb::handle /*unused*/) { return A::value() ; },
        [](nb::handle /*unused*/, int value) { A::set_value(value); });
```

template<typename **Getter**, typename ...**Extra**>
*class_* &**def_prop_ro_static**(const char *name, *Getter* &&getter, const *Extra*&... extra)

Construct a *read-only* (hence the `ro` suffix) Python `property` and assign it to the class member *name*. Every read access will call the function `getter` with *T*'s Python type object.

The variable length *extra* parameter can be used to pass a docstring and other *function binding annotations*.

Note that this function implicitly assigns the *rv_policy::reference* return value policy to *getter* (as opposed to the usual *rv_policy::automatic*). Provide an explicit return value policy as part of the *extra* argument to override this.

Example: the example below uses *def_prop_ro_static()* to expose a static C++ getter as a more "Pythonic" property:

```
class A {
public:
   static int value() { return s_value; }
private:
   inline static int s_value = 5;
};

nb::class_<A>(m, "A")
    .def_prop_ro_static("value",
        [](nb::handle /*unused*/) { return A::value() ; });
```

template<detail::op_id **id**, detail::op_type **ot**, typename **L**, typename **R**, typename ...**Extra**>
*class_* &**def**(const detail::op_<*id*, *ot*, *L*, *R*> &op, const *Extra*&... extra)

This interface provides convenient syntax sugar to replace relatively lengthy method bindings with shorter operator bindings. To use it, you will need an extra include directive:

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

Below is an example type with three arithmetic operators in C++ (unary negation and 2 binary subtraction overloads) along with corresponding bindings.

Example:

```
struct A {
   float value;

   A operator-() const { return { -value }; }
   A operator-(const A &o) const { return { value - o.value }; }
   A operator-(float o) const { return { value - o }; }
};

nb::class_<A>(m, "A")
    .def(nb::init<float>())
    .def(-nb::self)
    .def(nb::self - nb::self)
    .def(nb::self - float());
```

Bind an arithmetic or comparison operator expressed in short-hand form (e.g., `.def(nb::self + nb::self)`).

template<detail::op_id **id**, detail::op_type **ot**, typename **L**, typename **R**, typename ...**Extra**>
*class_* &**def_cast**(const detail::op_<*id*, *ot*, *L*, *R*> &op, const *Extra*&... extra)

Like the above `.def()` variant, but furthermore cast the result of the operation back to *T*.

template<typename **T**>
class **enum_** : public *class_*<*T*>

Class binding helper for scoped and unscoped C++ enumerations.

template<typename ...**Extra**>
NB_INLINE **enum_**(*handle* scope, const char *name, const *Extra*&... extra)

Bind the enumeration of type *T* to the identifier *name* within the scope *scope*. The variable length *extra* parameter can be used to pass a docstring and other *enum binding annotations* (currently, only *is_arithmetic* is supported).

*enum_* &**value**(const char *name, *T* value, const char *doc = nullptr)

Add the entry *value()* to the enumeration using the identifier *name*, potentially with a docstring provided via *doc* (optional).

*enum_* &**export_values**()

Export all entries of the enumeration into the parent scope.

template<typename **T**>
class **exception** : public *object*

Class binding helper for declaring new Python exception types

exception(*handle* scope, const char *name, *handle* base = PyExc_Exception)

Create a new exception type identified by *name* that derives from *base*, and install it in *scope*. The constructor also calls *register_exception_translator()* to register a new exception translator that converts caught C++ exceptions of type *T* into the newly created Python equivalent.

template<typename ...**Args**>
struct **init**

nanobind uses this simple helper class to capture the signature of a constructor. It is only meant to be used in binding declarations done via *class_::def()*.

Sometimes, it is necessary to bind constructors that don't exist in the underlying C++ type (meaning that they are specific to the Python bindings). Because *init* only works for existing C++ constructors, this requires a manual workaround noting that

```
nb::class_<MyType>(m, "MyType")
    .def(nb::init<const char*, int>());
```

is syntax sugar for the following lower-level implementation using "placement new":

```
nb::class_<MyType>(m, "MyType")
    .def("__init__",
         [](MyType* t, const char* arg0, int arg1) {
             new (t) MyType(arg0, arg1);
         });
```

The provided lambda function will be called with a pointer to uninitialized memory that has already been allocated (this memory region is co-located with the Python object for reasons of efficiency). The lambda function can then either run an in-place constructor and return normally (in which case the instance is assumed to be correctly constructed) or fail by raising an exception.

template<typename **Arg**>
struct **init_implicit**

See *init* for detail on binding constructors. The main difference between *init* and *init_implicit* is that the latter only supports constructors taking a single argument *Arg*, and that it marks the constructor as usable for implicit conversions from *Arg*.

Sometimes, it is necessary to bind implicit conversion-capable constructors that don't exist in the underlying C++ type (meaning that they are specific to the Python bindings). This can be done manually noting that

```
nb::class_<MyType>(m, "MyType")
    .def(nb::init_implicit<const char*>());
```

can be replaced by the lower-level code

```
nb::class_<MyType>(m, "MyType")
    .def("__init__",
         [](MyType* t, const char* arg0) {
             new (t) MyType(arg0);
         });

nb::implicitly_convertible<const char*, MyType>();
```

template<typename **Func**>

struct **new_**

This is a small helper class that indicates to *class_::def()* that a particular lambda or static method provides a Python object constructor (__new__) for the class being bound. Normally, you would use *init* instead if possible, in order to cooperate with nanobind's usual object creation process. Using *new_* replaces that process entirely. This is principally useful when some C++ type of interest can only provide pointers to its instances, rather than allowing them to be constructed directly.

Like *init*, the only use of a *new_* object is as an argument to *class_::def()*.

Example use:

```
class MyType {
private:
    MyType();
public:
    static std::shared_ptr<MyType> create();
    int value = 0;
};

nb::class_<MyType>(m, "MyType")
    .def(nb::new_(&MyType::create));
```

Given this example code, writing `MyType()` in Python would produce a Python object wrapping the result of `MyType::create()` in C++. If multiple calls to `create()` return pointers to the same C++ object, these will turn into references to the same Python object as well.

See the discussion of *customizing Python object creation* for more information.

24.17 GIL Management

These two RAII helper classes acquire and release the *Global Interpreter Lock* (GIL) in a given scope. The *gil_scoped_release* helper is often combined with the *call_guard*, as in

```
m.def("expensive", &expensive, nb::call_guard<nb::gil_scoped_release>());
```

This releases the interpreter lock while `expensive` is running, which permits running it in parallel from multiple Python threads.

struct **gil_scoped_acquire**

gil_scoped_acquire()

Acquire the GIL

~gil_scoped_acquire()

Release the GIL

struct **gil_scoped_release**

gil_scoped_release()

Release the GIL (**must** be currently held)

In *free-threaded extensions*, this operation also temporarily releases all *argument locks* held by the current thread.

~gil_scoped_release()

Reacquire the GIL

24.18 Free-threading

Nanobind provides abstractions to implement *additional* locking that is needed to ensure the correctness of free-threaded Python extensions.

struct **ft_mutex**

Object-oriented wrapper representing a PyMutex. It can be slightly more efficient than OS/language-provided primitives (e.g., `std::thread`, `pthread_mutex_t`) and should generally be preferred when adding critical sections to Python bindings.

In Python builds *without* free-threading, this class does nothing. It has no attributes and the *lock()* and *unlock()* functions return immediately.

ft_mutex()

Create a new (unlocked) mutex.

void **lock**()

Acquire the mutex.

void **unlock**()

Release the mutex.

struct **ft_lock_guard**

This class provides a RAII lock guard analogous to `std::lock_guard` and `std::unique_lock`.

ft_lock_guard(*ft_mutex* &mutex)

Call *mutex.lock()* (no-op in non-free-threaded builds).

~ft_lock_guard()

Call *mutex.unlock()* (no-op in non-free-threaded builds).

struct **ft_object_guard**

This class provides a RAII guard that locks a single Python object within a local scope (in contrast to *ft_lock_guard*, which locks a mutex).

It is a thin wrapper around the Python critical section API. Please refer to the Python documentation for details on the semantics of this relaxed form of critical section (in particular, Python critical sections may release previously held locks).

In Python builds *without* free-threading, this class does nothing—the constructor and destructor return immediately.

ft_object_guard(*handle* h)

Lock the object `h` (no-op in non-free-threaded builds)

~ft_object_guard()

Unlock the object `h` (no-op in non-free-threaded builds)

struct **ft_object2_guard**

This class provides a RAII guard that locks *two* Python object within a local scope (in contrast to *ft_lock_guard*, which locks a mutex).

It is a thin wrapper around the Python critical section API. Please refer to the Python documentation for details on the semantics of this relaxed form of critical section (in particular, Python critical sections may release previously held locks).

In Python builds *without* free-threading, this class does nothing—the constructor and destructor return immediately.

ft_object2_guard(*handle* h1, *handle* h2)

Lock the objects `h1` and `h2` (no-op in non-free-threaded builds)

~ft_object2_guard()

Unlock the objects `h1` and `h2` (no-op in non-free-threaded builds)

24.19 Low-level type and instance access

nanobind exposes a low-level interface to provide fine-grained control over the sequence of steps that instantiates a Python object wrapping a C++ instance. A thorough explanation of these features is provided in a *separate section*.

24.19.1 Type objects

bool **type_check**(*handle* h)

Returns `true` if `h` is a type that was previously bound via *class_*.

size_t **type_size**(*handle* h)

Assuming that *h* represents a bound type (see *type_check()*), return its size in bytes.

size_t **type_align**(*handle* h)

Assuming that *h* represents a bound type (see *type_check()*), return its alignment in bytes.

const std::type_info &**type_info**(*handle* h)

Assuming that *h* represents a bound type (see *type_check()*), return its C++ RTTI record.

template<typename **T**>
T &**type_supplement**(*handle* h)

Return a reference to supplemental data stashed in a type object. The type `T` must exactly match the type specified in the *nb::supplement<T>* annotation used when creating the type; no type check is performed, and invalid supplement accesses may crash the interpreter. Also refer to *nb::supplement<T>*.

str **type_name**(*handle* h)

Return the full (module-qualified) name of a type object as a Python string.

void ***type_get_slot**(*handle* h, int slot_id)

On Python 3.10+, this function is a simple wrapper around the Python C API function `PyType_GetSlot` that provides stable API-compatible access to type object members. On Python 3.9 and earlier, the official function did not work on non-heap types. The nanobind version consistently works on heap and non-heap types across Python versions.

24.19.2 Instances

The documentation below refers to two per-instance flags with the following meaning:

- *ready*: is the instance fully constructed? nanobind will not permit passing the instance to a bound C++ function when this flag is unset.
- *destruct*: should nanobind call the C++ destructor when the instance is garbage-collected?

bool **inst_check**(*handle* h)

Returns `true` if *h* represents an instance of a type that was previously bound via *class_*.

template<typename **T**>
T ***inst_ptr**(*handle* h)

Assuming that *h* represents an instance of a type that was previously bound via *class_*, return a pointer to the underlying C++ instance.

The function *does not check* that *h* actually contains an instance with C++ type *T*.

object **inst_alloc**(*handle* h)

Assuming that *h* represents a type object that was previously created via *class_* (see *type_check()*), allocate an unitialized object of type *h* and return it. The *ready* and *destruct* flags of the returned instance are both set to `false`.

object **inst_alloc_zero**(*handle* h)

Assuming that *h* represents a type object that was previously created via *class_* (see *type_check()*), allocate a zero-initialized object of type *h* and return it. The *ready* and *destruct* flags of the returned instance are both set to `true`.

This operation is equivalent to calling *inst_alloc()* followed by *inst_zero()*.

object **inst_reference**(*handle* h, void *p, *handle* parent = *handle*())

Assuming that *h* represents a type object that was previously created via *class_* (see *type_check()*) create an object of type *h* that wraps an existing C++ instance *p*.

The *ready* and *destruct* flags of the returned instance are respectively set to `true` and `false`.

This is analogous to casting a C++ object with return value policy *rv_policy::reference*.

If a *parent* object is specified, the instance keeps this parent alive while the newly created object exists. This is analogous to casting a C++ object with return value policy *rv_policy::reference_internal*.

object **inst_take_ownership**(*handle* h, void *p)

Assuming that *h* represents a type object that was previously created via *class_* (see *type_check()*) create an object of type *h* that wraps an existing C++ instance *p*.

The *ready* and *destruct* flags of the returned instance are both set to `true`.

This is analogous to casting a C++ object with return value policy *rv_policy::take_ownership*.

void **inst_zero**(*handle* h)

Zero-initialize the contents of *h*. Sets the *ready* and *destruct* flags to `true`.

bool **inst_ready**(*handle* h)

Query the *ready* flag of the instance *h*.

std::pair<bool, bool> **inst_state**(*handle* h)

Separately query the *ready* and *destruct* flags of the instance *h*.

void **inst_mark_ready**(*handle* h)

Simultaneously set the *ready* and *destruct* flags of the instance *h* to `true`.

void **inst_set_state**(*handle* h, bool ready, bool destruct)

Separately set the *ready* and *destruct* flags of the instance *h*.

void **inst_destruct**(*handle* h)

Destruct the instance *h*. This entails calling the C++ destructor if the *destruct* flag is set and then setting the *ready* and *destruct* fields to `false`.

void **inst_copy**(*handle* dst, *handle* src)

Copy-construct the contents of *src* into *dst* and set the *ready* and *destruct* flags of *dst* to `true`.

dst should be an uninitialized instance of the same type. Note that setting the *destruct* flag may be problematic if *dst* is an offset into an existing object created using *inst_reference()* (the destructor will be called multiple times in this case). If so, you must use *inst_set_state()* to disable the flag following the call to *inst_copy()*.

New in nanobind v2.0.0: The function is a no-op when `src` and `dst` refer to the same object.

void **inst_move**(*handle* dst, *handle* src)

Analogous to *inst_copy()*, except that the move constructor is used instead of the copy constructor.

void **inst_replace_copy**(*handle* dst, *handle* src)

Destruct the contents of *dst* (even if the *destruct* flag is `false`). Next, copy-construct the contents of *src* into *dst* and set the *ready* flag of `dst`. The value of the *destruct* flag is subsequently set to its value prior to the call.

This operation is useful to replace the contents of one instance with that of another regardless of whether *dst* has been created using *inst_alloc()*, *inst_reference()*, or *inst_take_ownership()*.

New in nanobind v2.0.0: The function is a no-op when `src` and `dst` refer to the same object.

void **inst_replace_move**(*handle* dst, *handle* src)

Analogous to *inst_replace_copy()*, except that the move constructor is used instead of the copy constructor.

str **inst_name**(*handle* h)

Return the full (module-qualified) name of the instance's type object as a Python string.

24.20 Global flags

bool **leak_warnings**() noexcept

Returns whether nanobind warns if any nanobind instances, types, or functions are still alive when the Python interpreter shuts down.

bool **implicit_cast_warnings**() noexcept

Returns whether nanobind warns if an implicit conversion was not successful.

void **set_leak_warnings**(bool value) noexcept

By default, nanobind loudly complains when any nanobind instances, types, or functions are still alive when the Python interpreter shuts down. Call this function to disable or re-enable leak warnings.

void **set_implicit_cast_warnings**(bool value) noexcept

By default, nanobind loudly complains when it attempts to perform an implicit conversion, and when that conversion is not successful. Call this function to disable or re-enable the warnings.

inline bool **is_alive**() noexcept

The function returns `true` when nanobind is initialized and ready for use. It returns `false` when the Python interpreter has shut down, causing the destruction various nanobind-internal data structures. Having access to this liveness status can be useful to avoid operations that are illegal in the latter context.

24.21 Miscellaneous

str **repr**(*handle* h)

Return a stringified version of the provided Python object. Equivalent to `repr(h)` in Python.

void **print**(*handle* value, *handle* end = *handle*(), *handle* file = *handle*())

Invoke the Python `print()` function to print the object *value*. If desired, a line ending *end* and file handle *file* can be specified.

void **print**(const char *str, *handle* end = *handle*(), *handle* file = *handle*())

Invoke the Python `print()` function to print the null-terminated C-style string *str* that is encoded using UTF-8 encoding. If desired, a line ending *end* and file handle *file* can be specified.

iterator **iter**(*handle* h)

Equivalent to `iter(h)` in Python.

object **none**()

Return an object representing the value `None`.

dict **builtins**()

Return the `__builtins__` dictionary.

dict **globals**()

Return the `globals()` dictionary.

Py_hash_t **hash**(*handle* h)

Hash the given argument like `hash()` in pure Python. The type of the return value (`Py_hash_t`) is an implementation-specific signed integer type.

template<typename **Source**, typename **Target**>
void **implicitly_convertible**()

Indicate that the type *Source* is implicitly convertible into *Target* (which must refer to a type that was previously bound via *class_*).

Note: the *init_implicit* interface generates more compact code and should be preferred, i.e., use

```
nb::class_<Target>(m, "Target")
    .def(nb::init_implicit<Source>());
```

instead of

```
nb::class_<Target>(m, "Target")
    .def(nb::init<Source>());

nb::implicitly_convertible<Source, Target>();
```

The function is provided for reasons of compatibility with pybind11, and as an escape hatch to enable use cases where *init_implicit* is not available (e.g., for custom binding-specific constructors that don't exist in *Target* type).

template<typename **T**, typename ...**Ts**>
class **typed**

This helper class provides an interface to parameterize generic types to improve generated Python function signatures (e.g., to turn `list` into `list[MyType]`).

Consider the following binding that iterates over a Python list.

```
m.def("f", [](nb::list l) {
    for (handle h : l) {
        // ...
    }
});
```

Suppose that `f` expects a list of `MyType` objects, which is not clear from the signature. To make this explicit, use the `nb::typed<T, Ts...>` wrapper to pass additional type parameters. This has no effect besides clarifying the signature—in particular, nanobind does *not* insert additional runtime checks!

```
m.def("f", [](nb::typed<nb::list, MyType> l) {
    for (nb::handle h : l) {
        // ...
    }
});
```

CHAPTER

TWENTYFIVE

C++ API REFERENCE (EXTRAS)

25.1 Operator overloading

The following optional include directive imports the special value `self`.

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

The underlying type exposes various C++ operators that enable a shorthand notation to bind operators to python. See the *operator overloading* example in the main documentation for details.

class `detail::self_t`

This is an internal class that should be accessed through the singleton `self` value.

It supports the overloaded operators listed below. Depending on whether `self` is the left or right argument of a binary operation, the binding will map to different Python methods as shown below.

C++ operator	Python method (left or right)
`operator-`	`__sub__`, `__rsub__`
`operator+`	`__add__`, `__radd__`
`operator*`	`__mul__`, `__rmul__`
`operator/`	`__truediv__`, `__rtruediv__`
`operator%`	`__mod__`, `__rmod__`
`operator<<`	`__lshift__`, `__rlshift__`
`operator>>`	`__rshift__`, `__rrshift__`
`operator&`	`__and__`, `__rand__`
`operator^`	`__xor__`, `__rxor__`
`operator|`	`__or__`, `__ror__`
`operator>`	`__gt__`, `__lt__`
`operator>=`	`__ge__`, `__le__`
`operator<`	`__lt__`, `__gt__`
`operator<=`	`__le__`, `__ge__`
`operator==`	`__eq__`
`operator!=`	`__ne__`
`operator+=`	`__iadd__`
`operator-=`	`__isub__`
`operator*=`	`__mul__`
`operator/=`	`__itruediv__`
`operator%=`	`__imod__`
`operator<<=`	`__ilrshift__`
`operator>>=`	`__ilrshift__`
`operator&=`	`__iand__`
`operator^=`	`__ixor__`
`operator|=`	`__ior__`
`operator-` (unary)	`__neg__`

continues on next page

Table 1 – continued from previous page

C++ operator	Python method (left or right)
`operator+` (unary)	`__pos__`
`operator~` (unary)	`__invert__`
`operator!` (unary)	`__bool__` (with extra negation)
`nb::abs(..)`	`__abs__`
`nb::hash(..)`	`__hash__`

detail::*self_t* **`self`**

25.2 Trampolines

The following macros to implement trampolines that forward virtual function calls to Python require an additional include directive:

```
#include <nanobind/trampoline.h>
```

See the section on *trampolines* for further detail.

`NB_TRAMPOLINE`(base, size)

Install a trampoline in an alias class to enable dispatching C++ virtual function calls to a Python implementation. Refer to the documentation on *trampolines* to see how this macro can be used.

`NB_OVERRIDE`(func, ...)

Dispatch the call to a Python method named `"func"` if it is overloaded on the Python side, and forward the function arguments specified in the variable length argument `...`. Otherwise, call the C++ implementation `func` in the base class.

Refer to the documentation on *trampolines* to see how this macro can be used.

`NB_OVERRIDE_PURE`(func, ...)

Dispatch the call to a Python method named `"func"` if it is overloaded on the Python side, and forward the function arguments specified in the variable length argument `...`. Otherwise, raise an exception. This macro should be used when the C++ function is pure virtual.

Refer to the documentation on *trampolines* to see how this macro can be used.

`NB_OVERRIDE_NAME`(name, func, ...)

Dispatch the call to a Python method named `name` if it is overloaded on the Python side, and forward the function arguments specified in the variable length argument `...`. Otherwise, call the C++ function `func` in the base class.

This function differs from *`NB_OVERRIDE()`* in that C++ and Python functions can be named differently (e.g., `operator+` and `__add__`). Refer to the documentation on *trampolines* to see how this macro can be used.

`NB_OVERRIDE_PURE_NAME`(name, func, ...)

Dispatch the call to a Python method named `name` if it is overloaded on the Python side, and forward the function arguments specified in the variable length argument `...`. Otherwise, raise an exception. This macro should be used when the C++ function is pure virtual.

This function differs from *`NB_OVERRIDE_PURE()`* in that C++ and Python functions can be named differently (e.g., `operator+` and `__add__`). Although the C++ base implementation cannot be called, its name is still important since nanobind uses it to infer the return value type. Refer to the documentation on *trampolines* to see how this macro can be used.

25.3 STL vector bindings

The following function can be used to expose `std::vector<...>` variants in Python. It is not part of the core nanobind API and requires an additional include directive:

```
#include <nanobind/stl/bind_vector.h>
```

template<typename **Vector**, *rv_policy* **Policy** = *rv_policy*::*automatic_reference*, typename ...**Args**>
*class_*<*Vector*> **bind_vector**(*handle* scope, const char *name, *Args*&&... args)

Bind the STL vector-derived type *Vector* to the identifier *name* and place it in *scope* (e.g., a *module_*). The variable argument list can be used to pass a docstring and other *class binding annotations*.

The type includes the following methods resembling `list`:

Signature	Documentation
`__init__(self)`	Default constructor
`__init__(self, arg: Vector)`	Copy constructor
`__init__(self, arg: typing.Sequence)`	Construct from another sequence type
`__len__(self) -> int`	Return the number of elements
`__repr__(self) -> str`	Generate a string representation
`__contains__(self, arg: Value)`	Check if the vector contains `arg`
`__eq__(self, arg: Vector)`	Check if the vector is equal to `arg`
`__ne__(self, arg: Vector)`	Check if the vector is not equal to `arg`
`__bool__(self) -> bool`	Check whether the vector is empty
`__iter__(self) -> iterator`	Instantiate an iterator to traverse the elements
`__getitem__(self, arg: int) -> Value`	Return an element from the list (supports negative indexing)
`__setitem__(self, arg0: int, arg1: Value)`	Assign an element in the list (supports negative indexing)
`__delitem__(self, arg: int)`	Delete an item from the list (supports negative indexing)
`__getitem__(self, arg: slice) -> Vector`	Slice-based getter
`__setitem__(self, arg0: slice, arg1: Value)`	Slice-based assignment
`__delitem__(self, arg: slice)`	Slice-based deletion
`clear(self)`	Remove all items from the list
`append(self, arg: Value)`	Append a list item
`insert(self, arg0: int, arg1: Value)`	Insert a list item (supports negative indexing)
`pop(self, index: int = -1)`	Pop an element at position `index` (the end by default)
`extend(self, arg: Vector)`	Extend `self` by appending elements from `arg`.
`count(self, arg: Value)`	Count the number of times that `arg` is contained in the vector
`remove(self, arg: Value)`	Remove all occurrences of `arg`.

In contrast to `std::vector<...>`, all bound functions perform range checks to avoid undefined behavior. When the type underlying the vector is not comparable or copy-assignable, some of these functions will not be generated.

The binding operation is a no-op if the vector type has already been registered with nanobind.

Warning: While this function creates a type resembling a Python `list`, it has a major caveat: the item accessor `__getitem__` copies the accessed element by default (the bottom of this paragraph explains how this copy can be avoided).

Consequently, writes to elements may not propagate in the expected way. Consider the following C++ bindings:

```
struct A {
    int value;
};

nb::class_<A>(m, "A")
    .def(nb::init<int>())
    .def_rw("value", &A::value);

nb::bind_vector<std::vector<A>>(m, "VecA");
```

On the Python end, they yield the following surprising behavior:

```
from my_ext import A, VecA

va = VecA()
va.append(A(123))
va[0].value = 456
assert va[0].value == 456 # <-- assertion fails!
```

To actually modify `va`, another write is needed.

```
v = va[0]
v.value = 456
va[0] = v
```

This may seem like a strange design, so it is worth explaining why the implementation works in this way.

The key issue is that any particular value (e.g., `va[0]`) lies within a memory buffer managed by the `std::vector`. It is not safe for nanobind to refer to objects within this buffer using their absolute or relative memory address. For example, inserting an element at position 0 will rearrange the buffer's contents and shift all subsequent `A` instances. If nanobind `A` objects could be "views" into the `std::vector`, then an insertion would cause the contents of unrelated `A` Python objects to change unexpectedly. Insertion may also require reallocation of the buffer, invalidating all current addresses, and this could lead to undefined behavior (use-after-free) if nanobind did not make a copy.

There are three situations in which the surprising behavior is avoided:

1. If the modification of the array is performed using in-place operations like

   ```
   v[i] += 5
   ```

   In-place operators automatically perform an array assignment, causing the issue to disappear. This means that if you work with a vector type like `std::vector<int>` or `std::vector<std::string>` with an immutable element type like `int` or `str` on the Python end, it will behave completely naturally in Python.

2. If the array contains STL shared pointers (e.g., `std::vector<std::shared_ptr<T>>`), the added indirection and ownership tracking removes the need for extra copies.

3. If the array contains pointers to reference-counted objects (e.g., `std::vector<ref<T>>` via the *`ref`* wrapper) and `T` uses the intrusive reference counting approach explained *here*, the added indirection and ownership tracking removes the need for extra copies.

   (It is usually unsafe to use this class to bind pointer-valued vectors `std::vector<T*>` when `T` does not use intrusive reference counting, because then there is nothing to prevent the Python objects returned by `__getitem__` from outliving the C++ `T` objects that they point to. But if you are able to guarantee through other means that the `T` objects will live long enough, the intrusive reference counting is not strictly required.)

Note: Previous versions of nanobind (before 2.0) and pybind11 return Python objects from `__getitem__`

that wrap *references* (i.e., views), meaning that they are only safe to use until the next insertion or deletion in the vector they were drawn from. As discussed above, any use after that point could **corrupt memory or crash your program**, which is why reference semantics are no longer the default.

If you truly need the unsafe reference semantics, and if you can guarantee that all use of your bindings will respect the memory layout and reference-invalidation rules of the underlying C++ container type, you can request the old behavior by passing a second template argument of *rv_policy::reference_internal* to *bind_vector()*. This will override nanobind's usual choice of *rv_policy::copy* for `__getitem__`.

```
nb::bind_vector<std::vector<MyType>,
                nb::rv_policy::reference_internal>(m, "ExampleVec");
```

Again, please avoid this if at all possible. It is *very* easy to cause problems if you're not careful, as the following example demonstrates.

```
def looks_fine_but_crashes(vec: ext.ExampleVec) -> None:
    # Trying to remove all the elements too much older than the last:
    last = vec[-1]
    # Even being careful to iterate backwards so we visit each
    # index only once...
    for idx in range(len(vec) - 2, -1, -1):
        if last.timestamp - vec[idx].timestamp > 5:
            del vec[idx]
            # Oops! After the first deletion, 'last' now refers to
            # uninitialized memory.
```

25.4 STL map bindings

The following function can be used to expose `std::map<...>` or `std::unordered_map<...>` variants in Python. It is not part of the core nanobind API and requires an additional include directive:

```
#include <nanobind/stl/bind_map.h>
```

template<typename **Map**, *rv_policy* **Policy** = *rv_policy*::*automatic_reference*, typename ...**Args**>
*class_*<*Map*> **bind_map**(*handle* scope, const char *name, *Args*&&... args)

Bind the STL map-derived type *Map* (ordered or unordered) to the identifier *name* and place it in *scope* (e.g., a *module_*). The variable argument list can be used to pass a docstring and other *class binding annotations*.

The type includes the following methods resembling `dict`:

Signature	Documentation
`__init__(self)`	Default constructor
`__init__(self, arg: Map)`	Copy constructor
`__init__(self, arg: dict)`	Construct from a Python dictionary
`__len__(self) -> int`	Return the number of elements
`__repr__(self) -> str`	Generate a string representation
`__contains__(self, arg: Key)`	Check if the map contains `arg`
`__eq__(self, arg: Map)`	Check if the map is equal to `arg`
`__ne__(self, arg: Map)`	Check if the map is not equal to `arg`
`__bool__(self) -> bool`	Check whether the map is empty
`__iter__(self) -> iterator`	Instantiate an iterator to traverse the set of map keys
`__getitem__(self, arg: Key) -> Value`	Return an element from the map
`__setitem__(self, arg0: Key, arg1: Value)`	Assign an element in the map
`__delitem__(self, arg: Key)`	Delete an item from the map
`clear(self)`	Remove all items from the list
`update(self, arg: Map)`	Update the map with elements from `arg`.
`keys(self, arg: Map) -> Map.KeyView`	Returns an iterable view of the map's keys
`values(self, arg: Map) -> Map.ValueView`	Returns an iterable view of the map's values
`items(self, arg: Map) -> Map.ItemView`	Returns an iterable view of the map's items

The binding operation is a no-op if the map type has already been registered with nanobind.

The binding routine ideally expects the involved types to be:

- copy-constructible
- copy-assignable
- equality-comparable

If not all of these properties are available, then a subset of the above methods will be omitted. Please refer to `bind_map.h` for details on the logic.

Warning: While this function creates a type resembling a Python `dict`, it has a major caveat: the item accessor `__getitem__` copies the accessed element by default.

Please refer to the *STL vector bindings* for a discussion of the problem and possible solutions. Everything applies equally to the map case.

Note: Unlike `std::vector`, the `std::map` and `std::unordered_map` containers are *node-based*, meaning their elements do have a consistent address for as long as they're stored in the map. (Note that this is generally *not* true of third-party containers with similar interfaces, such as `absl::flat_hash_map`.)

If you are binding a node-based container type, and you want `__getitem__` to return a reference to the accessed element rather than copying it, it is *somewhat* safer than it would be with *`bind_vector()`* to use the unsafe workaround discussed there:

```
nb::bind_map<std::map<std::string, SomeValue>,
             nb::rv_policy::reference_internal>(m, "ExampleMap");
```

With a node-based container, the only situation where a reference returned from `__getitem__` would be invalidated is if the individual element that it refers to were removed from the map. Unlike with `std::vector`, additions and removals of *other* elements would not present a danger.

It is still easy to cause problems if you're not careful, though:

```
def unsafe_pop(map: ext.ExampleMap, key: str) -> ext.SomeValue:
    value = map[key]
    del map[key]
    # Oops! `value` now points to a dangling element. Anything you
    # do with it now is liable to crash the interpreter.
    return value  # uh-oh...
```

25.5 Unique pointer deleter

The following *deleter* should be used to gain maximal flexibility in combination with `std::unique_ptr<..>`. It requires the following additional include directive:

```
#include <nanobind/stl/unique_ptr.h>
```

See the two documentation sections on unique pointers for further detail (*#1*, *#2*).

template<typename **T**>
struct **deleter**

deleter() = default

Create a deleter that destroys the object using a `delete` expression.

deleter(*handle* h)

Create a deleter that destroys the object by reducing the Python reference count.

bool **owned_by_python**() const

Check if the object is owned by Python.

bool **owned_by_cpp**() const

Check if the object is owned by C++.

void **operator()**(void *p) noexcept

Destroy the object at address *p*.

25.6 Iterator bindings

The following functions can be used to expose existing C++ iterators in Python. They are not part of the core nanobind API and require an additional include directive:

```
#include <nanobind/make_iterator.h>
```

template<*rv_policy* **Policy** = *rv_policy*::*automatic_reference*, typename **Iterator**, typename **Sentinel**, typename ...**Extra**>
auto **make_iterator**(*handle* scope, const char *name, *Iterator* first, *Sentinel* last, *Extra*&&... extra)

Create a Python iterator wrapping the C++ iterator represented by the range `[first, last)`. The *Extra* parameter can be used to pass additional function binding annotations.

This function lazily creates a new Python iterator type identified by *name*, which is stored in the given *scope*. Usually, some kind of *keep_alive* annotation is needed to tie the lifetime of the parent container to that of the iterator.

The return value is a typed iterator (*iterator* wrapped using *typed*), whose template parameter is given by the type of `*first`.

Here is an example of what this might look like for a STL vector:

```
using IntVec = std::vector<int>;

nb::class_<IntVec>(m, "IntVec")
    .def("__iter__",
         [](const IntVec &v) {
             return nb::make_iterator(nb::type<IntVec>(), "iterator",
                                      v.begin(), v.end());
         }, nb::keep_alive<0, 1>());
```

Note: Pre-2.0 versions of nanobind and pybind11 return *references* (views) into the underlying sequence.

This is convenient when

1. Iterated elements are used to modify the underlying container.
2. Iterated elements should reflect separately made changes to the underlying container.

But this strategy is *unsafe* if the allocated memory region or layout of the container could change (e.g., through insertion of removal of elements).

Because of this, iterators now copy by default. There are two ways to still obtain references to the target elements:

1. If the iterator is over STL shared pointers, the added indirection and ownership tracking removes the need for extra copies.
2. If the iterator is over reference-counted objects (e.g., `ref<T>` via the *ref* wrapper) and `T` uses the intrusive reference counting approach explained *here*, the added indirection and ownership tracking removes the need for extra copies.

If you truly need the unsafe reference semantics, and if you can guarantee that all use of your bindings will respect the memory layout and reference-invalidation rules of the underlying C++ container type, you can request the old behavior by passing *rv_policy::reference_internal* to the `Policy` template argument of this function.

---

template<*rv_policy* **Policy** = *rv_policy*::*automatic_reference*, typename **Type**, typename ...**Extra**>
auto **make_iterator**(*handle* scope, const char *name, *Type* &value, *Extra*&&... extra)

This convenience wrapper calls the above *make_iterator()* variant with `first` and `last` set to `std::begin(value)` and `std::end(value)`, respectively.

template<*rv_policy* **Policy** = *rv_policy*::*automatic_reference*, typename **Iterator**, typename **Sentinel**, typename ...**Extra**>
iterator **make_key_iterator**(*handle* scope, const char *name, *Iterator* first, *Sentinel* last, *Extra*&&... extra)

make_iterator() specialization for C++ iterators that return key-value pairs. *make_key_iterator()* returns the first pair element to iterate over keys.

The return value is a typed iterator (*iterator* wrapped using *typed*), whose template parameter is given by the type of `(*first).first`.

template<*rv_policy* **Policy** = *rv_policy*::*automatic_reference*, typename **Iterator**, typename **Sentinel**, typename ...**Extra**>
iterator **make_value_iterator**(*handle* scope, const char *name, *Iterator* first, *Sentinel* last, *Extra*&&... extra)

make_iterator() specialization for C++ iterators that return key-value pairs. *make_value_iterator()* returns the second pair element to iterate over values.

The return value is a typed iterator (*iterator* wrapped using *typed*), whose template parameter is given by the type of `(*first).second`.

25.7 N-dimensional array type

The following type can be used to exchange n-dimension arrays with frameworks like NumPy, PyTorch, Tensorflow, JAX, CuPy, and others. It requires an additional include directive:

```
#include <nanobind/ndarray.h>
```

Detailed documentation including example code is provided in a *separate section*.

bool **ndarray_check**(*handle* h) noexcept

Test whether the Python object represents an ndarray.

Objects with a `__dlpack__` attribute or objects that implement the buffer protocol are considered as ndarray objects. In addition, arrays from NumPy, PyTorch, TensorFlow and XLA are also regarded as ndarrays.

template<typename ...**Args**>
class **ndarray**

type **Scalar**

The scalar type underlying the array (or `void` if not specified)

static constexpr bool **ReadOnly**

A `constexpr` Boolean value that is `true` if the ndarray template arguments (*`Args...`*) include the `nb::ro` annotation or a `const`-qualified scalar type.

static constexpr char **Order**

A `constexpr` character value set based on the ndarray template arguments (*`Args...`*). It equals

- `'C'` if *`c_contig`* is specified,
- `'F'` if *`f_contig`* is specified,
- `'A'` if *`any_contig`* is specified,
- `'\0'` otherwise.

static constexpr int **DeviceType**

A `constexpr` integer value set to the device type ID extracted from the ndarray template arguments (*`Args...`*), or `device::none::value` when none was specified.

using **VoidPtr** = std::conditional_t<*ReadOnly*, const void*, void*>

A potentially `const`-qualified `void*` pointer type used by some of the `ndarray` constructors.

ndarray() = default

Create an invalid array.

template<typename ...**Args2**>
explicit **ndarray**(const *ndarray*<*Args2*...> &other)

Reinterpreting constructor that wraps an existing nd-array (parameterized by *`Args...`*) into a new ndarray (parameterized by *`Args2...`*). No copy or conversion is made.

Dropping parameters is always safe. For example, a function that returns different array types could call it to convert `ndarray<T>` to `ndarray<>`. When adding constraints, the constructor is only safe to use following a runtime check to ensure that newly created array actually possesses the advertised properties.

ndarray(const *ndarray*&)

Copy constructor. Increases the reference count of the referenced array.

ndarray(*ndarray*&&)

Move constructor. Steals the referenced array without changing reference counts.

~ndarray()

Decreases the reference count of the referenced array and potentially destroy it.

ndarray &**operator=**(const *ndarray*&)

Copy assignment operator. Increases the reference count of the referenced array. Decreases the reference count of the previously referenced array and potentially destroy it.

ndarray &**operator=**(*ndarray*&&)

Move assignment operator. Steals the referenced array without changing reference counts. Decreases the reference count of the previously referenced array and potentially destroy it.

ndarray(*VoidPtr* data, const std::initializer_list<size_t> shape = {}, *handle* owner = {}, std::initializer_list<int64_t> strides = {}, dlpack::*dtype* dtype = nanobind::*dtype*<*Scalar*>(), int32_t device_type = *DeviceType*, int32_t device_id = 0, char order = *Order*)

Create an array wrapping an existing memory allocation.

Only the *data* parameter is strictly required, while some other parameters can be be inferred from static *nb::ndarray<...>* template parameters.

The parameters have the following meaning:

- *data*: a CPU/GPU/.. pointer to the memory region storing the array data.

  When the array is parameterized by a `const` scalar type, or when it has a *nb::ro* read-only annotation, a `const` pointer can be passed here.

- *shape*: an initializer list that simultaneously specifies the number of dimensions and the size along each axis. If left at its default `{}`, the *nb::shape* template parameter will take precedence (if present).

- *owner*: if provided, the array will hold a reference to this object until its destruction. This makes it possible to create zero-copy views into other data structures, while guaranteeing the memory safety of array accesses.

- *strides*: an initializer list explaining the layout of the data in memory. Each entry denotes the number of elements to jump over to advance to the next item along the associated axis.

  strides must either have the same size as *shape* or be empty. In the latter case, strides are automatically computed according to the *order* parameter.

  Note that strides in nanobind express *element counts* rather than *byte counts*. This convention differs from other frameworks (e.g., NumPy) and is a consequence of the underlying DLPack protocol.

- *dtype* describes the numeric data type of array elements (e.g., floating point, signed/unsigned integer) and their bit depth.

  You can use the *nb::dtype<T>()* function to obtain the right value for a given type.

- *device_type* and *device_id* specify where the array data is stored. The *device_type* must be an enumerant like *nb::device::cuda::value*, while the meaning of the device ID is unspecified and platform-dependent.

  Note that the *device_id* is set to `0` by default and cannot be inferred by nanobind. If your extension creates arrays on multiple different compute accelerators, you *must* provide this parameter.

- The *order* parameter denotes the coefficient order in memory and is only relevant when *strides* is empty. Specify `'C'` for C-style or `'F'` for Fortran-style. When this parameter is not explicitly specified, the implementation uses the order specified as an ndarray template argument, or C-style order as a fallback.

Both `strides` and `shape` will be copied by the constructor, hence the targets of these initializer lists do not need to remain valid following the constructor call.

> **Warning:** The Python *global interpreter lock* (GIL) must be held when calling this function.

ndarray(*VoidPtr* data, size_t ndim, const size_t *shape, *handle* owner, const int64_t *strides = nullptr, dlpack::*dtype* dtype = nanobind::*dtype*<*Scalar*>(), int device_type = *DeviceType*, int device_id = 0, char order = *Order*)

Alternative form of the above constructor, which accepts the *shape* and *strides* arguments using pointers instead of initializer lists. The number of dimensions must be specified via the *ndim* parameter in this case.

See the previous constructor for details, the remaining behavior is identical.

dlpack::*dtype* **dtype**() const

Return the data type underlying the array

size_t **ndim**() const

Return the number of dimensions.

size_t **size**() const

Return the size of the array (i.e. the product of all dimensions).

size_t **itemsize**() const

Return the size of a single array element in bytes. The returned value is rounded up to the next full byte in case of bit-level representations (query `dtype::bits` for bit-level granularity).

size_t **nbytes**() const

Return the size of the entire array bytes. The returned value is rounded up to the next full byte in case of bit-level representations.

size_t **shape**(size_t i) const

Return the size of dimension *i*.

int64_t **stride**(size_t i) const

Return the stride (in number of elements) of dimension *i*.

const int64_t ***shape_ptr**() const

Return a pointer to the shape array. Note that the return type is `const int64_t*`, which may be unexpected as the scalar version *shape()* casts its result to a `size_t`.

This is a consequence of the DLPack tensor representation that uses signed 64-bit integers for all of these fields.

const int64_t ***stride_ptr**() const

Return pointer to the stride array.

bool **is_valid**() const

Check whether the array is in a valid state.

int **device_type**() const

ID denoting the type of device hosting the array. This will match the `value` field of a device class, such as *device::cpu::value* or *device::cuda::value*.

int **device_id**() const

In a multi-device/GPU setup, this function returns the ID of the device storing the array.

Scalar ***data**() const

Return a pointer to the array data. If *ReadOnly* is true, a pointer-to-const is returned.

template<typename ...**Args2**>

auto &**operator()**(*Args2*... indices)

Return a reference to the element stored at the provided index/indices. If *ReadOnly* is true, a reference-to-const is returned. Note that `sizeof...(Args2)` must match *ndim()*.

This accessor is only available when the scalar type and array dimension were specified as template parameters.

This function should only be used when the array storage is accessible through the CPU's virtual memory address space.

template<typename ...**Extra**>
auto **view**()

Returns an nd-array view that is optimized for fast array access on the CPU. You may optionally specify additional ndarray constraints via the *Extra* parameter (though a runtime check should first be performed to ensure that the array possesses these properties).

The returned view provides the operations `data()`, `ndim()`, `shape()`, `stride()`, and `operator()` following the conventions of the *ndarray* type.

auto **cast**(*rv_policy* policy = *rv_policy*::*automatic_reference*, *handle* parent = {})

The expression `array.cast(policy, parent)` is almost equivalent to *nb::cast(array, policy, parent)*.

The main difference is that the return type of *nb::cast* is *nb::object*, which renders as a rather non-descriptive `object` in Python bindings. The `.cast()` method returns a custom wrapper type that still derives from *nb::object*, but whose type signature in bindings reproduces that of the original nd-array.

25.7.1 Data types

Nanobind uses the DLPack ABI to represent metadata describing n-dimensional arrays (even when they are exchanged using the buffer protocol). Consequently, the set of possible dtypes is *more restricted* than that of other nd-array libraries (e.g., NumPy). Relevant data structures are located in the `nanobind::dlpack` sub-namespace.

enum class `dlpack::dtype_code` : uint8_t

This enumeration characterizes the elementary array data type regardless of bit depth.

enumerator **Int** = 0

Signed integer format

enumerator **UInt** = 1

Unsigned integer format

enumerator **Float** = 2

IEEE-754 floating point format

enumerator **Bfloat** = 4

"Brain" floating point format

enumerator **Complex** = 5

Complex numbers parameterized by real and imaginary component

struct `dlpack::dtype`

Represents the data type underlying an n-dimensional array. Use the *dtype<T>()* function to return a populated instance of this data structure given a scalar C++ arithmetic type.

uint8_t **code** = 0;

This field must contain the value of one of the *dlpack::dtype_code* enumerants.

uint8_t **bits** = 0;

Number of bits per entry (e.g., 32 for a C++ single precision `float`)

uint16_t **lanes** = 0;

Number of SIMD lanes (typically 1)

template<typename **T**>
dlpack::*dtype* **dtype**()

Returns a populated instance of the *dlpack::dtype* structure given a scalar C++ arithmetic type.

25.7.2 Array annotations

The *ndarray<..>* class admits optional template parameters. They constrain the type of array arguments that may be passed to a function.

The following are supported:

Data type

The data type of the underlying scalar element. The following are supported.

- `[u]int8_t` up to `[u]int64_t` and other variations (`unsigned long long`, etc.)
- `float`, `double`
- `bool`

Annotate the data type with `const` to indicate a read-only array. Note that only the buffer protocol/NumPy interface considers `const`-ness at the moment; data exchange with other array libraries will ignore this annotation.

When the is unspecified (e.g., to accept arbitrary input arrays), the *ro* annotation can instead be used to denote read-only access:

class **ro**

Indicate read-only access (use only when no data type is specified.)

nanobind does not support non-standard types as documented in the section on *dtype limitations*.

Shape

template<ssize_t... **Is**>
class **shape**

Require the array to have `sizeof...(Is)` dimensions. Each entry of *Is* specifies a fixed size constraint for that specific dimension. An entry equal to `-1` indicates that *any* size should be accepted for this dimension.

(An alias named `nb::any` representing `-1` was removed in nanobind 2).

template<size_t **N**>
class **ndim**

Alternative to the above that only constrains the array dimension. `nb::ndim<2>` is equivalent to `nb::shape<-1, -1>`.

Contiguity

class **c_contig**

Request that the array storage uses a C-contiguous representation.

class **f_contig**

Request that the array storage uses a F (Fortran)-contiguous representation.

class **any_contig**

Accept both C- and F-contiguous arrays.

If you prefer not to require contiguity, simply do not provide any of the `*_contig` template parameters listed above.

Device type

class **device**

The following helper classes can be used to constrain the device and address space of an array. Each class has a `static constexpr int32_t value` field that will then match up with *ndarray::device_id()*.

class **cpu**

CPU heap memory

class **cuda**

NVIDIA CUDA device memory

class **cuda_host**

NVIDIA CUDA host-pinned memory

class **cuda_managed**

NVIDIA CUDA managed memory

class **vulkan**

Vulkan device memory

class **metal**

Apple Metal device memory

class **rocm**

AMD ROCm device memory

class **rocm_host**

AMD ROCm host memory

class **oneapi**

Intel OneAPI device memory

Framework

Framework annotations cause *nb::ndarray* objects to convert into an equivalent representation in one of the following frameworks:

class **numpy**

class **tensorflow**

class **pytorch**

class **jax**

class **cupy**

25.8 Eigen convenience type aliases

The following helper type aliases require an additional include directive:

```
#include <nanobind/eigen/dense.h>
```

using **DStride** = Eigen::Stride<Eigen::Dynamic, Eigen::Dynamic>

This type alias refers to an Eigen stride object that is sufficiently flexible so that can be easily called with NumPy arrays and array slices.

template<typename **T**>

using **DRef** = Eigen::Ref<*T*, 0, *DStride*>

This templated type alias creates an `Eigen::Ref<..>` with flexible strides for zero-copy data exchange between Eigen and NumPy.

template<typename **T**>
using **DMap** = Eigen::Map<*T*, 0, *DStride*>

This templated type alias creates an `Eigen::Map<..>` with flexible strides for zero-copy data exchange between Eigen and NumPy.

25.9 Timestamp and duration conversions

nanobind supports bidirectional conversions of timestamps and durations between their standard representations in Python (`datetime.datetime`, `datetime.timedelta`) and in C++ (`std::chrono::time_point`, `std::chrono::duration`). A few unidirectional conversions from other Python types to these C++ types are also provided and explained below.

These type casters require an additional include directive:

```
#include <nanobind/stl/chrono.h>
```

25.9.1 An overview of clocks in C++11

The C++11 standard defines three different clocks, and users can define their own. Each `std::chrono::time_point` is defined relative to a particular clock. When using the `chrono` type caster, you must be aware that only `std::chrono::system_clock` is guaranteed to convert to a Python `datetime` object; other clocks may convert to `timedelta` if they don't represent calendar time.

The first clock defined by the standard is `std::chrono::system_clock`. This clock measures the current date and time, much like the Python `time.time()` function. It can change abruptly due to administrative actions, daylight savings time transitions, or synchronization with an external time server. That makes this clock a poor choice for timing purposes, but a good choice for wall-clock time.

The second clock defined by the standard is `std::chrono::steady_clock`. This clock ticks at a steady rate and is never adjusted, like `time.monotonic()` in Python. That makes it excellent for timing purposes, but the value in this clock does not correspond to the current date and time. Often this clock will measure the amount of time your system has been powered on. This clock will never be the same clock as the system clock, because 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 all the clocks in the system. It is normally an alias for either `system_clock` or `steady_clock`, but can be its own independent clock. Due to this uncertainty, conversions of time measured on the `high_resolution_clock` to Python produce platform-dependent types: you'll get a `datetime` if `high_resolution_clock` is an alias for `system_clock` on your system, or a `timedelta` value otherwise.

25.9.2 Provided conversions

The C++ types described in this section may be instantiated with any precision. Conversions to a less-precise type will round towards zero. Since Python's built-in date and time objects support only microsecond precision, any precision beyond that on the C++ side will be lost when converting to Python.

C++ to Python

- **std::chrono::system_clock::time_point** → **datetime.datetime**
  A system clock time will be converted to a Python `datetime` instance. The result describes a time in the local timezone, but does not have any timezone information attached to it (it is a naive datetime object).
- **std::chrono::duration** → **datetime.timedelta**
  A duration will be converted to a Python `timedelta`. Any precision beyond microseconds is lost by rounding towards zero.
- **std::chrono::[other_clock]::time_point** → **datetime.timedelta**
  A time on any clock except the system clock will be converted to a Python `timedelta`, which measures the number of seconds between the clock's epoch and the time point of interest.

Python to C++

- **datetime.datetime** or **datetime.date** or **datetime.time** → **std::chrono::system_clock::time_point**
  A Python date, time, or datetime object can be converted into a system clock timepoint. A `time` with no date information is treated as that time on January 1, 1970. A `date` with no time information is treated as midnight on that date. **Any timezone information is ignored.**
- **datetime.timedelta** → **std::chrono::duration**
  A Python time delta object can be converted into a duration that describes the same number of seconds (modulo precision limitations).
- **datetime.timedelta** → **std::chrono::[other_clock]::time_point**
  A Python time delta object can be converted into a timepoint on a clock other than the system clock. The resulting timepoint will be that many seconds after the target clock's epoch time.
- **float** → **std::chrono::duration**
  A floating-point value can be converted into a duration. The input is treated as a number of seconds, and fractional seconds are supported to the extent representable.
- **float** → **std::chrono::[other_clock]::time_point**
  A floating-point value can be converted into a timepoint on a clock other than the system clock. The input is treated as a number of seconds, and fractional seconds are supported to the extent representable. The resulting timepoint will be that many seconds after the target clock's epoch time.

25.10 Evaluating Python expressions from strings

The following functions can be used to evaluate Python functions and expressions. They require an additional include directive:

```
#include <nanobind/eval.h>
```

Detailed documentation including example code is provided in a *separate section*.

enum class **eval_mode**

This enumeration specifies how the content of a string should be interpreted. Used in Py_CompileString().

enumerator **eval_expr** = Py_eval_input

Evaluate a string containing an isolated expression

enumerator **eval_single_statement** = Py_single_input

Evaluate a string containing a single statement. Returns c None

enumerator **eval_statements** = Py_file_input

Evaluate a string containing a sequence of statement. Returns c None

template<*eval_mode* **start** = eval_expr, size_t **N**>
object **eval**(const char (&s)[*N*], *handle* global = *handle*(), *handle* local = *handle*())

Evaluate the given Python code in the given global/local scopes, and return the value.

inline void **exec**(const *str* &expr, *handle* global = *handle*(), *handle* local = *handle*())

Execute the given Python code in the given global/local scopes.

25.11 Intrusive reference counting helpers

The following functions and classes can be used to augment user-provided classes with intrusive reference counting that greatly simplifies shared ownership in larger C++/Python binding projects.

This functionality requires the following include directives:

```
#include <nanobind/intrusive/counter.h>
#include <nanobind/intrusive/ref.h>
```

These headers reference several functions, whose implementation must be provided. You can do so by including the following file from a single `.cpp` file of your project:

```
#include <nanobind/intrusive/counter.inl>
```

The functionality in these files consist of the following classes and functions:

class **intrusive_counter**

Simple atomic reference counter that can optionally switch over to Python-based reference counting.

The various copy/move assignment/constructors intentionally don't transfer the reference count. This is so that the contents of classes containing an `intrusive_counter` can be copied/moved without disturbing the reference counts of the associated instances.

intrusive_counter() noexcept = default

Initialize with a reference count of zero.

intrusive_counter(const *intrusive_counter* &o)

Copy constructor, which produces a zero-initialized counter. Does *not* copy the reference count from *o*.

intrusive_counter(*intrusive_counter* &&o)

Move constructor, which produces a zero-initialized counter. Does *not* copy the reference count from *o*.

intrusive_counter &**operator=**(const *intrusive_counter* &o)

Copy assignment operator. Does *not* copy the reference count from *o*.

intrusive_counter &**operator=**(*intrusive_counter* &&o)

Move assignment operator. Does *not* copy the reference count from *o*.

void **inc_ref**() const noexcept

Increase the reference count. When the counter references an object managed by Python, the operation calls `Py_INCREF()` to increase the reference count of the Python object instead.

The *inc_ref()* top-level function encapsulates this logic for subclasses of *intrusive_base*.

bool **dec_ref**() const noexcept

Decrease the reference count. When the counter references an object managed by Python, the operation calls `Py_DECREF()` to decrease the reference count of the Python object instead.

When the C++-managed reference count reaches zero, the operation returns `true` to signal to the caller that it should use a *delete expression* to destroy the instance.

The *dec_ref()* top-level function encapsulates this logic for subclasses of *intrusive_base*.

void **set_self_py**(PyObject *self)

Set the Python object associated with this instance. This operation is usually called by nanobind when ownership is transferred to the Python side.

Any references from prior calls to *intrusive_counter::inc_ref()* are converted into Python references by calling `Py_INCREF()` repeatedly.

PyObject ***self_py**()

Return the Python object associated with this instance (or `nullptr`).

class **intrusive_base**

Simple polymorphic base class for a intrusively reference-counted object hierarchy. The member functions expose corresponding functionality of *intrusive_counter*.

void **inc_ref**() const noexcept

See *intrusive_counter::inc_ref()*.

bool **dec_ref**() const noexcept

See *intrusive_counter::dec_ref()*.

void **set_self_py**(PyObject *self)

See *intrusive_counter::set_self_py()*.

PyObject ***self_py**()

See *intrusive_counter::self_py()*.

void **intrusive_init**(void (*intrusive_inc_ref_py)(PyObject*) noexcept, void (*intrusive_dec_ref_py)(PyObject*) noexcept)

Function to register reference counting hooks with the intrusive reference counter class. This allows its implementation to not depend on Python.

You would usually call this function as follows from the initialization routine of a Python extension:

```
NB_MODULE(my_ext, m) {
    nb::intrusive_init(
        [](PyObject * o) noexcept {
            nb::gil_scoped_acquire guard;
            Py_INCREF(o);
        },
        [](PyObject * o) noexcept {
            nb::gil_scoped_acquire guard;
            Py_DECREF(o);
        });

    // ...
}
```

inline void **inc_ref**(*intrusive_base* *o) noexcept

Reference counting helper function that calls `o->inc_ref()` if `o` is not equal to `nullptr`.

inline void **dec_ref**(*intrusive_base* *o) noexcept

Reference counting helper function that calls `o->dec_ref()` if `o` is not equal to `nullptr` and `delete o` when the reference count reaches zero.

template<typename **T**>
class **ref**

RAII scoped reference counting helper class

ref<T> is a simple RAII wrapper class that encapsulates a pointer to an instance with intrusive reference counting.

It takes care of increasing and decreasing the reference count as needed and deleting the instance when the count reaches zero.

For this to work, compatible functions *inc_ref()* and *dec_ref()* must be defined before including the file `nanobind/intrusive/ref.h`. Default implementations for subclasses of the type *intrusive_base* are already provided as part of the file `counter.h`.

ref() = default

Create a null reference

ref(*T* *ptr)

Create a reference from a pointer. Increases the reference count of the object (if not `nullptr`).

ref(const *ref* &r)

Copy a reference. Increase the reference count of the object (if not `nullptr`).

ref(*ref* &&r) noexcept

Move a reference. Object reference counts are unaffected by this operation.

~ref()

Destroy a reference. Decreases the reference count of the object (if not `nullptr`).

ref &**operator=**(*ref* &&r) noexcept

Move-assign another reference into this one.

ref &**operator=**(const *ref* &r)

Copy-assign another reference into this one.

ref &**operator=**(const *T* *ptr)

Overwrite this reference with a pointer to another object

void **reset**()

Clear the reference and reduces the reference count of the object (if not `nullptr`)

bool **operator==**(const *ref* &r) const

Compare this reference with another reference (pointer equality)

bool **operator!=**(const *ref* &r) const

Compare this reference with another reference (pointer inequality)

bool **operator==**(const *T* *ptr) const

Compare this reference with another object (pointer equality)

bool **operator!=**(const *T* *ptr) const

Compare this reference with another object (pointer inequality)

T ***operator->**()

Access the object referenced by this reference

const *T* ***operator->**() const

Access the object referenced by this reference (const version)

T &**operator***()

Return a C++ reference to the referenced object

const *T* &**operator***() const

Return a C++ reference to the referenced object (const version)

T ***get**()

Return a C++ pointer to the referenced object

const *T* ***get**() const

Return a C++ pointer to the referenced object (const version)

25.12 Typing

The following functions for typing-related functionality require an additional include directive:

```
#include <nanobind/typing.h>
```

template<typename ...**Args**>
object **type_var**(*Args*&&... args)

Create a type variable (i.e., an instance of `typing.TypeVar`). All arguments of the original Python construction are supported, e.g.:

```
m.attr("T") = nb::type_var("T",
                           "contravariant"_a = true,
                           "covariant"_a = false,
                           "bound"_a = nb::type<MyClass>());
```

template<typename ...**Args**>
object **type_var_tuple**(*Args*&&... args)

Analogousto *type_var()*, create a type variable tuple (i.e., an instance of `typing.TypeVarTuple`).

object **any_type**()

Convenience wrapper, which returns `typing.Any`.

CHAPTER

TWENTYSIX

CMAKE API REFERENCE

nanobind's CMake API simplifies the process of building python extension modules. This is needed because quite a few steps are involved: nanobind must build the module, a library component, link the two together, and add a different set of compilation and linker flags depending on the target platform.

If you prefer another build system, then you have the following options:

- Nicholas Junge has created a Bazel interface to nanobind. Please report Bazel-specific issues there.
- Will Ayd has created a Meson WrapDB package for nanobind. Please report Meson-specific issues on the Meson WrapDB repository.
- You could create a new build system from scratch that takes care of these steps. See this file for inspiration on how to do this on Linux. Note that you will be on your own if you choose to go this route—I unfortunately do not have the time to respond to GitHub tickets related to custom build systems.

The section on *building extensions* provided an introductory example of how to set up a basic build system via the *nanobind_add_module()* command, which is the *high level* build interface. The defaults chosen by this function are somewhat opinionated, however. For this reason, nanobind also provides an alternative *low level* interface that decomposes it into smaller steps.

A later part of this section explains how a Git submodule dependency can be *avoided* in exchange for a system-provided package.

Finally, the section ends with an explanation of the CMake convenience interface for *stub generation*.

26.1 High-level interface

The high-level interface consists of just one CMake command:

nanobind_add_module

Compile a nanobind extension module using the specified target name, optional flags, and source code files. Use it as follows:

```
nanobind_add_module(
  my_ext                  # Target name
  NB_STATIC STABLE_ABI LTO # Optional flags (see below)
  my_ext.h                # Source code files below
  my_ext.cpp)
```

It supports the following optional parameters:

STABLE_	Perform a stable ABI build, making it possible to use a compiled extension across Python minor versions. The flag is ignored on Python versions older than < 3.12.
FREE_TH	Compile an Python extension that opts into free-threaded (i.e., GIL-less) Python behavior, which requires a special free-threaded build of Python 3.13 or newer. The flag is ignored on unsupported Python versions.
NB_STAT	Compile the core nanobind library as a static library. This simplifies redistribution but can increase the combined binary storage footprint when a project contains many Python extensions (this is the default).
NB_SHAR	The opposite of NB_STATIC: compile the core nanobind library as a shared library for use in projects that consist of multiple extensions.
PROTECT	Don't remove stack smashing-related protections.
LTO	Perform link time optimization.
NOMINSI	Don't perform optimizations to minimize binary size.
NOSTRIP	Don't strip unneded symbols and debug information from the compiled extension when performing release builds.
NB_DOMA <name>	Restrict the inter-extension type visibility to a named subdomain. See the associated *FAQ entry* for details.
MUSL_DY	When cibuildwheel is used to produce musllinux wheels, don't statically link against libstdc++ and libgcc (which is an optimization that nanobind does by default in this specific case). If this explanation sounds confusing, then you can ignore it. See the detailed description below for more information on this step.

nanobind_add_module() performs the following steps to produce bindings.

- It creates a CMake library via `add_library(target_name MODULE ...)` and enables the use of C++17 features during compilation.
- It creates a CMake target for an internal library component required by nanobind (named `nanobind-..` where `..` depends on the compilation flags). This is only done once when compiling multiple extensions.

  This library component can either be a static or shared library depending on whether the optional `NB_STATIC` or `NB_SHARED` parameter was provided to `nanobind_add_module()`. The default is a static build, which simplifies redistribution (only one shared library must be deployed).

  When a project contains many Python extensions, a shared build is preferable to avoid unnecessary binary size overheads that arise from redundant copies of the `nanobind-...` component.
- It links the newly created library against the `nanobind-..` target.
- It appends the library suffix (e.g., `.cpython-39-darwin.so`) based on information provided by CMake's `FindPython` module.
- When requested via the optional `STABLE_ABI` parameter, the build system will create a stable ABI extension module with a different suffix (e.g., `.abi3.so`).

  Once compiled, a stable ABI extension can be reused across Python minor versions. In contrast, ordinary builds are only compatible across patch versions. This feature requires Python >= 3.12 and is ignored on older versions. Note that use of the stable ABI come at a small performance cost since nanobind can no longer access the internals of various data structures directly. If in doubt, benchmark your code to see if the cost is acceptable.
- In non-debug modes, it compiles with *size optimizations* (i.e., `-Os`). This is generally the mode that you will want to use for C++/Python bindings. Switching to `-O3` would enable further optimizations like vectorization, loop unrolling, etc., but these all increase compilation time and binary size with no real benefit for bindings.

  If your project contains portions that benefit from `-O3`-level optimizations, then it's better to run two separate compilation steps. An example is shown below:

```
# Compile project code with current optimization mode configured in CMake
add_library(example_lib STATIC source_1.cpp source_2.cpp)
# Need position independent code (-fPIC) to link into 'example_ext' below
set_target_properties(example_lib PROPERTIES POSITION_INDEPENDENT_CODE ON)

# Compile extension module with size optimization and add 'example_lib'
nanobind_add_module(example_ext common.h source_1.cpp source_2.cpp)
target_link_libraries(example_ext PRIVATE example_lib)
```

Size optimizations can be disabled by specifying the optional `NOMINSIZE` argument, though doing so is not recommended.

- `nanobind_add_module()` also disables stack-smashing protections (i.e., it specifies `-fno-stack-protector` to Clang/GCC). Protecting against such vulnerabilities in a Python VM seems futile, and it adds non-negligible extra cost (+8% binary size in benchmarks). This behavior can be disabled by specifying the optional `PROTECT_STACK` flag. Either way, is not recommended that you use nanobind in a setting where it presents an attack surface.
- It sets the default symbol visibility to `hidden` so that only functions and types specifically marked for export generate symbols in the resulting binary. This substantially reduces the size of the generated binary.
- In release builds, it strips unreferenced functions and debug information names from the resulting binary. This can substantially reduce the size of the generated binary and can be disabled using the optional `NOSTRIP` argument.
- Link-time optimization (LTO) is *not active* by default; benefits compared to pybind11 are relatively low, and this can make linking a build bottleneck. That said, the optional `LTO` argument can be specified to enable LTO in release builds.
- nanobind's CMake build system is often combined with cibuildwheel to automate the generation of wheels for many different platforms. One such platform called musllinux exists to create tiny self-contained binaries that are cheap to install in a container environment (Docker, etc.). An issue of the combination with nanobind is that `musllinux` doesn't include the `libstdc++` and `libgcc` libraries which nanobind depends on. `cibuildwheel` then has to ship those along in each wheel, which actually increases their size rather dramatically (by a factor of >5x for small projects). To avoid this, nanobind prefers to link against these libraries *statically* when it detects a `cibuildwheel` build targeting `musllinux`. Pass the `MUSL_DYNAMIC_LIBCPP` parameter to avoid this behavior.
- If desired (via the optional `NB_DOMAIN` parameter), nanobind will restrict the visibility of symbols to a named subdomain to avoid conflicts between bindings. See the associated *FAQ entry* for details.

26.2 Low-level interface

Instead of *nanobind_add_module()* nanobind also exposes a more fine-grained interface to the underlying operations. The following

```
nanobind_add_module(my_ext NB_SHARED LTO my_ext.cpp)
```

is equivalent to

```
# Build the core parts of nanobind once
nanobind_build_library(nanobind SHARED)

# Compile an extension library
add_library(my_ext MODULE my_ext.cpp)

# .. and link it against the nanobind parts
```

(continues on next page)

(continued from previous page)

```
target_link_libraries(my_ext PRIVATE nanobind)

# .. enable size optimizations
nanobind_opt_size(my_ext)

# .. enable link time optimization
nanobind_lto(my_ext)

# .. set the default symbol visibility to 'hidden'
nanobind_set_visibility(my_ext)

# .. strip unneeded symbols and debug info from the binary (only active in release␣
 ↪builds)
nanobind_strip(my_ext)

# .. disable the stack protector
nanobind_disable_stack_protector(my_ext)

# .. set the Python extension suffix
nanobind_extension(my_ext)

# .. set important compilation flags
nanobind_compile_options(my_ext)

# .. set important linker flags
nanobind_link_options(my_ext)

# Statically link against libstdc++/libgcc when targeting musllinux
nanobind_musl_static_libcpp(my_ext)
```

The various commands are described below:

nanobind_build_library

Compile the core nanobind library. The function expects only the target name and uses a slightly unusual parameter passing policy: its behavior changes based on whether or not one the following substrings is detected in the target name:

`-static`	Perform a static library build (without this suffix, a shared build is used)
`-abi3`	Perform a stable ABI build targeting Python v3.12+.
`-ft`	Perform a build that opts into the Python 3.13+ free-threaded behavior.

```
# Normal shared library build
nanobind_build_library(nanobind)

# Static ABI3 build
nanobind_build_library(nanobind-static-abi3)
```

nanobind_opt_size

This function enable size optimizations in `Release`, `MinSizeRel`, `RelWithDebInfo` builds. It expects a single target as argument, as in

```
nanobind_opt_size(my_target)
```

nanobind_set_visibility

This function sets the default symbol visibility to `hidden` so that only functions and types specifically marked for export generate symbols in the resulting binary. It expects a single target as argument, as in

```
nanobind_trim(my_target)
```

This substantially reduces the size of the generated binary.

nanobind_strip

This function strips unused and debug symbols in `Release` and `MinSizeRel` builds on Linux and macOS. It expects a single target as argument, as in

```
nanobind_strip(my_target)
```

nanobind_disable_stack_protector

The stack protector affects the binary size of bindings negatively (+8% on Linux in benchmarks). Protecting from stack smashing in a Python VM seems in any case futile, so this function disables it for the specified target when performing a build with optimizations. Use it as follows:

```
nanobind_disable_stack_protector(my_target)
```

nanobind_extension

This function assigns an extension name to the compiled binding, e.g., `.cpython-311-darwin.so`. Use it as follows:

```
nanobind_extension(my_target)
```

nanobind_extension_abi3

This function assigns a stable ABI extension name to the compiled binding, e.g., `.abi3.so`. Use it as follows:

```
nanobind_extension_abi3(my_target)
```

nanobind_compile_options

This function sets recommended compilation flags. Currently, it specifies `/bigobj` and `/MP` on MSVC builds, and it does nothing other platforms or compilers. Use it as follows:

```
nanobind_compile_options(my_target)
```

nanobind_link_options

This function sets recommended linker flags. Currently, it controls link time handling of undefined symbols on Apple platforms related to Python C API calls, and it does nothing other platforms. Use it as follows:

```
nanobind_link_options(my_target)
```

nanobind_musl_static_libcpp

This function passes the linker flags `-static-libstdc++` and `-static-libgcc` to `gcc` when the environment variable `AUDITWHEEL_PLAT` contains the string `musllinux`, which indicates a cibuildwheel build targeting that platform.

The function expects a single target as argument, as in

```
nanobind_musl_static_libcpp(my_target)
```

26.3 Submodule dependencies

nanobind includes a dependency (a fast hash map named `tsl::robin_map`) as a Git submodule. If you prefer to use another (e.g., system-provided) version of this dependency, set the `NB_USE_SUBMODULE_DEPS` variable before importing nanobind into CMake. In this case, nanobind's CMake scripts will internally invoke `find_dependency(tsl-robin-map)` to locate the associated header files.

26.4 Stub generation

Nanobind's CMake tooling includes a convenience command to interface with the `stubgen` program explained in the section on *stub generation*.

nanobind_add_stub

Import the specified module (`MODULE` parameter), generate a stub, and write it to the specified file (`OUTPUT` parameter). Here is an example use:

```
nanobind_add_stub(
    my_ext_stub
    MODULE my_ext
    OUTPUT my_ext.pyi
    PYTHON_PATH $<TARGET_FILE_DIR:my_ext>
    DEPENDS my_ext
)
```

The target name (`my_ext_stub` in this example) must be unique but has no other significance.

`stubgen` will add all paths specified as part of the `PYTHON_PATH` block and then execute `import my_ext` in a Python session. If the extension is not importable, this will cause stub generation to fail.

This command supports the following parameters:

INSTA	By default, stub generation takes place at build time following generation of all dependencies (see DEPENDS). When this parameter is specified, stub generation is instead postponed to the installation phase.
MODUI	Specifies the name of the module that should be imported. Mandatory.
OUTPU	Specifies the name of the stub file that should be written. The path is relative to CMAKE_CURRENT_BINARY_DIR for build-time stub generation and relative to CMAKE_INSTALL_PREFIX for install-time stub generation. Mandatory.
PYTHO	List of search paths that should be considered when importing the module. The paths are relative to CMAKE_CURRENT_BINARY_DIR for build-time stub generation and relative to CMAKE_INSTALL_PREFIX for install-time stub generation. The current directory (".") is always included and does not need to be specified. The parameter may contain CMake generator expressions when *nanobind_add_stub()* is used for build-time stub generation. Otherwise, generator expressions should not be used. Optional.
DEPEN	Any targets listed here will be marked as a dependencies. This should generally be used to list the target names of one or more prior *nanobind_add_module()* declarations. Note that this parameter tracks *build-time* dependencies and does not need to be specified when stub generation occurs at install time (see INSTALL_TIME). Optional.
VERBO	Show status messages generated by stubgen.
EXCLU	Generate a stub containing only typed signatures without docstrings.
INCLU	Also include private members, whose names begin or end with a single underscore.
MARKI	Typed extensions normally identify themselves via the presence of an empty file named py.typed in each module directory. When this parameter is specified, *nanobind_add_stub()* will automatically generate such an empty file as well.
PATTI	Specify a pattern file used to replace declarations in the stub. The syntax is described in the section on *stub generation*.
COMPO	Specify a component when INSTALL_TIME stub generation is used. This is analogous to install(..., COMPONENT [name]) in other install targets.
EXCLU	If specified, the file is only installed as part of a component-specific installation when INSTALL_TIME stub generation is used. This is analogous to install(..., EXCLUDE_FROM_ALL) in other install targets.

CHAPTER

TWENTYSEVEN

BAZEL API REFERENCE (3RD PARTY)

This page contains a reference of the basic APIs of nanobind-bazel.

27.1 Rules

nanobind-bazel's rules can be used to declare different types of targets in your Bazel project. Each of these rules is a thin wrapper around a corresponding builtin Bazel rule producing the equivalent C++ target.

The main tool to build nanobind extensions is the `nanobind_extension` rule.

nanobind_extension()

Declares a Bazel target representing a nanobind extension, which contains the Python bindings of your C++ code.

```
def nanobind_extension(
    name,
    domain = "",
    srcs = [],
    copts = [],
    deps = [],
    local_defines = [],
    **kwargs):
```

It corresponds directly to the builtin cc_binary rule, with all keyword arguments being directly forwarded to a `cc_binary` target.

The `domain` argument can be used to build the target extension under a different ABI domain, as described in the *FAQ* section.

To generate typing stubs for an extension, you can use the `nanobind_stubgen` rule.

nanobind_stubgen()

Declares a Bazel target for generating a stub file from a previously built nanobind bindings extension.

```
def nanobind_stubgen(
    name,
    module,
    output_file = None,
    imports = [],
    pattern_file = None,
    marker_file = None,
    include_private_members = False,
    exclude_docstrings = False):
```

It generates a py_binary rule with a corresponding runfiles distribution, which invokes nanobind's builtin stubgen script, outputs a stub file and, optionally, a typing marker file into the build output directory (commonly called "bindir" in Bazel terms).

All arguments (except the name, which is used only to refer to the target in Bazel) correspond directly to nanobind's stubgen command line interface, which is described in more detail in the *typing documentation*.

New in nanobind-bazel version 2.1.0.

To build a C++ library with nanobind as a dependency, use the `nanobind_library` rule.

nanobind_library()

Declares a Bazel target representing a C++ library depending on nanobind.

```
def nanobind_library(
    name,
    copts = [],
    deps = [],
    **kwargs):
```

It corresponds directly to the builtin cc_library rule, with all keyword arguments being directly forwarded to a `cc_library` target.

To build a C++ shared library with nanobind as a dependency, use the `nanobind_shared_library` rule.

nanobind_shared_library()

Declares a Bazel target representing a C++ shared library depending on nanobind.

```
def nanobind_shared_library(
    name,
    deps = [],
    **kwargs):
```

It corresponds directly to the builtin cc_shared_library rule, with all keyword arguments being directly forwarded to a `cc_shared_library` target.

New in nanobind-bazel version 2.1.0.

To build a C++ test target requiring nanobind, use the `nanobind_test` rule.

nanobind_test()

Declares a Bazel target representing a C++ test depending on nanobind.

```
def nanobind_test(
    name,
    copts = [],
    deps = [],
    **kwargs):
```

It corresponds directly to the builtin cc_test rule, with all keyword arguments being directly forwarded to a `cc_test` target.

27.2 Flags

To customize some of nanobind's build options, nanobind-bazel exposes the following flag settings.

@nanobind_bazel//:minsize (boolean)

Apply nanobind's size optimizations to the built extensions. Size optimizations are turned on by default, similarly to the CMake build. To turn off size optimizations, you can use the shorthand notation `--no@nanobind_bazel//:minsize`.

@nanobind_bazel//:py-limited-api (string)

Build nanobind extensions against the stable ABI of the configured Python version. Allowed values are `"cp312"`, `"cp313"`, which target the stable ABI starting from Python 3.12 or 3.13, respectively. By default, all extensions are built without any ABI limitations.

`@nanobind_bazel//:free_threading (boolean)`

Build nanobind extensions with a Python toolchain in free-threaded mode. If given, the currently configured Python toolchain must support free-threading, otherwise, the build will result in a compilation error. Only relevant for CPython 3.13+, since support for free-threaded Python was introduced in CPython 3.13. For more information on free-threaded extension support in nanobind, refer to the relevant *documentation section*.

New in nanobind-bazel version 2.2.0.

INDEX