cppyy Documentation

Release 1.6.0

Wim Lavrijsen

Feb 10, 2020

Contents

cppyy is an automatic, run-time, Python-C++ bindings generator, for calling C++ from Python and Python from C++. Run-time generation enables detailed specialization for higher performance, lazy loading for reduced memory use in large scale projects, Python-side cross-inheritance and callbacks for working with C++ frameworks, run-time template instantiation, automatic object downcasting, exception mapping, and interactive exploration of C++ libraries. cppyy delivers this without any language extensions, intermediate languages, or the need for boiler-plate hand-written code. For design and performance, see this PyHPC paper, albeit that the CPython/cppyy performance has been vastly improved since.

cppyy is based on Cling, the C++ interpreter, to match Python's dynamism, interactivity, and run-time behavior. Consider this session, showing dynamic, interactive, mixing of C++ and Python features (more examples are in the tutorial):

```
>>> import cppyy
>>> cppyy.cppdef("""
... class MyClass {
... public:
...     MyClass(int i) : m_data(i) {}
...     virtual ~MyClass() {}
...     virtual int add_int(int i) { return m_data + i; }
...     int m_data;
... };""")
True
>>> from cppyy.gbl import MyClass
>>> m = MyClass(42)
>>> cppyy.cppdef("""
... void say_hello(MyClass* m) {
...     std::cout << "Hello, the number is: " << m->m_data << std::endl;
... }""")
True
>>> MyClass.say_hello = cppyy.gbl.say_hello
>>> m.say_hello()
Hello, the number is: 42
>>> m.m_data = 13
>>> m.say_hello()
Hello, the number is: 13
>>> class PyMyClass(MyClass):
...     def add_int(self, i):  # python side override (CPython only)
...         return self.m_data + 2*i
...
>>> cppyy.cppdef("int callback(MyClass* m, int i) { return m->add_int(i); }")
True
>>> cppyy.gbl.callback(m, 2)             # calls C++ add_int
15
>>> cppyy.gbl.callback(PyMyClass(1), 2)  # calls Python-side override
5
>>>
```

With a modern C++ compiler having its back, cppyy is future-proof. Consider the following session using `boost::any`, a capsule-type that allows for heterogeneous containers in C++. The Boost library is well known for its no holds barred use of modern C++ and heavy use of templates:

```
>>> import cppyy
>>> cppyy.include('boost/any.hpp')
>>> from cppyy.gbl import std, boost
>>> val = boost.any()                     # the capsule
>>> val.__assign__(std.vector[int]())     # assign it a std::vector<int>
<cppyy.gbl.boost.any object at 0xf6a8a0>
```

(continues on next page)

(continued from previous page)

```
>>> val.type() == cppyy.typeid(std.vector[int])     # verify type
True
>>> extract = boost.any_cast[int](std.move(val))    # wrong cast
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
cppyy.gbl.boost.bad_any_cast: Could not instantiate any_cast<int>:
  int boost::any_cast(boost::any&& operand) =>
    wrapexcept<boost::bad_any_cast>: boost::bad_any_cast: failed conversion using
↪boost::any_cast
>>> extract = boost.any_cast[std.vector[int]](val) # correct cast
>>> type(extract) is std.vector[int]
True
>>> extract += xrange(100)
>>> len(extract)
100
>>> val.__assign__(std.move(extract))     # move forced
<cppyy.gbl.boost.any object at 0xf6a8a0>
>>> len(extract)                          # now empty (or invalid)
0
>>> extract = boost.any_cast[std.vector[int]](val)
>>> list(extract)
[0, 1, 2, 3, 4, 5, 6, ..., 97, 98, 99]
>>>
```

Of course, there is no reason to use Boost from Python (in fact, this example calls out for *pythonizations*), but it shows that cppyy seamlessly supports many advanced C++ features.

cppyy is available for both CPython (v2 and v3) and PyPy, reaching C++-like performance with the latter. It makes judicious use of precompiled headers, dynamic loading, and lazy instantiation, to support C++ programs consisting of millions of lines of code and many thousands of classes. cppyy minimizes dependencies to allow its use in distributed, heterogeneous, development environments.

CHAPTER 1

Changelog

For convenience, this changelog keeps tracks of changes with version numbers of the main cppyy package, but many of the actual changes are in the lower level packages, which have their own releases. See *packages*, for details on the package structure. PyPy support lags CPython support.

1.1 Master: 1.6.2

- Respect __len__ when using bound C++ objects in boolean expressions
- Support UTF-8 encoded unicode through std::string
- Support for std::byte
- Enable assignment to function pointer variable
- Allow passing cppyy.nullptr where are function pointer is expected
- Disable copy construction into constructed object (use `__assign__` instead)
- Lower priority of implicit conversion to temporary with initializer_list ctor
- Remove '-isystem' from makepch flags
- Extended documentation

1.2 2020-01-04: 1.6.1

- Mapped C++ exception reporting detailing
- Mapped C++ exception cleanup bug fix
- STL vector constructor passes the CPython sequence construction
- STL vector slicing passes the CPython sequence slicing tests
- Extended documentation

1.3 2019-12-23: 1.6.0

- Classes derived from std::exception can be used as Python exceptions
- Template handling detailing (for Eigen)
- Support keyword arguments
- Added add_library_path at module level
- Extended documentation
- Fix regression bugs: #176, #179, #180, #182

1.4 2019-11-07: 1.5.7

- Allow implicit converions for move arguments
- Choose vector over initializer_list if part of the template argument list

1.5 2019-11-03: 1.5.6

- Added public C++ API for some CPyCppyy core functions (CPython only)
- Support for char16_t/char16_t* and char32_t/char32_t*
- Respect std::hash in __hash__
- Fix iteration over vector of shared_ptr
- Length checking on global variables of type 'signed char[N]'
- Properly support overloaded templated with non-templated __setitem__
- Support for array of const char* as C-strings
- Enable type resolution of clang's builtin __type_pack_element
- Fix for inner class type naming when it directly declares a variable

1.6 2019-10-16: 1.5.5

- Added signal -> exception support in cppyy.ll
- Support for lazily combining overloads of operator*/+-
- No longer call trivial destructors
- Support for free function unary operators
- Refactored and optimized operator==/!= usage
- Refactored converters/executors for lower memory usage
- Bug fixes in rootcling and _cppyy_generator.py

1.7 2019-09-25: 1.5.4

- operator+/* now respect C++-side associativity
- Fix potential crash if modules are reloaded
- Fix some portability issues on Mac/Windows of cppyy-cling

1.8 2019-09-15: 1.5.3

- Performance improvements
- Support for anonymous/unnamed/nested unions
- Extended documentation

1.9 2019-09-06: 1.5.2

- Added a "low level" interface (cppyy.ll) for hard-casting and ll types
- Extened support for passing ctypes arguments through ptr, ref, ptr-ptr
- Fixed crash when creating an array of instances of a scoped inner struct
- Extended documentation

1.10 2019-08-26: 1.5.1

- Upgrade cppyy-cling to 6.18.2
- Various patches to upstream's pre-compiled header generation and use
- Instantiate templates with larger integer types if argument values require
- Improve cppyy.interactive and partially enable it on PyPy, IPython, etc.
- Let __overload__ be more flexible in signature matching
- Make list filtering of dir(cppyy.gbl) on Windows same as Linux/Mac
- Extended documentation

1.11 2019-08-18: 1.5.0

- Upgrade cppyy-cling to 6.18.0
- Allow python-derived classes to be used in templates
- Stricter template resolution and better caching/performance
- Detailed memory management for make_shared and shared_ptr
- Two-way memory management for cross-inherited objects
- Reduced memory footprint of proxy objects in most common cases

- Allow implicit conversion from a tuple of arguments
- Data set on namespaces reflected on C++ even if data not yet bound
- Generalized resolution of binary operators in wrapper generation
- Proper naming of arguments in namespaces for std::function<>
- Cover more cases of STL-liker iterators
- Allow std::vector initialization with a list of constructor arguments
- Consistent naming of `__cppname__` to `__cpp_name__`
- Added __set_lifeline__ attribute to overloads
- Fixes to the cmake fragments for Ubuntu
- Fixes linker errors on Windows in some configurations
- Support C++ naming of typedef of bool types
- Basic views of 2D arrays of builtin types
- Extended documentation

1.12 2019-07-01 : 1.4.12

- Automatic conversion of python functions to std::function arguments
- Fix for templated operators that can map to different python names
- Fix on p3 crash when setting a detailed exception during exception handling
- Fix lookup of std::nullopt
- Fix bug that prevented certain templated constructors from being considered
- Support for enum values as data members on “enum class” enums
- Support for implicit conversion when passing by-value

1.13 2019-05-23 : 1.4.11

- Workaround for JITed RTTI lookup failures on 64b MS Windows
- Improved overload resolution between f(void*) and f<>(T*)
- Minimal support for char16_t (Windows) and char32_t (Linux/Mac)
- Do not unnecessarily autocast smart pointers

1.14 2019-05-13 : 1.4.10

- Imported several FindCppyy.cmake improvements from Camille’s cppyy-bbhash
- Fixes to cppyy-generator for unresolved templates, void, etc.
- Fixes in typedef parsing for template arguments in unknown namespaces
- Fix in templated operator code generation

- Fixed ref-counting error for instantiated template methods

1.15 2019-04-25 : 1.4.9

- Fix import error on pypy-c

1.16 2019-04-22 : 1.4.8

- std::tuple is now iterable for return assignments w/o tie
- Support for opaque handles and typedefs of pointers to classes
- Keep unresolved enums desugared and provide generic converters
- Treat int8_t and uint8_t as integers (even when they are chars)
- Fix lookup of enum values in global namespace
- Backported name mangling (esp. for static/global data lookup) for 32b Windows
- Fixed more linker problems with malloc on 64b Windows
- Consistency in buffer length calculations and c_int/c_uint handling on Windows
- Properly resolve overloaded functions with using of templates from bases
- Get templated constructor info from decl instead of name comparison
- Fixed a performance regression for free functions.

1.17 2019-04-04 : 1.4.7

- Enable initializer_list conversion on Windows as well
- Improved mapping of operator() for indexing (e.g. for matrices)
- Implicit conversion no longer uses global state to prevent recursion
- Improved overload reordering
- Fixes for templated constructors in namespaces

1.18 2019-04-02 : 1.4.6

- More transparent use of smart pointers such as shared_ptr
- Expose versioned std namespace through using on Mac
- Improved error handling and interface checking in cross-inheritance
- Argument of (const/non-const) ref types support in callbacks/cross-inheritance
- Do template argument resolution in order: reference, pointer, value
- Fix for return type deduction of resolved but uninstantiated templates
- Fix wrapper generation for defaulted arguments of private types

- Several linker fixes on 64b Windows

1.19 2019-03-25 : 1.4.5

- Allow templated free functions to be attached as methods to classes
- Allow cross-derivation from templated classes
- More support for 'using' declarations (methods and inner namespaces)
- Fix overload resolution for std::set::rbegin()/rend() operator ==
- Fixes for bugs #61, #67
- Several pointer truncation fixes for 64b Windows
- Linker and lookup fixes for Windows

1.20 2019-03-20 : 1.4.4

- Support for 'using' of namespaces
- Improved support for alias templates
- Faster template lookup
- Have rootcling/genreflex respect compile-time flags (except for –std if overridden by CLING_EXTRA_FLAGS)
- Utility to build dictionarys on Windows (32/64)
- Name mangling fixes in Cling for JITed global/static variables on Windows
- Several pointer truncation fixes for 64b Windows

1.21 2019-03-10 : 1.4.3

- Cross-inheritance from abstract C++ base classes
- Preserve 'const' when overriding virtual functions
- Support for by-ref (using ctypes) for function callbacks
- Identity of nested typedef'd classes matches actual
- Expose function pointer variables as std::function's
- More descriptive printout of global functions
- Ensure that standard pch is up-to-date and that it is removed on uninstall
- Remove standard pch from wheels on all platforms
- Add -cxxflags option to rootcling
- Install clang resource directory on Windows

CHAPTER 2

License and copyright

2.1 Additional copyright holders

In addition to LBNL/UC Berkeley, this package contains files copyrighted by one or more of the following people and organizations, and licensed under the same conditions (except for some compatible licenses as retained in the source code):

- CERN
- Simone Bacchio
- Robert Bradshaw
- Ellis Breen
- Antonio Cuni
- Aditi Dutta
- Shaheed Haque
- Jonsomi
- Max Kolin
- Alvaro Moran
- Tarmo Pikaro
- Matti Picus
- Camille Scott
- Toby StClere-Smithe
- Stefan Wunsch

Conda-forge recipes were provided by Julian Rueth and Isuru Fernando.

2.2 External code

The create_src_directory.py script will pull in ROOT and LLVM sources, which are licensed differently:

LLVM: distributed under University of Illinois/NCSA Open Source License https://opensource.org/licenses/UoI-NCSA.php

ROOT: distributed under LGPL 2.1 https://root.cern.ch/license

The ROOT and LLVM/Clang codes are modified/patched, as part of the build process.

CHAPTER 3

Installation

cppyy requires a (modern) C++ compiler. When installing through conda-forge, conda will install the compiler for you, to match the other conda-forge packages. When using pip and the wheels from PyPI, you minimally need gcc5, clang5, or MSVC'17. When installing from source, the only requirement is full support for C++11 (e.g. minimum gcc 4.8.1 on GNU/Linux), but older compilers than the ones listed for the wheels have not been tested.

With CPython on Linux or MacOS, probably by far the easiest way to install cppyy, is through conda-forge on Anaconda (or miniconda). A Windows recipe for conda is not available yet, but is forthcoming, so use pip for that platform for now (see below). PyPI always has the authoritative releases (conda-forge pulls the sources from there), so conda-forge may sometimes lag PyPI. If you absolutely need the latest release, use PyPI or consider *building from source*.

To install using conda, create and/or activate your (new) work environment and install from the conda-forge channel:

```
$ conda create -n WORK
$ conda activate WORK
(WORK) $ conda install -c conda-forge cppyy
(WORK) [current compiler] $
```

To install with pip through PyPI, it is recommend to use virtualenv (or module venv for modern pythons). The use of virtualenv prevents pollution of any system directories and allows you to wipe out the full installation simply by removing the virtualenv created directory ("WORK" in this example):

```
$ virtualenv WORK
$ source WORK/bin/activate
(WORK) $ python -m pip install cppyy
(WORK) $
```

If you use the `--user` option to pip and use pip directly on the command line, instead of through python, make sure that the PATH envar points to the bin directory that will contain the installed entry points during the installation, as the build process needs them. You may also need to install wheel first if you have an older version of pip and/or do not use virtualenv (which installs wheel by default). Example:

```
$ python -m pip install wheel --user
$ PATH=$HOME/.local/bin:$PATH python -m pip install cppyy --user
```

3.1 Wheels on PyPI

Wheels for the backend (`cppyy-cling`) are available on PyPI for GNU/Linux, MacOS-X, and MS Windows (both 32b and 64b).

The Linux wheels are built on manylinux, but with gcc 5.5, not the 4.8.2 that ships with manylinux1, since cppyy exposes C++ APIs and g++ introduced ABI incompatibilities starting with its 5 series forward. Using 4.8.2 would have meant that any software using cppyy would have to be (re)compiled for the older gcc ABI, which the odds don't favor. Note that building cppyy fully with 4.8.2 (and requiring the old ABI across the board) does work.

The wheels for MS Windows were build with MSVC Community Edition 2017.

There are no wheels for the `CPyCppyy` and `cppyy` packages, to allow the C++ standard chosen to match the local compiler.

3.2 pip with conda

Although installing `cppyy` through conda-forge is recommended, it is possible to build/install with `pip` under Anaconda/miniconda.

Typical Python extensions only expose a C interface for use through the Python C-API, requiring only calling conventions (and the Python C-API version, of course) to match to be binary compatible. Here, cppyy differs because it exposes C++ APIs: it thus requires a C++ run-time that is ABI compatible with the C++ compiler that was used during build-time.

A set of modern compilers is available through conda-forge, but are only intended for use with `conda-build`. In particular, the corresponding run-time is installed (for use through rpath when building), but not set up. That is, the conda compilers are added to `PATH` but not their libraries to `LD_LIBRARY_PATH` (Mac, Linux; `PATH` for both on MS Windows). Thus, you get the conda compilers and your system libraries mixed in the same build environment, unless you set `LD_LIBRARY_PATH` (`PATH` on Windows) explicitly, e.g. by adding `$CONDA_PREFIX/lib`. Note that the conda documentation recommends against this. Furthermore, the compilers from conda-forge are not vanilla distributions: header files have been modified, which can can lead to parsing problems if your system C library does not support C11, for example.

Nevertheless, with the above caveats, if your system C/C++ run-times are new enough, the following can be made to work:

```
$ conda create -n WORK
$ conda activate WORK
(WORK) $ conda install python
(WORK) $ conda install -c conda-forge compilers
(WORK) [current compiler] $ python -m pip install cppyy
```

3.3 C++ standard with pip

The C++17 standard is the default for Mac and Linux (both PyPI and conda-forge); but it is C++14 for MS Windows (compiler limitation). When installing from PyPI using `pip`, you can control the standard selection by setting the `STDCXX` envar to '17', '14', or '11' (for Linux, the backend does not need to be recompiled). Note that the build will lower your choice if the compiler used does not support a newer standard.

3.4 Install from source

To build an existing release from source, tell `pip` to not download any binary wheels. Build-time only dependencies are `cmake` (for general build), `python` (obviously, but also for LLVM), and a modern C++ compiler (one that supports at least C++11). Use the envar `STDCXX` to control the C++ standard version; `MAKE` to change the `make` command, `MAKE_NPROCS` to control the maximum number of parallel jobs allowed, and `VERBOSE=1` to see full build/compile commands. Example (using `--verbose` to see `pip` progress):

```
$ STDCXX=17 MAKE_NPROCS=32 pip install --verbose cppyy --no-binary=cppyy-cling
```

Compilation of the backend, which contains a customized version of Clang/LLVM, can take a long time, so by default the setup script will use all cores (x2 if hyperthreading is enabled). Once built, however, the wheel of `cppyy-cling` is reused by pip for all versions of CPython and for PyPy, thus the long compilation is needed only once for all different versions of Python on the same machine.

See the *section on repos* for more details/options.

3.5 PyPy

PyPy 5.7 and 5.8 have a built-in module `cppyy`. You can still install the cppyy package, but the built-in module takes precedence. To use cppyy, first import a compatibility module:

```
$ pypy
[PyPy 5.8.0 with GCC 5.4.0] on linux2
>>>> import cppyy_compat, cppyy
>>>>
```

You may have to set `LD_LIBRARY_PATH` appropriately if you get an `EnvironmentError` (it will indicate the needed directory).

Note that your python interpreter (whether CPython or `pypy-c`) may not have been linked by the C++ compiler. This can lead to problems during loading of C++ libraries and program shutdown. In that case, re-linking is highly recommended.

Very old versions of PyPy (5.6.0 and earlier) have a built-in `cppyy` based on Reflex, which is less feature-rich and no longer supported. However, both the *distribution utilities* and user-facing Python codes are very backwards compatible, making migration straightforward.

3.6 Precompiled header

For performance reasons (reduced memory and CPU usage), a precompiled header (PCH) of the system and compiler header files will be installed or, failing that, generated on startup. Obviously, this PCH is not portable and should not be part of any wheel.

Some compiler features, such as AVX, OpenMP, fast math, etc. need to be active during compilation of the PCH, as they depend both on compiler flags and system headers (for intrinsics, or API calls). You can control compiler flags through the `EXTRA_CLING_ARGS` envar and thus what is active in the PCH. In principle, you can also change the C++ language standard by setting the appropriate flag on `EXTRA_CLING_ARGS` and rebuilding the PCH. However, if done at this stage, that disables some automatic conversion for C++ types that were introduced after C++11 (such as `string_view` and `optional`).

If you want multiple PCHs living side-by-side, you can generate them yourself (note that the given path must be absolute):

```
>>> import cppyy_backend.loader as l
>>> l.set_cling_compile_options(True)         # adds defaults to EXTRA_CLING_ARGS
>>> install_path = '/full/path/to/target/location/for/PCH'
>>> l.ensure_precompiled_header(install_path)
```

You can then select the appropriate PCH with the `CLING_STANDARD_PCH` envar:

```
$ export CLING_STANDARD_PCH=/full/path/to/target/location/for/PCH/allDict.cxx.pch
```

Or disable it completely by setting that envar to “none”.

CHAPTER 4

Trying it out

This is a basic guide to try cppyy and see whether it works for you. Large code bases will benefit from more advanced features such as *pythonizations* for a cleaner interface to clients; precompiled modules for faster parsing and reduced memory usage; "*dictionaries*" to package locations and manage dependencies; and mapping files for automatic, lazy, loading. You can, however, get very far with just the basics and it may even be completely sufficient for small packages with fewer classes.

cppyy works by parsing C++ definitions through `cling`, generating tiny wrapper codes to honor compile-time features and create standardized interfaces, then compiling/linking those wrappers with the `clang` JIT. It thus requires only those two ingredients: *C++ definitions* and *linker symbols*. All cppyy uses, the basic and the more advanced, are variations on the theme of bringing these two together at the point of use.

Definitions typically live in header files and symbols in libraries. Headers can be loaded with `cppyy.include` and libraries with the `cppyy.load_library` call. Loading the header is sufficient to start exploring, with `cppyy.gbl` the starting point of all things C++, while the linker symbols are only needed at the point of first use.

Here is an example using the zlib library, which is likely available on your system:

```
>>> import cppyy
>>> cppyy.include('zlib.h')          # bring in C++ definitions
>>> cppyy.load_library('libz')       # load linker symbols
>>> cppyy.gbl.zlibVersion()          # use a zlib API
'1.2.11'
>>>
```

Since header files can include other header files, it is easy to aggregate all relevant ones into a single header to include. If there are project-specific include paths, you can add those paths through `cppyy.add_include_path`. If a header is C-only and not set for use with C++, use `cppyy.c_include`, which adds `extern "C"` around the header.

Library files can be aggregated by linking all relevant ones to a single library to load. Using the linker for this purpose allows regular system features such as `rpath` and envars such as `LD_LIBRARY_PATH` to be applied as usual. Note that any mechanism that exposes the library symbols will work. For example, you could also use the standard module `ctypes` through `ctypes.CDLL` with the `ctypes.RTLD_GLOBAL` option.

To explore, start from `cppyy.gbl` to access your namespaces, classes, functions, etc., etc. directly; or use python's `dir` (or tab-completion) to see what is available. Use python's `help` to see list the methods and data members of classes and see the interfaces of functions.

Now try this out for some of your own headers, libraries, and APIs!

CHAPTER 5

Examples

The detailed feature lists have examples that work using a header file, and there is the tutorial that shows mixing of C++ and Python interactively. More complete examples that show packaging include these repos (in alphabetical order):

- cppyy-bbhash
- PyEtaler
- gmpxxyy
- cppyy-knearestneighbors
- libsemigroups_cppyy
- python-vspline

In addition, cppyy is being re-integrated in its roots, which includes several sophisticated pythonizations. See the "experimental" version of the PyROOT project.

CHAPTER 6

Bugs and feedback

Please report bugs, ask questions, request improvements, and post general comments on the issue tracker or on stack overflow.

CHAPTER 7

Basic types

C++ has a far richer set of builtin types than Python. Most Python code can remain relatively agnostic to that, and `cppyy` provides automatic conversions as appropriate. On the other hand, Python builtin types such as lists and maps are far richer than any builtin types in C++. These are mapped to their Standard Template Library equivalents instead.

The C++ code used for the examples below can be found *here*, and it is assumed that that code is loaded at the start of any session. Download it, save it under the name `features.h`, and load it:

```
>>> import cppyy
>>> cppyy.include('features.h')
>>>
```

7.1 *Builtins*

The selection of builtin data types varies greatly between Python and C++. Where possible, builtin data types map onto the expected equivalent Python types, with the caveats that there may be size differences, different precision or rounding, etc. For example, a C++ `float` is returned as a Python `float`, which is in fact a C++ `double`. If sizes allow, conversions are automatic. For example, a C++ `unsigned int` becomes a Python2 `long` or Python3 `int`, but unsigned-ness is still honored:

```
>>> cppyy.gbl.gUint
0L
>>> type(cppyy.gbl.gUint)
<type 'long'>
>>> cppyy.gbl.gUint = -1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot convert negative integer to unsigned
>>>
```

Some types are builtin in Python, but (STL) classes in C++. Examples are `str` vs. `std::string` and `complex` vs. `std::complex`. These classes have been pythonized to behave the same wherever possible. For example, string comparison work directly, and `std::complex` has `real` and `imag` properties:

```
>>> c = cppyy.gbl.std.complex['double'](1, 2)
>>> c
(1+2j)
>>> c.real, c.imag
(1.0, 2.0)
>>> s = cppyy.gbl.std.string("aap")
>>> type(s)
<class cppyy.gbl.std.string at 0x7fa75edbf8a0>
>>> s == "aap"
True
>>>
```

To pass an argument through a C++ `char` (signed or unsigned) use a Python string of size 1. In many cases, the explicit C types from module `ctypes` can also be used, but that module does not have a public API (for type conversion or otherwise), so support is somewhat limited.

There are automatic conversions between C++'s `std::vector` and Python's `list` and `tuple`, where possible, as they are often used in a similar manner. These datatypes have completely different memory layouts, however, and the `std::vector` requires that all elements are of the same type and laid out consecutively in memory. Conversion thus requires type checks, memory allocation, and copies. This can be rather expensive. See the section on *STL*.

7.2 Arrays

Builtin arrays are supported through arrays from module `array` (or any other builtin-type array that implements the Python buffer interface, such as numpy arrays) and a low-level view type from `cppyy` for returns and variable access (that implements the buffer interface as well). Out-of-bounds checking is limited to those cases where the size is known at compile time. Example:

```
>>> from cppyy.gbl import Concrete
>>> from array import array
>>> c = Concrete()
>>> c.array_method(array('d', [1., 2., 3., 4.]), 4)
1 2 3 4
>>> c.m_data[4] # static size is 4, so out of bounds
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: buffer index out of range
>>>
```

Arrays of arrays are supported through the C++ low-level view objects. This only works well if sizes are known at compile time or can be inferred. If sizes are not known, the size is set to a large integer (depending on the array element size) to allow access. It is then up to the developer not to access the array out-of-bounds. There is limited support for arrays of instances, but those should be avoided in C++ anyway:

```
>>> cppyy.cppdef('std::string str_array[3][2] = {{"aa", "bb"}, {"cc", "dd"},
↪{"ee", "ff"}};')
True
>>> type(cppyy.gbl.str_array[0][1])
<class cppyy.gbl.std.string at 0x7fd650ccb650>
>>> cppyy.gbl.str_array[0][1]
'bb'
>>> cppyy.gbl.str_array[4][0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(continues on next page)

(continued from previous page)

```
IndexError: tuple index out of range
>>>
```

7.3 *Pointers*

When the C++ code takes a pointer or reference type to a specific builtin type (such as an `unsigned int` for example), then types need to match exactly. `cppyy` supports the types provided by the standard modules `ctypes` and `array` for those cases. Example of using a reference to builtin:

```
>>> from ctypes import c_uint
>>> u = c_uint(0)
>>> c.uint_ref_assign(u, 42)
>>> u.value
42
>>>
```

For objects, an object, a pointer to an object, and a smart pointer to an object are represented the same way, with the necessary (de)referencing applied automatically. Pointer variables are also bound by reference, so that updates on either the C++ or Python side are reflected on the other side as well.

7.4 *Enums*

Named, anonymous, and class enums are supported. The Python-underlying type of an enum is implementation dependent and may even be different for different enums on the same compiler. Typically, however, the types are `int` or `unsigned int`, which translates to Python's `int` or `long` on Python2 or class `int` on Python3. Separate from the underlying, all enums have their own Python type to allow them to be used in template instantiations:

```
>>> from cppyy.gbl import kBanana   # classic enum, globally available
>>> print(kBanana)
29
>>> cppyy.gbl.EFruit
<class '__main__.EFruit'>
>>> print(cppyy.gbl.EFruit.kApple)
78
>>> cppyy.gbl.E1                    # C++11 class enum, scoped
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: <namespace cppyy.gbl at 0x7ff2766a4af0> has no attribute 'E1
↪'.
>>> cppyy.gbl.NamedClassEnum.E1
42
>>>
```

CHAPTER 8

Classes

Both Python and C++ support object-oriented code through classes and thus it is logical to expose C++ classes as Python ones, including the full inheritance hierarchy.

The C++ code used for the examples below can be found *here*, and it is assumed that that code is loaded at the start of any session. Download it, save it under the name `features.h`, and load it:

```
>>> import cppyy
>>> cppyy.include('features.h')
>>>
```

8.1 *Basics*

All bound C++ code starts off from the global C++ namespace, represented in Python by `gbl`. This namespace, as any other namespace, is treated as a module after it has been loaded. Thus, we can import C++ classes that live underneath it:

```
>>> from cppyy.gbl import Concrete
>>> Concrete
<class cppyy.gbl.Concrete at 0x2058e30>
>>>
```

Placing classes in the same structure as imposed by C++ guarantees identity, even if multiple Python modules bind the same class. There is, however, no necessity to expose that structure to end-users: when developing a Python package that exposes C++ classes through `cppyy`, consider `cppyy.gbl` an “internal” module, and expose the classes in any structure you see fit. The C++ names will continue to follow the C++ structure, however, as is needed for e.g. pickling:

```
>>> from cppyy.gbl import Namespace
>>> Concrete == Namespace.Concrete
False
>>> n = Namespace.Concrete.NestedClass()
>>> type(n)
```

(continues on next page)

(continued from previous page)

```
<class cppyy.gbl.Namespace.Concrete.NestedClass at 0x22114c0>
>>> type(n).__name__
NestedClass
>>> type(n).__module__
cppyy.gbl.Namespace.Concrete
>>> type(n).__cpp_name__
Namespace::Concrete::NestedClass
>>>
```

8.2 *Constructors*

Python and C++ both make a distinction between allocation (`__new__` in Python, `operator new` in C++) and initialization (`__init__` in Python, the constructor call in C++). When binding, however, there comes a subtle semantic difference: the Python `__new__` allocates memory for the proxy object only, and `__init__` initializes the proxy by creating or binding the C++ object. Thus, no C++ memory is allocated until `__init__`. The advantages are simple: the proxy can now check whether it is initialized, because the pointer to C++ memory will be NULL if not; it can be a reference to another proxy holding the actual C++ memory; and it can now transparently implement a C++ smart pointer. If `__init__` is never called, eg. when a call to the base class `__init__` is missing in a derived class override, then accessing the proxy will result in a Python `ReferenceError` exception.

8.3 *Destructors*

There should not be a reason to call a destructor directly in CPython, but PyPy uses a garbage collector and that makes it sometimes useful to destruct a C++ object where you want it destroyed. Destructors are accessible through the conventional `__destruct__` method. Accessing an object after it has been destroyed will result in a Python `ReferenceError` exception.

8.4 *Inheritance*

The output of help shows the inheritance hierarchy, constructors, public methods, and public data. For example, `Concrete` inherits from `Abstract` and it has a constructor that takes an `int` argument, with a default value of 42. Consider:

```
>>> from cppyy.gbl import Abstract
>>> issubclass(Concrete, Abstract)
True
>>> a = Abstract()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: cannot instantiate abstract class 'Abstract'
>>> c = Concrete()
>>> isinstance(c, Concrete)
True
>>> isinstance(c, Abstract)
True
>>> d = Concrete(13)
>>>
```

Just like in C++, interface classes that define pure virtual methods, such as `Abstract` does, can not be instantiated, but their concrete implementations can. As the output of `help` showed, the `Concrete` constructor takes an integer argument, that by default is 42.

8.5 *Cross-inheritance*

Python classes that derive from C++ classes can override virtual methods as long as those methods are declared on class instantiation (adding methods to the Python class after the fact will not provide overrides on the C++ side, only on the Python side). Example:

```
>>> from cppyy.gbl import Abstract, call_abstract_method
>>> class PyConcrete(Abstract):
...     def abstract_method(self):
...         print("Hello, Python World!\n")
...     def concrete_method(self):
...         pass
...
>>> pc = PyConcrete()
>>> call_abstract_method(pc)
Hello, Python World!
>>>
```

Note that it is not necessary to provide a constructor (`__init__`), but if you do, you *must* call the base class constructor through the `super` mechanism.

8.6 *Methods*

C++ methods are represented as Python ones: these are first-class objects and can be bound to an instance. If a method is virtual in C++, the proper concrete method is called, whether or not the concrete class is bound. Similarly, if all classes are bound, the normal Python rules apply:

```
>>> c.abstract_method()
called Concrete::abstract_method
>>> c.concrete_method()
called Concrete::concrete_method
>>> m = c.abstract_method
>>> m()
called Concrete::abstract_method
>>>
```

8.7 *Data members*

Data members are implemented as properties, using descriptors. For example, The `Concrete` instances have a public data member `m_int`:

```
>>> c.m_int, d.m_int
(42, 13)
>>>
```

Note however, that the data members are typed: setting them results in a memory write on the C++ side. This is different in Python, where references are replaced, and thus any type will do:

```
>>> c.m_int = 3.14   # a float does not fit in an int
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: int/long conversion expects an integer object
>>> c.m_int = int(3.14)
>>> c.m_int, d.m_int
(3, 13)
>>>
```

Private and protected data members are not accessible, contrary to Python data members, and C++ const-ness is respected:

```
>>> c.m_const_int = 71    # declared 'const int' in class definition
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: assignment to const data not allowed
>>>
```

Static C++ data members act like Python class-level data members. They are also represented by property objects and both read and write access behave as expected:

```
>>> Concrete.s_int       # access through class
321
>>> c.s_int = 123        # access through instance
>>> Concrete.s_int
123
```

8.8 *Operators*

Many C++ operators can be mapped to their Python equivalent. When the operators are part of the C++ class definition, this is done directly. If they are defined globally, the lookup is done lazily (ie. can resolve after the class definition by loading the global definition or by defining them interactively). Some operators have no Python equivalent and are instead made available by mapping them onto the following conventional functions:

C++	Python
`operator=`	`__assign__`
`operator++(int)`	`__postinc__`
`operator++()`	`__preinc__`
`operator--(int)`	`__postdec__`
`operator--()`	`__predec__`
`unary operator*`	`__deref__`
`operator->`	`__follow__`
`operator&&`	`__dand__`
`operator||`	`__dor__`
`operator,`	`__comma__`

Here is an example of operator usage, using STL iterators directly (note that this is not necessary in practice as STL and STL-like containers work transparently in Python for-loops):

```
>>> v = cppyy.gbl.std.vector[int](range(3))
>>> i = v.begin()
>>> while (i != v.end()):
...     print(i.__deref__())
...     _ = i.__preinc__()
...
0
1
2
>>>
```

Overridden `operator new` and `operator delete`, as well as their array equivalents, are not accessible but will be called as appropriate.

8.9 *Templates*

Templated classes are instantiated using square brackets. (For backwards compatibility reasons, parentheses work as well.) The instantiation of a templated class yields a class, which can then be used to create instances.

Templated classes need not pre-exist in the bound code, just their declaration needs to be available. This is true for e.g. all of STL:

```
>>> cppyy.gbl.std.vector                # template metatype
<cppyy.Template 'std::vector' object at 0x7fffed2674d0>
>>> cppyy.gbl.std.vector(int)           # instantiates template -> class
<class cppyy.gbl.std.vector<int> at 0x1532190>
cppyy.gbl.std.vector[int]()             # instantiates class -> object
<cppyy.gbl.std.vector<int> object at 0x2341ec0>
>>>
```

The template arguments may be actual types or their names as a string, whichever is more convenient. Thus, the following are equivalent:

```
>>> from cppyy.gbl.std import vector
>>> type1 = vector[Concrete]
>>> type2 = vector['Concrete']
>>> type1 == type2
True
>>>
```

8.10 *Typedefs*

Typedefs are simple python references to the actual classes to which they refer.

```
>>> from cppyy.gbl import Concrete_t
>>> Concrete is Concrete_t
True
>>>
```

CHAPTER 9

Functions

C++ functions are first-class objects in Python and can be used wherever Python functions can be used, including for dynamically constructing classes.

The C++ code used for the examples below can be found *here*, and it is assumed that that code is loaded at the start of any session. Download it, save it under the name `features.h`, and load it:

```
>>> import cppyy
>>> cppyy.include('features.h')
>>>
```

Function argument type conversions follow the expected rules, with implicit conversions allowed, including between Python builtin types and STL types, but it is rather more efficient to make conversions explicit.

9.1 *Free functions*

All bound C++ code starts off from the global C++ namespace, represented in Python by `gbl`. This namespace, as any other namespace, is treated as a module after it has been loaded. Thus, we can directly import C++ functions from it and other namespaces that themselves may contain more functions. All lookups on namespaces are done lazily, thus if loading more headers bring in more functions (incl. new overloads), these become available dynamically.

```
>>> from cppyy.gbl import global_function, Namespace
>>> global_function == Namespace.global_function
False
>>> from cppyy.gbl.Namespace import global_function
>>> global_function == Namespace.global_function
True
>>> from cppyy.gbl import global_function
>>>
```

Free functions can be bound to a class, following the same rules as apply to Python functions: unless marked as static, they will turn into member functions when bound to an instance, but act as static functions when called through the class. Consider this example:

```
>>> from cppyy.gbl import Concrete, call_abstract_method
>>> c = Concrete()
>>> Concrete.callit = call_abstract_method
>>> Concrete.callit(c)
called Concrete::abstract_method
>>> c.callit()
called Concrete::abstract_method
>>> Concrete.callit = staticmethod(call_abstract_method)
>>> c.callit()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: void ::call_abstract_method(Abstract* a) =>
    TypeError: takes at least 1 arguments (0 given)
>>> c.callit(c)
called Concrete::abstract_method
>>>
```

9.2 *Static methods*

Class static functions are treated the same way as free functions, except that they are accessible either through the class or through an instance, just like Python's `staticmethod`.

9.3 *Methods*

For class methods, see the *methods section* under the *classes heading*.

9.4 *Operators*

Globally defined operators are found lazily (ie. can resolve after the class definition by loading the global definition or by defining them interactively) and are mapped onto a Python equivalent when possible. See the *operators section* under the *classes heading* for more details.

9.5 *Templates*

Templated functions (and class methods) can either be called using square brackets (`[]`) to provide the template arguments explicitly, or called directly, through automatic lookup. The template arguments may either be a string of type names (this results in faster code, as it needs no further lookup/verification) or a list of the actual types to use (which tends to be more convenient).

Note: the Python type `float` maps to the C++ type `float`, even as Python uses a C `double` as its internal representation. The motivation is that doing so makes the Python code more readable (and Python may anyway change its internal representation in the future). The same has been true for Python `int`, which used to be a C `long` internally.

Examples, using multiply from *features.h*:

```
>>> mul = cppyy.gbl.multiply
>>> mul(1, 2)
2
```

(continues on next page)

(continued from previous page)

```
>>> mul(1., 5)
5.0
>>> mul[int](1, 1)
1
>>> mul[int, int](1, 1)
1
>>> mul[int, int, float](1, 1)
1.0
>>> mul[int, int](1, 'a')
 TypeError: Template method resolution failed:
 none of the 6 overloaded methods succeeded. Full details:
 int ::multiply(int a, int b) =>
   TypeError: could not convert argument 2 (int/long conversion expects an␣
↪integer object)
 ...
 Failed to instantiate "multiply(int,std::string)"
>>> mul['double, double, double'](1., 5)
5.0
>>>
```

9.6 *Overloading*

C++ supports overloading, whereas Python supports "duck typing", thus C++ overloads have to be selected dynamically in response to the available "ducks". This may lead to additional lookups or template instantiations. However, pre-existing methods (incl. auto-instantiated methods) are always preferred over new template instantiations:

```
>>> global_function(1.)         # selects 'double' overload
2.718281828459045
>>> global_function(1)          # selects 'int' overload
42
>>>
```

C++ does a static dispatch at compile time based on the argument types. The dispatch is a selection among overloads (incl. templates) visible at that point in the *translation unit*. Bound C++ in Python does a dynamic dispatch: it considers all overloads visible *globally* at that point in the execution. Because the dispatch is fundamentally different (albeit in line with the expectation of the respective languages), differences can occur. Especially if overloads live in different header files and are only an implicit conversion apart, or if types that have no direct equivalent in Python, such as e.g. `unsigned short`, are used.

There are two rounds to finding an overload. If all overloads fail argument conversion during the first round, where implicit conversions are not allowed, _and_ at least one converter has indicated that it can do implicit conversions, a second round is tried. In this second round, implicit conversions are allowed, including class instantiation of temporaries. During some template calls, implicit conversions are not allowed at all, to make sure new instantiations happen instead.

In the rare occasion where the automatic overload selection fails, the `__overload__` function can be called to access a specific overload matching a specific function signature:

```
>>> global_function.__overload__('double')(1)    # int implicitly converted
2.718281828459045
>>>
```

Note that `__overload__` only does a lookup; it performs no (implicit) conversions. To see all available overloads, use `help()` or look at the `__doc__` string of the function:

```
>>> print(global_function.__doc__)
int ::global_function(int)
double ::global_function(double)
>>>
```

9.7 *Return values*

Most return types are readily amenable to automatic memory management: builtin returns, by-value returns, (const-)reference returns to internal data, smart pointers, etc. The important exception is pointer returns.

A function that returns a pointer to an object over which Python should claim ownership, should have its `__creates__` flag set through its *pythonization*. Well-written APIs will have clear clues in their naming convention about the ownership rules. For example, functions called `New...`, `Clone...`, etc. can be expected to return freshly allocated objects. A simple name-matching in the pythonization then makes it simple to mark all these functions as creators.

The return values are *auto-casted*.

9.8 **args and **kwds*

C++ default arguments work as expected. Keywords, however, are a Python language feature that does not exist in C++. Many C++ function declarations do have formal arguments, but these are not part of the C++ interface (the argument names are repeated in the definition, making the names in the declaration irrelevant: they do not even need to be provided). Thus, although `cppyy` will map keyword argument names to formal argument names from the C++ declaration, use of this feature is not recommended unless you have a guarantee that the names in C++ the interface are maintained. Example:

```
>>> from cppyy.gbl import Concrete
>>> c = Concrete()       # uses default argument
>>> c.m_int
42
>>> c = Concrete(13)     # uses provided argument
>>> c.m_int
13
>>> args = (27,)
>>> c = Concrete(*args)  # argument pack
>>> c.m_int
27
>>> c = Concrete(n=17)
>>> c.m_int
17
>>> kwds = {'n' : 18}
>>> c = Concrete(**kwds)
>>> c.m_int
18
>>>
```

9.9 *Callbacks*

Python callables (functions/lambdas/instances) can be passed to C++ through function pointers and/or `std::function`. This involves creation of a temporary wrapper, which has the same life time as the Python callable it wraps, so the callable needs to be kept alive on the Python side if the C++ side stores the callback. Example:

```
>>> from cppyy.gbl import call_int_int
>>> print(call_int_int.__doc__)
int ::call_int_int(int(*)(int,int) f, int i1, int i2)
>>> def add(a, b):
...     return a+b
...
>>> call_int_int(add, 3, 7)
7
>>> call_int_int(lambda x, y: x*y, 3, 7)
21
>>>
```

CHAPTER 10

Type conversions

Most type conversions are done automatically, e.g. between Python `str` and C++ `std::string` and `const char*`, but low-level APIs exist to perform explicit conversions.

The C++ code used for the examples below can be found *here*, and it is assumed that that code is loaded at the start of any session. Download it, save it under the name `features.h`, and load it:

```
>>> import cppyy
>>> cppyy.include('features.h')
>>>
```

10.1 *Auto-casting*

Object pointer returns from functions provide the most derived class known (i.e. exposed in header files) in the hierarchy of the object being returned. This is important to preserve object identity as well as to make casting, a pure C++ feature after all, superfluous. Example:

```
>>> from cppyy.gbl import Abstract, Concrete
>>> c = Concrete()
>>> Concrete.show_autocast.__doc__
'Abstract* Concrete::show_autocast()'
>>> d = c.show_autocast()
>>> type(d)
<class '__main__.Concrete'>
>>>
```

As a consequence, if your C++ classes should only be used through their interfaces, then no bindings should be provided to the concrete classes (e.g. by excluding them using a *selection file*). Otherwise, more functionality will be available in Python than in C++.

Sometimes, however, full control over a cast is needed. For example, if the instance is bound by another tool or even a 3rd party, hand-written, extension library. Assuming the object supports the `PyCapsule` or `CObject` abstraction, then a C++-style reinterpret_cast (i.e. without implicitly taking offsets into account), can be done by taking and rebinding the address of an object:

```
>>> from cppyy import addressof, bind_object
>>> e = bind_object(addressof(d), Abstract)
>>> type(e)
<class '__main__.Abstract'>
>>>
```

10.2 *Operators*

If conversion operators are defined in the C++ class and a Python equivalent exists (i.e. all builtin integer and floating point types, as well as `bool`), then these will map onto those Python conversions. Note that `char*` is mapped onto `__str__`. Example:

```
>>> from cppyy.gbl import Concrete
>>> print(Concrete())
Hello operator const char*!
>>>
```

C++ code can overload conversion operators by providing methods in a class or global functions. Special care needs to be taken for the latter: first, make sure that they are actually available in some header file. Second, make sure that headers are loaded in the desired order. I.e. that these global overloads are available before use.

CHAPTER 11

STL

Parts of the Standard Template Library (STL), in particular its container types, are the de facto equivalent of Python's builtin types. STL is written in C++ and Python bindings of it are fully functional as-is, but are much more useful when pluggable into idiomatic expressions where Python builtin containers are expected (e.g. in list contractions).

There are two extremes to achieve such drop-in behavior: copy into Python builtins, so that the Python-side always deals with true Python objects; or adjust the C++ interfaces to be the same as their Python equivalents. Neither is very satisfactory: the former is not because of the existence of global/static variables and return-by-reference. If only a copy is available, then expected modifications do not propagate. Copying is also either slow (when copying every time) or memory intensive (if the results are cached). Filling out the interfaces may look more appealing, but all operations then involve C++ function calls, which can be slower than the Python equivalents, and C++-style error handling.

Given that neither choice will satisfy all cases, `cppyy` aims to maximize functionality and minimum surprises based on common use. Thus, for example, `std::vector` grows a pythonistic `__len__` method, but does not lose its C++ `size` method. Passing a Python container through a const reference to a `std::vector` will trigger automatic conversion, but such an attempt through a non-const reference will fail since a non-temporary C++ object is required[1] to return any updates/changes.

`std::string` is almost always converted to Python's `str` on function returns (the exception is return-by-reference when assigning), but not when its direct use is more likely such as in the case of (global) variables or when iterating over a `std::vector<std::string>`.

The rest of this section shows examples of how STL containers can be used in a natural, pythonistic, way.

11.1 *vector*

A `std::vector` is the most commonly used C++ container type because it is more efficient and performant than specialized types such as `list` and `map`, unless the number of elements gets very large. Python has several similar types, from the builtin `tuple` and `list`, the `array` from builtin module `array`, to "as-good-as-builtin" `numpy.ndarray`. A vector is more like the latter two in that it can contain only one type, but more like the former two in that

[1] The meaning of "temporary" differs between Python and C++: in a statement such as `func(std.vector[int]((1, 2, 3)))`, there is no temporary as far as Python is concerned, even as there clearly is in the case of a similar statement in C++. Thus that call will succeed even if `func` takes a non-const reference.

it can contain objects. In practice, it can interplay well with all these containers, but e.g. efficiency and performance can differ significantly.

A vector can be instantiated from any sequence, including generators, and vectors of objects can be recursively constructed:

```
>>> from cppyy.gbl.std import vector, pair
>>> v = vector[int](range(10))
>>> len(v)
10
>>> vp = vector[pair[int, int]](((1, 2), (3, 4)))
>>> len(vp)
2
>>> vp[1][0]
3
>>>
```

To extend a vector in-place with another sequence object, use +=, just as would work for Python's list:

```
>>> v += range(10, 20)
>>> len(v)
20
>>>
```

The easiest way to print the full contents of a vector, is by using a list and printing that instead. Indexing and slicing of a vector follows the normal Python slicing rules:

```
>>> v[1]
1
>>> v[-1]
19
>>> v[-4:]
<cppyy.gbl.std.vector<int> object at 0x7f9051057650>
>>> list(v[-4:])
[16, 17, 18, 19]
>>>
```

The usual iteration operations work on vector, but the C++ rules still apply, so a vector that is being iterated over can *not* be modified in the loop body. (On the plus side, this makes it much faster to iterate over a vector than, say, a numpy ndarray.)

```
>>> for i in v[2:5]:
...     print(i)
...
2
3
4
>>> 2 in v
True
>>> sum(v)
190
>>>
```

When a function takes a non-l-value (const-ref, move, or by-value) vector as a parameter, another sequence can be used and cppyy will automatically generate a temporary. Typically, this will be faster than coding up such a temporary on the Python side, but if the same sequence is used multiple times, creating a temporary once and re-using it will be the most efficient approach.o

```
>>> cppyy.cppdef("""
... int sumit1(const std::vector<int>& data) {
...   return std::accumulate(data.begin(), data.end(), 0);
... }
... int sumit2(std::vector<int> data) {
...   return std::accumulate(data.begin(), data.end(), 0);
... }
... int sumit3(const std::vector<int>&& data) {
...   return std::accumulate(data.begin(), data.end(), 0);
... }""")
...
True
>>> cppyy.gbl.sumit1(range(5))
10
>>> cppyy.gbl.sumit2(range(6))
16
>>> cppyy.gbl.sumit3(range(7))
21
>>>
```

The temporary vector is created using the vector constructor taking an `std::initializer_list`, which is more flexible than constructing a temporary vector and filling it: it allows the data in the container to be implicitly converted (e.g. from `int` to `double` type, or from pointer to derived to pointer to base class). As a consequence, however, with STL containers being allowed where Python containers are, this in turn means that you can pass e.g. an `std::vector<int>` (or `std::list<int>`) where a `std::vector<double>` is expected and a temporary is allowed:

```
>>> cppyy.cppdef("""
... double sumit4(const std::vector<double>& data) {
...   return std::accumulate(data.begin(), data.end(), 0);
... }""")
...
True
>>> cppyy.gbl.sumit4(vector[int](range(7)))
21.0
>>>
```

Normal overload resolution rules continue to apply, however, thus if an overload were available that takes an `const std::vector<int>&`, it would be preferred.

When templates are involved, overload resolution is stricter, to ensure that a better matching instantiation is preferred over an implicit conversion. However, that does mean that as-is, C++ is actually more flexible: it has the curly braces initializer syntax to explicitly infer an `std::initializer_list`, with no such equivalent in Python.

Although in general this approach guarantees the intended result, it does put some strictures on the Python side, requiring careful use of types. However, an easily fixable error is preferable over an implicitly wrong result. Note the type of the init argument in the call resulting in an (attempted) implicit instantiation in the following example:

```
>>> cppyy.cppdef("""
... template<class T>
... T sumit_T(const std::vector<T>& data, T init) {
...  return std::accumulate(data.begin(), data.end(), init);
... }""")
...
True
>>> cppyy.gbl.sumit_T(vector['double'](range(7)), 0)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
TypeError: Template method resolution failed:
  Failed to instantiate "sumit_T(std::vector<double>&,int)"
  Failed to instantiate "sumit_T(std::vector<double>*,int)"
  Failed to instantiate "sumit_T(std::vector<double>,int)"
>>> cppyy.gbl.sumit_T(vector['double'](range(7)), 0.)
21.0
>>>
```

To be sure, the code is *too* strict in the simplistic example above, and with a future version of Cling it should be possible to lift some of these restrictions without causing incorrect results.

CHAPTER 12

Exceptions

All C++ exceptions are converted to Python exceptions and all Python exceptions are converted to C++ exceptions, to allow exception propagation through multiple levels of callbacks, while retaining the option to handle the outstanding exception as needed in either language. To preserve an exception across the language boundaries, it must derive from `std::exception`. If preserving the exception (or its type) is not possible, generic exceptions are used to propagate the exception: `Exception` in Python or `CPyCppyy::PyException` in C++.

In the most common case of an instance of a C++ exception class derived from `std::exception` that is thrown from a compiled library and which is copyable, the exception can be caught and handled like any other bound C++ object (or with `Exception` on the Python and `std::exception` on the C++ side). If the exception is not copyable, but derived from `std::exception`, the result of its `what()` reported with an instance of Python's `Exception`. In all other cases, including exceptions thrown from interpreted code (due to limitations of the Clang JIT), the exception will turn into an instance of `Exception` with a generic message.

The standard C++ exceptions are explicitly not mapped onto standard Python exceptions, since other than a few simple cases, the mapping is too crude to be useful as the typical usage in each standard library is too different. Thus, for example, a thrown `std::runtime_error` instance will become a `cppyy.gbl.std.runtime_error` instance on the Python side (with Python's `Exception` as its base class), not a `RuntimeError` instance.

The C++ code used for the examples below can be found *here*, and it is assumed that that code is loaded at the start of any session. Download it, save it under the name `features.h`, and load it:

```
>>> import cppyy
>>> cppyy.include('features.h')
>>>
```

In addition, the examples require the `throw` to be in compiled code. Save the following and build it into a shared library `libfeatures.so` (or `libfeatures.dll` on MS Windows):

```
#include "features.h"

void throw_an_error(int i) {
    if (i) throw SomeError{"this is an error"};
    throw SomeOtherError{"this is another error"};
}
```

And load the resulting library:

```
>>> cppyy.load_library('libfeatures')
>>>
```

Then try it out:

```
>>> cppyy.gbl.throw_an_error(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
cppyy.gbl.SomeError: void ::throw_an_error(int i) =>
    SomeError: this is an error
>>>
```

Note how the full type is preserved and how the result of `what()` is used for printing the exception. By preserving the full C++ type, it is possible to call any other member functions the exception may provide beyond `what` or access any additional data it carries.

To catch the exception, you can either use the full type, or any of its base classes, including `Exception` and `cppyy.gbl.std.exception`:

```
>>> try:
...     cppyy.gbl.throw_an_error(0)
... except cppyy.gbl.SomeOtherError as e:  # catch by exact type
...     print("received:", e)
...
received: <cppyy.gbl.SomeOtherError object at 0x7f9e11d3db10>
>>> try:
...     cppyy.gbl.throw_an_error(0)
... except Exception as e:                 # catch through base class
...     print("received:", e)
...
received: <cppyy.gbl.SomeOtherError object at 0x7f9e11e00310>
>>>
```

CHAPTER 13

Python

The C++ code used for the examples below can be found *here*, and it is assumed that that code is loaded at the start of any session. Download it, save it under the name `features.h`, and load it:

```
>>> import cppyy
>>> cppyy.include('features.h')
>>>
```

13.1 *PyObject*

Arguments and return types of `PyObject*` can be used, and passed on to CPython API calls (or through `cpyext` in PyPy).

13.2 *Doc strings*

The documentation string of a method or function contains the C++ arguments and return types of all overloads of that name, as applicable. Example:

```
>>> from cppyy.gbl import Concrete
>>> print Concrete.array_method.__doc__
void Concrete::array_method(int* ad, int size)
void Concrete::array_method(double* ad, int size)
>>>
```

13.3 *Help*

Bound C++ class is first-class Python and can thus be inspected like any Python objects can. For example, we can ask for `help()`:

```
>>> help(Concrete)
Help on class Concrete in module gbl:

class Concrete(Abstract)
 |  Method resolution order:
 |      Concrete
 |      Abstract
 |      CPPInstance
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __assign__(self, const Concrete&)
 |      Concrete& Concrete::operator=(const Concrete&)
 |
 |  __init__(self, *args)
 |      Concrete::Concrete(int n = 42)
 |      Concrete::Concrete(const Concrete&)
 |
 etc. ....
```

CHAPTER 14

Low-level code

C code and older C++ code sometimes makes use of low-level features such as pointers to builtin types, some of which do not have any Python equivalent (e.g. `unsigned short*`). Furthermore, such codes tend to be ambiguous: the information from header file is not sufficient to determine the full purpose. For example, an `int*` type may refer to the address of a single `int` (an out-parameter, say) or it may refer to an array of `int`, the ownership of which is not clear either. cppyy provides a few low-level helpers and integration with the Python ctypes module to cover these cases.

Use of these low-level helpers will obviously lead to very "C-like" code and it is recommended to *pythonize* the code, perhaps using the Cling JIT and embedded C++.

Note: the low-level module is not loaded by default (since its use is, or should be, uncommon). It needs to be imported explicitly:

```
>>> import cppyy.ll
>>>
```

14.1 *C/C++ casts*

C++ instances are auto-casted to the most derived available type, so do not require explicit casts even when a function returns a pointer to a base class or interface. However, when given only a `void*` or `intptr_t` type on return, a cast is required to turn it into something usable.

- **bind_object**: This is the preferred method to proxy a C++ address, and lives in `cppyy`, not `cppyy.ll`, as it is not a low-level C++ cast, but a `cppyy` API that is also used internally. It thus plays well with object identity, references, etc. Example:

```
>>> cppyy.cppdef("""
... struct MyStruct { int fInt; };
... void* create_mystruct() { return new MyStruct{42}; }
... """)
...
>>> s = cppyy.gbl.create_mystruct()
```

(continues on next page)

(continued from previous page)

```
>>> print(s)
<cppyy.LowLevelView object at 0x10559d430>
>>> sobj = cppyy.bind_object(s, 'MyStruct')
>>> print(sobj)
<cppyy.gbl.MyStruct object at 0x7ff25e28eb20>
>>> print(sobj.fInt)
42
>>>
```

Instead of the type name as a string, `bind_object` can also take the actual class (here: `cppyy.gbl.MyStruct`).

- **Typed nullptr**: A Python side proxy can pass through a pointer to pointer function argument, but if the C++ side allocates memory and stores it in the pointer, the result is a memory leak. In that case, use `bind_object` to bind `cppyy.nullptr` instead, to get a typed nullptr to pass to the function. Example (continuing from the example above):

```
>>> cppyy.cppdef("""
... void create_mystruct(MyStruct** ptr) { *ptr = new MyStruct{42}; }
... """)
...
>>> s = cppyy.bind_object(cppyy.nullptr, 'MyStruct')
>>> print(s)
<cppyy.gbl.MyStruct object at 0x0>
>>> cppyy.gbl.create_mystruct(s)
>>> print(s)
<cppyy.gbl.MyStruct object at 0x7fc7d85b91c0>
>>> print(s.fInt)
42
>>>
```

- **C-style cast**: This is the simplest option for builtin types. The syntax is "template-style", example:

```
>>> cppyy.cppdef("""
... void* get_data(int sz) {
... int* iptr = (int*)malloc(sizeof(int)*sz);
... for (int i=0; i<sz; ++i) iptr[i] = i;
... return iptr;
... }""")
...
>>> NDATA = 4
>>> d = cppyy.gbl.get_data(NDATA)
>>> print(d)
<cppyy.LowLevelView object at 0x1068cba30>
>>> d = cppyy.ll.cast['int*'](d)
>>> d.reshape((NDATA,))
>>> print(list(d))
[0, 1, 2, 3]
>>>
```

- **C++-style casts**: Similar to the C-style cast, there are `ll.static_cast` and `ll.reinterpret_cast`. There should never be a reason for a `dynamic_cast`, since that only applies to objects, for which auto-casting will work. The syntax is "template-style", just like for the C-style cast above.

14.2 *NumPy casts*

The `cppyy.LowLevelView` type returned for pointers to basic types, including for `void*`, is a simple and light-weight view on memory, given a pointer, type, and number of elements (or unchecked, if unknown). It only supports basic operations such as indexing and iterations, but also the buffer protocol for integration with full-fledged functional arrays such as NumPy's `ndarray`.

In addition, specifically when dealing with `void*` returns, you can use NumPy's low-level `frombuffer` interface to perform the cast. Example:

```
>>> cppyy.cppdef("""
... void* create_float_array(int sz) {
...     float* pf = (float*)malloc(sizeof(float)*sz);
...     for (int i = 0; i < sz; ++i) pf[i] = 2*i;
...     return pf;
... }""")
...
>>> import numpy as np
>>> NDATA = 8
>>> arr = cppyy.gbl.create_float_array(NDATA)
>>> print(arr)
<cppyy.LowLevelView object at 0x109f15230>
>>> arr.reshape((NDATA,))    # adjust the llv's size
>>> v = np.frombuffer(arr, dtype=np.float32, count=NDATA)   # cast to float
>>> print(len(v))
8
>>> print(v)
array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14.], dtype=float32)
>>>
```

Note that NumPy will internally check the total buffer size, so if the size you are casting *to* is larger than the size you are casting *from*, then the number of elements set in the `reshape` call needs to be adjusted accordingly.

14.3 *Capsules*

It is not possible to pass proxies from cppyy through function arguments of another binder (and vice versa, with the exception of `ctypes`, see below), because each will use a different internal representation, including for type checking and extracting the C++ object address. However, all Python binders are able to rebind (just like `bind_object` above for cppyy) the result of at least one of the following:

- **ll.addressof**: Takes a cppyy bound C++ object and returns its address as an integer value. Takes an optional `byref` parameter and if set to true, returns a pointer to the address instead.
- **ll.as_capsule**: Takes a cppyy bound C++ object and returns its address as a PyCapsule object. Takes an optional `byref` parameter and if set to true, returns a pointer to the address instead.
- **ll.as_cobject**: Takes a cppyy bound C++ object and returns its address as a PyCObject object for Python2 and a PyCapsule object for Python3. Takes an optional `byref` parameter and if set to true, returns a pointer to the address instead.
- **as_ctypes**: Takes a cppyy bound C++ object and returns its address as a `ctypes.c_void_p` object. Takes an optional `byref` parameter and if set to true, returns a pointer to the address instead.

14.4 *ctypes*

The ctypes module has been part of Python since version 2.5 and provides a Python-side foreign function interface. It is clunky to use and has very bad performance, but it is guaranteed to be available. It does not have a public C interface, only the Python one, but its internals have been stable since its introduction, making it safe to use for tight and efficient integration at the C level (with a few Python helpers to assure lazy lookup).

Objects from `ctypes` can be passed through arguments of functions that take a pointer to a single C++ builtin, and `ctypes` pointers can be passed when a pointer-to-pointer is expected, e.g. for array out-parameters. This leads to the following set of possible mappings:

C++	ctypes
by value (ex.: `int`)	`.value` (ex.: `c_int(0).value`)
by const reference (ex.: `const int&`)	`.value` (ex.: `c_int(0).value`)
by reference (ex.: `int&`)	direct (ex.: `c_int(0)`)
by pointer (ex.: `int*`)	direct (ex.: `c_int(0)`)
by ptr-ref (ex.: `int*&`)	`pointer` (ex.: `pointer(c_int(0))`)
by ptr-ptr **in** (ex.: `int**`)	`pointer` (ex.: `pointer(c_int(0))`)
by ptr-ptr **out** (ex.: `int**`)	`POINTER` (ex.: `POINTER(c_int)()`)

The `ctypes` pointer objects (from `POINTER`, `pointer`, or `byref`) can also be used for pass by reference or pointer, instead of the direct object, and `ctypes.c_void_p` can pass through all pointer types. The addresses will be adjusted internally by cppyy.

Note that `ctypes.c_char_p` is expected to be a NULL-terminated C string, not a character array (see the ctypes module documentation), and that `ctypes.c_bool` is a C `_Bool` type, not C++ `bool`.

14.5 *Memory*

C++ has three ways of allocating heap memory (`malloc`, `new`, and `new[]`) and three corresponding ways of deallocation (`free`, `delete`, and `delete[]`). Direct use of `malloc` and `new` should be avoided for C++ classes, as these may override `operator new` to control their allocation own. However these low-level allocators can be necessary for builtin types on occassion if the C++ side takes ownership (otherwise, prefer either `array` from the builtin module `array` or `ndarray` from Numpy).

The low-level module adds the following functions:

- **ll.malloc**: an interface on top of C's malloc. Use it as a template with the number of elements (not the number types) to be allocated. The result is a `cppyy.LowLevelView` with the proper type and size:

  ```
  >>> arr = cppyy.ll.malloc[int](4)    # allocates memory for 4 C ints
  >>> print(len(arr))
  4
  >>> print(type(arr[0]))
  <type 'int'>
  >>>
  ```

  The actual C malloc can also be used directly, through `cppyy.gbl.malloc`, taking the number of *bytes* to be allocated and returning a `void*`.

- **ll.free**: an interface to C's free, to deallocate memory allocated by C's malloc. To continue to example above:

  ```
  >>> cppyy.ll.free(arr)
  >>>
  ```

The actual C free can also be used directly, through `cppyy.gbl.free`.

- **ll.array_new**: an interface on top of C++'s `new[]`. Use it as a template; the result is a `cppyy.LowLevelView` with the proper type and size:

```
>>> arr = cppyy.ll.array_new[int](4)    # allocates memory for 4 C ints
>>> print(len(arr))
4
>>> print(type(arr[0]))
<type 'int'>
>>>
```

- **ll.array_delete**: an interface on top of C++'s `delete[]`. To continue to example above:

```
>>> cppyy.ll.array_delete(arr)
>>>
```

CHAPTER 15

Miscellaneous

15.1 File features.h

```cpp
#include <cmath>
#include <iostream>
#include <vector>


//-----
unsigned int gUint = 0;

//-----
class Abstract {
public:
    virtual ~Abstract() {}
    virtual void abstract_method() = 0;
    virtual void concrete_method() = 0;
};

void Abstract::concrete_method() {
    std::cout << "called Abstract::concrete_method" << std::endl;
}

//-----
class Concrete : Abstract {
public:
    Concrete(int n=42) : m_int(n), m_const_int(17) {}
    ~Concrete() {}

    virtual void abstract_method() {
        std::cout << "called Concrete::abstract_method" << std::endl;
    }

    virtual void concrete_method() {
```

(continues on next page)

(continued from previous page)

```
        std::cout << "called Concrete::concrete_method" << std::endl;
    }

    void array_method(int* ad, int size) {
        for (int i=0; i < size; ++i)
            std::cout << ad[i] << ' ';
        std::cout << '\n';
    }

    void array_method(double* ad, int size) {
        for (int i=0; i < size; ++i)
            std::cout << ad[i] << ' ';
        std::cout << '\n';
    }

    void uint_ref_assign(unsigned int& target, unsigned int value) {
        target = value;
    }

    Abstract* show_autocast() {
        return this;
    }

    operator const char*() {
        return "Hello operator const char*!";
    }

public:
    double m_data[4];
    int m_int;
    const int m_const_int;

    static int s_int;
};

typedef Concrete Concrete_t;

int Concrete::s_int = 321;

void call_abstract_method(Abstract* a) {
    a->abstract_method();
}

//-----
int global_function(int) {
    return 42;
}

double global_function(double) {
    return std::exp(1);
}

int call_int_int(int (*f)(int, int), int i1, int i2) {
    return f(i1, i2);
}

template<class A, class B, class C = A>
```

(continues on next page)

(continued from previous page)

```cpp
C multiply(A a, B b) {
    return C{a*b};
}

//-----
namespace Namespace {

    class Concrete {
    public:
        class NestedClass {
        public:
            std::vector<int> m_v;
        };

    };

    int global_function(int i) {
        return 2*::global_function(i);
    }

    double global_function(double d) {
        return 2*::global_function(d);
    }

} // namespace Namespace

//-----
enum EFruit {kApple=78, kBanana=29, kCitrus=34};
enum class NamedClassEnum { E1 = 42 };

//-----
void throw_an_error(int i);

class SomeError : public std::exception {
public:
    explicit SomeError(const std::string& msg) : fMsg(msg) {}
    const char* what() const throw() override { return fMsg.c_str(); }

private:
    std::string fMsg;
};

class SomeOtherError : public SomeError {
public:
    explicit SomeOtherError(const std::string& msg) : SomeError(msg) {}
    SomeOtherError(const SomeOtherError& s) : SomeError(s) {}
};
```

This is a collection of a few more features listed that do not have a proper place yet in the rest of the documentation.

The C++ code used for the examples below can be found *here*, and it is assumed that that code is loaded at the start of any session. Download it, save it under the name `features.h`, and load it:

```python
>>> import cppyy
>>> cppyy.include('features.h')
>>>
```

15.2 *Special variables*

There are several conventional "special variables" that control behavior of functions or provide (internal) information. Often, these can be set/used in pythonizations to handle memory management or Global Interpreter Lock (GIL) release.

- `__python_owns__`: a flag that every bound instance carries and determines whether Python or C++ owns the C++ instance (and associated memory). If Python owns the instance, it will be destructed when the last Python reference to the proxy disappears. You can check/change the ownership with the __python_owns__ flag that every bound instance carries. Example:

```
>>> from cppyy.gbl import Concrete
>>> c = Concrete()
>>> c.__python_owns__         # True: object created in Python
True
>>>
```

- `__creates__`: a flag that every C++ overload carries and determines whether the return value is owned by C++ or Python: if `True`, Python owns the return value, otherwise C++.
- `__set_lifeline__`: a flag that every C++ overload carries and determines whether the return value should place a back-reference on `self`, to prevent the latter from going out of scope before the return value does. The default is `False`, but will be automatically set at run-time if a return value's address is a C++ object pointing into the memory of `this`, or if `self` is a by-value return.
- `__release_gil__`: a flag that every C++ overload carries and determines whether the Global Interpreter Lock (GIL) should be released during the C++ call to allow multi-threading. The default is `False`.
- `__useffi__`: a flag that every C++ overload carries and determines whether generated wrappers or direct foreign functions should be used. This is for PyPy only; the flag has no effect on CPython.
- `__sig2exc__`: a flag that every C++ overload carries and determines whether C++ signals (such as SIGABRT) should be converted into Python exceptions.
- `__cppname__`: a string that every C++ bound class carries and contains the actual C++ name (as opposed to `__name__` which has the Python name). This can be useful for template instantiations, documentation, etc.

15.3 *STL algorithms*

It is usually easier to use a Python equivalent or code up the effect of an STL algorithm directly, but when operating on a large container, calling an STL algorithm may offer better performance. It is important to note that all STL algorithms are templates and need the correct types to be properly instantiated. STL containers offer typedefs to obtain those exact types and these should be used rather than relying on the usual implicit conversions of Python types to C++ ones. For example, as there is no `char` type in Python, the `std::remove` call below can not be instantiated using a Python string, but the `std::string::value_type` must be used instead:

```
>>> cppstr = cppyy.gbl.std.string
>>> n = cppstr('this is a C++ string')
>>> print(n)
this is a C++ string
>>> n.erase(cppyy.gbl.std.remove(n.begin(), n.end(), cppstr.value_type(' ')))
<cppyy.gbl.__wrap_iter<char*> object at 0x7fba35d1af50>
>>> print(n)
thisisaC++stringing
>>>
```

15.4 *Reduced typing*

Typing `cppyy.gbl` all the time gets old rather quickly, but the dynamic nature of `cppyy` makes something like `from cppyy.gbl import *` impossible. For example, classes can be defined dynamically after that statement and then they would be missed by the import. In scripts, it is easy enough to rebind names to achieve a good amount of reduction in typing (and a modest performance improvement to boot, because of fewer dictionary lookups), e.g.:

```
import cppyy
std = cppyy.gbl.std
v = std.vector[int](range(10))
```

But even such rebinding becomes annoying for (brief) interactive sessions.

For CPython only (and not with tools such as IPython or in IDEs that replace the interactive prompt), there is a fix, using `from cppyy.interactive import *`. This makes lookups in the global dictionary of the current frame also consider everything under `cppyy.gbl`. This feature comes with a performance *penalty* and is not meant for production code. Example usage:

```
>>> from cppyy.interactive import *
>>> v = std.vector[int](range(10))
>>> print(list(v))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> cppdef("struct SomeStruct {};")
True
>>> s = SomeStruct()          # <- dynamically made available
>>> s
<cppyy.gbl.SomeStruct object at 0x7fa9b8624320>
>>>
```

For PyPy, IPython, etc. `cppyy.gbl` is simply rebound as `g` and `cppyy.gbl.std` is made available as `std`. Not as convenient as full lookup, and missing any other namespaces that may be available, but still saves some typing in may cases.

15.5 *Odds and ends*

- **namespaces**: Are represented as python classes. Namespaces are more open-ended than classes, so sometimes initial access may result in updates as data and functions are looked up and constructed lazily. Thus the result of `dir()` on a namespace shows the classes and functions available for binding, even if these may not have been created yet. Once created, namespaces are registered as modules, to allow importing from them. The global namespace is `cppyy.gbl`.
- **NULL**: Is represented as `cppyy.nullptr`. Starting C++11, the keyword `nullptr` is used to represent `NULL`. For clarity of intent, it is recommended to use this instead of `None` (or the integer `0`, which can serve in some cases), as `None` is better understood as `void` in C++.

CHAPTER 16

Debugging

By default, the `clang` JIT as used by cppyy does not generate debugging information. This is first of all because it has proven to be not reliable in all cases, but also because in a production setting this information, being internal to the wrapper generation, goes unused. However, that does mean that a debugger that starts from python will not be able to step through JITed code into the C++ function that needs debugging, even when such information is available for that C++ function.

To enable debugging information in JITed code, set the `EXTRA_CLING_ARGS` envar to `-g` (and any further compiler options you need, e.g. add `-O2` to debug optimized code).

On a crash in C++, the backend will attempt to provide a stack trace. This works quite well on Linux (through `gdb`) and decently on MacOS (through `unwind`), but is currently unreliable on MS Windows. To prevent printing of this trace, which can be slow to produce, set the envar `CPPYY_CRASH_QUIET` to '1'.

It is even more useful to obtain a traceback through the Python code that led up to the problem in C++. Many modern debuggers allow mixed-mode C++/Python debugging (for example gdb and MSVC), but cppyy can also turn abortive C++ signals (such as a segmentation violation) into Python exceptions, yielding a normal traceback. This is particularly useful when working with cross-inheritance and other cross-language callbacks.

To enable the signals to exceptions conversion, import the lowlevel module `cppyy.ll` and use:

```
import cppyy.ll
cppyy.ll.set_signals_as_exception(True)
```

Call `set_signals_as_exception(False)` to disable the conversion again. It is recommended to only have the conversion enabled around the problematic code, as it comes with a performance penalty. If the problem can be localized to a specific function, you can use its `__sig2exc__` flag to only have the conversion active in that function. Finally, for convenient scoping, you can also use:

```
with cppyy.ll.signals_as_exception():
    # crashing code goes here
```

The translation of signals to exceptions is as follows (all of the exceptions are subclasses of `cppyy.ll.FatalError`):

C++ signal	Python exception
SIGSEGV	cppyy.ll.SegmentationViolation
SIGBUS	cppyy.ll.BusError
SIGABRT	cppyy.ll.AbortSignal
SIGILL	cppyy.ll.IllegalInstruction

As an example, consider the following cross-inheritance code that crashes with a segmentation violation in C++, because a `nullptr` is dereferenced:

```
import cppyy
import cppyy.ll

cppyy.cppdef("""
   class Base {
   public:
      virtual ~Base() {}
      virtual int runit() = 0;
   };

   int callback(Base* b) {
       return b->runit();
   }

   void segfault(int* i) { *i = 42; }
""")

class Derived(cppyy.gbl.Base):
    def runit(self):
        print("Hi, from Python!")
        cppyy.gbl.segfault(cppyy.nullptr)
```

If now used with `signals_as_exception`, e.g. like so:

```
d = Derived()
with cppyy.ll.signals_as_exception():
    cppyy.gbl.callback(d)
```

it produces the following, very informative, Python-side trace:

```
Traceback (most recent call last):
  File "crashit.py", line 25, in <module>
    cppyy.gbl.callback(d)
cppyy.ll.SegmentationViolation: int ::callback(Base* b) =>
    SegmentationViolation: void ::segfault(int* i) =>
    SegmentationViolation: segfault in C++; program state was reset
```

whereas without, there would be no Python-side information at all.

CHAPTER 17

Pythonizations

Automatic bindings generation mostly gets the job done, but unless a C++ library was designed with expressiveness and interactivity in mind, using it will feel stilted. Thus, if you are not the end-user of a set of bindings, it is beneficial to implement *pythonizations*. Some of these are already provided by default, e.g. for STL containers. Consider the following code, iterating over an STL map, using naked bindings (i.e. "the C++ way"):

```
>>> from cppyy.gbl import std
>>> m = std.map[int, int]()
>>> for i in range(10):
...     m[i] = i*2
...
>>> b = m.begin()
>>> while b != m.end():
...     print(b.__deref__().second, end=' ')
...     b.__preinc__()
...
0 2 4 6 8 10 12 14 16 18
>>>
```

Yes, that is perfectly functional, but it is also very clunky. Contrast this to the (automatic) pythonization:

```
>>> for key, value in m:
...    print(value, end=' ')
...
0 2 4 6 8 10 12 14 16 18
>>>
```

Such a pythonization can be written completely in Python using the bound C++ methods, with no intermediate language necessary. Since it is written on abstract features, there is also only one such pythonization that works for all STL map instantiations.

17.1 Python callbacks

Since bound C++ entities are fully functional Python ones, pythonization can be done explicitly in an end-user facing Python module. However, that would prevent lazy installation of pythonizations, so instead a callback mechanism is provided.

A callback is a function or callable object taking two arguments: the Python proxy class to be pythonized and its C++ name. The latter is provided to allow easy filtering. This callback is then installed through `cppyy.py.add_pythonization` and ideally only for the relevant namespace (installing callbacks for classes in the global namespace is supported, but beware of name clashes).

Pythonization is most effective of well-structured C++ libraries that have idiomatic behaviors. It is then straightforward to use Python reflection to write rules. For example, consider this callback that looks for the conventional C++ function `GetLength` and replaces it with Python's `__len__`:

```
import cppyy

def replace_getlength(klass, name):
    try:
        klass.__len__ = klass.__dict__['GetLength']
    except KeyError:
        pass

cppyy.py.add_pythonization(replace_getlength, 'MyNamespace')

cppyy.cppdef("""
namespace MyNamespace {
class MyClass {
public:
    MyClass(int i) : fInt(i) {}
    int GetLength() { return fInt; }

private:
    int fInt;
};
}""")

m = cppyy.gbl.MyNamespace.MyClass(42)
assert len(m) == 42
```

17.2 C++ callbacks

If you are familiar with the Python C-API, it may sometimes be beneficial to add unique optimizations to your C++ classes to be picked up by the pythonization layer. There are two conventional function that cppyy will look for (no registration of callbacks needed):

```
static void __cppyy_explicit_pythonize__(PyObject* klass, const std::string&);
```

which is called *only* for the class that declares it. And:

```
static void __cppyy_pythonize__(PyObject* klass, const std::string&);
```

which is also called for all derived classes.

Just as with the Python callbacks, the first argument will be the Python class proxy, the second the C++ name, for easy filtering. When called, cppyy will be completely finished with the class proxy, so any and all changes, including such low-level ones such as the replacement of iteration or buffer protocols, are fair game.

CHAPTER 18

Utilities

The cppyy-backend package brings in the following utilities to help with repackaging and redistribution:

- cling-config: for compile time flags
- rootcling and genreflex: for dictionary generation
- cppyy-generator: part of the *CMake interface*

18.1 Compiler/linker flags

cling-config is a small utility to provide access to the as-installed configuration, such as compiler/linker flags and installation directories, of other components. Usage examples:

```
$ cling-config --help
Usage: cling-config [--cflags] [--cppflags] [--cmake]
$ cling-config --cmake
/usr/local/lib/python2.7/dist-packages/cppyy_backend/cmake
```

18.2 Dictionaries

Loading header files or code directly into cling is fine for interactive work and smaller packages, but large scale applications benefit from pre-compiling code, using the automatic class loader, and packaging dependencies in so-called "dictionaries."

A *dictionary* is a generated C++ source file containing references to the header locations used when building (and any additional locations provided), a set of forward declarations to reduce the need of loading header files, and a few I/O helper functions. The name "dictionary" is historic: before cling was used, it contained the complete generated C++ reflection information, whereas now that is derived at run-time from the header files. It is still possible to fully embed header files rather than only storing their names and search locations, to make the dictionary more self-contained.

After generating the dictionary, it should be compiled into a shared library. This provides additional dependency control: by linking it directly with any further libraries needed, you can use standard mechanisms such as `rpath` to locate those library dependencies. Alternatively, you can add the additional libraries to load to the mapping files of the class loader (see below).

In tandem with any dictionary, a pre-compiled module (.pcm) file will be generated. C++ modules are still on track for inclusion in the C++20 standard and most modern C++ compilers, `clang` among them, already have implementations. The benefits for cppyy include faster bindings generation, lower memory footprint, and isolation from preprocessor macros and compiler flags. The use of modules is transparent, other than the requirement that they need to be co-located with the compiled dictionary shared library.

Optionally, the dictionary generation process also produces a mapping file, which lists the libraries needed to load C++ classes on request (for details, see the section on the class loader below).

Structurally, you could have a single dictionary for a project as a whole, but more likely a large project will have a pre-existing functional decomposition that can be followed, with a dictionary per functional unit.

18.2.1 Generation

There are two interfaces onto the same underlying dictionary generator: `rootcling` and `genreflex`. The reason for having two is historic and they are not complete duplicates, so one or the other may suit your preference better. It is foreseen that both will be replaced once C++ modules become more mainstream, as that will allow simplification and improved robustness.

rootcling

The first interface is called `rootcling`:

```
$ rootcling
Usage: rootcling [-v][-v0-4] [-f] [out.cxx] [opts] file1.h[+][-][!] file2.h[+][-][!] .
↪..[Linkdef.h]
For more extensive help type: /usr/local/lib/python2.7/dist-packages/cppyy_backend/
↪bin/rootcling -h
```

Rather than providing command line options, the main steering of `rootcling` behavior is done through #pragmas in a Linkdef.h file, with most pragmas dedicated to selecting/excluding (parts of) classes and functions. Additionally, the Linkdef.h file may contain preprocessor macros.

The output consists of a dictionary file (to be compiled into a shared library), a C++ module, and an optional mapping file, as described above.

genreflex

The second interface is called `genreflex`:

```
$ genreflex
Generates dictionary sources and related ROOT pcm starting from an header.
Usage: genreflex headerfile.h [opts] [preproc. opts]
...
```

`genreflex` has a richer command line interface than `rootcling` as can be seen from the full help message.

Selection/exclusion is driven through a selection file using an XML format that allows both exact and pattern matching to namespace, class, enum, function, and variable names.

Example

Consider the following basic example code, living in a header "MyClass.h":

```
class MyClass {
public:
    MyClass(int i) : fInt(i) {}
    int get_int() { return fInt; }

private:
    int fInt;
};
```

and a corresponding "Linkdef.h" file, selecting only `MyClass`:

```
#ifdef __ROOTCLING__
#pragma link off all classes;
#pragma link off all functions;
#pragma link off all globals;
#pragma link off all typedef;

#pragma link C++ class MyClass;

#endif
```

For more pragmas, see the rootcling manual. E.g., a commonly useful pragma is one that selects all C++ entities that are declared in a specific header file:

```
#pragma link C++ defined_in "MyClass.h";
```

Next, use `rootcling` to generate the dictionary (here: `MyClass_rflx.cxx`) and module files:

```
$ rootcling -f MyClass_rflx.cxx MyClass.h Linkdef.h
```

Alternatively, define a "myclass_selection.xml" file:

```
<lcgdict>
    <class name="MyClass" />
</lcgdict>
```

serving the same purpose as the Linkdef.h file above (in fact, `rootcling` accepts a "selection.xml" file in lieu of a "Linkdef.h"). For more tags, see the selection file documentation. Commonly used are `namespace`, `function`, `enum`, or `variable` instead of the `class` tag, and `pattern` instead of `name` with wildcarding in the value string.

Next, use `genreflex` to generate the dictionary (here: `MyClass_rflx.cxx`) and module files:

```
$ genreflex MyClass.h --selection=myclass_selection.xml -o MyClass_rflx.cxx
```

From here, compile and link the generated dictionary file with the project and/or system specific options and libraries into a shared library, using `cling-config` for the relevant cppyy compiler/linker flags. (For work on MS Windows, this helper script may be useful.) To continue the example, assuming Linux:

```
$ g++ `cling-config --cppflags` -fPIC -O2 -shared MyClass_rflx.cxx -o MyClassDict.so
```

Instead of loading the header text into `cling`, you can now load the dictionary:

```
>>> import cppyy
>>> cppyy.load_reflection_info('MyClassDict')
>>> cppyy.gbl.MyClass(42)
<cppyy.gbl.MyClass object at 0x7ffb9f230950>
>>> print(_.get_int())
42
>>>
```

and use the selected C++ entities as if the header was loaded.

The dictionary shared library can be relocated, as long as it can be found by the dynamic loader (e.g. through `LD_LIBRARY_PATH`) and the header file is fully embedded or still accessible (e.g. through a path added to `cppyy.add_include_path` at run-time, or with `-I` to `rootcling`/`genreflex` during build time). When relocating the shared library, move the .pcm with it. Once support for C++ modules is fully fleshed out, access to the header file will no longer be needed.

18.2.2 Class loader

Explicitly loading dictionaries is fine if this is hidden under the hood of a Python package and thus transparently done on `import`. Otherwise, the automatic class loader is more convenient, as it allows direct use without having to manually find and load dictionaries (assuming these are locatable by the dynamic loader).

The class loader utilizes so-called rootmap files, which by convention should live alongside the dictionary shared library (and C++ module file). These are simple text files, which map C++ entities (such as classes) to the dictionaries and other libraries that need to be loaded for their use.

With `genreflex`, the mapping file can be automatically created with `--rootmap-lib=MyClassDict`, where "MyClassDict" is the name of the shared library (without the extension) build from the dictionary file. With `rootcling`, create the same mapping file with `-rmf MyClassDict.rootmap -rml MyClassDict`. It is necessary to provide the final library name explicitly, since it is only in the separate linking step where these names are fixed and those names may not match the default choice.

With the mapping file in place, the above example can be rerun without explicit loading of the dictionary:

```
>>> import cppyy
>>> from cppyy.gbl import MyClass
>>> MyClass(42).get_int()
42
>>>
```

18.3 Bindings collection

`cppyy-generator` is a clang-based utility program which takes a set of C++ header files and generates a JSON output file describing the objects found in them. This output is intended to support more convenient access to a set of cppyy-supported bindings:

```
$ cppyy-generator --help
usage: cppyy-generator [-h] [-v] [--flags FLAGS] [--libclang LIBCLANG]
                       output sources [sources ...]
...
```

This utility is mainly used as part of the *CMake interface*.

CHAPTER 19

CMake interface

CMake fragments are provided for an Automated generation of an end-user bindings package from a CMake-based project build. The bindings generated by rootcling, are 'raw' in the sense that:

- The .cpp file be compiled. The required compilation steps are platform-dependent.
- The bindings are not packaged for distribution. Typically, users expect to have a pip-compatible package.
- The binding are in the 'cppyy.gbl' namespace. This is an inconvenience at best for users who might expect C++ code from KF5::Config to appear in Python via "import KF5.Config".
- The bindings are loaded lazily, which limits the discoverability of the content of the bindings.
- `cppyy` supports customization of the bindings via 'Pythonization' but there is no automated way to load them.

These issues are addressed by the CMake support. This is a blend of Python packaging and CMake where CMake provides:

- Platform-independent scripting of the creation of a Python 'wheel' package for the bindings.
- An facility for CMake-based projects to automate the entire bindings generation process, including basic automated tests.

19.1 Python packaging

Modern Python packaging usage is based on the 'wheel'. This is places the onus on the creation of binary artifacts in the package on the distributor. In this case, this includes the platform-dependent steps necessary to compile the .cpp file.

The generated package also takes advantage of the __init__.py load-time mechanism to enhance the bindings:

- The bindings are rehosted in a "native" namespace so that C++ code from KF5::Config appears in Python via "import KF5.Config".
- (TBD) Load Pythonizations.

Both of these need/can use the output of the *cppyy-generator* (included in the package) as well as other runtime support included in `cppyy`.

19.2 CMake usage

The CMake usage is via two modules:

- FindLibClang.cmake provides some bootstrap support needed to locate clang. This is provided mostly as a temporary measure; hopefully upstream support will allow this to be eliminated in due course.
- FindCppyy.cmake provides the interface described further here.

Details of the usage of these modules is within the modules themselves, but here is a summary of the usage. `FindLibClang.cmake` sets the following variables:

```
LibClang_FOUND             - True if libclang is found.
LibClang_LIBRARY           - Clang library to link against.
LibClang_VERSION           - Version number as a string (e.g. "3.9").
LibClang_PYTHON_EXECUTABLE - Compatible python version.
```

`FindCppyy.cmake` sets the following variables:

```
Cppyy_FOUND - set to true if Cppyy is found
Cppyy_DIR - the directory where Cppyy is installed
Cppyy_EXECUTABLE - the path to the Cppyy executable
Cppyy_INCLUDE_DIRS - Where to find the ROOT header files.
Cppyy_VERSION - the version number of the Cppyy backend.
```

and also defines the following functions:

```
cppyy_add_bindings - Generate a set of bindings from a set of header files.
cppyy_find_pips - Return a list of available pip programs.
```

19.2.1 cppyy_add_bindings

Generate a set of bindings from a set of header files. Somewhat like CMake's add_library(), the output is a compiler target. In addition ancillary files are also generated to allow a complete set of bindings to be compiled, packaged and installed:

```
cppyy_add_bindings(
    pkg
    pkg_version
    author
    author_email
    [URL url]
    [LICENSE license]
    [LANGUAGE_STANDARD std]
    [LINKDEFS linkdef...]
    [IMPORTS pcm...]
    [GENERATE_OPTIONS option...]
    [COMPILE_OPTIONS option...]
    [INCLUDE_DIRS dir...]
    [LINK_LIBRARIES library...]
    [H_DIRS H_DIRSectory]
    H_FILES h_file...)
```

The bindings are based on https://cppyy.readthedocs.io/en/latest/, and can be used as per the documentation provided via the cppyy.gbl namespace. First add the directory of the <pkg>.rootmap file to the LD_LIBRARY_PATH environment variable, then "import cppyy; from cppyy.gbl import <some-C++-entity>".

Alternatively, use “import <pkg>”. This convenience wrapper supports “discovery” of the available C++ entities using, for example Python 3’s command line completion support.

The bindings are complete with a setup.py, supporting Wheel-based packaging, and a test.py supporting pytest/nosetest sanity test of the bindings.

The bindings are generated/built/packaged using 3 environments:

- One compatible with the header files being bound. This is used to generate the generic C++ binding code (and some ancillary files) using a modified C++ compiler. The needed options must be compatible with the normal build environment of the header files.
- One to compile the generated, generic C++ binding code using a standard C++ compiler. The resulting library code is “universal” in that it is compatible with both Python2 and Python3.
- One to package the library and ancillary files into standard Python2/3 wheel format. The packaging is done using native Python tooling.

Arguments and options	Description
pkg	The name of the package to generate. This can be either of the form “simplename” (e.g. “Akonadi”), or of the form “namespace.simplename” (e.g. “KF5.Akonadi”).
pkg_version	The version of the package.
author	The name of the library author.
author_email	The email address of the library author.
URL url	The home page for the library. Default is “https://pypi.python.org/pypi/<pkg>”.
LICENSE license	The license, default is “LGPL 2.0”.
LANGUAGE_STANDARD std	The version of C++ in use, “14” by default.
IMPORTS pcm	Files which contain previously-generated bindings which pkg depends on.
GENERATE_OPTIONS optio	Options which are to be passed into the rootcling command. For example, bindings which depend on Qt may need “-D__PIC__;-Wno-macro-redefined” as per https://sft.its.cern.ch/jira/browse/ROOT-8719.
LINKDEFS def	Files or lines which contain extra #pragma content for the linkdef.h file used by rootcling. See https://root.cern.ch/root/html/guides/users-guide/AddingaClass.html#the-linkdef.h-file. In lines, literal semi-colons must be escaped: “;”.
EXTRA_CODES code	Files which contain extra code needed by the bindings. Customization is by routines named “c13n_<something>”; each such routine is passed the module for <pkg>: :: code-block python **def c13n_doit(pkg_module):** print(pkg_module.__dict__) The files and individual routines within files are processed in alphabetical order.
EXTRA_HEADERS hdr	Files which contain extra headers needed by the bindings.
EXTRA_PYTHONS py	Files which contain extra Python code needed by the bindings.
COMPILE_OPTIONS option	Options which are to be passed into the compile/link command.
INCLUDE_DIRS dir	Include directories.
LINK_LIBRARIES library	Libraries to link against.
H_DIRS directory	Base directories for H_FILES.
H_FILES h_file	Header files for which to generate bindings in pkg. Absolute filenames, or filenames relative to H_DIRS. All definitions found directly in these files will contribute to the bindings. (NOTE: This means that if “forwarding headers” are present, the real “legacy” headers must be specified as H_FILES). All header files which contribute to a given C++ namespace should be grouped into a single pkg to ensure a 1-to-1 mapping with the implementing Python class.

Returns via PARENT_SCOPE variables:

```
target                  The CMake target used to build.
setup_py                The setup.py script used to build or install pkg.
```

Examples:

```
find_package(Qt5Core NO_MODULE)
find_package(KF5KDcraw NO_MODULE)
get_target_property(_H_DIRS KF5::KDcraw INTERFACE_INCLUDE_DIRECTORIES)
get_target_property(_LINK_LIBRARIES KF5::KDcraw INTERFACE_LINK_LIBRARIES)
set(_LINK_LIBRARIES KF5::KDcraw ${_LINK_LIBRARIES})
include(${KF5KDcraw_DIR}/KF5KDcrawConfigVersion.cmake)

cppyy_add_bindings(
    "KDCRAW" "${PACKAGE_VERSION}" "Shaheed" "srhaque@theiet.org"
    LANGUAGE_STANDARD "14"
    LINKDEFS "../linkdef_overrides.h"
    GENERATE_OPTIONS "-D__PIC__;-Wno-macro-redefined"
    INCLUDE_DIRS ${Qt5Core_INCLUDE_DIRS}
    LINK_LIBRARIES ${_LINK_LIBRARIES}
    H_DIRS ${_H_DIRS}
    H_FILES "dcrawinfocontainer.h;kdcraw.h;rawdecodingsettings.h;rawfiles.h")
```

There is a fuller example of embedding the use of cppyy_add_bindings for a large set of bindings:

```
https://cgit.kde.org/pykde5.git/plain/KF5/CMakeLists.txt?h=include_qt_binding
```

19.2.2 cppyy_find_pips

Return a list of available pip programs.

CHAPTER 20

PyPI Packages

20.1 Cppyy

The `cppyy` module is a frontend (see *Package Structure*), and most of the code is elsewhere. However, it does contain the docs for all of the modules, which are built using Sphinx: http://www.sphinx-doc.org/en/stable/ and published to http://cppyy.readthedocs.io/en/latest/index.html using a webhook. To create the docs:

```
$ pip install sphinx_rtd_theme
Collecting sphinx_rtd_theme
...
Successfully installed sphinx-rtd-theme-0.2.4
$ cd docs
$ make html
```

The Python code in this module supports:

- Interfacing to the correct backend for CPython or PyPy.
- Pythonizations (TBD)

20.2 Cppyy-backend

The `cppyy-backend` module contains two areas:

- A patched copy of cling
- Wrapper code

20.3 Package structure

There are four PyPA packages involved in a full installation, with the following structure:

```
              (A) _cppyy (PyPy)
          /                         \
(1) cppyy                             (3) cling-backend -- (4) cppyy-cling
          \                         /
            (2) CPyCppyy (CPython)
```

The user-facing package is always `cppyy` (1). It is used to select the other (versioned) required packages, based on the python interpreter for which it is being installed.

Below (1) follows a bifurcation based on interpreter. This is needed for functionality and performance: for CPython, there is the CPyCppyy package (2). It is written in C++, makes use of the Python C-API, and installs as a Python extension module. For PyPy, there is the builtin module `_cppyy` (A). This is not a PyPA package. It is written in RPython as it needs access to low-level pointers, JIT hints, and the `_cffi_backend` backend module (itself builtin).

Shared again across interpreters is the backend, which is split in a small wrapper (3) and a large package that contains Cling/LLVM (4). The former is still under development and expected to be updated frequently. It is small enough to download and build very quickly. The latter, however, takes a long time to build, but since it is very stable, splitting it off allows the creation of binary wheels that need updating only infrequently (expected about twice a year).

All code is publicly available; see the *section on repositories*.

CHAPTER 21

Repositories

The cppyy module is a frontend that requires an intermediate (Python interpreter dependent) layer, and a backend (see *Package Structure*). Because of this layering and because it leverages several existing packages through reuse, the relevant codes are contained across a number of repositories.

- Frontend, cppyy: https://bitbucket.org/wlav/cppyy
- CPython (v2/v3) intermediate: https://bitbucket.org/wlav/cpycppyy
- PyPy intermediate (module _cppyy): https://bitbucket.org/pypy/pypy/
- Backend, cppyy: https://bitbucket.org/wlav/cppyy-backend

The backend repo contains both the cppyy-cling (under "cling") and cppyy-backend (under "clingwrapper") packages.

21.1 Building from source

Except for cppyy-cling, the structure in the repositories follows a normal PyPA package and they are thus ready to build with setuptools: simply clone the package and either run `python setup.py`, or use `pip`.

It is highly recommended to follow the dependency chain when manually upgrading packages individually (i.e. `cppyy-cling`, `cppyy-backend`, `CPyCppyy` if on CPython, and then finally `cppyy`), because upstream packages expose headers that are used by the ones downstream. Of course, if only building for a patch/point release, there is no need to re-install the full chain (or follow the order). Always run the local updates from the package directories (i.e. where the `setup.py` file is located), as some tools rely on the package structure.

The `STDCXX` envar can be used to control the C++ standard version; use `MAKE` to change the `make` command; and `MAKE_NPROCS` to control the maximum number of parallel jobs. Compilation of the backend, which contains a customized version of Clang/LLVM, can take a long time, so by default the setup script will use all cores (x2 if hyperthreading is enabled).

On MS Windows, some temporary path names may be too long, causing the build to fail. To resolve this issue, point the `TMP` and `TEMP` envars to an existing directory with a short name before the build: For example:

```
> set TMP=C:\TMP
> set TEMP=C:\TMP
```

Start with the `cppyy-cling` package (cppyy-backend repo, subdirectory "cling"), which requires source to be pulled in from upstream, and thus takes a few extra steps:

```
$ git clone https://bitbucket.org/wlav/cppyy-backend.git
$ cd cppyy-backend/cling
$ python setup.py egg_info
$ python create_src_directory.py
$ python -m pip install . --upgrade
```

The `egg_info` setup command is needed for `create_src_directory.py` to find the right version. That script in turn downloads the proper release from upstream, trims and patches it, and installs the result in the "src" directory. When done, the structure of `cppyy-cling` looks again like a PyPA package and can be used/installed as expected, here using `pip`.

The `cppyy-cling` package, because it contains Cling/Clang/LLVM, is rather large to build, so by default the setup script will use all cores (x2 if hyperthreading is enabled). You can change this behavior with the `MAKE_NPROCS` envar. The wheel of `cppyy-cling` is reused by pip for all versions of CPython and PyPy, thus the long compilation is needed only once for all different versions of Python on the same machine.

Unless you build on the manylinux1 docker images, wheels for `cppyy`, `CPyCppyy`, and `cppyy-backend` are disabled, because `setuptools` (as used by `pip`) does not properly resolve dependencies for wheels. You will see a harmless "error" message to that effect fly by in the (verbose) output. You can force manual build of those wheels, as long as you make sure that you have the proper dependencies *installed*, using `--force-bdist`, when building from the repository.

Next up is `cppyy-backend` (cppyy-backend, subdirectory "clingwrapper"; omit the first step if you already cloned the repo for `cppyy-cling`):

```
$ git clone https://bitbucket.org/wlav/cppyy-backend.git
$ cd cppyy-backend/clingwrapper
$ python -m pip install . --upgrade
```

Upgrading `CPyCppyy` (if on CPython; it's not needed for PyPy) and `cppyy` is very similar:

```
$ git clone https://bitbucket.org/wlav/CPyCppyy.git
$ cd CPyCppyy
$ python -m pip install . --upgrade
```

Finally, the top-level package `cppyy`:

```
$ git clone https://bitbucket.org/wlav/cppyy.git
$ cd cppyy
$ python -m pip install . --upgrade
```

Please see the pip documentation for more options, such as developer mode.

CHAPTER 22

Test suite

The cppyy tests live in the top-level cppyy package, can be run for both CPython and PyPy, and exercises the full setup, including the backend. Most tests are standalone and can be run independently, with a few exceptions in the template tests (see file `test_templates.py`).

To run the tests, first install cppyy by any usual means, then clone the cppyy repo, and enter the `test` directory:

```
$ git clone https://bitbucket.org/wlav/cppyy.git
$ cd cppyy/test
```

Next, build the dictionaries, the manner of which depends on your platform. On Linux or MacOS-X, run `make`:

```
$ make all
```

On Windows, run the dictionary building script:

```
$ python make_dict_win32.py all
```

Next, make sure you have pytest installed, for example with `pip`:

```
$ python -m pip install pytest
```

and finally run the tests:

```
$ python -m pytest -sv
```

On Linux and MacOS-X, all tests should succeed. On MS Windows 32bit there are 4 failing tests, on 64bit there are 5 still failing.

CHAPTER 23

Bugs and feedback

Please report bugs or requests for improvement on the issue tracker.