Journal of Physics: Conference Series **664** (2015) 062029 doi:10.1088/1742-6596/664/6/062029

Python in the Cling World

W Lavrijsen[1]

[1] Lawrence Berkeley National Laboratory (LBNL), Berkeley, California, United States

E-mail: WLavrijsen@lbl.gov

Abstract. The language improvements in C++11/14 greatly reduce the amount of boilerplate code required and allow resource ownership to be clarified in interfaces. On top, the Cling C++ interpreter brings a truly interactive experience and real dynamic behavior to the language. Taken together, these developments bring C++ much closer to Python in ability, allowing the combination of PyROOT/cppyy and Cling to integrate the two languages on a new level. This paper describes the current state of the art, including cross-language callbacks, automatic template instantiations, and the ability to use Python from Cling.

1. ROOT6 and Cling

Two important new features come with ROOT6: support for C++11/14 and a true interactive C++ interpreter[1]. For Python developers, C++11/14 offers important improvements that make interfacing easier. For example, the need for "naked" pointers in C++ has been virtually eliminated with new wrapper classes that clarify memory ownership rules in interfaces and can thus be automatically handled. Further, C++ has become more "pythonic" with auto typing, range-based `for` loops, lambda expressions, etc. Use of these features in C++ requires some minimal standardization to conform, which the bindings can use. For example, to enable range-based `for` loops on your own C++ classes, they need to have the same basic signature as STL containers, so that the C++ compiler can derive the intended code automatically. Knowing this standard interface is enough for a Python bindings generator to do the same.

Having a true C++ interpreter behind the scenes allows automatic instantiations of templates like a C++ compiler would, as well as the generation of helper codes to smooth over rough edges between the two languages. Likewise, code generation makes it easier to use Python from C++ by providing the necessary stubs at run-time, rather than requiring developers to preselect and generate dictionaries for what they think they (or their users) will need. Finally, having the ability to generate correct language information such as offsets for classes that have not been seen until run-time unlocks performance optimizations.

This paper discusses each of these new features in detail, then looks at the refactoring work done for PyROOT, and concludes with an overview of future work.

2. Dynamic Templates

Templates in C++ are used for several programming paradigms. A very important one, generic programming, is the expression of the same logical construct as applied to different types, with a single code and with the only requirement that the types used conform to a specific interface signature. In Python, because of dynamic typing, this is the normal modus operandi, and is referred to as "duck typing"[2]. In C++, however, each use of a template on a different type leads

to code generation ("template instantiation"), which then needs to be compiled before use. As a practical matter it was therefore required in ROOT5 to predefine the expected instantiations, and those and only those would then be available at run-time in Python. In ROOT6, this is no longer a restriction, as the Cling interpreter can instantiate templates on-the-fly.

The simplest case, as it is in C++, is for classes. Here, the full template types are either provided explicitly by the developer or are part of a templated function's return type. As a result, the full desired type of the template is available at evaluation of the Python expression that uses it, and instantiation can thus proceed as needed. The most common case is the use of STL containers with developer provided classes. For example, assume a C++ class "`Data`" is available to the auto-loader, then the following would cause instantiations of the STL `map`, as well of the required iterator types for the loop, just-in-time for their use:

```
results = std.map( std.string, Data )()
results[ 'summary' ] = Data(42)
for tag, data in results:
   print tag, ':', data
```

It is quite different for functions and methods: the automatic template instantiation rules for functions in C++ cover several exceptions, such as specializations, choices between implicit conversion of arguments v.s. instantiation of a new function, and selection scopes when dealing with different namespaces. Most importantly, since the C++ compiler can often derive the template types from the call argument types, it does not require them to be specified, and so neither should that be required in Python. Furthermore in Python, the evaluation of the method instantiation (the call to the method descriptor) is a separate expression from the method call itself. Therefore PyROOT can not distinguish between an instantiation and a call, and must do the latter without information about the former. PyROOT follows a set of rules to balance expected developer intent with the needs of the run-time implementation to minimize surprises; together with explicit developer control to resolve ambiguous cases. These rules are, in order:

1: Non-templated overloads come first (as is the case in C++).
2: Assume the arguments are for template lookup (not instantiation), as this is most likely to fail if that is *not* the intent, thus reducing the chances for false positives.
3: Choose among the existing instantiations, if any. This allows the developer to drive the decisions by providing these explicitly.
4: Use the types of the arguments instead of the arguments for template instantiation, unless the arguments are types themselves.
5: Or finally, use the arguments directly for instantiation.

To understand these rules better, here are some examples. Assume that the class `Data` has a method "`Method`" which is templated on its first argument type. If `Method` is called with an argument of type `float` (C++'s `double`), rule 1 will fail (no non-templated overloads); rule 2 will fail (the instantiation with the `float` *value* does not exist); rule 3 will fail (no pre-existing instantiations); but finally rule 4 will succeed (template instantiation with the type of the argument). A second call using an `int` argument type will succeed through rule 3, as the pre-existing instantiation can be used (`int` is implicitly convertible to `double`). If on the other hand differences of instantiations between `int` and `double` matter, then the developer can provide these instantiations in the dictionary, and rule 3 will select the correct overload.

Rule 5 exists, because templates can be instantiated with values as well as with types (e.g. `Method<1>()` rather than `Method<int>(1)` in C++). Recall that the instantiation and the call are two expressions in Python that are evaluated sequentially, so that it is not possible (and indeed, no way to roll back) to know at run-time which is which. Using the argument values directly for the instantiation highly likely leads to uncompilable code, which is silently ignored,

so rule 5 could have been preferred over rule 4, as false positives are unlikely. However, the reverse is equally true, yet instantiation with values is less common. Rule 2 and rule 3 have the reverse order to allow disambiguation through explicit instantiation by the developer.

3. Python from Cling

PyROOT is a two-way bridge between Python and C++, and in particular between the Python interpreter and Cling. In ROOT5, there were important limitations in code generation for CINT, and hence the dynamic nature of Python was difficult to handle from it. Cling has no limitations for code generation, but it does have limitations of scope, its basic unit, contrary to the line-by-line evaluation of CINT. Thus, Python types need to be loaded in a separate scope from where they are used, but once loaded, there are much fewer limits to their use. In particular, new features include support for free functions, modules as as namespaces, and the ability to derive C++ classes from Python ones. Casts are still needed on the C++ side, as this is a language requirement. For backwards compatibility reasons, a new method `TPython::Import` has been added to expose the new functionality.

As an example, consider this Python module, "`pymod.py`":

```
class PyKlass(object):
   def pass_it(self, value):
      return value

def func(value):
   return value
```

After importing this module into Cling, it can be used like so:

```
root[0] TPython::Import("pymod")
root[1] (double)pymod::func(3.14)
(double) 3.140000e+00
root[1] struct Aap : public pymod::PyKlass {
   Aap() : fInt(42) { };
   int fInt;
};
root[2] Aap a;
root[2] (char*)a.pass_it("hello")
(char *) "hello"
root[3] TPython::Prompt()
>>> a = ROOT.Aap()
>>> print a.fInt
42
```

As is seen from the example, the module name becomes a namespace, scoping the functions and classes within it. The result of a function call is of a generic Python wrapper type (`TPyReturn`) that needs to be cast explicitly to the expected type before the result can be used. The number of expected arguments are declared to Cling (each argument is presented by another generic wrapper type, `TPyArg`), allowing it to check and report proper errors through the normal means *before* attempting the call.

New C++ classes can be derived from existing Python classes, and these new classes (and their instances) can be used both in C++ as well as in Python as normal. The example shows that after dropping into the Python interpreter from Cling, access to the newly defined C++ class works just like any other bound class.

4. Performance Improvements

There is nothing intrinsically faster in Cling than in CINT for bindings.[1] Both make use of wrappers with a known interface, albeit generated (and compiled) at run-time for the former, and compiled into dictionaries for the latter. Most of the cost of calling from Python into C++ is in conversion functions (the unboxing of Python variables) and their type checking, and thus increased type correctness could even be expected to result in a performance *loss*. Since the Python interpreter performs code verification and the PyROOT layer performs type checks (in order to do the unboxing of the Python variables), such checks in Cling itself are superfluous for use from Python. Fortunately, Cling allows by-passing most call layers, giving direct access to the generated wrappers, which no longer perform checks or input sanitation. A significant reduction in overhead results, leading to a noticeable speedup both for direct calls as well as for callbacks.

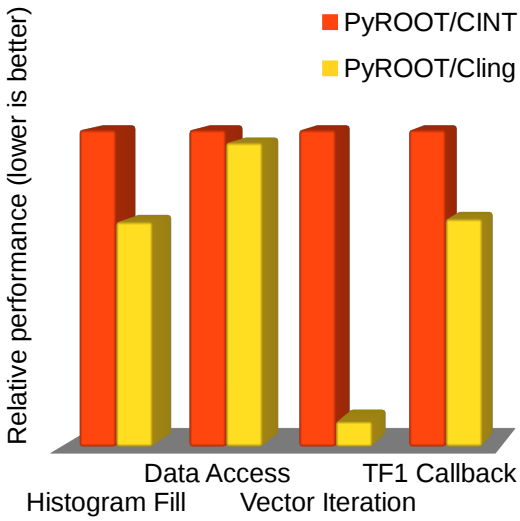

Figure 1. *Performance comparison between PyROOT on top of Cling/ROOT6 and on CINT/ROOT5 for various micro-benchmarks of common use cases.*

Data access already by-passed most of CINT, and still does in the case of Cling: all that is required are memory addresses. Once resolved, there are no further call layers — other than from the Python interpreter and PyROOT for the usual (un)boxing. Part of the refactoring (see next section) did result in small performance improvements, though.

An important use case is the iteration over C++'s `std::vector`. Because this consists of direct pointer access (after all optimizations have run) in C++, it is very fast. In Python, the overhead of the iterator protocol is considerable, but worse, inlining of C++ calls that access elements is not possible. C++11, which Cling brings to the table, provides direct access to the memory underlying the `std::vector`, as well as the guarantee that this memory is contiguous. Further, Cling can provide the correct size for strides. This way, the Python iterator protocol can by-pass the access methods, and the result is a $10x$ speedup for `std::vector` iteration.

5. Bindings Generator Refactored

The PyROOT code base is more than a decade old and its use as a C++-Python bindings generator has extended well beyond just ROOT. A similar approach, reworked from scratch for the PyPy[3][4] project, cppyy [5]. PyPy is a fully compatible Python interpreter that employs just-in-time compilation and dynamic optimizations to greatly improve execution speed and lower memory footprint. Its cppyy module exists without the ROOT heritage: contrary to PyROOT, where the access to reflection information is throughout the code as-needed, cppyy has a well defined C-API separating it from the underlying reflection technology. As an important side-effect of that approach, the code that accesses and manipulates Python objects, and thus needs to be holding the global interpreter lock (GIL), is well separated from code that manipulates only C++ objects and thus does not need to hold this lock.

Having learned from the cppyy development, and to improve the existing PyROOT code

[1] Thanks to just-in-time compilation, Cling's execution of C++ code is considerably faster, however.

base, a refactorization was undertaken with the following goals:

- Separation of a backend cppyy module that provides the pure bindings, and a ROOT module that provides the ROOT pythonizations. The cppyy module is compatible between PyPy and CPython, as well as backwards compatible with the PyCintex module (now obsolete).
- Separation of Python object handling, and placement of all C++ reflection access in a backend library that can be shared between PyPy and CPython. This sharing greatly reduces maintenance effort. The bindings implementations for PyPy and CPython will have to remain separate however, as the former is written in RPython, the latter in C++.
- Clear paths for GIL acquisition and hand-off. This is very important for multi-threading frameworks that use Python algorithms, as mismanagement of the GIL quickly leads to deadlocks or corruption of Python objects.

The PyROOT library itself has not yet been split, even as most of the code compiled into it now is, but doing so is the next step.

Taken together, this refactorization ensures the relevance of PyROOT, now cppyy, when the field moves to multi-threading and Python to just-in-time compilation for performance.

6. Conclusions and Future Work

The functionality of PyROOT on Cling now greatly exceeds that of PyROOT on CINT: more C++ code can be bound well, templates can be automatically handled at run-time, usage of Python from Cling is vastly superior. Work on its performance has only just begun, and already it meets or exceeds the original implementation in terms of CPU usage (memory overhead is still a work in progress).

The clear path of GIL hand-off allows usage of PyROOT/cppyy in multi-threading frameworks, even through multiple layers of callbacks. More fine-grained control for the developer through a pythonization API and the use of software transactional memory is the logical next step.

Finally, the refactorization needs to be brought to completion by separating the library and with separate distribution of the bindings-only and ROOT-specific pythonizations.

References

[1] P. Canal *et al.*, "ROOT 6 and beyond: TObject, C++14 and many cores.", Computing in High Energy Physics 2015, these proceedings.
[2] `http://en.wikipedia.org/wiki/Duck_typing`
[3] `http://pypy.org`
[4] `https://root.cern.ch/drupal/content/pypyroot`
[5] `http://pypy.readthedocs.org/en/latest/cppyy.html`