REFLECTION-BASED PYTHON-C++ BINDINGS

J. Generowicz,[*] P. Mato, CERN, Geneva, Switzerland
W. T. L. P. Lavrijsen, M. Marino, LBNL, CA 94530, USA

Abstract

Python is a flexible, powerful, high-level language with excellent interactive and introspective capabilities and a very clean syntax. As such, it can be a very effective tool for driving physics analysis.

Python is designed to be extensible in low-level C-like languages, and its use as a scientific steering language has become quite widespread. To this end, existing and custom-written C or C++ libraries are bound to the Python environment as so-called extension modules. A number of tools for easing the process of creating such bindings exist, such as SWIG and BOOST.PYTHON. Yet, the process still requires a considerable amount of effort and expertise.

The C++ language has few built-in introspective capabilities, but tools such as LCGDICT and CINT add this by providing so-called dictionaries: libraries that contain information about the names, entry points, argument types, etc. of other libraries. The reflection information from these dictionaries can be used for the creation of bindings and so the process can be fully automated, as dictionaries are already provided for many end-user libraries for other purposes, such as object persistency.

PYLCGDICT is a Python extension module that uses LCG dictionaries, as PYROOT uses CINT reflection information, to allow Python users to access C++ libraries with essentially no preparation on the users' behalf. In addition, and in a similar way, PYROOT gives ROOT users access to Python libraries.

INTRODUCTION

Programming languages that are dynamically typed and compiled on-the-fly, often referred to as "scripting languages," are well known to make software developers more productive than when they develop in system level languages. This is because of the dramatically reduced time spent in waiting for the computer to build and link newly developed code, and because dynamic typing reduces clutter in the syntax: these languages are more expressive and high-level, making them easier to learn and program with.

Dynamic languages are often designed to interoperate smoothly with other languages, thus allowing the programmer to re-use existing libraries. In doing so, the existing codes become well separated components of the new program and the dynamic language becomes a "glue" that binds them together by providing cross-calls and data transfers. The Python programming language [2] in particular, is widely used for this role.

Python is simple, elegant, easy to learn, and is one of the most popular dynamic languages. It is open source, freely available, and comes with an interactive interpreter as well as a wide range of standard modules. Python has many applications in scientific programming, including extensions for high performance and distributed parallel code [3], and is already in use today in many high energy physics (HEP) experiments.

The large, existing, C/C++ code base in HEP can be made available to Python users, by providing so-called *language bindings*: wrapper code that translates back and forth between C/C++ and Python (see Fig. 1). This paper describes some of the existing tools for creating such bindings, as well as two tools, PYLCGDICT and PYROOT, that were developed specifically with the physicist end-user in mind.

TECHNOLOGY OVERVIEW

It is possible to write Python-C++ bindings by hand, based on the Python C-API. But even for medium-sized projects, that quickly becomes cumbersome and unmaintainable, therefore several tools are available that automate the generation of bindings. These tools fall, broadly speaking, in two categories: *static* and *dynamic wrappers*. The differences between the two approaches, which are important to highlight because they seep through to the end-user level, are described below. It must be kept in mind, though, that the actual implementations of tools do not strictly fall into one category or the other.

In either approach, the wrapper needs to resolve the following issues:

- **Object and parameter conversions.** Function cross-calls require that the call arguments and the return values are converted from one language into the other.
- **Missing or incomplete types.** Function signatures and class definitions may include pointers to types that are only declared, not defined.
- **Memory management.** With C++, memory is managed by hand or by custom developed strategies, whereas Python uses reference counted objects and a cyclic garbage collector.
- **C++ function overloading.** Python does not need overloading, because of the dynamic typing, but it is required to select the correct C++ function from a set of overloaded ones, when calling into a C++ library.

[*] Funded by: the Particle Physics and Astronomy Research Council, Swindon, UK.

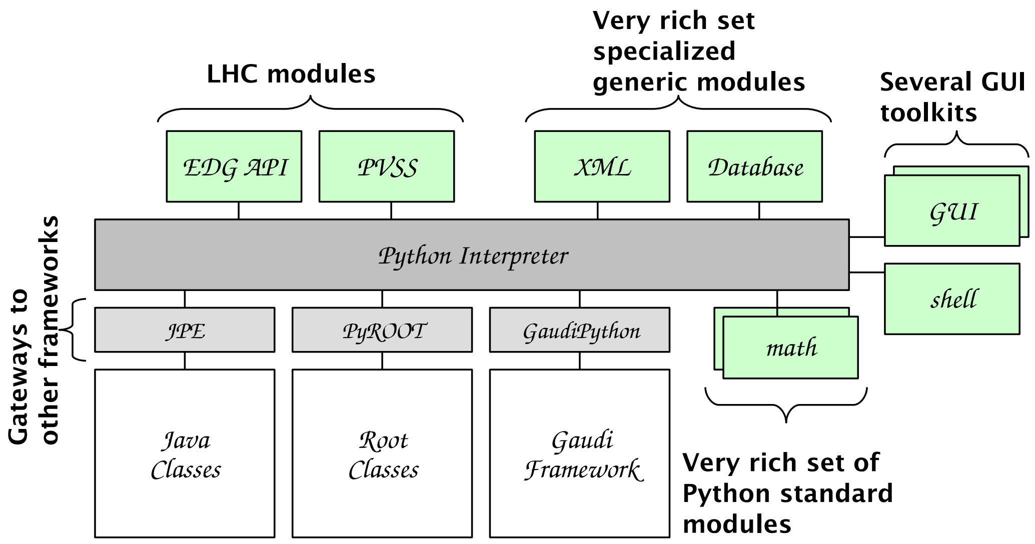

Figure 1: An overview of how the Python interpreter, with the appropriate bindings where necessary, is used as "glue" between HEP libraries, thus giving the Python user access to all existing functionality.

- **C++ templates.** A naming convention is needed to distinguish between templates, especially since additional instantiations can be loaded at run-time.

An issue like memory management is handled in more or less the same way by all tools: the end-user can set policies, apply explicit memory management also on the Python side, or let the tool make educated guesses. Implementations for overloading and templates tend to differ only in details that the end-user will hardly ever see.

Static Wrapping

A generic overview of the process of statically wrapping a library is shown in Fig. 2 (A). An interface definition file (in the case of C++: a header file) is parsed by a wrapper engine that generates the appropriate wrapper code. An interface selection file can be used to steer the engine: it determines which parts of the interface to use, which parts to skip, etc. The generated wrapper code is typically in a low-level system language such as C, and it can subsequently be compiled and linked with the implementation that is being wrapped. The end result is an *extension module* that can be used by the Python interpreter just like any other module.

In a Python session, multiple extension modules can be loaded, and the types from one module must be available to, and usable in, the other modules. In particular, it should be transparent to load types that were left undefined in one module, which usually means that a stub was generated for them, from another module. To this end, static wrappers commonly keep a conversion repository, e.g. to be able to integrally replace stubs. These conversion repositories differ from tool to tool, therefore it is not always straightforward to combine extension modules from different tools.

Note that return values that are of type pointer, or reference, to base class will typically[1] be represented on the Python side as references to that base class, as that is the only information available at code generation time. The Python end-user can explicitly cast the return value, as needed, just as it would have been possible in C++.

Dynamic Wrapping

Dictionaries are libraries that contain reflection information such as names, entry points, argument types, etc., of other libraries. If this information is available, bindings to a library can be dynamically created, as shown in Fig. 2 (B).

In the dynamic approach, Python classes and functions are loaded and created on-demand. Upon use, or upon any other explicit request, class names are looked up in the dictionary and all type information is retrieved to construct a Python class in memory. This may result in the loading of additional dictionaries, as needed.

There are no stubs: all classes are actual Python classes, although some of them may not be complete if type information is missing. Thus, there is no need of a separate bookkeeping of classes and there are potentially fewer problems when mixing with bindings generated by other tools.

The availability of reflection information allows the binding to always use the concrete types for the return values of functions. Thus, if the C++ signature uses type pointer, or reference, to base class, the Python user will see the derived class that is actually returned and no further casting is required.

[1]It is technically possible to downcast to the actual derived class using C++ run-time type information (RTTI). In practice, however, this is not commonly done.

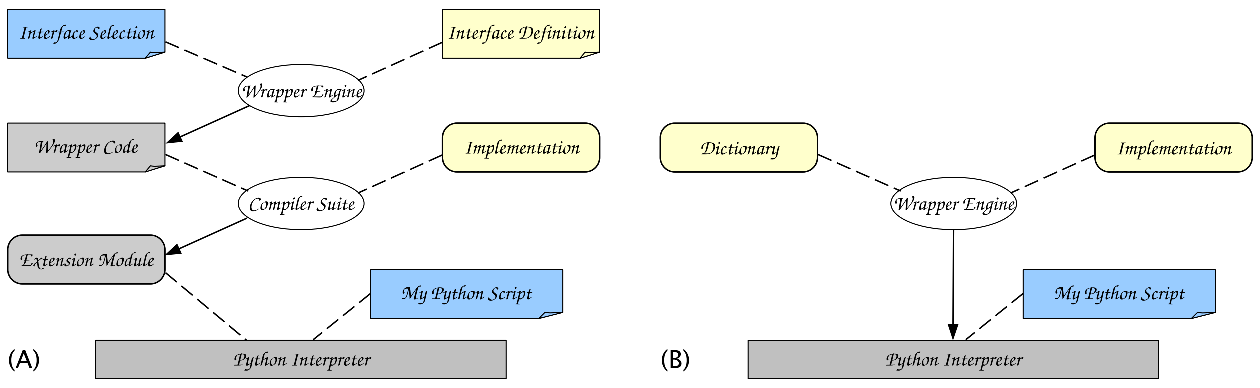

Figure 2: Two conceptually different ways of wrapping existing libraries: static (A), which works from the source code; and dynamic (B), which utilizes reflection information.

Tools

The problem of generating Python-C++ bindings has been around for some time, and several mature tools are available:

- SWIG. This is probably the original wrapper tool. C++ is only one of its inputs and it has, including Python, 13 target languages (see [4]). SWIG is a static wrapper and the individual steps required to produce an extension module are often automated with MAKE by the end-user. That, however, makes it less than trivial to use.
- BOOST.PYTHON. Specifically developed for generating Python-C++ bindings, BOOST.PYTHON is a static wrapper and boasts a large feature set (see [5]). It is, however, primarily an improvement on the flat Python C-API for bindings. Automated generation of bindings is provided by PYSTE, which is now part of BOOST.PYTHON. Note, however, that tracing errors in the generated code is often complicated, because of the heavy reliance on C++ template metaprogramming.
- PYLCGDICT. Part of the SEAL project [6], PYLCGDICT is a dynamic wrapper that uses LCG dictionaries (LCGDICT) to generate bindings. LCGDICT was developed in the SEAL project, primarily to provide C++ introspection for the POOL persistency framework.
- PYROOT. Originally part of SEAL, now part of the ROOT project [7], PYROOT [1] is a dynamic wrapper that uses dictionaries in the CINT format [8] to generate bindings. PYROOT also allows the CINT interpreter to call into Python, and to mix sessions of the two interpreters.
- There are various other projects (CABLE, CXX, SIP, etc.), but their development has either been discontinued or they have been written for very specific applications only.

From an end-user's standpoint, dynamic wrapping is preferred. Especially in HEP, where dictionaries are often already available for other uses (e.g. persistency and analysis code), making the dynamic approach much simpler. This is the reason for the existence of PYLCGDICT and PYROOT: to make it as easy as possible on the physicist end-user to get access to her code and data objects from the Python interpreter.

PYLCGDICT AND PYROOT

The LCG dictionary (LCGDICT) is used to persistify, among others, end-user data objects with the POOL persistency framework [9]. These dictionaries are subsequently used by PYLCGDICT to automatically provide bindings for these objects, without any further work being needed from the end-user.

With scripts made available by the REFLECTIONBUILDER package, also part of SEAL, it is straightforward to create bindings not only for data objects, but also for code. This is done for the GAUDI/ATHENA software framework, where these scripts and PYLCGDICT are used to provide interactivity. This interactive use allows for new ways of doing physics analysis, and gives the developer more diagnostic tools.

Similarly, the widely used ROOT class library, and applications as well as analysis codes based on it, provides many dictionaries in the CINT format that are used by PYROOT to automatically provide bindings. PYROOT works bidirectionally: when loaded into the CINT interpreter, it allows access to Python and the interchanging of the two interpreters. In addition, PYROOT has specific features for it to work easily with ROOT: it hooks into the ROOT memory handler to pick up notifications of the deletion of objects, provides Python sequence protocols for ROOT arrays and lists, allows Python functions to be plotted and used to fit histograms, etc.

There is a new reflection API that the LCGDICT and ROOT developers have agreed on: REFLEX. The core of PYLCGDICT can be used to create a new package: PYREFLEX, which can then also form the basis of PYROOT.

CONCLUSIONS AND OUTLOOK

The physicist's toolkit has been beefed up with PyLCGDict and PyROOT which make it possible to easily use the Python interpreter together with existing C/C++ HEP libraries and with end-user objects that often have associated dictionaries. In doing so, the rich and varied set of standard and third-party Python modules, in addition to any other existing C++ libraries with Python bindings, has been added to the physicist's toolkit as well.

The dictionaries used by LCG and ROOT are to converge on a common API: Reflex. Once this has been accomplished, the core code of PyLCGDict and PyROOT can be shared: PyReflex, with small add-ons for specific uses such as PyROOT currently provides for ROOT.

REFERENCES

[1] http://cern.ch/wlav/pyroot.

[2] G. van Rossum and F. L. Drake, Jr. (eds.), "Python Reference Manual", Release 2.3.4, PythonLabs, May 2004. http://www.python.org.

[3] http://www.scipy.org.

[4] http://www.swig.org.

[5] http://www.boost.org/libs/python.

[6] J. Generowicz, *et al.*, "SEAL: Common Core Libraries and Services for LHC Applications", physics/0306033, June 2003. http://cern.ch/seal.

[7] R. Brun, F. Rademakers, S. Panacek, "ROOT, an object oriented data analysis framework", 23rd CERN School of Computing, Conference Proceedings. http://root.cern.ch.

[8] http://root.cern.ch/root/Cint.html.

[9] D. Düllmann, "The LCG POOL Project, General Overview and Project Structure", CHEP 2003, Conference Proceedings. http://lcgapp.cern.ch/project/persist/.