Idiomatic Python from idiomatic C++: removing barriers to rapid scientific development

Toby St Clere Smithe

Project Overview

- New infrastructure in ROOT 6
  - cling C++ interpreter
  - based on LLVM → knows about modern C++
- “Point and go” C++ bindings
  - just tell PyROOT / cppyy where the source or precompiled module is, then use it like any Python extension!
- But: some things require manual tuning
  - no one-one mapping from C++ to Python
- Longer term: loosen direct dependency on ROOT
- **Code, tests and documentation should be merged soon!**

“Pythonizations”

- The core of the project: how can we make interacting with C++ more like idiomatic Python?
  - for both the CPython and PyPy versions
- “Pythonizations” are strategies to implement this camouflage.
  - In particular, they are usually functions that are called when a new type is instantiated.
  - If the name of the new type matches a user-supplied pattern, then the function applies some change to the type.

New Pythonizations

- GIL policy: should this function release the GIL?
- Return value ownership policy (eg, for raw pointers)
- Smart pointer policy: do we “hide” the smart pointer?
  - more on this later
- Class attribute renaming
- Adding overloads to class methods
- Class method composition
- Automatic transformation of getter and setter methods into Python properties

C++ exception mapping

- Catches the C++ exception in the C++ part of the wrapper (if it is of type derived from std::exception) then uses the reflection information to identify its full type
- Matches the full type against any user-defined mappings; if a match is found, raises the appropriate python exception with the C++ exception message (the C++ “what()”)

Type pinning

- Sometimes, it is desirable to cast objects explicitly from one type to another
- Looking at ROOT-6073, it can be useful to do so systematically
- This part of the project adds type pinning, so that one can say, *“instead of returning MyDerivedType, give me MyBaseType every time”*.
- It is now also possible to add exceptions to the pinning, and cast objects individually.

Smart pointer handling: memory management

- Because Python is garbage-collected, the user doesn't usually think about memory management
- Directly interfacing with C++ code forces us to be more careful, since we want the C++ objects by default to live as long as we need them, but no longer
  - sensible defaults, with manual tuning

Memory management

- For instance, if a C++ function returns by value, we take ownership of the resulting object.
- If it returns by reference, we do not.
- But also if it returns by raw pointer, we do not, because we do not know what semantics are expected, and we want to be safe by default.
  - We do not want to free memory prematurely, but this leads to leaks.

Smart pointers

- Objects that hold a raw pointer, and encode the associated ownership semantics
  - eg, reference counting with shared_ptr or sole ownership with unique_ptr
- Usually passed and returned by value
  - when C++ code returns one, the Python wrapper takes ownership (nb: not of the underlying pointer).
- If the smart pointer object is deleted, then its destructor decides what to do with the underlying pointer.
- But the Python user doesn't care!
  - only interested in using their type T, not shared_ptr<T> or even my_smart_ptr<T>!

Manage smart pointers transparently

- Functions that return shared_ptr<T> now look like functions that return T, and the resulting python wrapper objects can be treated the same.
  - Includes being passed to functions taking either raw T* or shared_ptr<T> (but not any other smart pointer).
  - Also possible to get a python object representing the associated smart pointer directly, if the user so wishes.
  - But if a python object wraps a raw T*, then it still cannot be passed automatically to a function expecting a smart pointer: these must be created explicitly.
- Still obvious to the user if some object is managed by a smart pointer
  - string representation says so
  - _get_smart_ptr method returns not None.
- However, the point is that the user shouldn't have to care any more!

On-going and future work

- Buffer protocol

- Separation of layers and shared PyPy/CPython internal API

Future work: buffer protocol

- Say you have a C++ object representing a matrix.
- Then it would be nice if it were natively available to NumPy operations.
- This requires changing the CPython type so that it can answer questions like *“how large is this buffer? what is the associated memory layout?”*
- (… PyPy buffer support is more rudimentary)

Future work: separation of layers

- Easy to imagine cppyy being broadly popular, outside of ROOT.
- But at the moment, the CPython version depends on a fairly large set of core ROOT functionality, and is built from within the ROOT tree, with all the attendant complexity.
- The PyPy version is quite different, but lags behind recent developments in ROOT 6.
- Therefore, there is on-going work to separate the layers of cppyy more satisfactorily, including refactoring the ROOT dependency into a thin layer, and providing a minimal build tree for non-root users.
- This would also help in unifying the PyPy and CPython versions.

Any questions?