Python bindings for C++ via PyRoot

User experience from PyCool in COOL

(and from PyCoral in CORAL)

Andrea Valassi

(CERN IT-DI-LCG)

2nd CERN Developers Forum – 30th May 2016

(~rerun of the talk given at the ROOT Saas-Fee Workshop in September 2015)

Outline

- Introduction to CORAL, COOL and PyCool
  - Who uses PyCool and the relationship to ROOT
- Implementation of PyCool over time
  - ROOT5 (and before) – gccxml, no c++11
  - ROOT6 – clang/cling (JIT in 6.02, ORCJIT in 6.04)
- A few comments and comparisons to alternatives
- Conclusions

CORAL, COOL and PyCool

- COOL: a set of libraries and tools for the handling of the time variation and versioning of "conditions data"
  - Used by ATLAS and LHCb throughout LHC data taking
    - e.g. detector calibration for Sep 2015 computed with latest algorithm

- CORAL: a generic relational database access layer
  - Used by ATLAS, CMS and LHCb and internally by COOL
    - Conditions data, trigger configuration data, geometry data...
    - Main entry point to physics data in Oracle (directly or via Frontier)

- Both COOL and CORAL are written in C++
  - Python bindings also exist: PyCool (2005), PyCoral (2006)
  - 10-year old products maintained through LHC data taking

COOL data model

Each COOL conditions object has

- **Metadata** ***(system-controlled)***
  - Data item identifier
  - Interval-of-validity [since, until]
  - Version information
- **Data "payload"** ***(user-defined)***
  - Physics values (temperatures, calibration parameters…)
  - Separate columns or a CLOB

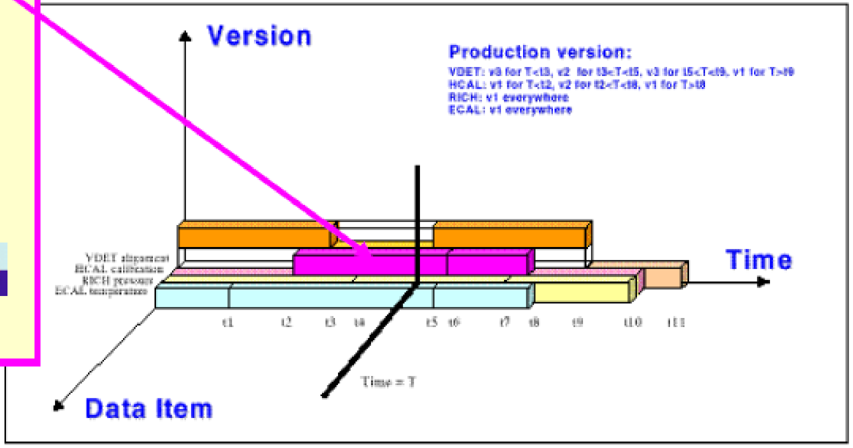
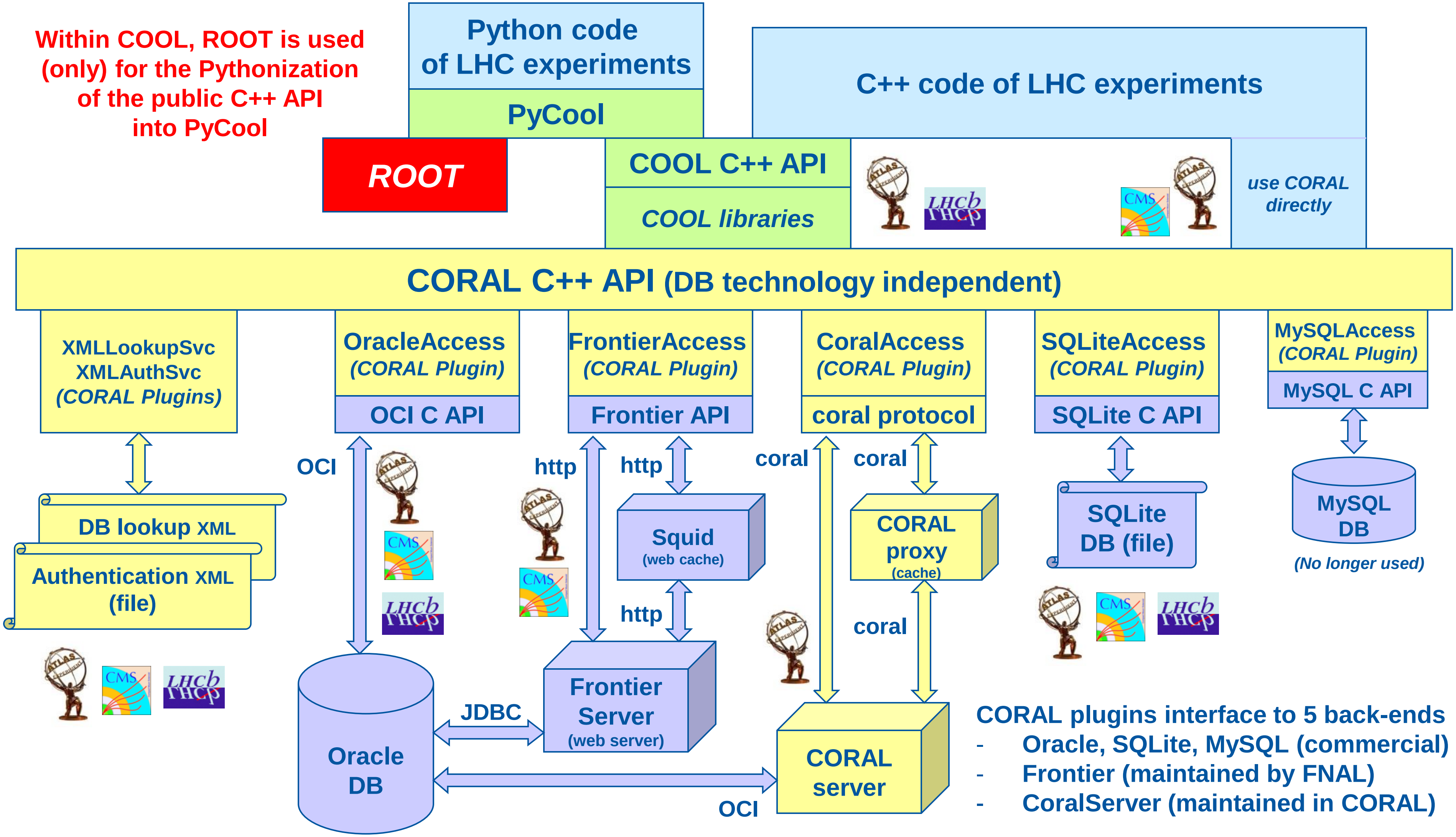


Figure 1 The three axes for identifying uniquely each data item in the condition database

COOL provides a technology-independent C++ API to handle the time variation and versioning of the conditions data of the LHC experiments. This is implemented using relational databases, based on CORAL.

Component architecture

Within COOL, ROOT is used (only) for the Pythonization of the public C++ API into PyCool



Python code of LHC experiments

PyCool

C++ code of LHC experiments

ROOT

COOL C++ API

COOL libraries

use CORAL directly

CORAL C++ API (DB technology independent)

XMLLookupSvc XMLAuthSvc (CORAL Plugins)

OracleAccess (CORAL Plugin)

OCI C API

FrontierAccess (CORAL Plugin)

Frontier API

CoralAccess (CORAL Plugin)

coral protocol

SQLiteAccess (CORAL Plugin)

SQLite C API

MySQLAccess (CORAL Plugin)

MySQL C API

DB lookup XML

Authentication XML (file)

OCI

http

http

Squid (web cache)

http

coral

coral

CORAL proxy (cache)

coral

SQLite DB (file)

MySQL DB

(No longer used)

Oracle DB

JDBC

Frontier Server (web server)

OCI

CORAL server

CORAL plugins interface to 5 back-ends

- Oracle, SQLite, MySQL (commercial)
- Frontier (maintained by FNAL)
- CoralServer (maintained in CORAL)

PyCool usage – inside ATLAS

- Most ATLAS ConditionsDB tools use Python and PyCool:
  - AtlCoolTag (and other tools for "tagging" condition data versions)
  - AtlCoolConsole (to look at "raw" conditions data in the database)
  - CoolCherryPy (RESTful interface for both read and write operations)
  - AtlRunQuery (for online operations)
  - COMA (conditions metadata manager)
  - CoolLumiUtilities (for luminosity management)

- Relevance of ROOT – constraints on COOL release schedule
  - COOL2 vs COOL3 (Boost vs c++11) in ROOT 5 vs ROOT 6
  - Workarounds for exception handling in ROOT 6.02 vs ROOT 6.04
  - See details in the following slides

PyCool usage – by the COOL team

- PyCool is also used internally by the COOL team for tests
  - Performance test suite for COOL/Oracle queries is fully Python based
  - Functional test suite for COOL includes both Python and C++ tests
    - Some functionalities are only tested in Python with no C++ equivalent
  - *Python is a great language for testing and interactive prototyping*

- Relevance of / for ROOT – PyCool tests also test PyROOT!
  - Extensive feedback to (and from) the ROOT team over the years
  - Especially intense testing during the move to ROOT6
    - Regular reports about PyCool at the weekly ROOT6 planning meetings
    - *Many issues in PyROOT and cppyy detected through the PyCool tests*

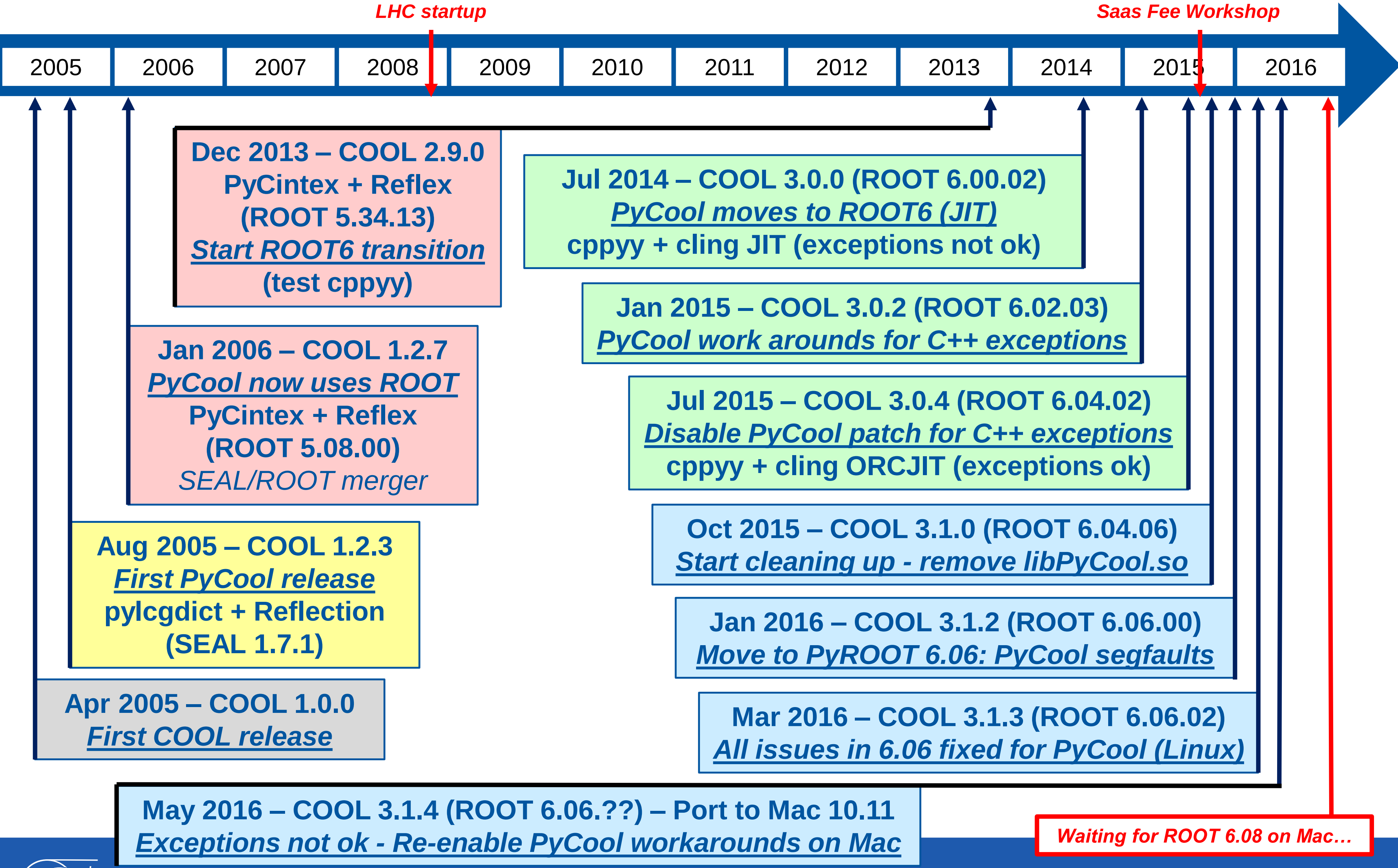
PyCoral – for comparison

- PyCoral provides Python bindings for the CORAL C++ code
  - Does essentially the same as PyCool for COOL
    - Developed around the same time in what was then a separate project
  - Direct CPython implementation, without going via ROOT

- PyCoral was initially developed on request by the experiments
  - Unclear if/how this is still used by any experiments
  - Some of the same features are provided by SQLAlchemy

- PyCoral is still used by the CORAL team
  - Functional test suite for CORAL includes both Python and C++ tests
    - Some functionalities are only tested in Python with no C++ equivalent
  - *Python is a great language for testing and interactive prototyping*

11 years of PyCool evolution



Dec 2013 – COOL 2.9.0
PyCintex + Reflex
(ROOT 5.34.13)
Start ROOT6 transition
(test cppyy)

Jul 2014 – COOL 3.0.0 (ROOT 6.00.02)
PyCool moves to ROOT6 (JIT)
cppyy + cling JIT (exceptions not ok)

Jan 2015 – COOL 3.0.2 (ROOT 6.02.03)
PyCool work arounds for C++ exceptions

Jan 2006 – COOL 1.2.7
PyCool now uses ROOT
PyCintex + Reflex
(ROOT 5.08.00)
SEAL/ROOT merger

Jul 2015 – COOL 3.0.4 (ROOT 6.04.02)
Disable PyCool patch for C++ exceptions
cppyy + cling ORCJIT (exceptions ok)

Oct 2015 – COOL 3.1.0 (ROOT 6.04.06)
Start cleaning up - remove libPyCool.so

Aug 2005 – COOL 1.2.3
First PyCool release
pylcgdict + Reflection
(SEAL 1.7.1)

Jan 2016 – COOL 3.1.2 (ROOT 6.06.00)
Move to PyROOT 6.06: PyCool segfaults

Apr 2005 – COOL 1.0.0
First COOL release

Mar 2016 – COOL 3.1.3 (ROOT 6.06.02)
All issues in 6.06 fixed for PyCool (Linux)

May 2016 – COOL 3.1.4 (ROOT 6.06.??) – Port to Mac 10.11
Exceptions not ok - Re-enable PyCool workarounds on Mac

Waiting for ROOT 6.08 on Mac…

What is/was in PyCool?

[i.e. what could it take to Pythonize your C++ API? not much!]

- A Python module (e.g. to instrument special classes)
  - – Handle C++ template methods and C++/Python types...
  - – Handle C++ shared pointers and C++ references...
  - – Translate C++ iterators to Python iterators...
  - – *Most of this is no longer needed with the latest ROOT*
- A set of C++ helper methods
  - – Hooks to selected internal C++ API calls
  - – Workarounds for Python binding issues (e.g. ROOT 6.02)
- Plus the machinery to build/bootstrap, some tests...
- Some of the above is likely to disappear eventually
  - – Things will be easier when ROOT ~~< 6.04~~ is no longer used

    < 6.08? (see recent Mac issues)

PyCool using ROOT5

- Binding C++/Python uses dictionaries as done in I/O
  - C++ header parsing: at build time (using gccxml)
    - *parse headers* to generate .cpp dictionaries from .h
    - compile .cpp dictionaries into .so
  - Python bootstrapping: use rootmap to load .so of PyCool
    - "import PyCintex" (loads libPyROOT.so, *bootstraps ROOT*)
    - "PyCintex.gbl.cool.IDatabaseSvc" (symbol lookup in the rootmap)
      - this *loads the Python bindings for C++* from libPyCoolDict.so
      - this in turn *loads the COOL C++ libraries* (libCoolApplication.so)

PyCool using ROOT6

- Dictionaries are only needed for I/O, not for Python!
  - C++ header parsing: at run time (using clang JIT)
    - no action needed at build time
  - Python bootstrapping: load .h of PyCool directly!
    - “import cppyy” (loads libPyROOT.so, *bootstraps ROOT*)
    - “cppyy.gbl.gSystem.Load('libCoolApplication.so')”
      - explicitly *loads the COOL C++ libraries* (libCoolApplication.so)
    - “cppyy.gbl.gInterpreter.Declare('#include PyCool_headers.h’)”
      - this *parses the headers* and *generates the Python bindings for C++*

- Now this looks so easy, but it took many iterations...
  - A useless empty PyCoolDict.so still exists in COOL 3.0.x

ROOT6 migration issues for PyCool

(1) Boost vs c++11 in ROOT5 vs ROOT6

- ROOT5 does not support c++11
  - The COOL C++ API with ROOT5 cannot contain c++11
    - It contains Boost instead – the old COOL2 (and CORAL2) releases

- ROOT6 was (is?) choking on headers using Boost
  - The COOL C++ API with ROOT6 can no longer use Boost
    - It was moved to c++11 instead – the new COOL3 (and CORAL3)

- Consequence: two COOL and CORAL code branches have been maintained in parallel for almost two years
  - (Largely reusing identical files with #ifdef's actually)

ROOT6 migration issues for PyCool

(2) C++ exceptions in ROOT 6.02 (JIT) vs 6.04 (ORCJIT)

- PyCool users (ATLAS and internal tests) heavily rely on C++ exception being properly translated to Python exceptions
  - And Python try/except is even more common than try/catch in C++!

- ROOT 6.00 and 6.02 (JIT) cause an unrecoverable Python abort on Linux whenever a C++ exception is thrown
  - C++ exceptions are only translated to Python in ROOT 6.04 (ORCJIT)

- Consequence: urgent workarounds were added to COOL to allow ATLAS ConditionsDB tools to work with ROOT 6.02
  - C++ helper functions: add a status return to convey C++ exception
  - These (many) ugly patches will be removed when only ~~6.04~~ 6.08? is left
    - *Same issue now found on Mac on 6.06: wait for 6.08 ☹*

Was ROOT the right choice for Pythonization?

- PyCool uses Reflection/Reflex/ROOT since always (10 years)
  - This seemed the road HEP would take (also for I/O), and it did
  - Advantages? Automatic parsing of headers... (and good support)
  - Disadvantages? Extra dependency on ROOT (on the release cycle, on the occasional bugs – and initially even on unwanted graphic libraries!)

- PyCoral uses manual coding of CPython bindings for CORAL
  - Developed around the same time in what was then a separate project
  - Advantages? Standalone solution, from first principles
  - Disadvantages? Maintainability (must follow all C++ API changes)
  - If anything, I would now move PyCoral to the PyCool ROOT solution

- Many other solutions out there now (Boost, cython...)
  - Maybe these were not mature 10 years ago, but they probably are now

Conclusions

- ROOT is successfully used in COOL since 10 years for creating Python bindings for C++ (PyCool)
  - Many evolutions so far (and more to come with PCM?)
- ROOT6 (especially ROOT6.04) has made this easier
  - But the road has been rough, and we have not arrived yet
- Just-in-time parsing makes ROOT6 an attractive and mature solution for creating C++/Python bindings
  - But alternative competitors are also more mature by now!
  - To make it even more attractive, making it more modular, lightweight (~~and ipython friendly!~~) can only help IMHO

(being) done! see next talk...

Many thanks to the ROOT team for their support of PyROOT!