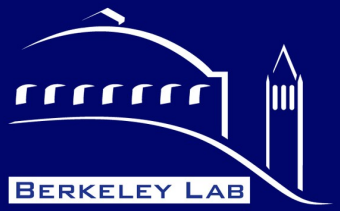
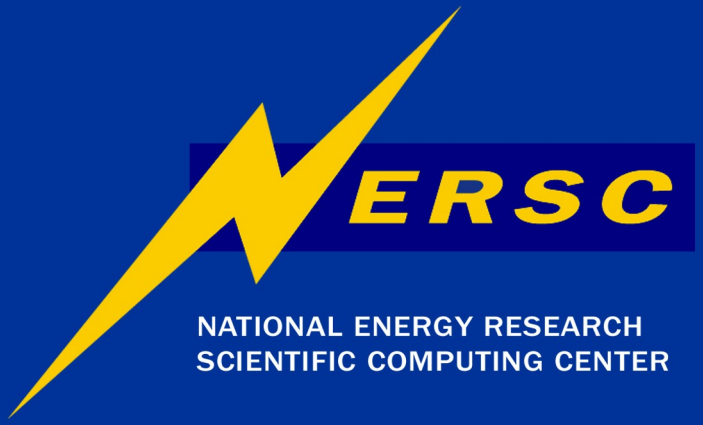




National Energy Research Scientific Computing Center (NERSC)

PyROOT Developments

Wim Lavrijsen

ROOT Workshop '07 – CERN, March 2007

Outline

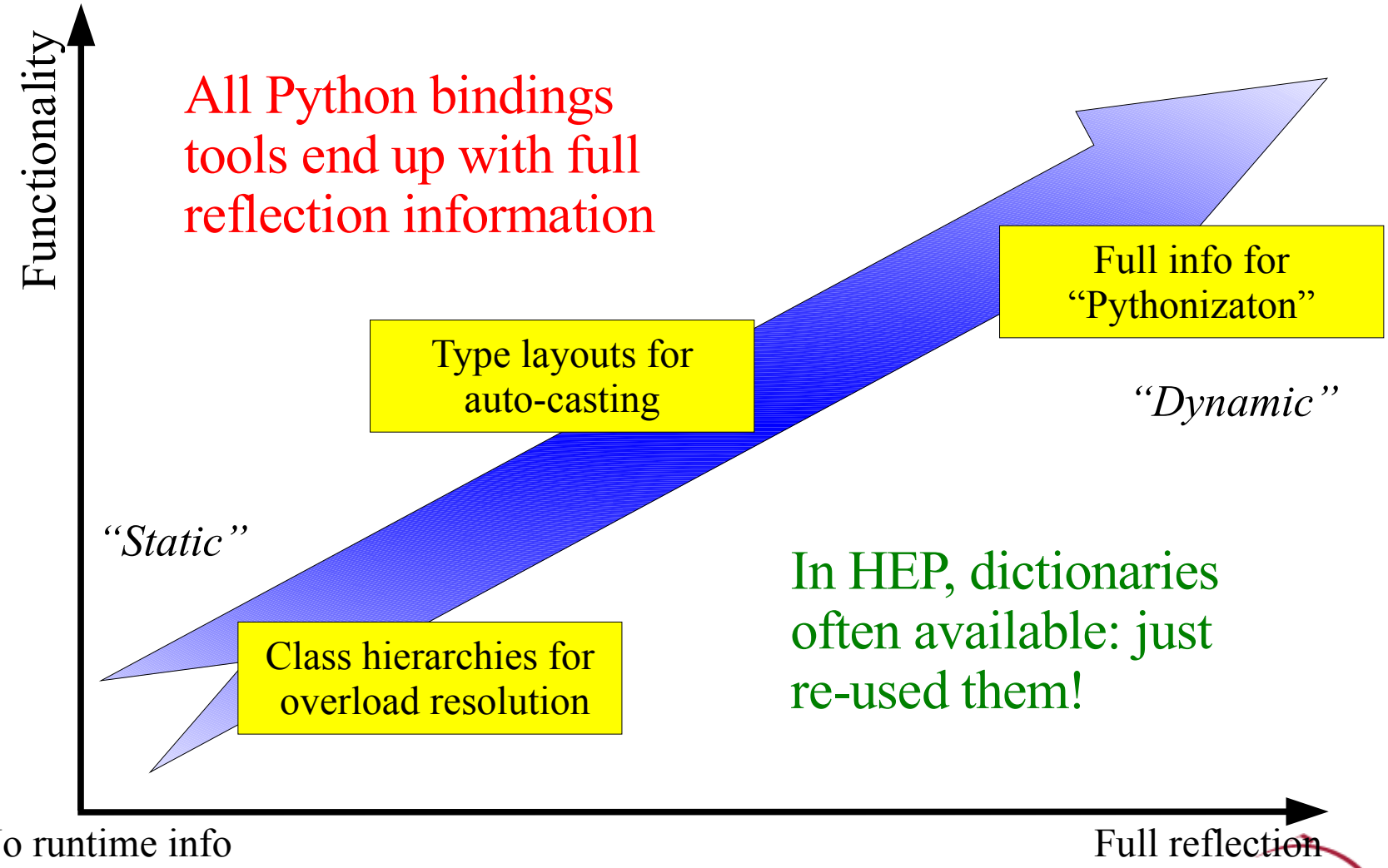
- Introduction
- Developments since ROOT'05
- Ongoing developments
  - TTree ease-of-use
  - Prototypes: cross-inheritance, Reflex
  - Interoperability with externals
- Optimizing Python code
- Conclusion
  - Future plans, further resources

Introduction

Overview

- **Python bindings to ROOT**
  - Python is a very popular dynamic language
  - Every app/lib out there has Python bindings
    - Including LHCb/Atlas framework: Gaudi/Athena
- **Two-way: access to Python from RINT**
  - E.g. ATLAS framework from CINT: athena.C
  - Interactive mixing of interpreters
- **Fully dictionary-based**
  - Often available, i.e. zero end-user effort

Functionality v.s. Available Runtime Language Info



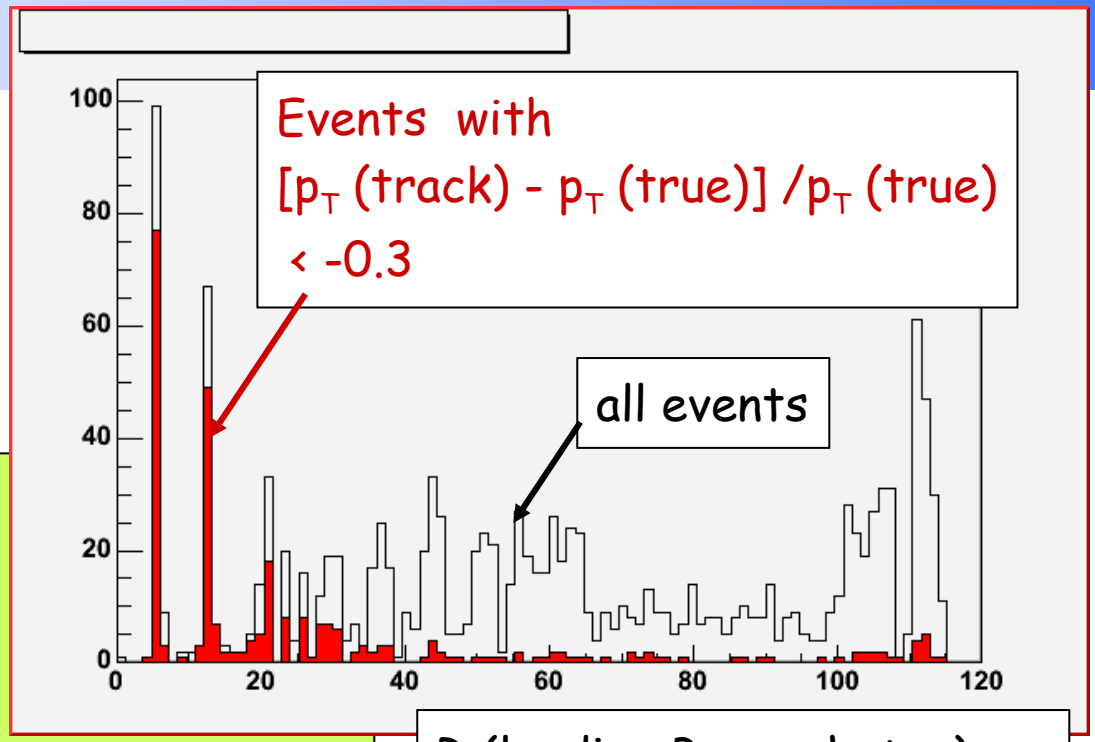
History

- **Pere Mato's RootPython (07/02)**
  - Based on boost.python v1, in Gaudi CVS
- **Rewrite: PyROOT in SEAL (03/03)**
  - Based on boost.python v2, in SEAL CVS
- **Python C-API based PyROOT (02/04)**
  - Released with ROOT 4.00/04 and later
- **Optimize and finish edges (06/05)**
  - Current core, satisfy Cintex/Reflex
- **Pythonizations (today)**
  - Support and consolidation

Example: ROOT from Gaudi

Interactive analysis of

Z –> ee events



PyROOT

```
import ROOT
import math
#
theApp.initialize()

h16 = ROOT.TH1F ("h16",
      'R G4 Brem photons',100, 0., 120.)
for i in range(5000):
  theApp.nextEvent()
  scon=PyParticleTools.getTruthParticles("SpclMC")

  for p in scon:
   t=p.getGenParticle()
   tver=t.production_vertex()
   if tver :
    R=math.pow(tver.point3d().y(),2)\
       +math.pow(tver.point3d().x(),2)
    R=math.sqrt(R)/10.

    if p.pdgId()==22 and math.fabs(t.barcode()) > 10000.:
     if math.fabs(p.eta())< 1.0:
       h16.Fill(R)
```

Access to Gaudi/Athena framework data objects from analysis tools

F. Gianotti

Example: Python from CINT/C++

```
class MyPyClass:
   def __init__( self ):
      print 'in MyPyClass.__init__'
   def gime( self, what ):
      return what

root [0] TPython::LoadMacro( "MyPyClass.py" );
root [1] MyPyClass m;
in MyPyClass.__init__
root [2] char* s = m.gime( "aap" );
root [3] s
(char* 0x41ee7754)"aap"
root [4] int i = m.gime( 2 );
root [5] i
(int)2
```

Supported Features

- **C++ types**
  - All C++ primitive and most of Python builtin types
  - Templates, inner classes, incomplete types
- **C++ namespaces**
  - Mapped onto Python classes
  - One global namespace (“ROOT”)
- **Class methods and functions**
  - Global, class static, and member functions
  - Arguments by value or by reference; defaults
  - Overloading through sequential dispatch, after sorting
    - Matches are cached based on hash of args types
  - Auto-casting of return values to dynamic type

Supported Features (2)

- **Additional supported features**
  - Public data members accessible as properties
  - Enums (const), typedefs (refs), global variables (properties)
  - Doc strings created from signatures
  - Settable operator[](int), smart pointers, etc.
  - C++ operators mapped to Python equivalents (+ extras)
- **Emulation of Python containers**
  - STL containers => Python collections (iterators, slicing)
  - Automatic tupling of pairs when looping over maps
- **Memory management**
  - Two schemes (heuristic and strict) + callbacks
  - Fine-tunable by the user (take, relinquish ownership)

Mapping C++ to Python

C++ construct	Python equivalent	Example
Primitive types: bool, char, short, int, long, unsigned log, float, double, char*, std::string	Converted to: bool, int, int, int, long, long, float, float, string, string	
Namespace separation (::)	scope separation (.)	`Gaudi.Time`
Global namespace (::)	ROOT, gaudimodule.gbl	`gbl.Gaudi.Time`
Template class <…>	<…> replaced by (…)	`std.vector( float )`
NULL pointer	None	`func(None, None)`
Dynamic cast	Always dynamic type	`tk = event('track')`
Complex mix of object value, pointers and references	Always object references	
Iterators	Iteration protocol	`for o in vector:` `  ...`

Developments since ROOT'05

Lots of Minor Improvements

- **Handling of fringe cases**
  - Missing dicts, creation order, shutdown sequence, double imports, null pointers, spurious lookups, float/double overloads
- **Python and C++ features:**
  - Reference types, smart pointers, data type limits, repr strings, char <=> string, settable operator[]
- **New/improved pythonizations:**
  - TClass casts, TClonesArray filling, TTree branch access, std::map loops, safe std::vector indexing
- **Python 2.5 and 64bit support**
- **400+ unit/regression tests and counting**

C++ Memory Problems

- **Python for close-out/post-mortem**
  - Often, C++ memory issues are constrained
    - E.g. at start of function => no/small side effects
  - Print error/stack trace and set exception

```
>>> Tstring().Puts(0)          # expects FILE* (opaque)
 *** Break *** segmentation violation
 Generating stack trace...
 0x402035bb in _IO_fputs + 0x2b from /lib/tls/libc.so.6
 0x40604c5f in TString::Puts(_IO_FILE*) + 0x21 from lib...
 0x4088bd03 in <unknown> from libCore.so
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SystemError: problem in C++; program state has been reset
>>>                          # <= back at normal python prompt
```

New Pythonizations

```
>>> from ROOT import *
>>> mymap = std.map( int, int )()
>>> for i in range( 3 ):
...     mymap[i] = i                    # __setitem__
...
>>> for key, value in mymap:            # __iter_ and unpacking
...   print key, value                  #  of pair<int,int>s
0 0
1 1
2 2
>>> cr = TClassRef( "TObject" )   # smart pointer object
>>> print cr.GetListOfAllPublicMethods.__doc__
TList* TClass::GetListOfAllPublicMethods()
>>> print len( cr.GetListOfAllPublicMethods() )
90                                      # from TClass object
>>>
```

TClonesArray

```
>>> from ROOT import *
>>> c = TClonesArray( "TVector3" )
>>> c[ 0 ] = TVector3()     # takes ownership of temporary
>>> c[ 1 ] = TVector3()
>>> v = TVector3()          # not a temporary
>>> c[ 1 ] = v              # calls dtor on old TVector3
>>> c[ 1 ] = TVector3()     # v is now destroyed
>>> print v
None
>>> print c[8]              # out-of-range (empty memory)
None
>>> print list(c)           # iterable-safe
[<ROOT.TVector3 object ("TVector3") at 0x86c9530>,
  <ROOT.TVector3 object ("TVector3") at 0x86c96d8>]
```

=> NOTE: this is just to make TClonesArray usable, the performance benefits are lost!

TTree

```
>>> f = TFile( 'test.root', 'recreate' )
>>> t = TTree( 'mytree', '' )
>>> v = std.vector( float )()      # std::vector< float >
>>> t.Branch( 'data', v )          # one branch "data"
>>> for i in range(10):
...     v.push_back( i )
...
>>> t.Fill()                       # write 0.0 ... 9.0
>>> f.Write()

>>> f = TFile( 'test.root' )       # mytree implicit
>>> for event in mytree:           # loop over entries in tree
>>>    for i in event.data:         # access data as if member
>>>       print i                   # pythonized vector access
0.0
[..]
```

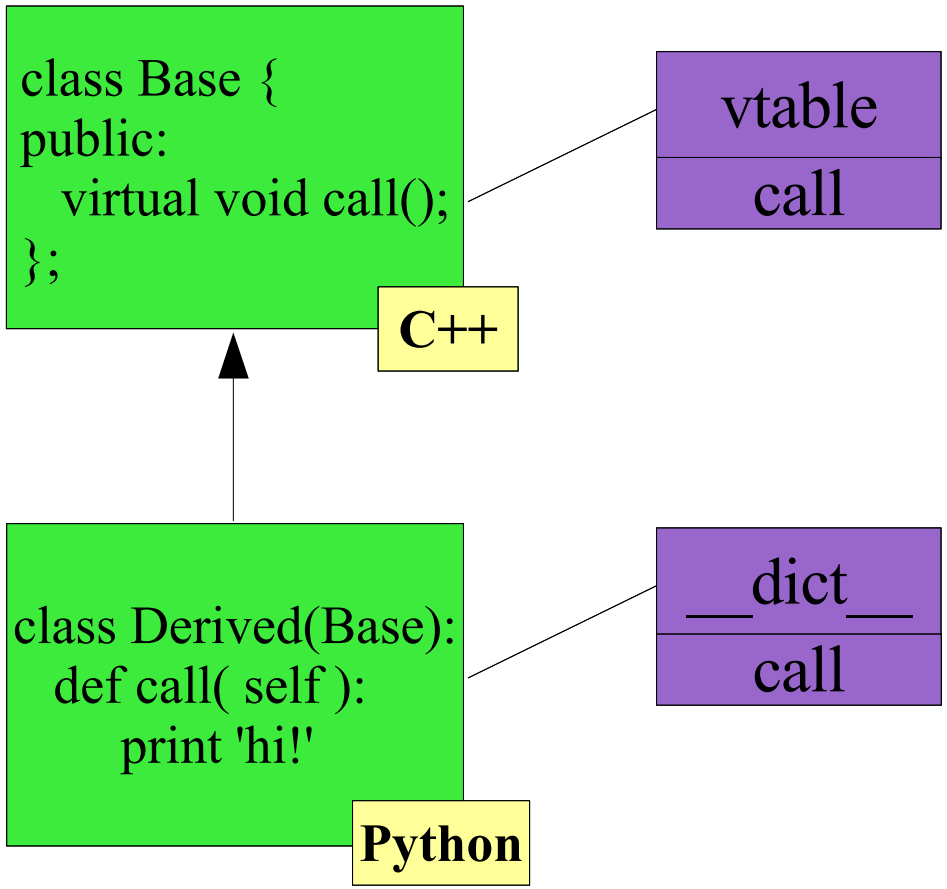
Optimizations needed: read branch on access only, removal of layers, loop optimizations

Ongoing Developments

TTree Ease-of-Use

- **Treat TTree as Python-style container**
  - Loop over TTree entries with *for*-loop ✓
  - Bounds and read-sanity checks ✓
  - Select slices (ranges)
- **Treat TTree as C-style struct**
  - Access branches as data members ✓
- **Optimize Python access and I/O**
  - Judicious use of caches
  - Read branches only on-demand

Cross-Inheritance



C++ sees only base class method “call”, which has an entry in the vtable

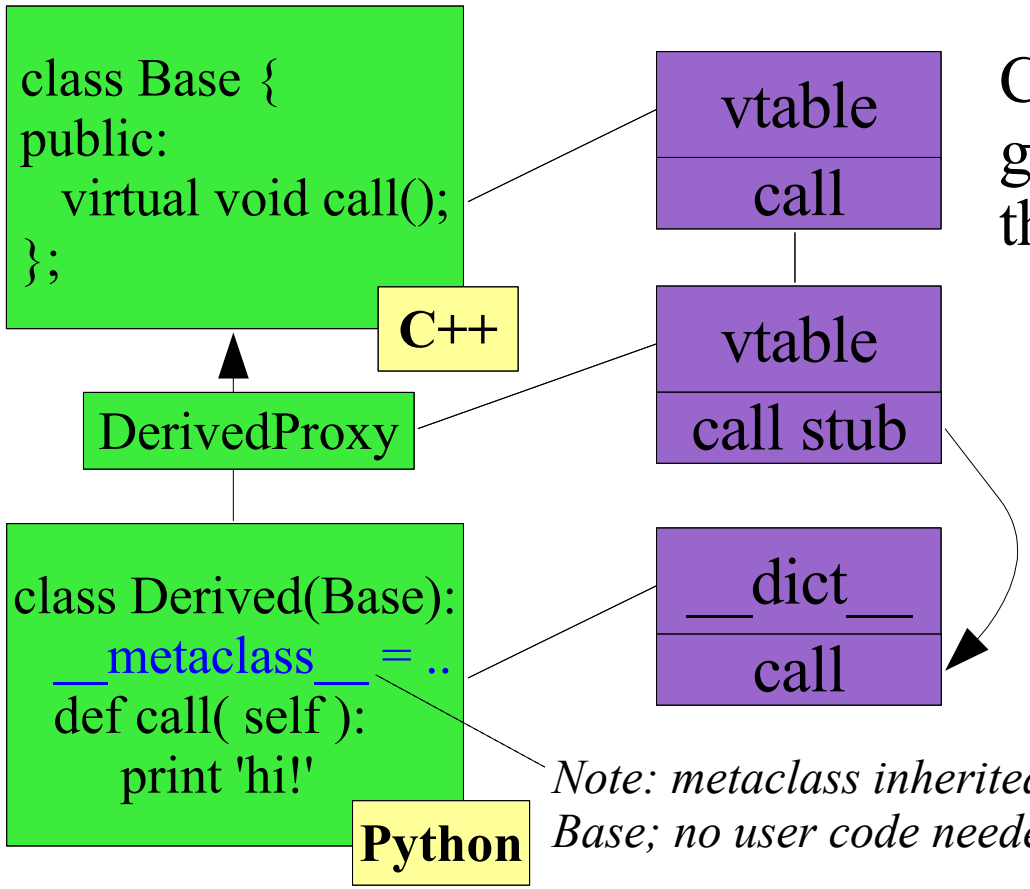
=> *no override*

Python sees both base and derived methods “call”, in the respective __dict__s

=> *override works*

Different results when calling from Python or C++!

Cross-Inheritance



C++ DerivedProxy with stubs generated by __metaclass__, then compiled with ACLiC

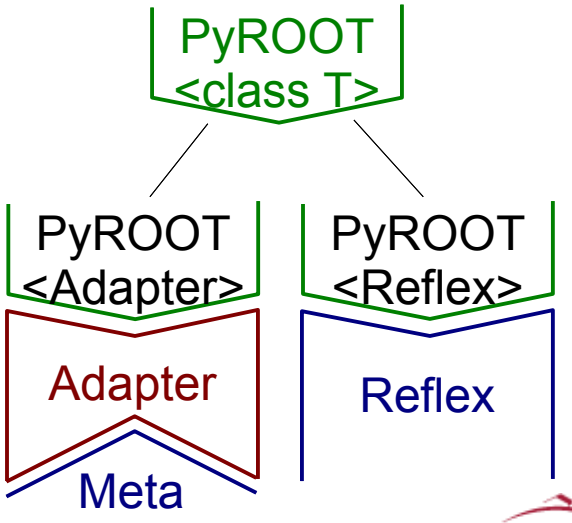
C++ calls DerivedProxy through normal vtable mechanism, which calls into Python through stub

Same results from C++ and Python!

Prototype

Direct Reflex Support

- **Implementation of Reflex support:**
  - Templated most of PyROOT
  - Put adapters on top of root/meta
    - Closely follow Reflex API ("as necessary")
  - Two instantiations
  - No change to CINT entry
  - Reflex by callback
    - Activated with a flag
- **Basic classes only**
  - To be continued ...



Interfacing External C/C++ Libs

- **Many different Python bindings**
  - SIP, SWIG, Boost.Python, C-API, etc.
- **Communication via opaque pointers**
  - "Trust the developer"
  - Currently not well supported in PyROOT
    - Prototype with CObject derived
- **PyROOT doesn't release GIL**
  - Need to experiment with allowing threads

Optimizing Python Code

Optimize What?

- **Flexibility is great, if it is needed**
  - W/o language support *you* have to write it!
    - You're unlikely to do better than the language
- **Flexibility is a burden, if unneeded**
  - You would pay for what you don't use
    - C++: many complex trade-offs possible
    - Python: typically one, straightforward way
- **In inner loops, typically:**
  - No flexibility needed
  - Constrained block of code

=> Optimization

Example

```
>>> from ROOT import gRandom, TCanvas, TH1F
>>> c1 = TCanvas('c1','Example',200,10,700,500)
>>> hpx = TH1F('hpx','px',100,-4,4)
>>> for i in xrange(25000):
...     px = gRandom.Gaus()
...     hpx.Fill(px)
...
>>> hpx.Draw()
>>> c1.Update()
```

} => inner loop

- Normal, straightforward Python code
- With (readable) tricks: gain ~20%

Note: simplistic example, could use TH1F::FillRandom('gaus', 25000)

ACLiC (C/C++ "baseline")

```
>>> fname = __file__[:-3]+'.C'
>>> open( fname, 'w' ).write( """#include "TRandom.h"
#include "TH1F.h"

void speed2( TH1F* h, Long_t nmax ) {
   for ( Long_t i = 0; i < nmax; ++i ) {
      double px = gRandom->Gaus();
      h->Fill( px );
   }
}
""" )
>>> gROOT.LoadMacro( fname + '+' )
>>> from ROOT import speed2
>>> hpx = TH1F('hpx','px',100,-4,4)
>>> speed2( hpx, 250000 )
```

Standard Python way:

1) Develop code in Python
2) Move slow parts to C/C++

- Pro: 20x speed-up
- Con: more code to write

The Issues

```
>>> from ROOT import gRandom, TCanvas, TH1F
>>> c1 = TCanvas('c1','Example',200,10,700,500)
>>> hpx = TH1F('hpx','px',100,-4,4)
>>> for i in xrange(25000):
...     px = gRandom.Gaus()
...     hpx.Fill(px)
...
>>> hpx.Draw()
>>> c1.Update()
```

Iterator temporary

Method lookups

Object temporary

Bound method temporary

} => inner loop

=> Python creates many temporaries

=> No type optimizations applied

Note: simplistic example, could use TH1F::FillRandom('gaus', 25000)

Optimization in PyROOT?

- **Main ingredients in inner loop codes:**
  - Numeric: math + builtins + arrays
    - Use NumPy with Psyco (JIT, x86 only)
  - ROOT extension library calls
    - Use dictionary info for static optimization
    - Dynamically change internal call settings
      - No autocast, return in local buffer, remove checks, etc.
- **Need fixed block of python code**
  - User defined, different optimization levels
  - But also such as THn::Fit and TFn::Draw

Prototyping Results

- **Effectiveness varies wildly ...**
  - Optimizations on builtins: no effect
  - Shortcircuit call stack: 45%
  - No check/cast on object ptrs: 10%
  - Drop bound method alloc: 25%
- **Effects are independent, so cumulative**
  - Total gain of about a factor of 3 in speed
- **Perhaps can be offered outside blocks**
  - Define configuration interface (or API)

Conclusion

Future Plans

- **Finish prototypes and release them**
  - Cross-inheritance, TTrees, Reflex
- **API to Converters/Executors needed**
  - Allow user to handle specific/new types
  - Possible today with ACLiC, but clumsy
- **User-side Python code-optimizations**
  - Especially for fixed/inner loops
- **Improve dependencies**
  - Load less libraries on startup

Resources

- **Documentation**
  - Chapter 20 of the ROOT User's Guide
  - http://cern.ch/wlav/pyroot
- **Examples**
  - $ROOTSYS/tutorials/pyroot/*.py
- **Code Repository**
  - root.cern.ch/viewcvs/pyroot
- **Python optimizations**
  - http://wiki.python.org/moin/PythonSpeed/
  - http://www.scipy.org/