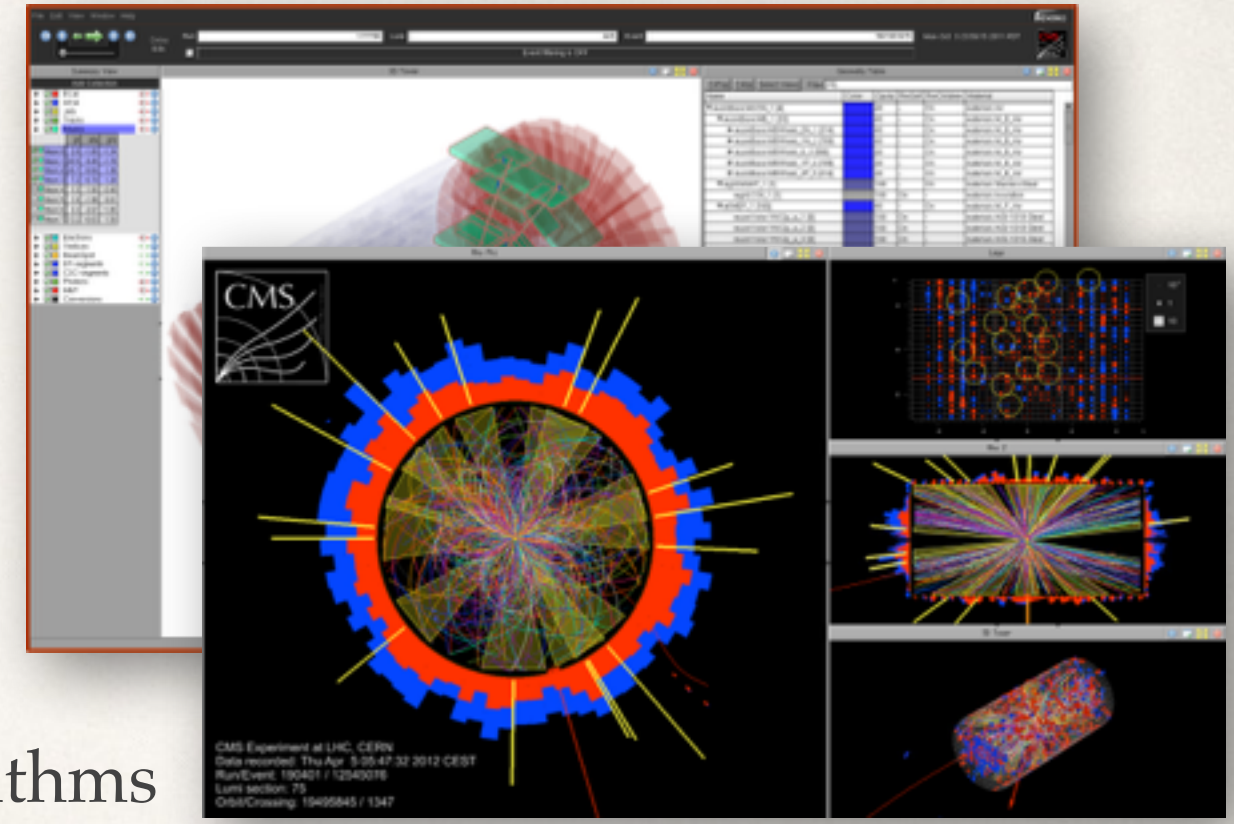
ROOT
Data Analysis Framework

PyROOT: Seamless Melting of C++ and Python

Pere MATO, Danilo PIPARO
on behalf of the ROOT Team

ROOT

- “At the root of the experiments”, project started in 1995
- Open Source project (LGPL2)
  - mainly written in C++; 4 MLOC
- ROOT provides (amongst other things):
  - C++ interpreter, Python bindings
  - Efficient data storage mechanism
  - Advanced statistical analysis algorithms
    - histogramming, fitting, minimization, statistical methods …
  - Multivariate analysis, machine learning methods
  - Scientific visualization: 2D/3D graphics, PDF, Latex
  - Geometrical modeler
  - PROOF parallel query engine

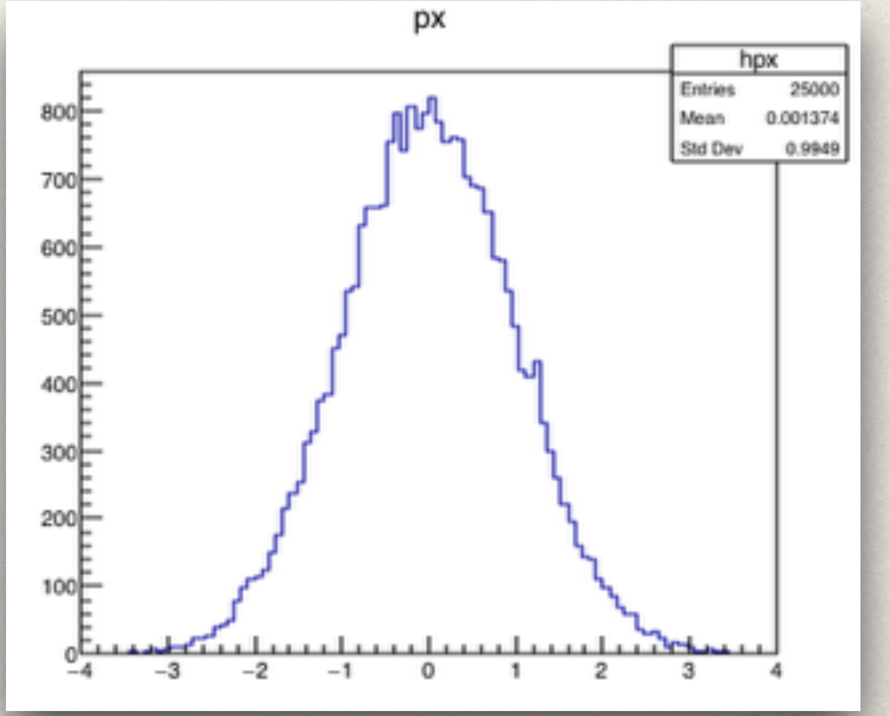


http://root.cern.ch

PyROOT

- The ROOT Python extension module (PyROOT) allows users to interact with any C++ class from Python
  - Generically, without the need to develop specific bindings
  - Mapping C++ constructs to Python equivalent
- Give access to the whole Python ecosystem

```
# Example: displaying a ROOT histogram from Python
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()
   i = hpx.Fill(px)
hpx.Draw()
```

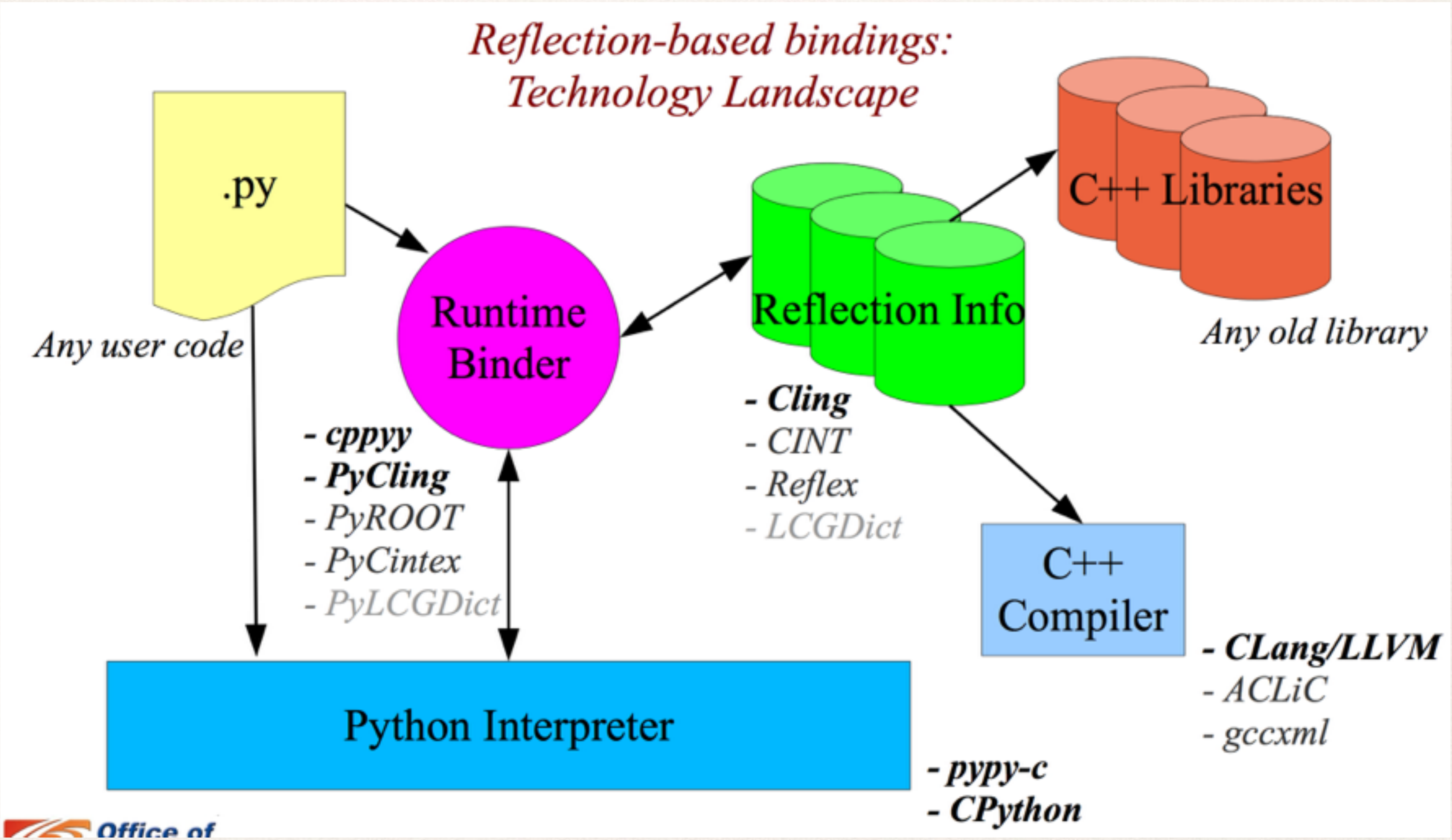


ROOT
Data Analysis Framework

Why PyROOT is Special?

- Python bindings based on C++ reflexion information
  - Python classes are created dynamically when needed
  - C++ globals and functions appear automatically in Python
  - Much less work than using binding tools (e.g. boost, swig)
- Uniform mapping of C++ idioms to Python equivalent
  - Same behavior everywhere
  - Essential for managing large code bases
- Standard "Pythonizations" of C++ classes and constructs
- Two-way interaction
  - Calling Python from C++, and calling C++ from Python

Reflexion-Based Bindings



C++ to Python Mapping

C++	Python
basic_types: short, int, float, double, std::string, char*, …	int, [long], float, str
basic_type*, C-array	array (module)
class, template class	class, class generator
STL classes	std.vector, std.list, std.shared_ptr,…
inheritance, dynamic_cast	inheritance, always final type
namespace	scope (dictionary)
pointer, reference	reference
exceptions	exceptions

ROOT Data Analysis Framework

Memory Management

- Python handles memory for the user by employing reference counting and a garbage collection. In C++ memory handling is done by hand or by a 'framework'
- Two global policies: *heuristics[default]* and *strict*
- For the *heuristic* policy the following rules are observed:
  - An object created on the python interpreter side is owned by python and will be deleted once the last python reference to it goes away
  - An object coming from a call is not owned. When the object goes out of scope on the C++ side, the python object will change type into an object that largely behaves like None
- Ownership can be set/relinquish for individual objects

ROOT Data Analysis Framework

Function Overloading

- A single Python function acts as a proxy to the set of overloaded C++ functions

```
void func(int) { cout << "func(int) called" << endl;}
void func(double) { cout << "func(double) called" << endl;}
void func(int, double) { cout << "func(int, double) called" << endl;}
```

```
from ROOT import func
func(10)
func(9.9)
func(1,2.)
```

```
func(int) called
func(double) called
func(int, double) called
```

```
func('a')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-fa56e76b52e2> in <module>()
----> 1 func('a')

TypeError: none of the 3 overloaded methods succeeded. Full details:
  void ::func(int) =>
    could not convert argument 1 (int/long converion expects an integer object)
  void ::func(double) =>
    could not convert argument 1 (a float is required)
  void ::func(int, double) =>
    takes at least 2 arguments (1 given)
```

ROOT Data Analysis Framework

Running Python from ROOT

* ROOT user can run any Python command and eventually switch to the python prompt from the ROOT prompt
* The interpreter state will be preserved in between calls

```
root [0] TPython::Exec( "print 1 + 1" )
2
root [1] b = TPython::Eval( "TBrowser()" )
(class TObject*)0x8d1daa0
root [2] TPython::Prompt()
>>> i = 2; ^D
root [3] TPython::Prompt()
>>> print i
2
```

ROOT Data Analysis Framework

Python Callbacks

```
from ROOT import TF1, TH1F, TCanvas

class Linear:
   def __call__( self, x, par ):
      return par[0] + x[0]*par[1]

# create a linear function for fitting
f = TF1( 'pyf3', Linear(), -1., 1., 2 )

# create and fill a histogram
h = TH1F( 'h', 'test', 100, -1., 1. )
f2 = TF1( 'cf2', '6.+x*4.5', -1., 1. )
h.FillRandom( 'cf2', 10000 )

# fit the histo with the python 'linear' function
h.Fit( f )

# print results
par = f.GetParameters()
print 'fit results: const =', par[0], ', pitch =', par[1]
```

```
 FCN=115.87 FROM MIGRAD    STATUS=CONVERGED      29 CALLS          30 TOTAL
                     EDM=1.68681e-15    STRATEGY= 1      ERROR MATRIX ACCURATE
  EXT PARAMETER                                STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  p0           9.88413e+01   9.94190e-01   4.56265e-03   6.71977e-08
   2  p1           7.55552e+01   1.53812e+00   7.05889e-03  -2.14706e-08
fit results: const = 98.8412955492 , pitch = 75.5551735795
```

E.g. Fit function in Python

ROOT Data Analysis Framework

Python inheriting from C++ class

C++

```
class Base {
   ...
   virtual void Foo() { cout << "base::Foo" << endl; }
   void CallFoo() { this->Foo(); }
};

class PyBase : public Base { ... };
```

Adaptor class

Python

```
class PyDerived(ROOT.PyBase):
   def __init__(self): ROOT.PyBase.__init__(self, self)
   def Foo(self): print 'Python::Foo'
```

```
d = PyDerived()
d.CallFoo()
o = ROOT.Base()
o.CallFoo()
```

```
Python::Foo
base::Foo
```

ROOT Data Analysis Framework

CLING

- Replaces CINT: a radical change at the core of ROOT
- Based on LLVM and CLANG libraries.
  - Piggy back on a production quality compiler rather than using an old C parser
  - Future-safe - CLANG is an active C++ compiler
  - Full support for C++11/14 with carefully selected extensions
  - Script's syntax is much stricter (proper C++)
  - Use a **C++ just in time compiler** (JIT)
  - A C++11 package (e.g. needs at least gcc 4.8 to build)
- Support for more architectures (ARM64, PowerPC64)

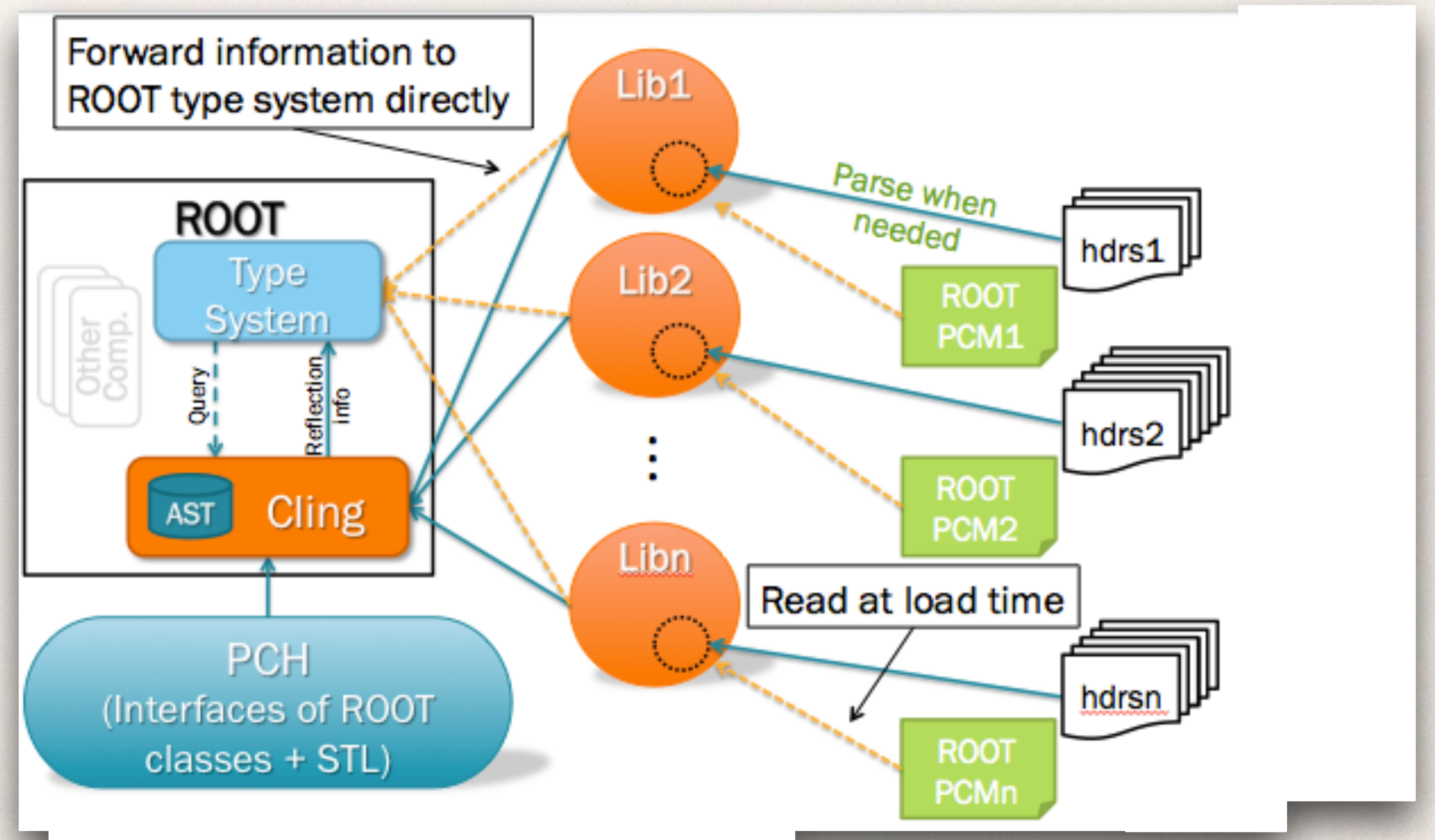
Clang, the AST and ROOT

- C++ entities in Clang: **Abstract Syntax Tree** (AST)
  - Classes, functions, templates, statements …
  - Exists in memory and can be persisted on disk in two forms:
    - 1) **Pre-Compiled Header**: can load only one, file granularity
    - 2) **Pre-Compiled Modules**: can load many, AST node granularity
  - Both queried lazily by the compiler
- Original ROOT6 design: AST source of information for
  - ROOT Dictionaries: a thin layer around portions of AST
  - Reflection and I/O
  - Interactive function calls

ROOT Data Analysis Framework

Evolution Reflex/CINT -> CLING

- With ROOT 6 CLING replaces CINT
- PyROOT has been adapted to the new way to obtain reflection information



ROOT Data Analysis Framework

Python with JIT Reflection

```
#include <iostream>
class A {
 public:
  A(const char* n) : m_name(n){}
  void printName() {std::cout<< m_name
                    << std::endl;}
 private:
  const std::string m_name;
};
int dummy() {return 42;}
```

A.h

```
import ROOT
ROOT.gInterpreter.ProcessLine('#include "A.h"')
a = ROOT.A('my name')
a.printName()
ROOT.dummy()
```

```
42

my name
```

python

No need for dictionaries!!

Great potential for 3rd party libraries

ROOT Data Analysis Framework

C++11

many C++11 keywords and concepts

C++

```
constexpr int data_size() { return 5; }
auto N = data_size();

template<class L, class R> struct MyMath {
    static auto add(L l, R r) -> decltype(l+r) { return l+r; }
};

template class MyMath<int, int>;
```

Python

```
print 'N =', ROOT.N
print '1+1 =', ROOT.MyMath(int, int).add(1,1)
```

```
N = 5
1+1 = 2
```

ROOT Data Analysis Framework

Recent Features

- Multiple virtual inheritance fully supported
  - Uses clang AST to get the relative offsets
- New C++11 declarations (resolve to simpler terms)
  - E.g. auto → real type after the compiler is done with it
- C++11 implementations not visible to bindings
  - => automatically okay
- Python3 support
  - Almost completed the support for recent versions of Python
  - LHC experiments using Python 2.x

Pythonizations

- PyROOT is mostly about automatic bindings
  - Sine qua non: unwieldy and unmaintainable otherwise
- Dictionaries (incl. for experiment data) widely available
  - and maintained for I/O and CLING in experiment releases
- Automatic bindings often feel too much like C++
  - Some C++ idiosyncrasies still visible
  - Memory management is not 100% solved
- Some limited, still generic, **Pythonizations** exist
  - E.g. TFn, TTree, looping over std::vector, etc.
- Other packages such as *rootpy* provides a more pythonic ROOT

ROOT
Data Analysis Framework

Pythonization Examples

```
from ROOT import vector
v = vector(str)()
v.push_back('a')
v.push_back('b')
v.push_back('c')
for c in v:
    print c
```

```
a
b
c
```

iterators

```
from ROOT import std, TH1, TH1F
p = std.shared_ptr<TH1>()
if not p : p = TH1F()
p
```

```
<ROOT.TH1F object at 0x10130a810>
```

shared pointers

ROOT Data Analysis Framework

Pythonization Examples

```
mymap = ROOT.map(str,int)()
mymap['a'] = 1
mymap['b'] = 2
```

operator []

```
print mymap
for label, value in mymap:
    print label,value
```

```
<ROOT.map<string,int> object at 0x11c9a9790>
a 1
b 2
```

iterators

std<pair> to tuple

```
print len(mymap)
```

```
2
```

std::map::size()

ROOT
Data Analysis Framework

Performance

```python
for i in xrange(100):   # make it measurable
  sumAge = 0
  for entry in tree:
    sumAge += entry.Age
print sumAge
```

Python

```
158151
CPU times: user 1.64 s, sys: 181 ms, total: 1.82 s
Wall time: 1.74 s
```

example reading a TTree (cernstaff.root)

```cpp
int sumAge;
for( int i = 0; i < 100; i++) { // make it measurable
    TTreeReader reader(tree);
    TTreeReaderValue<int> age(reader, "Age");
    sumAge = 0;
    while (reader.Next()) {
        sumAge += *age;
    }
}
cout << sumAge << endl;
```

C++

```
158151
CPU times: user 51.9 ms, sys: 2.74 ms, total: 54.6 ms
Wall time: 54.8 ms
```

No surprise, C++ is much faster!!

ROOT Data Analysis Framework

Functional Chains

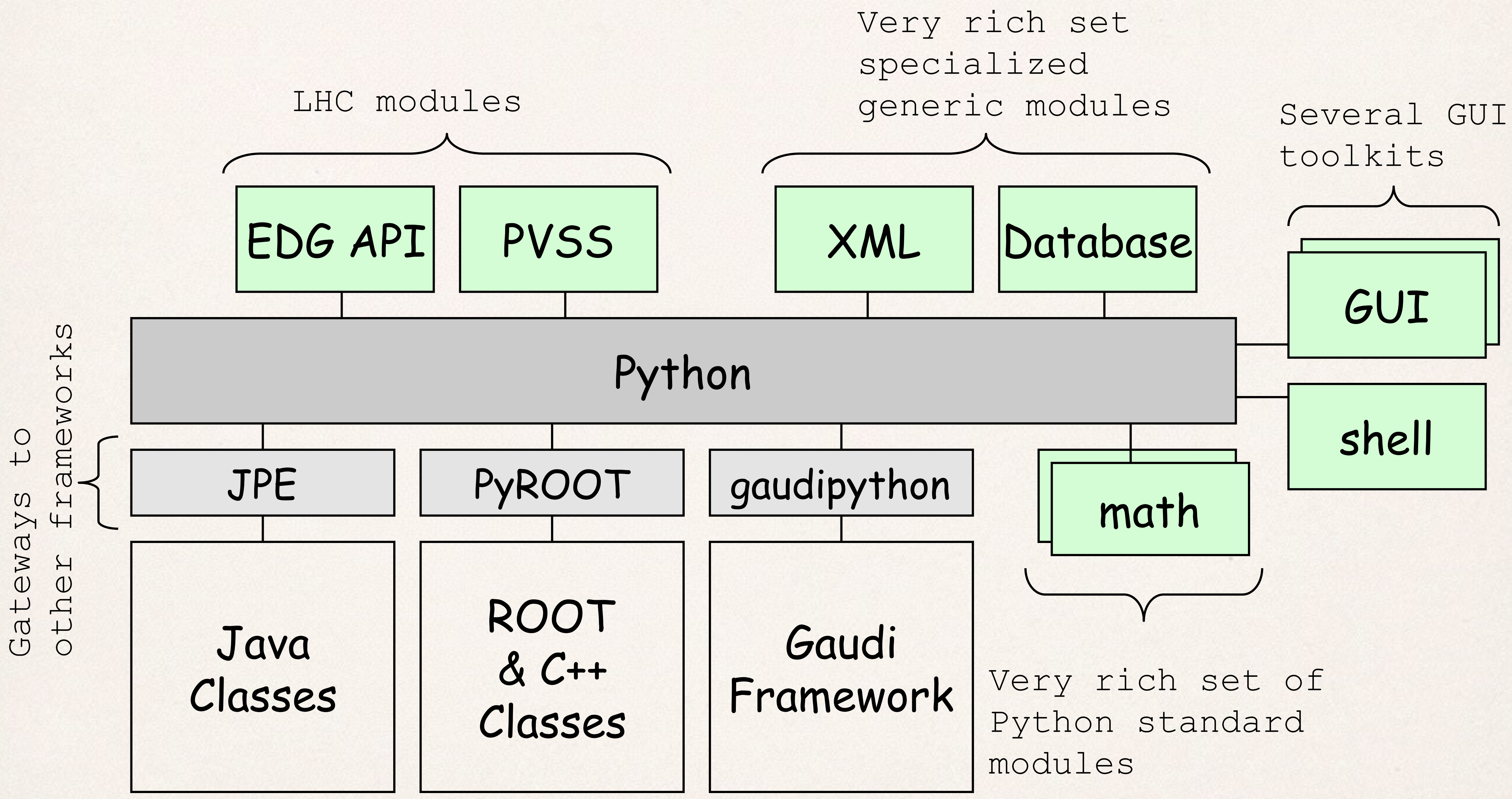
- Prototyping some ideas of 'declarative/functional' chains of basic concepts such as map, filter, reduce, accumulate, etc.
  - Inspired from data analytic tools such as Spark
- The user specifies the **What** and system chooses **How**
  - Actions are only triggered at the end of the chain
  - Great opportunity for optimizations (partitioning, caching, re-ordering, etc.)

```
hist = ttree.filter(lambda event: event.Emiss > 40)
            .flatMap(lambda event: event.tracks)
            .map(lambda track: sqrt(px**2 + py**2))
            .histo(100, 0, 20)
```

The chain is only executed when is completed

ROOT
Data Analysis Framework

Python as a Software Bus



Conclusions

- PyROOT provides a very complete and generic binding between C++ and Python (not limited to ROOT classes)
  - See next two talks for applications of PyROOT in different domains
- PyROOT has been delivering good service to many physicists that prefer to use Python for interacting with ROOT
  - Difficult to asses the adoption level
  - From last survey, ~52% of ROOT users use the Python interface
- The JIT compilation coming with ROOT 6 provides even a more flexible and dynamic interface
  - JIT can help to recover the bad performance of interpreted Python by generating code for the “loops” and “number crunching”