A More Pythonic, Interoperable and Modern PyROOT

Stephan Hageboeck, Enric Tejedor, Stefan Wunsch
for the ROOT team

ACAT 2019
Saas Fee (Switzerland)

ROOT
Data Analysis Framework
https://root.cern

Introduction

ROOT

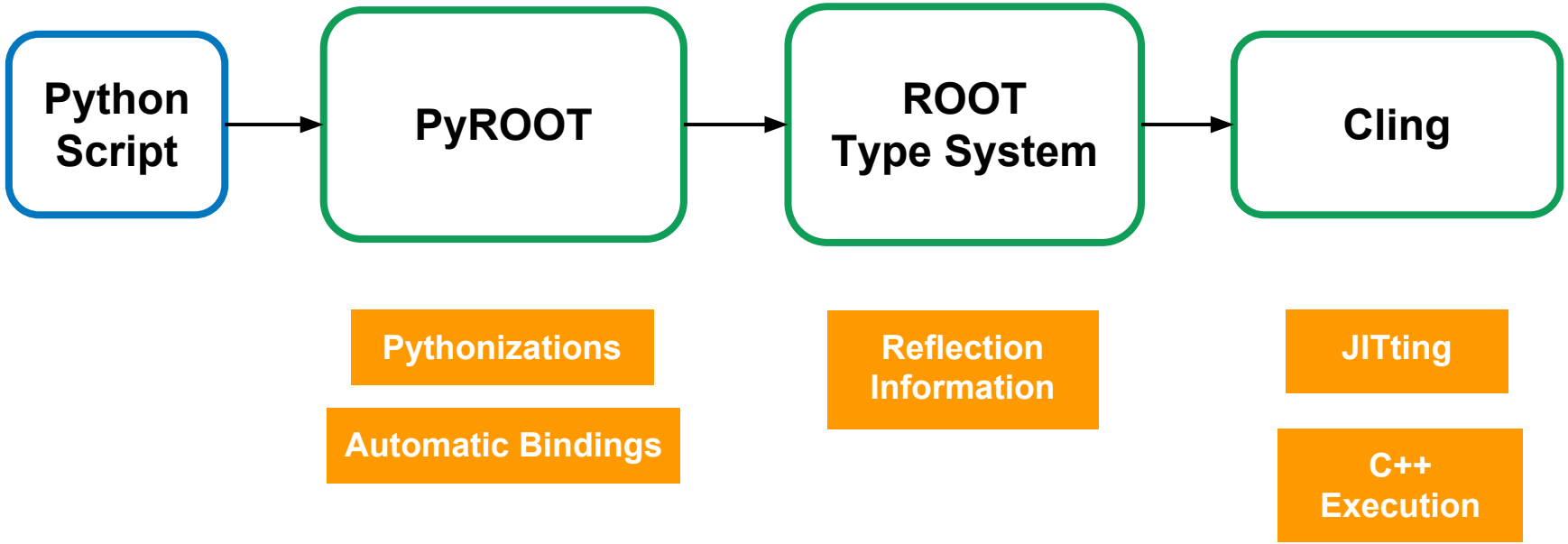
Data Analysis Framework

https://root.cern

PyROOT

- Python bindings offered by ROOT
- Access all the (not only!) ROOT C++ functionality from Python
  - Python façade, C++ performance
- Automatic, **dynamic**
  - No static wrapper generation
  - Dynamic python proxies for C++ entities
  - Lazy class/variable lookup
- Powered by the ROOT type system and Cling
  - Reflection information, JIT C++ compilation, execution
- **Pythonizations**
  - Make it simpler, more pythonic

The Structure



A Concrete Example

- Automatic bindings + Pythonizations

```
import ROOT

f = ROOT.TFile('myfile.root')

t = f.mytree

for event in t:
  ...
```

`TFile` is a (dynamic) Python proxy of a C++ class

`f` is a (dynamic) Python proxy of a C++ object

Pythonization: access tree as an attribute

Pythonization: iterate over tree events in a pythonic way

Three Main Development Lines

- The ROOT team has increased the effort in PyROOT
  - We are aware of the importance of Python for HEP!
- Main objective is to improve PyROOT in three ways:
  1. Modernize PyROOT with a new design
  2. Consolidate current PyROOT: add new features, fix issues
  3. Support better interoperability with data science Python ecosystem (NumPy, pandas)

New Features

ROOT

Data Analysis Framework

https://root.cern

C++ Vectors to Numpy…

- Zero-copy C++ to NumPy array conversion
  - Objects with contiguous data (`std::vector`, RVec)
  - Pythonization: tell NumPy about data and shape

Since 6.14

```
import ROOT
import numpy as np

vec = ROOT.std.vector('int')(2)
arr = np.asarray(vec) # zero-copy operation
vec[0], vec[1] = 1, 2

assert arr[0] == 1 and arr[1] == 2
```

Memory adopted!

… and Back!

Coming in 6.16/02

- Convert NumPy arrays to RVecs
  - Pass them into C++ functions
  - Conversion could be done implicitly in the future

```
arr = np.array([1,2,3])
vec = ROOT.AsRVec(arr) # zero-copy operation
my_cpp_fun(vec)
```

Reading TTrees as NumPy Arrays

- Read a TTree into a NumPy array
  - Branches of arithmetic types (ntuples)

Since 6.14

```
myTree # Contains branches x and y of type float

# Convert to numpy array and calculate mean values of all
# branches
myArray = myTree.AsMatrix()
m = np.mean(myArray, axis = 0)

# Read only specific branches
onlyX = myTree.AsMatrix(columns = ['x'])
```

RDataFrame to NumPy

- Even more powerful way to read TTrees into NumPy
  - All `RDataFrame` operations available
  - Optional parallelism

```
from ROOT import RDataFrame

df = RDataFrame('myTree', 'file.root')

# Apply cuts, define new columns
df = df.Filter('x > 0').Define('z', 'x*y')

# Column dictionary, each column is a NumPy array
cols = df.AsNumpy()
```

JITted C++ expression

Coming in 6.16/02

RDataFrame to pandas

```
# Run input pipeline with C++ performance that can process TBs of data, reads from remote, ...
df = RDataFrame('tree', 'file.root')
     .Filter('pT_j0>30', 'Trigger requirement')
     .Filter('n_jet >= 2', 'Jet multiplicity cut')
     .Define('r_j0', 'sqrt(eta_j0*eta_j0 + phi_j0*phi_j0)')

# Read out final selection with defined variables as NumPy arrays
col_dict = df.AsNumpy(['r_j0', 'eta_j0', 'phi_j0'])

# Wrap data with pandas
import pandas
p = pandas.DataFrame(col_dict)
print(p)

   r_j0  eta_j0  phi_j0
0  0.26  0.1    -0.5
1  1.0  -1.0     0.0
2  4.45  2.1     0.2
```

Coming in 6.16/02

(Py)ROOT Installation with Conda

- New and easy way to install PyROOT and its dependencies
- Currently available on Linux, Mac support underway
- Brief set of instructions:
  - Installing

```
conda create --name myenv --channel conda-forge python=3 root
```

  - Activating the environment

```
conda activate myenv
```

  - Deactivating the environment

```
conda deactivate
```

C. Burr,
E. Guiraud

The New PyROOT

ROOT

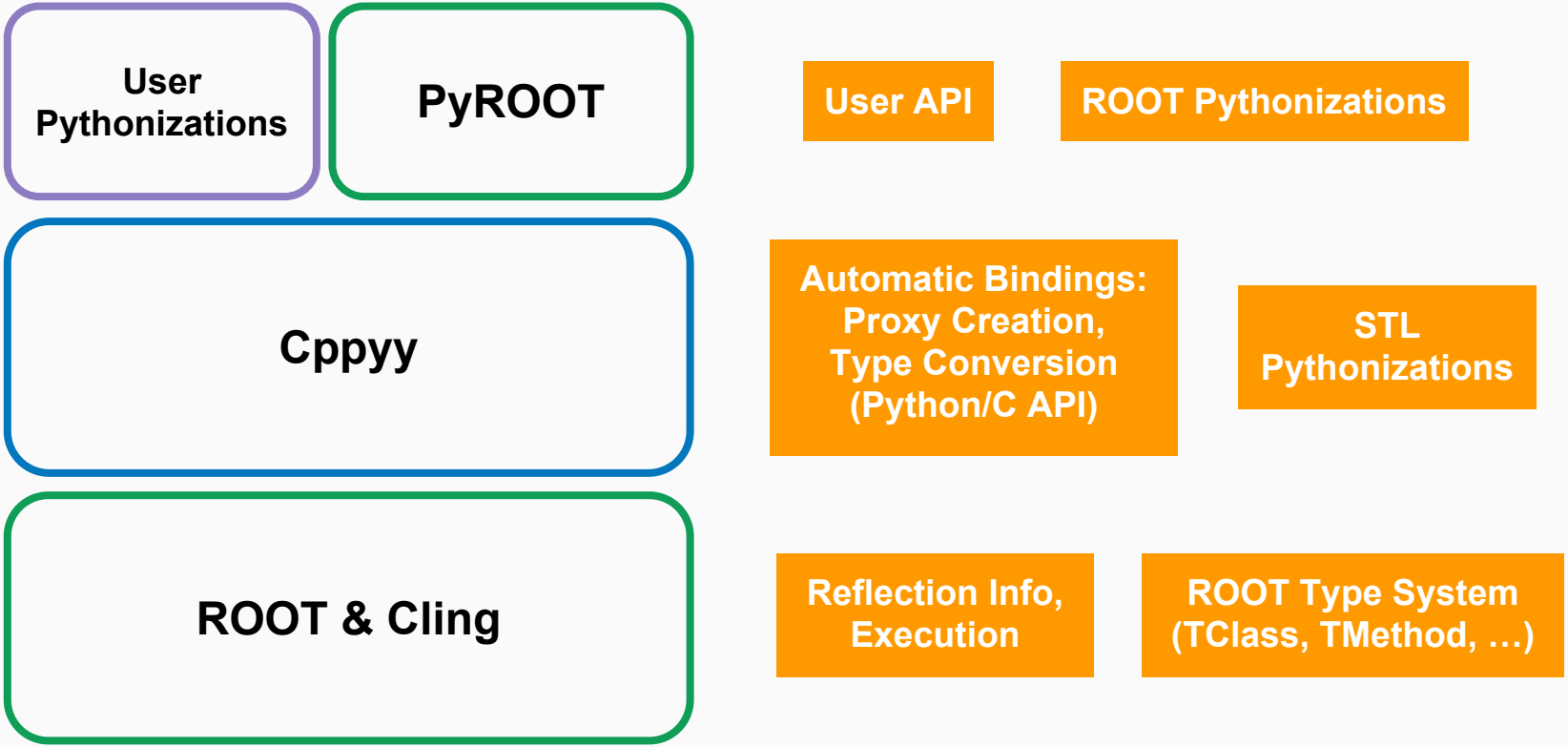
Data Analysis Framework

https://root.cern

The New PyROOT

- A new (experimental) PyROOT implementation is in the making
  - Already available in ROOT master (link)
  - **-Dpyroot_experimental=ON**
- Based on current Cppyy
  - Set of packages for automatic Python-C++ bindings generation
  - Forked from PyROOT by Wim Lavrijsen
- Goal: benefit from all the new features of Cppyy
- ROOT-specific Pythonizations added on top
  - A few available at the moment, many more will come: JIRA item

The New Structure



New PyROOT: Variadic Templates

- Support for variadic template arguments of functions

```
>>> import ROOT
>>> ROOT.gInterpreter.ProcessLine("""
template<typename... myTypes>
int f() { return sizeof...(myTypes); }
""")
>>> ROOT.f['int', 'double', 'void*']()
3
```

New PyROOT: Variadic Templates

- Example: RooFit. Want to move to variadic template arguments
  - Clean up documentation and function signatures
  - Make it work with any number of arguments
  - Arguments need not be same type

```
# Old signature in C++:
# RooFitResult * RooAbsPdf::fitTo(RooAbsData & data, const RooCmdArg & arg1 = RooCmdArg::none(),
# const RooCmdArg & arg2 = RooCmdArg::none(), const RooCmdArg & arg3 = RooCmdArg::none(), ...)
result = myPdf.fitTo(data, ROOT.RooFit.Save(), ROOT.RooFit.Minos(), … )
```

```
# New signature:
# template <typename ...CmdArg>
# RooFitResult * RooAbsPdf::fitTo(RooAbsData & data, CmdArg… args)
result = myPdf.fitTo(data, ROOT.RooFit.Save(), ROOT.RooFit.Minos(), … )
```

Works in new PyROOT!

New PyROOT: Lambdas

- Possible to use C++ lambdas from Python

```
>>> import ROOT
>>> ROOT.gInterpreter.ProcessLine(
"auto mylambda = [](int i) { return i*i; };")
>>> ROOT.mylambda
<cppyy.gbl.function<int(int)>* object at 0x35f9570>
>>> ROOT.mylambda(2)
4
```

Future Plans

ROOT
Data Analysis Framework
https://root.cern

Python2 & Python3

- PyROOT can work with either Python2 or Python3
- Not in our plans to discontinue support for Python2
  - At least in the next few years
  - However, end of life for Py2 is very close (2020)
- Building ROOT: we will remove the limitation of one Python version per build
  - If requested, PyROOT libraries will be generated for both Py2 and Py3
- Installation in Python2/3 directories
  - Don’t need to rely on $PYTHONPATH

More on Pythonizations

- User Pythonizations: allow ROOT users to define pythonizations for their own classes
  - Lazily executed

```
@pythonization('MyCppClass')
def my_pythonizor_function(klass):
    # Inject new behaviour in the class
    klass.__iter__ = my_iter_function
    ...
```

Python proxy of the class

Pythonic RooFit

- RooFit collections becoming faster & more STL-like (ACAT19)
- Can be leveraged to pythonize RooFit
  - PyROOT provides automatic bindings for STL-style iterators
  - Add a few additional pythonisations like dictionary syntax

```
# Loop over nuisance parameters without Pythonizations
it = modelConfig.GetNuisanceParameters().createIterator()
# Need to remember how RooFit-specific iterators work
nuis = it.Next()
while nuis:
    plot.DrawChainScatter(firstPOI, nuis)
    nuis = it.Next()
```

↓

```
# In the future:
for nuis in modelConfig['NuisanceParameters']:
    plot.DrawChainScatter(firstPOI, nuis)
```

Summary

ROOT
Data Analysis Framework
https://root.cern

Summary

- PyROOT's automatic Python bindings: unique!
- The ROOT team is aware of the importance of Python in HEP
  - Increasing effort in PyROOT
- Modernizing PyROOT
  - Modern C++ with Cppyy, new features
- Pythonizations are key for usability
  - Will be included in the Doxygen documentation

Backup Slides

ROOT
Data Analysis Framework
https://root.cern

C++ to Python Mapping

C++	Python
basic_types: short, int, long, 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

New PyROOT: Move Semantics

- Support for rvalue reference parameters

```
>>> import ROOT
>>> ROOT.gInterpreter.ProcessLine(
'void myfunction(std::vector<int>&& v) {
   for (auto i : v) std::cout << i << " ";
 }')
0L
>>> v = ROOT.std.vector['int'](range(10))
>>> ROOT.myfunction(ROOT.std.move(v))
0 1 2 3 4 5 6 7 8 9
>>> ROOT.myfunction(ROOT.std.vector['int'](range(10)))
0 1 2 3 4 5 6 7 8 9
```