PyROOT
Redesign and New Features

Enric Tejedor, Stefan Wunsch, Guilherme Amadio
for the ROOT team

ROOT Workshop 2018
Sarajevo, Bosnia and
Herzegovina

ROOT
Data Analysis Framework

https://root.cern

Introduction

ROOT

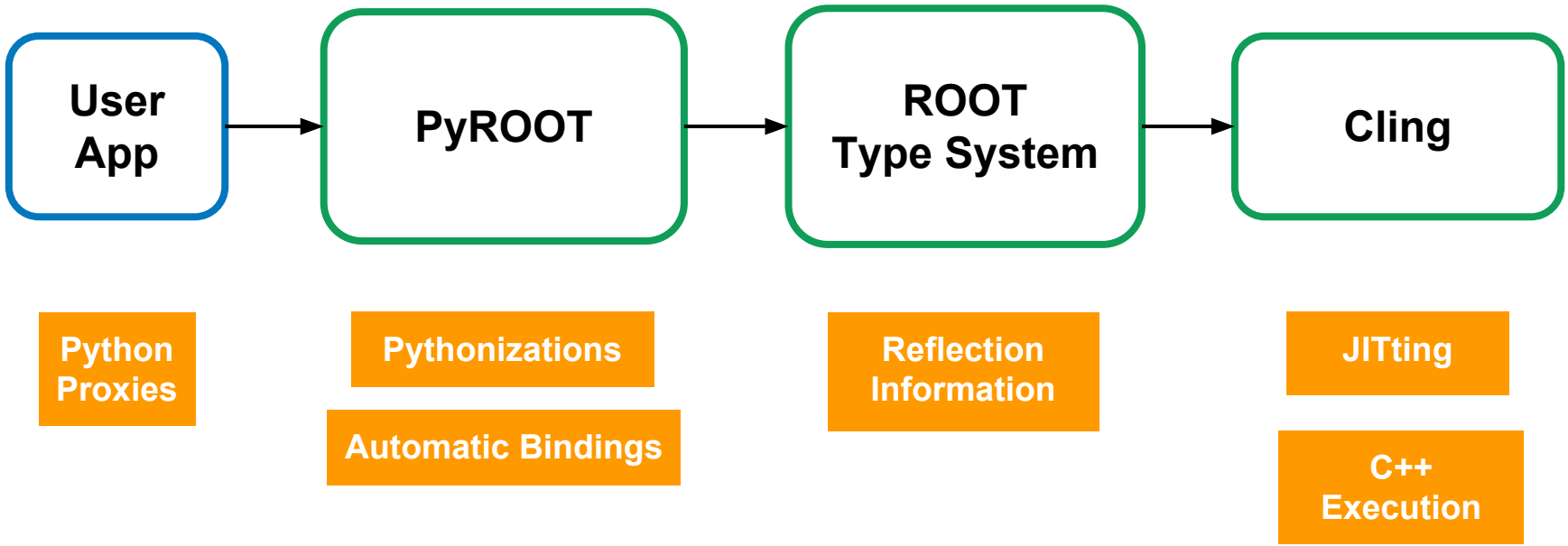
Data Analysis Framework

https://root.cern

PyROOT

- Python bindings offered by ROOT
- Access all the 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 (e.g. NumPy)

New Features in 6.14

ROOT

Data Analysis Framework

https://root.cern

New Features

- Zero-copy C++ to NumPy array conversion

New in 6.14

  - Objects with contiguous data (`std::vector`, `RVec`)
  - Pythonization: tell NumPy about data and shape

```
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!

New Features (II)

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

New in 6.14

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

# Convert to numpy array and apply numpy methods
myArray = myTree.AsMatrix()
m = np.mean(myArray, axis = 0)

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

Forthcoming Features

ROOT

Data Analysis Framework

https://root.cern

Forthcoming Features

- RDataFrame to NumPy
  - All RDataFrame operations available
  - Implicit parallelism

[Enrico's RDataFrame talk]

```
import ROOT
from ROOT import RDataFrame

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

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

np_arr = df.AsMatrix()
```

JITted C++ expression

Forthcoming Features (II)

- TMVA interoperability with NumPy
  - RTensor class on the C++ side

Stefan's TMVA talk

```
# Load some trained model
bdt = ROOT.TMVA.BDT('parameters.root')

# Create numpy array
inputs = numpy.array(some_data)

# Pass array to TMVA for prediction
predictions = bdt.Predict(inputs)
```

The New PyROOT

ROOT

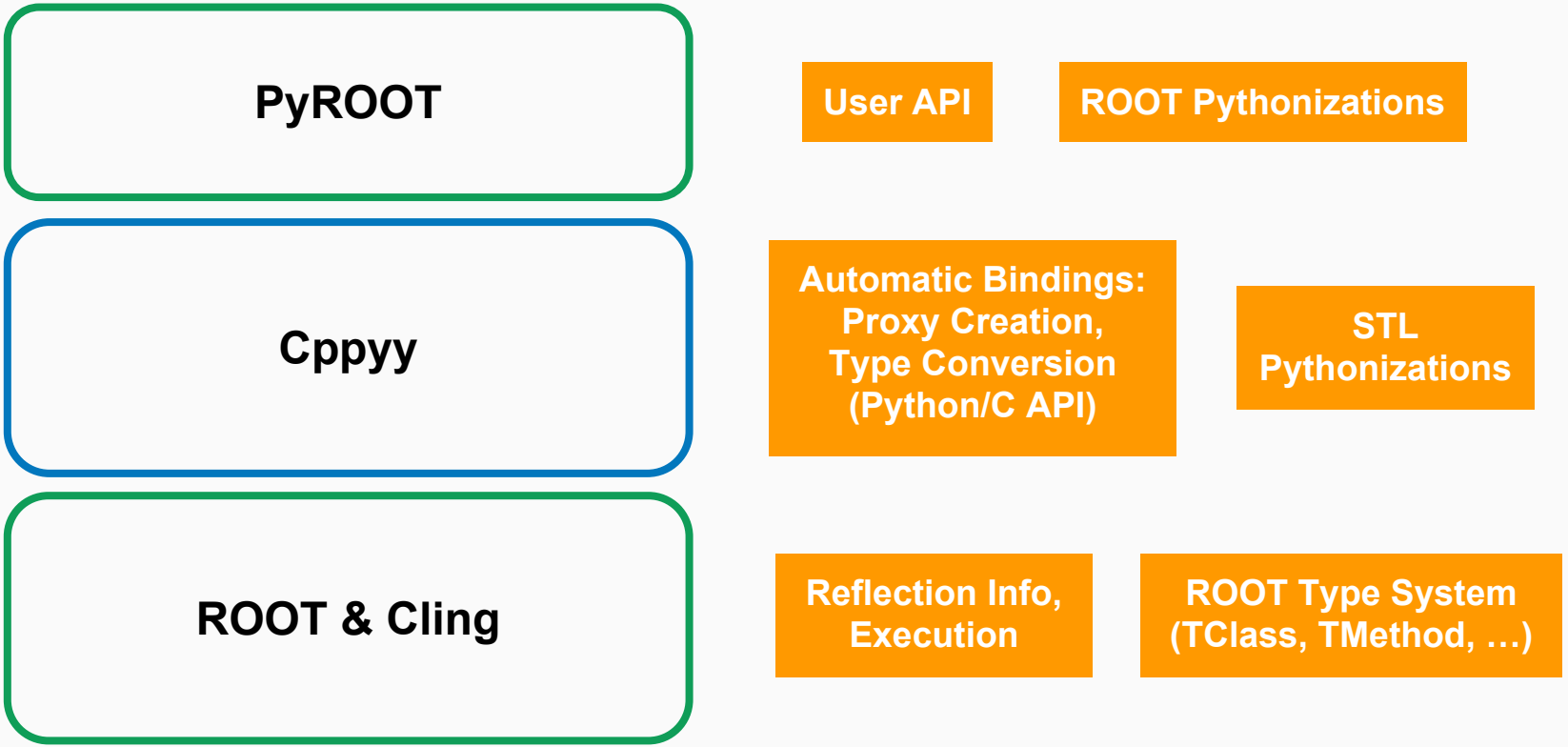
Data Analysis Framework

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, more will come

The New Structure



New PyROOT: Lambdas

- Possible to use C++ lambdas from Python

```
>>> import ROOT
>>> ROOT.gInterpreter.ProcessLine(
"auto mylambda = [](int i) { std::cout << i << std::endl; };")
140518947094560L
>>> ROOT.mylambda
<cppyy.gbl.function<void(int)>* object at 0x35f9570>
>>> ROOT.mylambda(2)
2
```

New PyROOT: Variadic Templates

- Support for variadic template arguments of functions

```
>>> import ROOT
>>> ROOT.gInterpreter.ProcessLine("""
template<typename... myTypes>
int f() { return sizeof...(myTypes); }
""")
0L
>>> ROOT.f['int', 'double', 'void*']()
3
```

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

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

Cppyy on Cling

- Both current PyROOT and Cppyy rely on ROOT meta classes (TClass, TMethod, …)
  - I.e. reflection data from ROOT
- Not needed: Cppyy could be rebased on top of Cling
  - Use cling and its clang binding directly
  - Access a more powerful API

Summary

ROOT

Data Analysis Framework

https://root.cern

Summary

- PyROOT's automatic Python bindings: unique!
- The ROOT team is aware of the growing importance of Python in HEP
  - Dedicating more effort to PyROOT
- Our goal is to modernize PyROOT
  - Modern C++ with Cppyy, new features
- Pythonizations are key for usability
  - Being tracked for PyROOT experimental: JIRA item

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
```

Forthcoming Features (III)

- Use Python callables in `RDataFrame`
  - For `Filter` and `Define` operations
  - Implementation with Numba to JIT Python code?

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

df.Filter('x > 0') # Already possible, jitted C++ expression

def my_cut(x):
 return x > 0

df.Filter(my_cut, ['x']) # Uses Python callable
```