A New PyROOT for ROOT 6.22

Enric Tejedor, Stefan Wunsch, Massimiliano Galli
for the ROOT team

PyHEP 2020

ROOT
Data Analysis Framework

https://root.cern

Outline

- Introduction: What is PyROOT?
  - Dynamic Python bindings to C++
- New PyROOT
  - Support for modern C++
  - Interoperability with Python data science libraries
  - Multi-Python builds
  - Docs
- Future Work
- Demo

Introduction

ROOT

Data Analysis Framework

https://root.cern

PyROOT

Python-C++ bindings

- Access to all C++ from Python (not just for ROOT!)
- Python façade, C++ performance

Automatic, dynamic

- No static wrapper generation (unlike e.g. SWIG, PyBind11)
- 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

A concrete example

Automatic bindings + Pythonizations

```
import ROOT

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

t = f.mytree
```

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

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

Pythonization: access the file content as an attribute

Dynamic access to C++

- JIT compile or load efficient C++, access it right away in Python

```
import ROOT
cpp_code = """
int f(int i) { return i*i; }

class A {
public:
  A() { cout << "Hello PyROOT!" << endl; }
};
"""
# Inject the code in the ROOT interpreter
ROOT.gInterpreter.ProcessLine(cpp_code)

# We find all the C++ entities in Python!
a = ROOT.A()   # this prints Hello PyROOT!
x = ROOT.f(3)  # x = 9
```

C++ code we want to invoke from Python

New PyROOT

ROOT

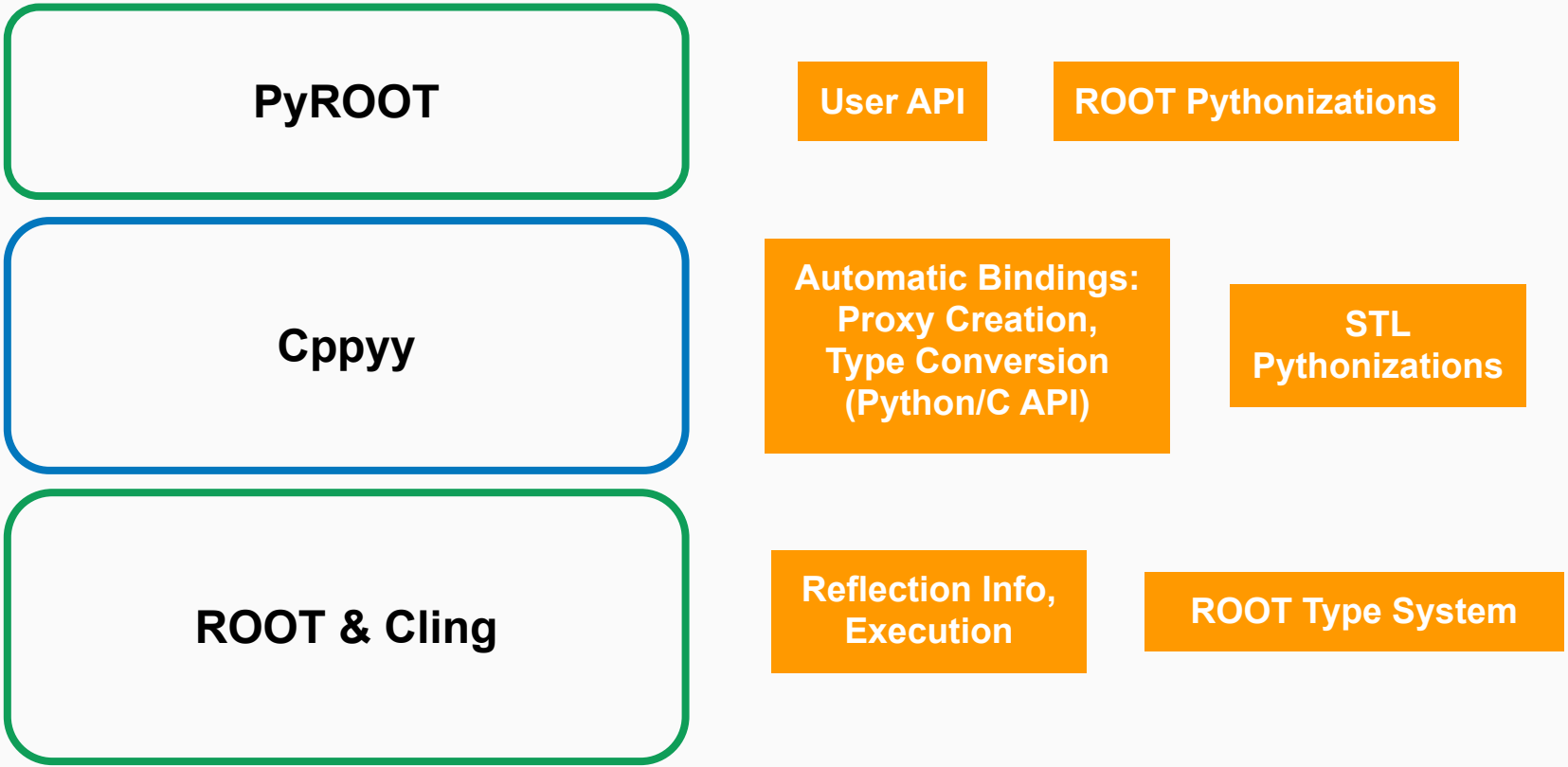
Data Analysis Framework

https://root.cern

New ROOT release, new PyROOT

- ROOT 6.22 has just been released with a new PyROOT
  - Previously in experimental mode
- Major revision of PyROOT
  - Support for modern C++
  - Interoperability features
  - More pythonizations
  - Python 2 & 3 in just one build

The New Structure



Improved C++ support

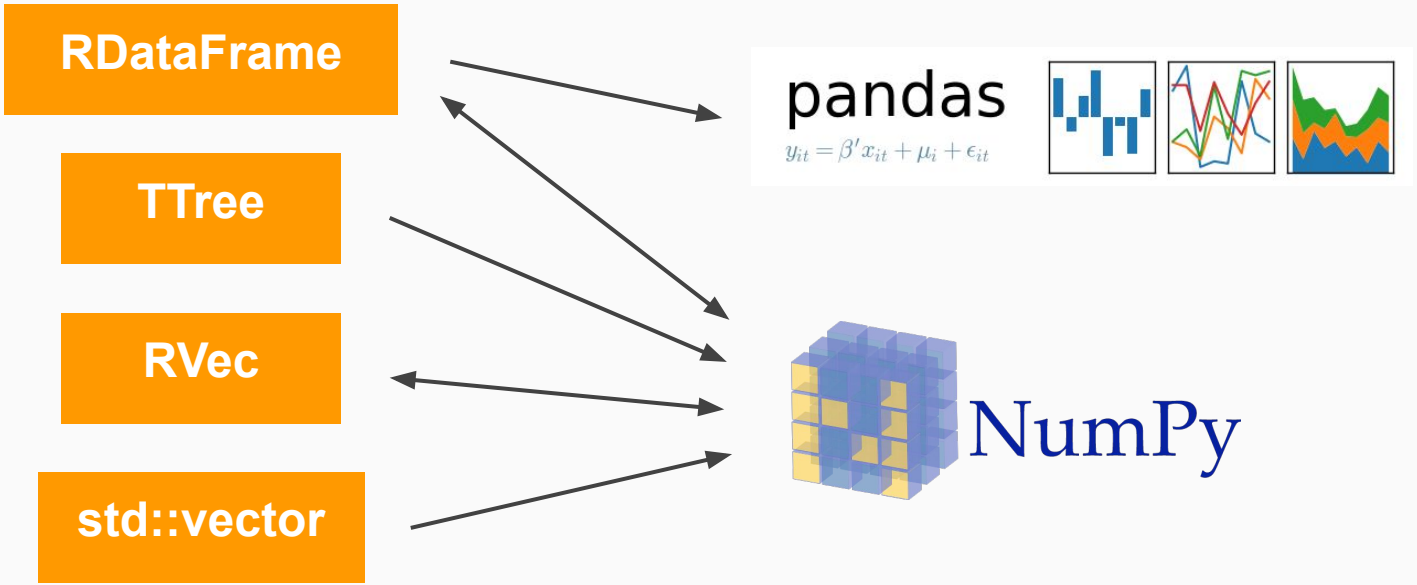
- Powered by cppyy
- Modern C++: variadic templates, move semantics, lambdas, ...

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

Interoperability with scientific Python

PyROOT

Scientific Python Libraries



RDataFrame to NumPy and pandas

- Efficient computations in C++ powered by `RDataFrame`, implicit parallelism
- Convert final dataset to NumPy (and pandas)

```
import ROOT

ROOT.EnableImplicitMT()

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

# Perform computations with RDataFrame
np_arrays = df.Filter('x > 0')
              .Define('z', 'x*y')
              .AsNumpy()

# Wrap data with pandas
import pandas
pdf = pandas.DataFrame(np_arrays)
```

Get dataset columns as NumPy arrays

C++ containers to NumPy

Zero-copy C++ to NumPy array conversion

- 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!

Using Python callables in C++

- Decorate Python callables to be used from C++, efficiently
  - Automatic wrapper is generated that calls a Numba-jitted version

```
import ROOT

@ROOT.Numba.Declare(['float', 'int'], 'float')
def pypow(x, y):
    return x**y

# Use the callable within an RDataFrame workflow
data = ROOT.RDataFrame('tree', 'file.root')
          .Define('x_pow3', 'Numba::pypow(x, 3)')
          .AsNumpy()
```

Multi-Python builds

- Single ROOT build for Python 2 & 3
  - Eases the transition from Python 2 to 3
  - Support Python 2 for as long as the experiments need it
- How it works
  - By default, it will build for both if it finds them in the system
  - Hints can be provided to point to specific Python installations
  - Building for a single Python version is still possible
  - More info here

Multi-Python builds: CMSSW

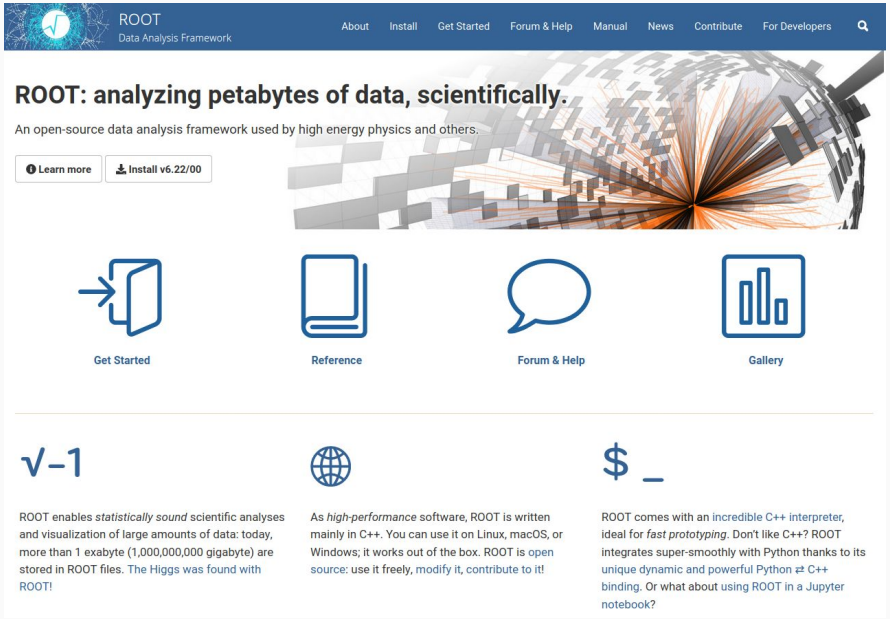
- CMSSW users are now able to use PyROOT from both Python 2 & 3

```
[etejedor@lxplus CMSSW_11_1_ROOT6_X_2020-04-27-2300]$ python2
Python 2.7.15+ (default, Mar 24 2020, 13:10:41)
>>> import ROOT
```

```
[etejedor@lxplus CMSSW_11_1_ROOT6_X_2020-04-27-2300]$ python3
Python 3.8.2 (default, Mar 24 2020, 13:29:26)
>>> import ROOT
```

Documentation

- A new ROOT website has been recently published!
- The website includes a PyROOT manual
  - User interface
  - Interactive graphics
  - Jupyter notebooks
  - … and more!
- Docs for pythonizations will be in the reference guide



Summary & Future Work

ROOT

Data Analysis Framework

https://root.cern

Summary

- ROOT 6.22 comes with new PyROOT
  - Modern, pythonic, interoperable
  - Support for Python 2 & 3 in the same installation
- Find here how to install it
- Future work
  - Installation in standard Python directories
  - API for user-defined pythonizations
  - Extend RDataFrame Python programming model

Demo

ROOT

Data Analysis Framework

https://root.cern

Demo

- Notebooks hosted on GitHub and interactively accessible via binder

- Alternative: Install ROOT locally in under 5 minutes
  - Powered by conda and thanks to Chris Burr, Enrico Guiraud and Henry Schreiner
  - Link to instructions

Backup Slides

ROOT

Data Analysis Framework

https://root.cern

C++ support: 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
```

RDataFrame to NumPy and pandas

```
# Run input pipeline with C++ performance that can process TBs of data
df = ROOT.RDataFrame('tree', 'file.root')
         .Filter('pT_j0 > 30')
         .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'])
print(col_dict)

{'r_j0': ndarray([0.26,1.,4.45]), 'eta_j0': ndarray(0.1,-1.,2.1), 'phi_j0': ndarray([-0.5,0.,0.2])}

# Wrap data with 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
```

TTree to NumPy

Read a TTree into a NumPy array

- Branches of arithmetic types

```
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, specify output type
xAsInts = myTree.AsMatrix(columns = ['x'], dtype = 'int')
```

User-defined 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.some_attr = ...
```

Python proxy of the class