

ROOT's new Python Interfaces based on CppInterOp: Performance, Correctness, and Future-Proof Design

Aaron Jomy¹, Vipul Cariappa^{1, 3}, Jonas Rembser¹, Vassil Vassilev²

¹ CERN EP-SFT

² Princeton University

³ Ramaiah University of Applied Sciences



May 28, 2026
CHEP, Bangkok



PRINCETON
UNIVERSITY



EP-SFT
Software Frameworks and Tools



Python and C++: Extensively used across the HENP software landscape

Python: Simple, interactive, easy prototyping

Good for rapid application development, data science friendly

C++: Industry standard for production-grade, performance-critical software.

Excels at numerically intensive computational kernels and scientific workflows

Python's interoperability with highly efficient production C++ libraries –

Crucial for long-term stability, usability and evolution of HENP analysis tools/workflows



ROOT: Analysis framework with the Cling C++ interpreter. Enables performant, exploratory programming with interactive C++

Provides users with automatic Python-C++ interoperability for rapid prototyping

Entrance to production-grade HEP C++ libraries from Python, like frameworks and steering code

```
import ROOT
```



```
using FourVector = ROOT::Math::XYZTVector;  
using FourVectorVec = std::vector<FourVector>;  
  
// A simple helper function to fill a test tree  
void fill_tree(const char *filename, const char  
*treeName)  
{  
    const double M = 0.13957; // set pi+ mass  
    TRandom3 R(1);  
  
    auto genTracks = [&]() {  
        FourVectorVec tracks;  
        const auto nPart = R.Poisson(15);  
        // ...  
    };
```



```
fileName = "df002_dataModel_py.root"  
treeName = "myTree"  
ROOT.fill_tree(fileName, treeName)  
d = ROOT.RDataFrame(treeName, fileName)
```





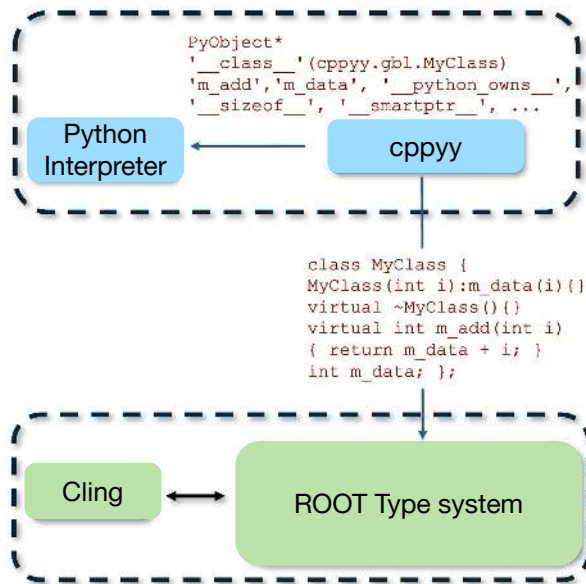
```
>>> import cppyy
>>> cppyy.cppdef("""
... class MyClass {
... public:
...     MyClass(int i) : m_data(i) {}
...     virtual ~MyClass() {}
...     virtual int add_int(int i) { return m_data + i; }
...     int m_data;
... };""")
True

>>> from cppyy.gbl import MyClass
>>> m = MyClass(42)

>>> class PyMyClass(MyClass):
...     def add_int(self, i): # python side override (CPython only)
...         return self.m_data + 2*i

>>> cppyy.cppdef("""int callback(MyClass* m, int i) {
...     return m->add_int(i); }""")
True

>>> cppyy.gbl.callback(m, 2)          # calls C++ add_int
85
>>> cppyy.gbl.callback(PyMyClass(1), 2) # calls Python-side override
5
```



The cppyy project is based on Cling:
<https://cppyy.readthedocs.io/en/latest/>



cppyy forks ROOT for interactive C++ (Cling) and type-system¹

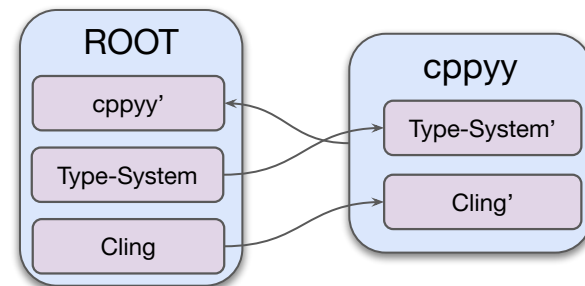
ROOT forks cppyy for automatic Python Interoperability²

Maintenance challenges:

- Both versions **diverged** while being **cross-dependent**
- Barriers in adopting latest technologies, stall in active feature development downstream

Compiler-level challenges:

- **ROOT Type system:** essential C++ reflection layer enabling ROOT I/O and other components, but **not optimised** for Python interoperability
- **Limited C++ features:** overload resolution, template instantiation, enum/lambda support, inadequate compiler-diagnostics, lack of newer C++ standard features
- **High memory footprint and suboptimal runtime performance**



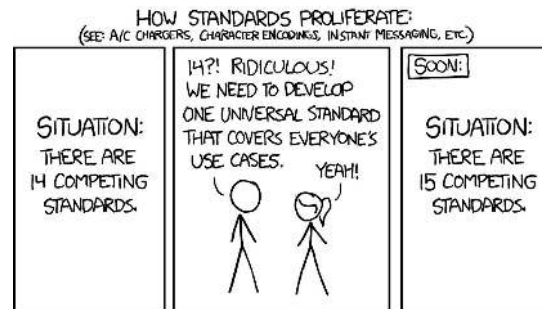
¹minus the rest of ROOT

²minus cppyy's copy of ROOT's type-system and Cling



How we broke a two-way fork: A third fork of [cppy](#)

But one that works universally, is sustainable,
more performant and conforms to C++ standards



In collaboration with [Compiler-Research](#) group at Princeton, cppy underwent significant R&D:

- Development of a **generalised** Clang-based interoperability library: **CppInterOp**
- **No string-maps:** opaque handles to representation of C++ entities on the Clang AST¹
- Conforms to LLVM/Clang for **correct type conversion and overload resolution rules**
- Fewer dependencies, more performant and crash-resilient
- Enables a **more complete feature set from the C++ standard**

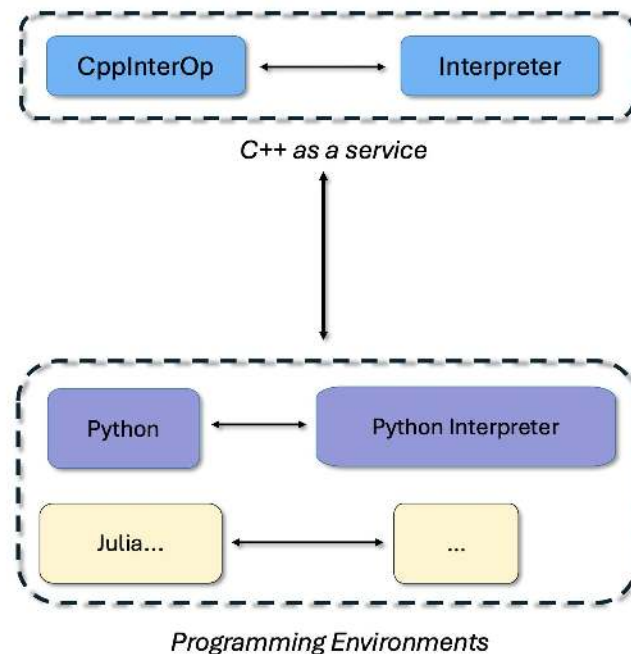
¹Clang AST (Abstract Syntax Tree): Compiler in-memory tree representation of C++ code



- Lightweight layer on top of LLVM/Clang
- Efficient, **on-demand reflection** and incremental **C++ JIT compilation** capabilities
- Embeds the **compiler as a service** in your framework in a **backwards-compatible** way
- Downstream tooling support: reflection API, interactive C++ for **language interoperability**
- Works with C/C++ libraries and LLVM from your favourite package manager

<https://github.com/compiler-research/CppInterOp>

See [talk at EuroLLVM 2026](#) for more details





Here we use CppInterOp's API and type-system to programmatically:

- Declare JIT-compiled templated struct S with a operator<
- Instantiate a type of S with int
- Construct an instance of S<int>
- Obtain a handle to the method of this object
- Invoke the method with an arbitrary parameter set

```
Cpp::Declare(R"(
    template<typename T>
    struct S {
        bool operator<(T &a) { return 0 < a; }
    };)");

Cpp::TemplateArgInfo TmplArgs[] = {Cpp::GetType("int")};
auto *InstanceTy = Cpp::InstantiateTemplate(Cpp::GetNamed("S"),
                                           TmplArgs, 1);

auto *obj = Cpp::Construct(InstanceTy);
std::vector<TCppFunction_t> ops;
Cpp::GetOperator(InstanceTy, Cpp::Operator::OP_Less, ops);
Cpp::JitCall FCI_Less = Cpp::MakeFunctionCallable(ops[0]);

int a = 5;
bool result = false;
void *args[1] = {(void *)&a};
FCI_Less.Invoke(&result, {args, /*arg_count = */ 1}, obj);
assert(result == true);
```

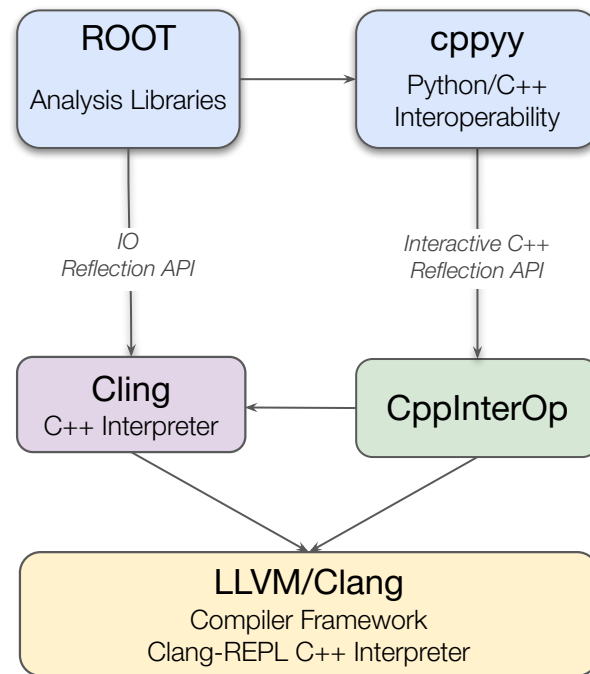


cppy now **works standalone**, no dependency on ROOT

CppInterOp is used in ROOT and supports Cling

CppInterOp also supports LLVM's **Clang-REPL** interpreter:
A path forward for ROOT to eventually use vanilla-LLVM

With no more domain-specific dependencies, this tool can
be leveraged by communities beyond high-energy physics



What this enables in ROOT:



Over 50 different use-cases each leveraging **unique** C++ language-features/idioms/usage-patterns/metaprogramming-techniques from Python:

Return smart pointer from callback

Derived classes access to protected members

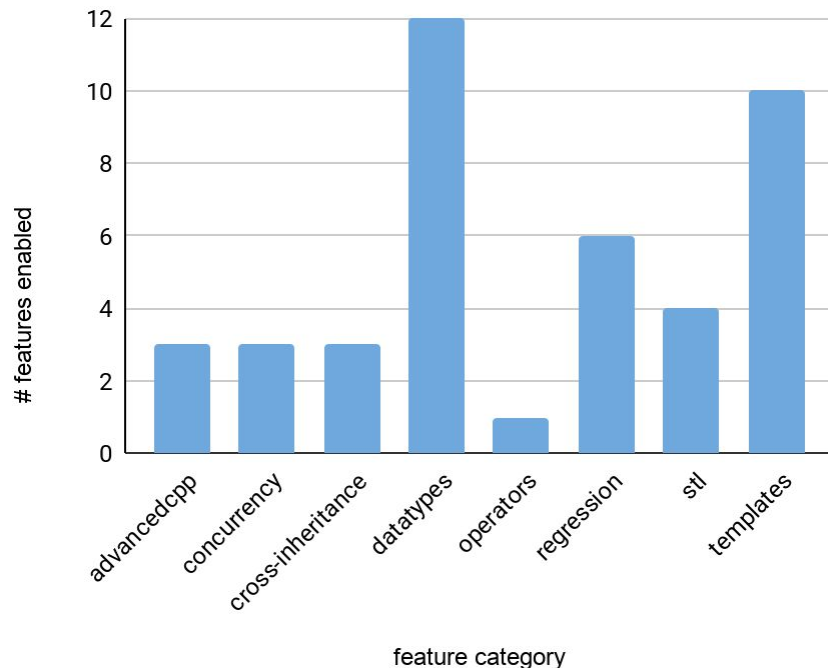
Inheriting from a non-copyable base class

Use of class enum with typedef'd type

Templated binary operators

Use of ctypes for pass-by-ref/ptr

Support for complex numpy arrays





Passing Python callbacks to higher order templated functions:

```
template <typename R, typename... U,  
          typename... A>  
R callme(R (*f)(U...), A &&...args) {  
    return f(args...);  
}
```



```
def callback(x: int, y: float) -> float:  
    return x + y  
  
ROOT.callme(callback, 123, 321.5) # 444.5
```



Fully automatic, no user-defined Python wrappers



Drive a C++ worker function from
Python with futures thread pool executor

With the new backend, cppyy's use of
the type system is now thread-safe

```
namespace Workers {  
    double calc(double d) { return d*42.; }  
}
```



```
import concurrent.futures as cf  
from cppyy.gbl.Workers import calc  
  
calc.__release_gil__ = True  
  
data = [3.1415, 2.7183, 1.4142, 1.3807, -9.2848]  
total = 0.  
  
pool = cf.ThreadPoolExecutor(max_workers=len(data))  
with pool as executor:  
    futures = [executor.submit(calc, d) for d in data]  
    for f in cf.as_completed(futures):  
        total += f.result()  
  
assert round(total+26.4642, 8) == 0.0
```





~20% of all open ROOT Python Issues :

[#19517](#) - Overloads like `foo(MyEnum, MyClass)` not correctly resolved

[#19358](#) - `cppyy.gbl.std.function` fails for template types like `ROOT::RVec<int>`

[#16852](#) - Regression in accessing protected data members in cross-inheritance

[#10680](#) - Namespace dropped in `std::function` cppyy binding

[#15062](#) - Non-trivial type traits do not work on cppyy

[#15085](#) - PyROOT/cppyy fails with template functions overload with SFINAE

[#15378](#) - Problem with templated binary operator implementations like `operator+`

[#9246](#) - Enums are not correctly resolved when using an alias within a class/struct

[#7077](#) - Incorrect/unstable overload resolution with enum class

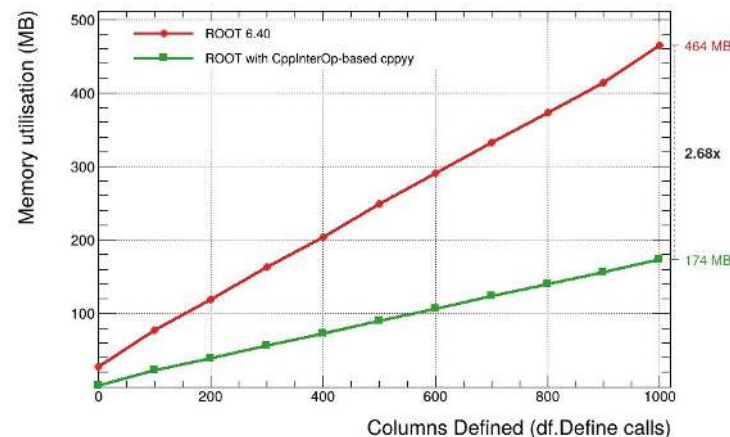
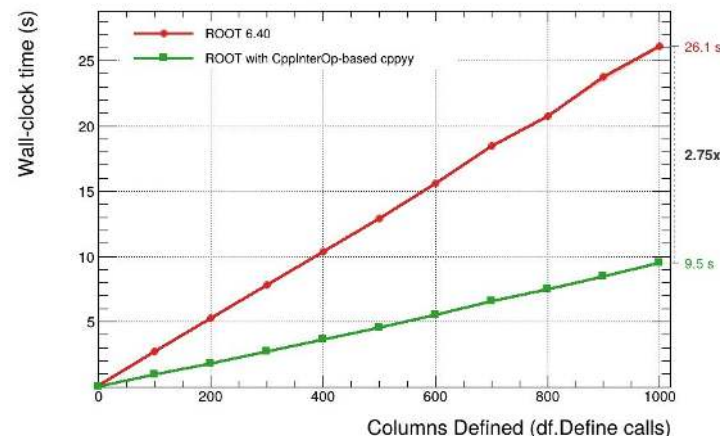


We use the current [cppyy migration branch](#) benchmarked against ROOT 6.40

Measures the call-overhead and memory footprint of template instantiation of a lightweight C++ dataframe class modelled after RDataFrame:

```
df.Define("c1", f1).Define("c2", f2)...
```

Clear performance benefits across the workflow
Gains from the leaner type system and more efficient use of Cling





CppInterOp is based on Clang and the LLVM JIT
Supports GPU and parallel programming API:

Experimental, Clang-REPL

```
Cpp::CreateInterpreter("--std=c++23", "--cuda")
```

```
#include <thrust/transform_reduce.h>
#include <thrust/execution_policy.h>
#include <thrust/iterator/counting_iterator.h>
#include <thrust/functional.h>
#include <thrust/complex.h>
```

```
double thrust_reduce(long N) {
    static_assert(THRUST_DEVICE_SYSTEM == THRUST_DEVICE_SYSTEM_CUDA);
    return thrust::transform_reduce(
        thrust::device,
        thrust::counting_iterator<long>(0),
        thrust::counting_iterator<long>(N),
        [] __device__ (long i) -> double {
            double u = double(i) * 0.05, mag = exp(-0.5 * 0.04 * u * u);
            return thrust::norm(thrust::complex<double>(mag * cos(u), -mag * sin(u)));
        },
        0.0, thrust::plus<double>{});
}
```



```
>>> N = 1 << 24
>>> cppyy.gbl.thrust_reduce(N)
89.12269254527581
```



<https://github.com/jank-lang/jank>

```
(ns jank.json
  (:include "fstream"
            "./json.hpp")
  (:refer-global :only [std ifstream nlohmann.json.parse]
                 :rename {std ifstream open-file
                          nlohmann.json.parse parse-json}))
(defn -main [& args]
  (let [file (open-file (cpp/cast std.string (first args)))
        json (parse-json file)]
    (println (.dump json 2))))
```

Dialect of Clojure built on LLVM, extends CppInterOp for C++ Interoperability (AOTCall, extra predicates and transformations)

First-class support for C++ includes, bringing C++ globals into a jank namespace and renaming them.



```
julia> import CppInterOp as Cpp

julia> using Test

julia> I = Cpp.create_interpreter()
CppInterOp.Interpreter{Ptr{CppInterOp.LibCppInterOp.CXI
nterpreterImpl}(0x000000001c2c0c60)}

julia> Cpp.process(I, "int x = 1 + 1;")
true

julia> Cpp.evaluate(I, "x")
2

julia> Cpp.undo(I, 2)
true

julia> Cpp.process(I, "float x = 42.0;")
true

julia> Cpp.evaluate(I, "x")
42.0f0
```

Thanks to Yupei Qi (@Gnimuc) for this prototype!

> pkg add <https://github.com/Gnimuc/CppInterOp.jl>



New Python-C++ Interoperability system is planned to ship with **ROOT 6.42 release**

Roadmap:

- Infrastructure hardening, tackling platform specific issues, resolving ~35 edge-cases
- Migrating to a standalone monorepo that is an external dependency for ROOT:
[CppJIT](#) - Next generation of cppy, alpha releases on PyPI in summer 2026
- Ensuring a smooth migration for existing ROOT Python workflows and user code:
 - Adapting to improvements in behaviour/semantics
 - Deprecation of non-standard features

Future opportunities:

- Leveraging more upstream LLVM features in ROOT via CppInterOp
- Return to active feature-development, community-building around the CppJIT project



Thank You!