A numba extension for PyROOT

140th PPP meeting

What is numba?

Numba is a compiler for Python array and numerical functions that can speed up applications with high performance functions written directly in Python.

```
# Function is translated into Python byte code
def go_slow(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace += np.tanh(a[i, i])
    return a + trace
```

```
# Function is compiled and runs in machine code
@jit(nopython=True)
def go_fast(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace += np.tanh(a[i, i])
    return a + trace
```

0.2 ms	**1st Run**	176 ms
0.2 ms	**Subsequent Runs**	0.046 ms

How does Numba compile Python code?

Numba translates the code into LLVM IR in multiple passes:

Pass 1: Analyze bytecode

- Generates CFG
- Tracks the lifetime of variables

Pass 2: Generate the Numba IR

- Converts code for stack machine to a register machine

Pass 3: Rewrite untyped IR

- Transformed to allow Python features such as raising exceptions

Pass 4: Infer types

- Runs numba.typeinfer
- Assigns a type to every intermediate var

Pass 5: Rewrite typed IR

- Optimizes the IR

Pass 6: Generate nopython LLVM IR

- Converts each Numba IR node to LLVM IR

Pass 7: LLVM IR to machine code

- Compiled by LLVM JIT compiler
- Python wrapper (Dispatcher) created

Benefits and Limitations of using Numba

Benefits

- It can make Python programs run as fast as C.
- It's easy to use. For most use cases you only need to add the JIT decorator and no additional information is required.
- Programs can be debugged either by using debuggers like gdb or in python by simply commenting out the decorator.
- It is well maintained.
- Good support for numpy.

Limitations

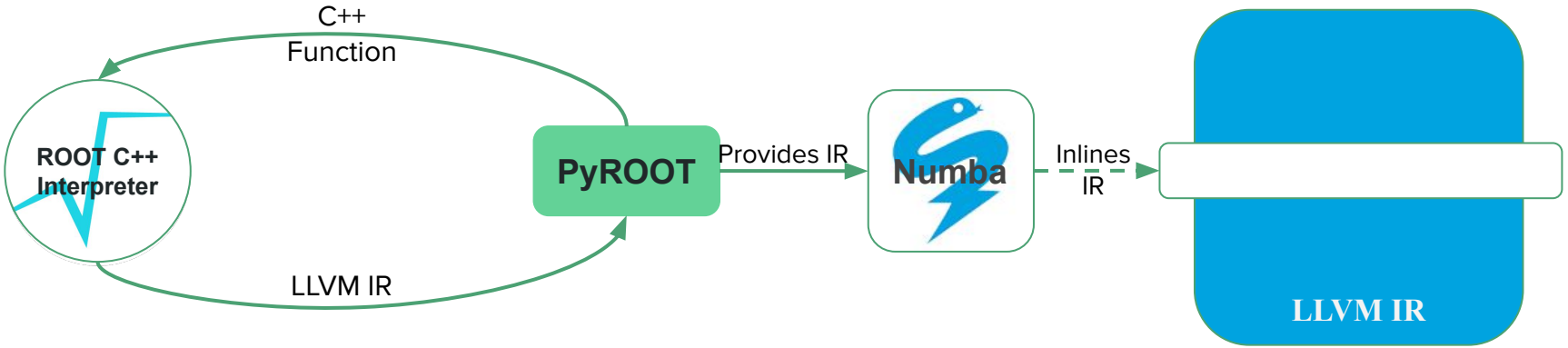
- A lot of libraries are not officially supported by numba - pandas, scipy etc.
- To support new Python classes you have to provide typing information which can be exhausting in a large codebase.
- Not applicable for all types of Python code - some cannot compile others don't provide speedup

PyROOT with Numba

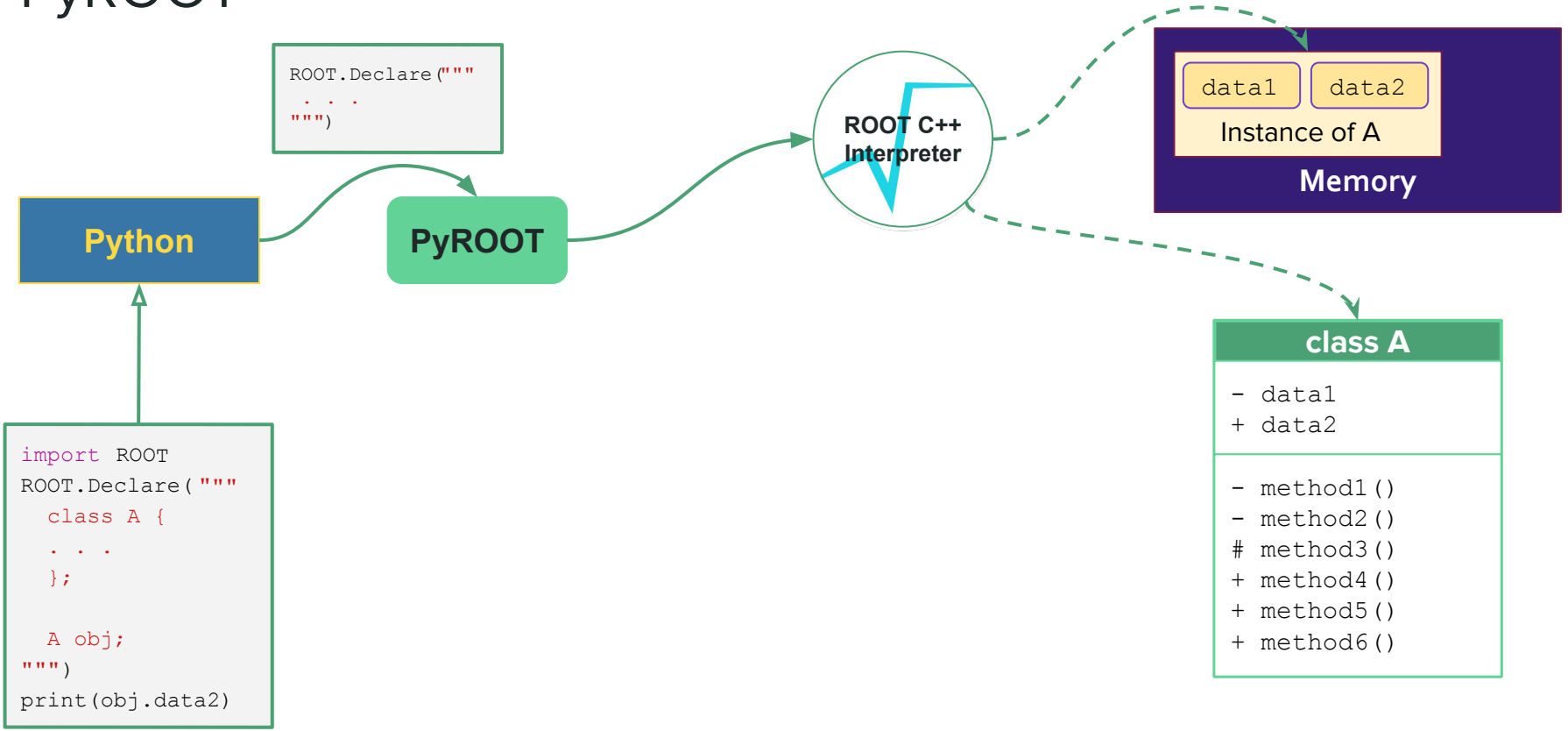
Motivation

PyROOT has made it possible for Python functions to use C++. Integration of C++ and Numba will give the same benefits as with PyROOT.

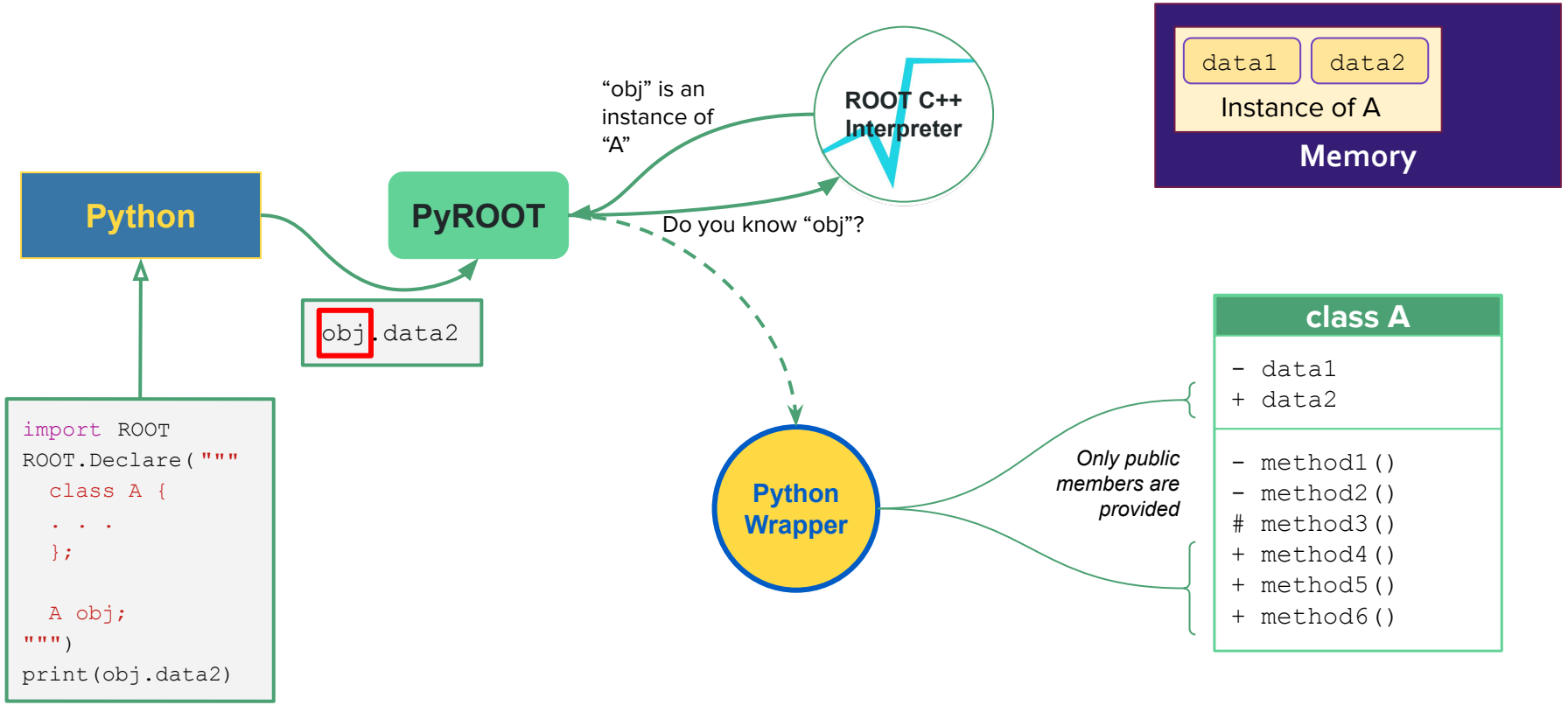
Since C++ can be compiled to LLVM IR our initial idea was something like this:



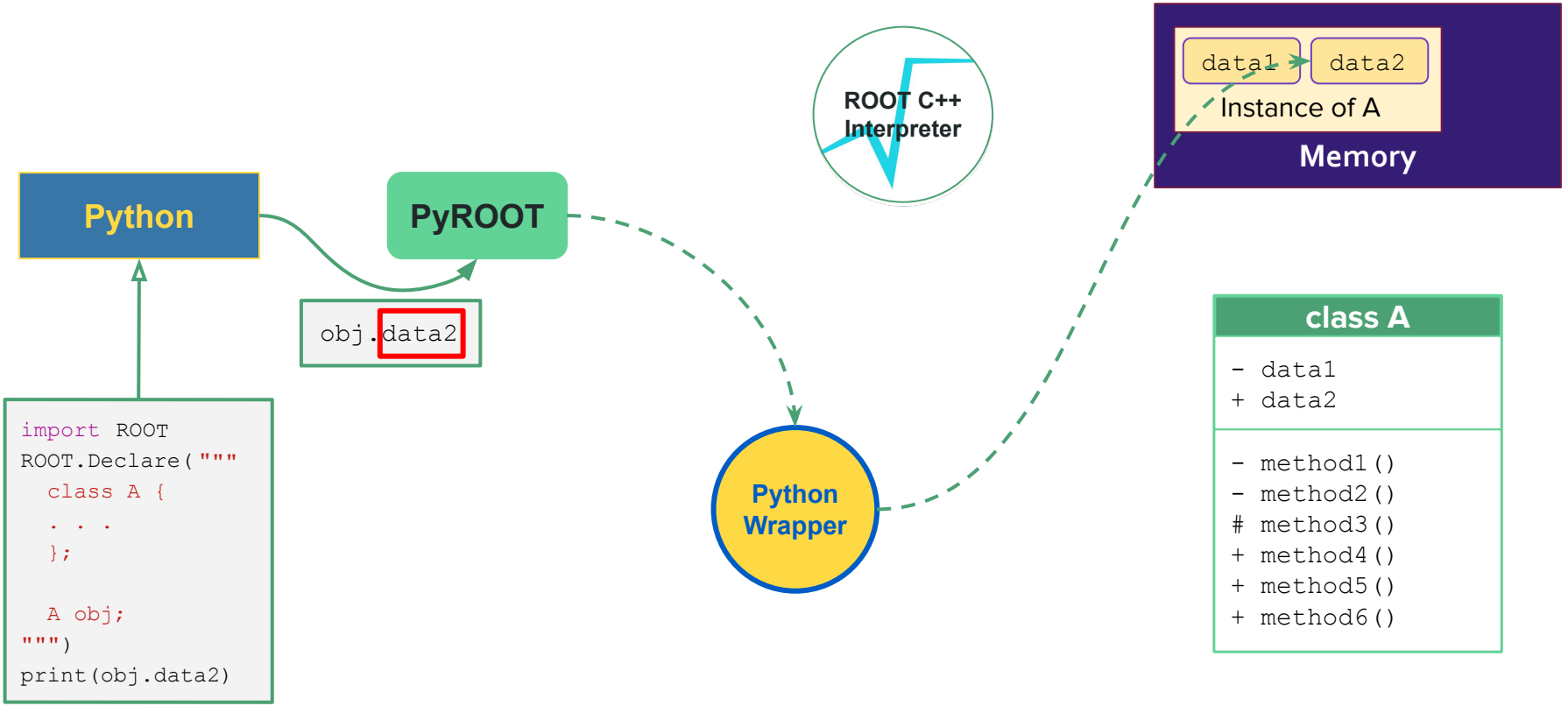
PyROOT


ROOT.Declare("""
. . .
""")
Python
PyROOT
ROOT C++
Interpreter
data1
data2
Instance of A
Memory
class A
- data1
+ data2
- method1()
- method2()
method3()
+ method4()
+ method5()
+ method6()
import ROOT
ROOT.Declare("""
class A {
. . .
};
A obj;
""")
print(obj.data2)

PyROOT

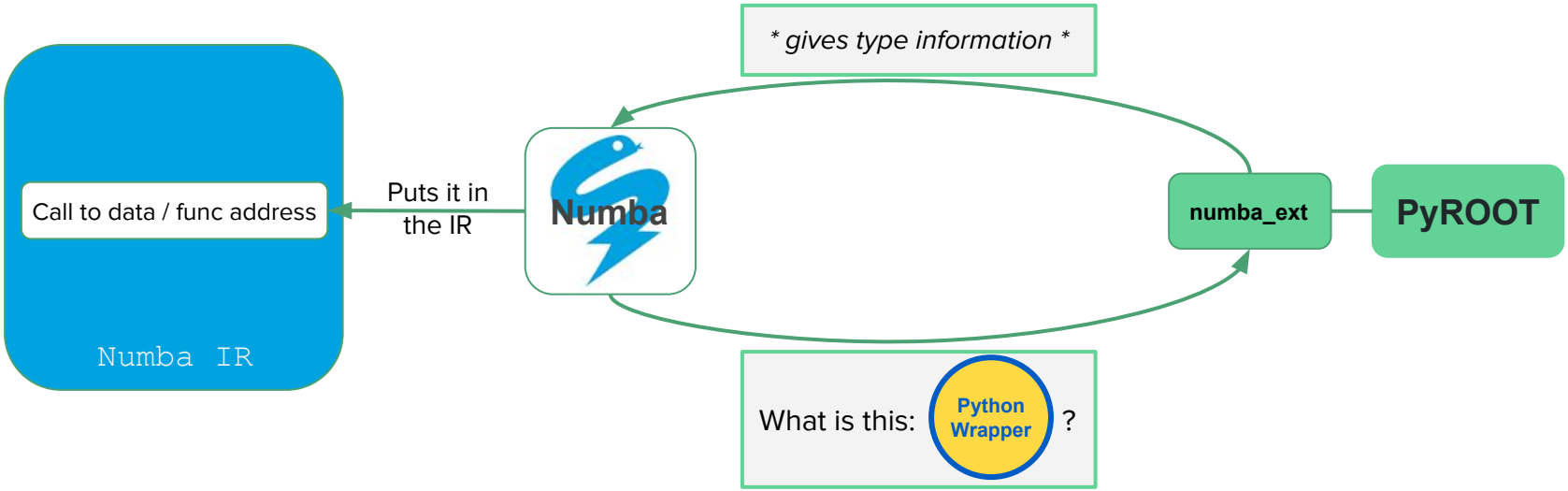

Python
PyROOT
obj.data2
"obj" is an instance of "A"
ROOT C++ Interpreter
Do you know "obj"?
data1
data2
Instance of A
Memory
import ROOT
ROOT.Declare("""
class A {
. . .
};
A obj;
""")
print(obj.data2)
Python Wrapper
Only public members are provided
class A
- data1
+ data2
- method1()
- method2()
method3()
+ method4()
+ method5()
+ method6()

PyROOT


Python
PyROOT
obj.data2
ROOT C++
Interpreter
data1
data2
Instance of A
Memory
Python
Wrapper
class A
- data1
+ data2
- method1()
- method2()
method3()
+ method4()
+ method5()
+ method6()
import ROOT
ROOT.Declare("""
class A {
. . .
};
A obj;
""")
print(obj.data2)

Current Design

Numba has more support for providing typing information than mentioned in its user manual. Since PyROOT wrappers are Python objects we just have to extend them to provide numba with the necessary typing information.



Example - Wrapper classes for C++ functions

The information that needs to be supplied to numba for it to support PyROOT functions is shown here:

Even though the logic is tiny it's not easy to add generic support for all the wrapper classes but this proves that it's possible.

```
class CppFunctionNumbaType(nb_types.Callable):
    def __init__(self, func, is_method=False):
        ### Store the cppyy function and create a
        ### dictionary of arguments and the func
        ### signature that can take in the arguments

    def get_call_type(self, context, args, kwds):
        ### Return the function signature if the
        ### args were supplied before or find the
        ### overload of the function suitable for the args

        @nb_iutils.lower_builtin(ol, *args)
        def lower_external_call(context, builder,
            sig, args, ty=..., pyval=self._func):
            ### Tell numba how to lower the function call

        ### Return the signature of the appropriate overload
        return ol.sig

    def get_pointer(self, func):
        ### Get the function address using ROOT
        return address

@nb_ext.typeof_impl.register(cpp_types.Function)
def typeof_function(val, c):
    return CppFunctionNumbaType(val)
```

Benefits of this extension

1. Access to C++ codebase (ROOT)
2. Ease of use - just plug in PyROOT and numba will run it without any hassles
3. Can support full C++ feature set in the future
4. Pythonizations can be easily supported

Current support

- C++ free (global) functions can be called and overloads will be selected, or a template will be instantiated, based on the provided types, assuming all types match explicitly (thus e.g. typedefs, implicit conversions, and default arguments are not yet supported).
- Instances of C++ classes can be passed into Numba traces. They can be returned from functions called within the trace, but can not yet be returned from the trace. Their public data is accessible but cannot be modified and their public methods can be called, but the same rules as free functions apply to them.

Demo

Thankyou