Lessons learned in Python-C++ integration

Jim Pivarski

Princeton University – IRIS-HEP

December 1, 2021

This talk is about lessons learned in Python-C++ integration.

Awkward Array is the case study.

Awkward
Array

What is Awkward Array?

Efficient representation of variable length, nested, JSON-like data with NumPy-like functions to quickly compute and restructure data in Python.

What is Awkward Array?

Efficient representation of variable length, nested, JSON-like data with NumPy-like functions to quickly compute and restructure data in Python.

```
array = ak.Array([
    [{"x": 1.1, "y": [1]}, {"x": 2.2, "y": [1, 2]}, {"x": 3.3, "y": [1, 2, 3]}],
    [],
    [{"x": 4.4, "y": [1, 2, 3, 4]}, {"x": 5.5, "y": [1, 2, 3, 4, 5]}]
])
```

What is Awkward Array?

Efficient representation of variable length, nested, JSON-like data with NumPy-like functions to quickly compute and restructure data in Python.

```
array = ak.Array([
    [{"x": 1.1, "y": [1]}, {"x": 2.2, "y": [1, 2]}, {"x": 3.3, "y": [1, 2, 3]}],
    [],
    [{"x": 4.4, "y": [1, 2, 3, 4]}, {"x": 5.5, "y": [1, 2, 3, 4, 5]}]
])
```

NumPy-like expression

```
output = np.square(array["y", ..., 1:])

output.to_list()
```

```
[
    [[], [4], [4, 9]],
    [],
    [[4, 9, 16], [4, 9, 16, 25]]
]
```

What is Awkward Array?

Efficient representation of variable length, nested, JSON-like data with NumPy-like functions to quickly compute and restructure data in Python.

```
array = ak.Array([
    [{"x": 1.1, "y": [1]}, {"x": 2.2, "y": [1, 2]}, {"x": 3.3, "y": [1, 2, 3]}],
    [],
    [{"x": 4.4, "y": [1, 2, 3, 4]}, {"x": 5.5, "y": [1, 2, 3, 4, 5]}]
])
```

NumPy-like expression

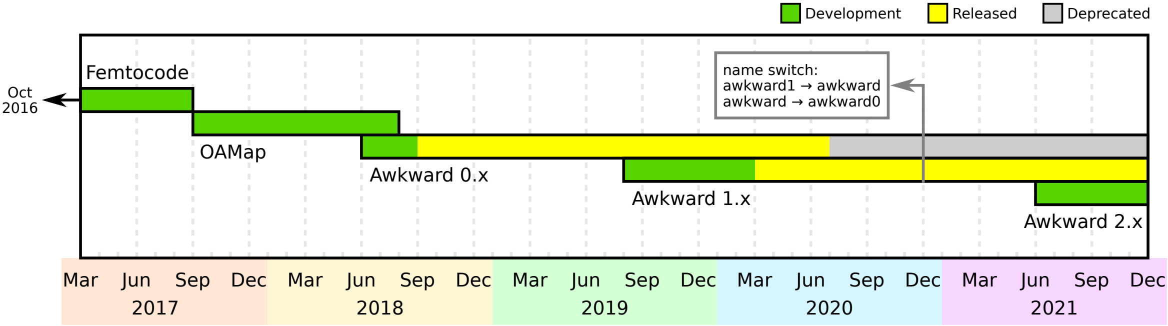
```
output = np.square(array["y", ..., 1:])

output.to_list()
```

```
[
    [[], [4], [4, 9]],
    [],
    [[4, 9, 16], [4, 9, 16, 25]]
]
```

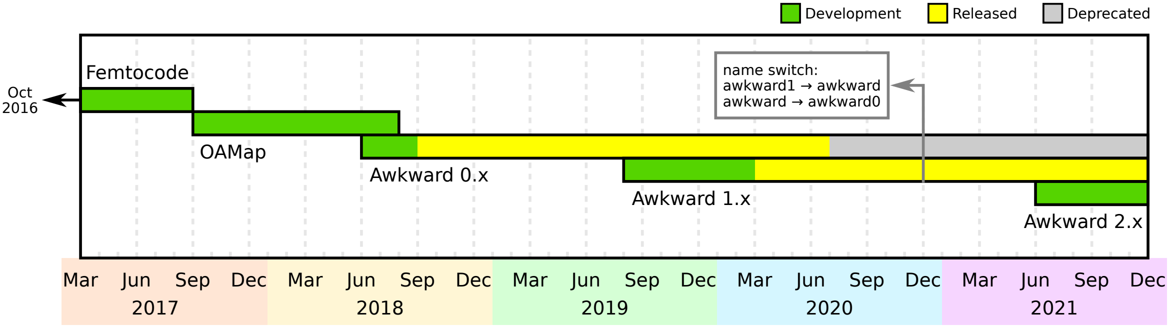
equivalent Python

```
output = []
for sublist in python_objects:
    tmp1 = []
    for record in sublist:
        tmp2 = []
        for number in record["y"][1:]:
            tmp2.append(np.square(number))
        tmp1.append(tmp2)
    output.append(tmp1)
```

History of the project



History of the project



- Prehistory: Femtocode (21 kLOC Python) and OAMap (8 kLOC Python)
- Awkward 0.x: first users, first API, pure Python + NumPy (9 kLOC Python)
- Awkward 1.x: rewrite for stability & API (22 kLOC Python, 70 kLOC C++, 14 kLOC C)
- Awkward 2.x: refactor for maintainability (29 kLOC Python, 10 kLOC C++, 9 kLOC C)

Language changes

Awkward 0.x → 1.x: complete rewrite with the intention of changing the API.

Also had to replace some NumPy function calls with precompiled C functions, so we built the infrastructure in C++ with Python only for the high-level front-end.

Language changes

Awkward 0.x → 1.x: complete rewrite with the intention of changing the API.

Also had to replace some NumPy function calls with precompiled C functions, so we built the infrastructure in C++ with Python only for the high-level front-end.

Awkward 1.x → 2.x: we are now porting the C++ infrastructure back to Python.

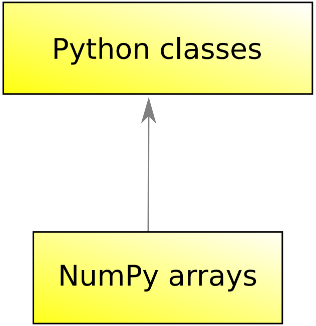
(For reasons described later in this talk.)

Python	−19.6 kLOC	+26.9 kLOC
C++	−59.8 kLOC	+0 kLOC
C	−4.9 kLOC	+0 kLOC

kLOC = thousands of non-blank, non-comment lines of code counted by `cloc`

Awkward 2.x is 75% done; the above is a projection

Evolution of architecture: Awkward 0.x



tree structures,
user interface

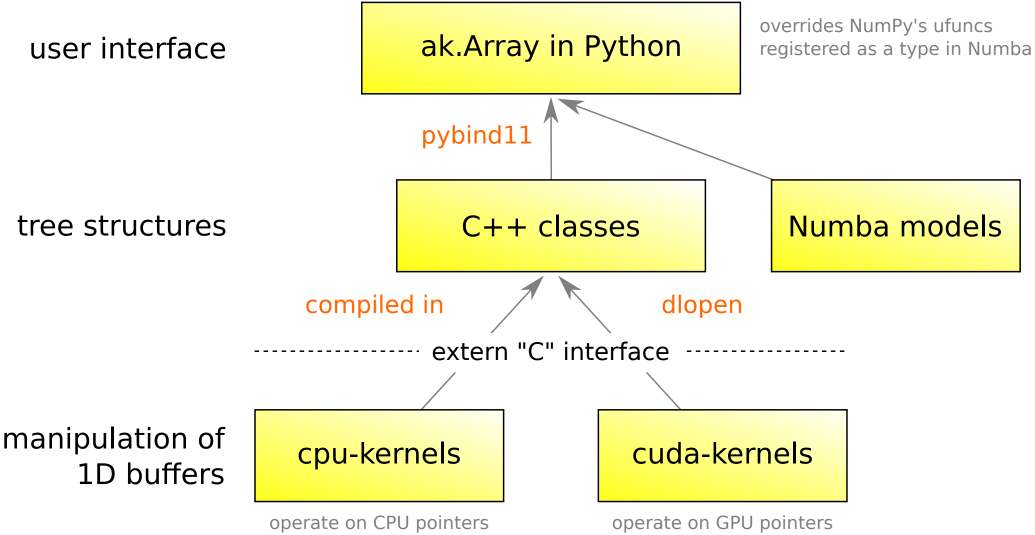
Python classes

manipulation of
1D buffers

NumPy arrays

operate on CPU pointers

Evolution of architecture: Awkward 1.x



user interface

ak.Array in Python

overrides NumPy's ufuncs
registered as a type in Numba

pybind11

tree structures

C++ classes

Numba models

compiled in

dlopen

extern "C" interface

manipulation of
1D buffers

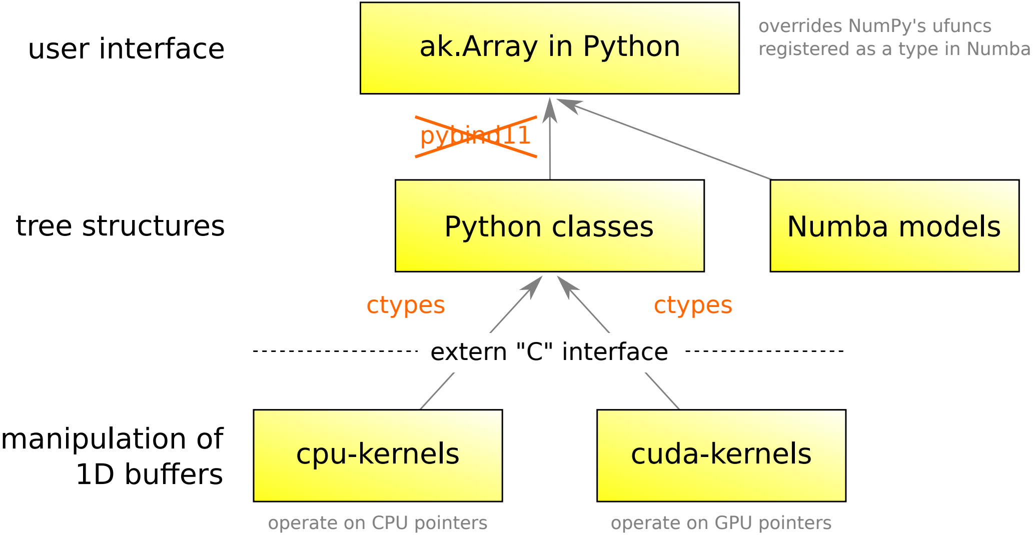
cpu-kernels

cuda-kernels

operate on CPU pointers

operate on GPU pointers

Evolution of architecture: Awkward 2.x



This language choice is unrelated to performance

All $\mathcal{O}(n)$ operations for arrays of length n are performed in the kernels layer (C).

Porting the tree structures from C++ to Python has no $\mathcal{O}(n)$ impact.

This language choice is unrelated to performance

All $\mathcal{O}(n)$ operations for arrays of length n are performed in the kernels layer (C).

Porting the tree structures from C++ to Python has no $\mathcal{O}(n)$ impact.

For an array of 30 million lists of records:

NumPy-like expression

```python
output = np.square(array["y", ..., 1:])
```

Awkward 0.15.5: 4.6 seconds
Awkward 1.5.1: 2.4 seconds
Awkward 2.0.0a: 1.5 seconds

equivalent Python

```python
output = []
for sublist in python_objects:
    tmp1 = []
    for record in sublist:
        tmp2 = []
        for number in record["y"][1:]:
            tmp2.append(np.square(number))
        tmp1.append(tmp2)
    output.append(tmp1)
```

equivalent Python: 140 seconds
because $\mathcal{O}(n)$ operations are performed in Python

Lessons learned in Python-C++ integration

Issue #1: Intention to use C++ as an ABI

Awkward 1.x design goal: connect to C++ HEP libraries through the C++ layer.

Awkward 2.x removes this capability.

Issue #1: Intention to use C++ as an ABI

Awkward 1.x design goal: connect to C++ HEP libraries through the C++ layer.

Awkward 2.x removes this capability.

- Python packages must be distributed as binaries, with compilation-on-install *as a last resort!* "`pip install awkward`" shouldn't have dependencies outside of `pip`, including the existence of a compiler.

Issue #1: Intention to use C++ as an ABI

Awkward 1.x design goal: connect to C++ HEP libraries through the C++ layer.

Awkward 2.x removes this capability.

- Python packages must be distributed as binaries, with compilation-on-install *as a last resort!* "`pip install awkward`" shouldn't have dependencies outside of `pip`, including the existence of a compiler.
- Dynamically linking to precompiled C++ exposes a project to ABI dependencies. Awkward-enabled HEP libraries would depend on ABI version.

  (See `PYBIND11_INTERNALS_VERSION` in `pybind11/include/pybind11/detail/internals.h`.)

Issue #1: Intention to use C++ as an ABI

Awkward 1.x design goal: connect to C++ HEP libraries through the C++ layer.

Awkward 2.x removes this capability.

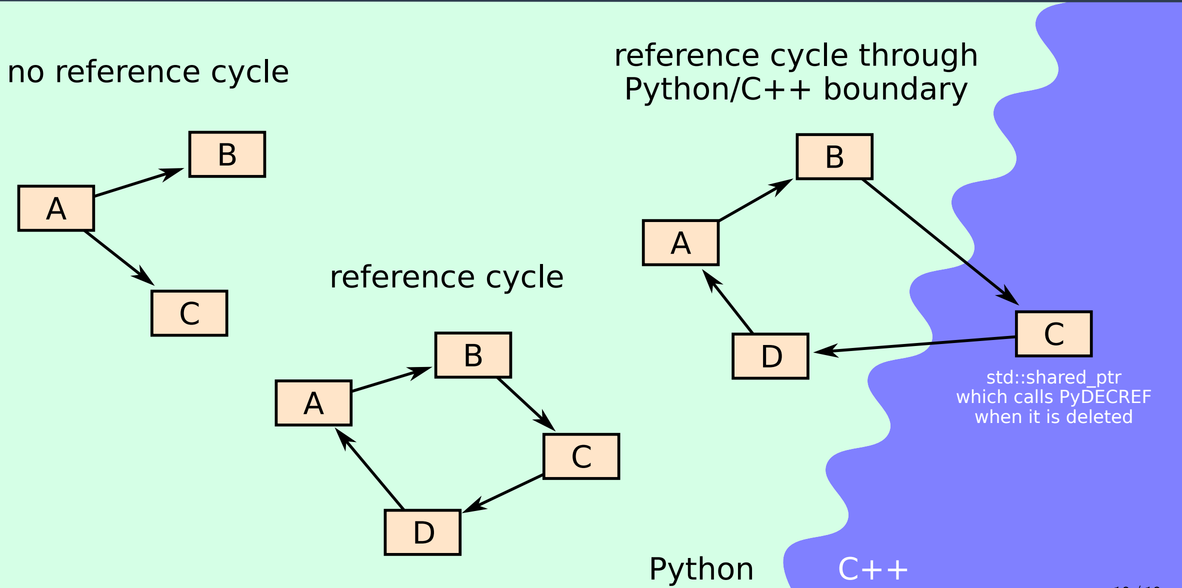
- Python packages must be distributed as binaries, with compilation-on-install *as a last resort!* "`pip install awkward`" shouldn't have dependencies outside of `pip`, including the existence of a compiler.
- Dynamically linking to precompiled C++ exposes a project to ABI dependencies. Awkward-enabled HEP libraries would depend on ABI version.

  (See `PYBIND11_INTERNALS_VERSION` in `pybind11/include/pybind11/detail/internals.h`.)
- **Not needed anyway:** fastjet, a Python/Awkward interface to FastJet (C++), interfaces through C data types (raw arrays) easily.

  Awkward Array does not need to be C++ to interface with C++.

See Aryan Roy's poster!

Issue #2: Reference cycles through C++



no reference cycle

reference cycle

reference cycle through Python/C++ boundary

std::shared_ptr
which calls PyDECREF
when it is deleted

Issue #2: Reference cycles through C++

- In most circumstances, Awkward Arrays have no reference cycles because they're immutable.

Issue #2: Reference cycles through C++

- In most circumstances, Awkward Arrays have no reference cycles because they're immutable.
- VirtualArrays (for lazy evaluation) maintain a cache to avoid repeat evaluations. Sometimes, a VirtualArray is put inside its own cache.

Issue #2: Reference cycles through C++

- In most circumstances, Awkward Arrays have no reference cycles because they're immutable.
- VirtualArrays (for lazy evaluation) maintain a cache to avoid repeat evaluations. Sometimes, a VirtualArray is put inside its own cache.
- Python garbage collector can detect reference cycles, immediately when a reference count goes to zero or during mark-and-sweep.

Issue #2: Reference cycles through C++

- In most circumstances, Awkward Arrays have no reference cycles because they're immutable.
- VirtualArrays (for lazy evaluation) maintain a cache to avoid repeat evaluations. Sometimes, a VirtualArray is put inside its own cache.
- Python garbage collector can detect reference cycles, immediately when a reference count goes to zero or during mark-and-sweep.
- But it can't detect a reference cycle through C++.

Issue #2: Reference cycles through C++

- In most circumstances, Awkward Arrays have no reference cycles because they're immutable.
- VirtualArrays (for lazy evaluation) maintain a cache to avoid repeat evaluations. Sometimes, a VirtualArray is put inside its own cache.
- Python garbage collector can detect reference cycles, immediately when a reference count goes to zero or during mark-and-sweep.
- But it can't detect a reference cycle through C++.
- We tried to fix this by making the VirtualArray cache a weakref, but this had its own share of issues: #230, #400, #432, #479, #541, #560, #597, #603, #655, #679, #783, #865, #899, #940, #1052...

Issue #2: Reference cycles through C++

- In most circumstances, Awkward Arrays have no reference cycles because they're immutable.
- VirtualArrays (for lazy evaluation) maintain a cache to avoid repeat evaluations. Sometimes, a VirtualArray is put inside its own cache.
- Python garbage collector can detect reference cycles, immediately when a reference count goes to zero or during mark-and-sweep.
- But it can't detect a reference cycle through C++.
- We tried to fix this by making the VirtualArray cache a weakref, but this had its own share of issues: #230, #400, #432, #479, #541, #560, #597, #603, #655, #679, #783, #865, #899, #940, #1052...
- Even sneakier: reference cycles can be created by function closures; VirtualArrays hold Python functions. (Anything mutable is an opening!)

Issue #2: Reference cycles through C++

- In most circumstances, Awkward Arrays have no reference cycles because they're immutable.
- VirtualArrays (for lazy evaluation) maintain a cache to avoid repeat evaluations. Sometimes, a VirtualArray is put inside its own cache.
- Python garbage collector can detect reference cycles, immediately when a reference count goes to zero or during mark-and-sweep.
- But it can't detect a reference cycle through C++.
- We tried to fix this by making the VirtualArray cache a weakref, but this had its own share of issues: #230, #400, #432, #479, #541, #560, #597, #603, #655, #679, #783, #865, #899, #940, #1052...
- Even sneakier: reference cycles can be created by function closures; VirtualArrays hold Python functions. (Anything mutable is an opening!)
- Dropping C++ solves this, but we're also replacing VirtualArrays with Dask.

Issue #3: Releasing the GIL

If you're using pybind11, be aware that it does not release Python's GIL by default. The GIL undermines multithreading.

(See `py::gil_scoped_acquire`, `py::gil_scoped_release`, and `py::call_guard`.)

Issue #3: Releasing the GIL

If you're using pybind11, be aware that it does not release Python's GIL by default. The GIL undermines multithreading.

(See `py::gil_scoped_acquire`, `py::gil_scoped_release`, and `py::call_guard`.)

Furthermore, you can't release the GIL if you ever need to change any Python object, even a reference counter.

Issue #3: Releasing the GIL

If you're using pybind11, be aware that it does not release Python's GIL by default. The GIL undermines multithreading.

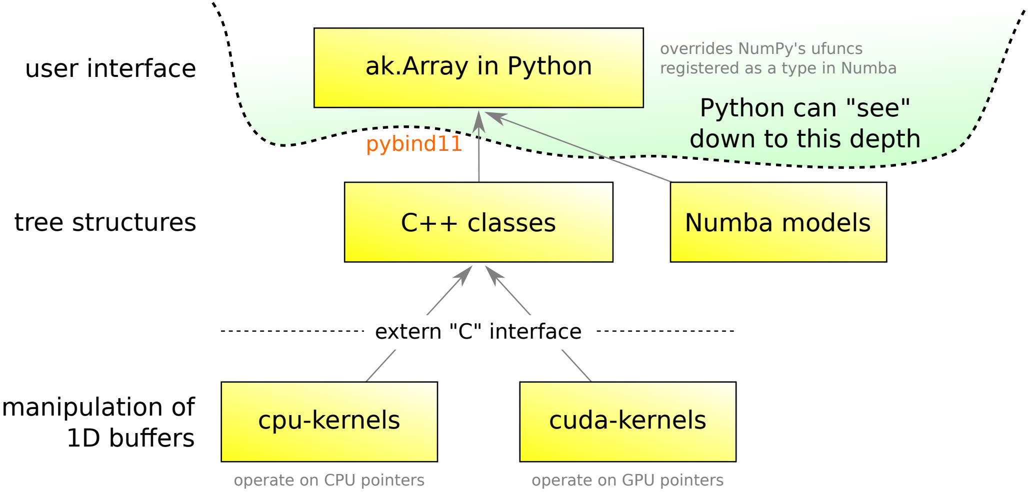
(See `py::gil_scoped_acquire`, `py::gil_scoped_release`, and `py::call_guard`.)

Furthermore, you can't release the GIL if you ever need to change any Python object, even a reference counter.

Awkward Array's C++ objects hold references to Python objects (mostly NumPy arrays) and decrement the reference count if the C++ object is deleted. Since that could happen at any time, we can't release the GIL without risking segfaults.

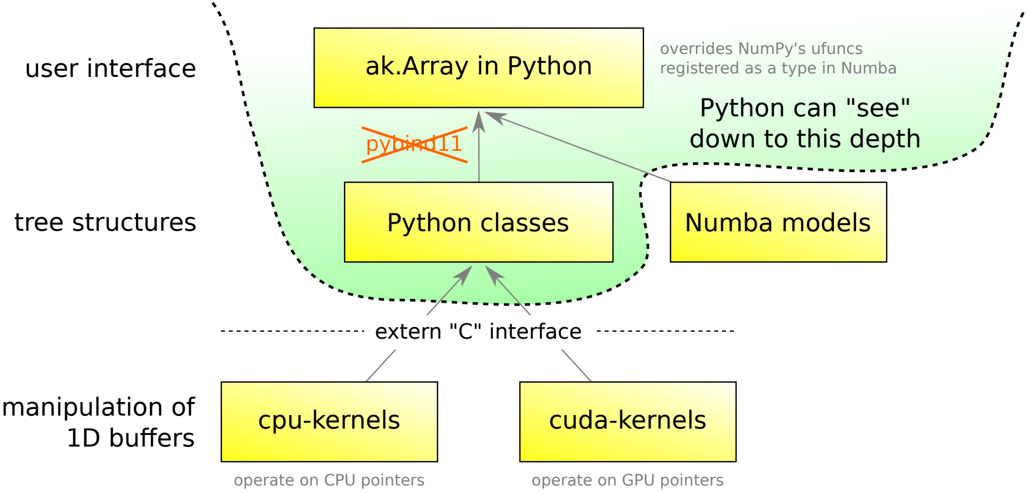
Issue #4: Algorithm visibility

The *final* reason for C++ → Python refactoring: third-party libraries like Dask and JAX couldn't "see" enough of what Awkward Array was doing.



Issue #4: Algorithm visibility

The *final* reason for C++ → Python refactoring: third-party libraries like Dask and JAX couldn't "see" enough of what Awkward Array was doing.



Issue #4: Algorithm visibility

These libraries use “tracers” to inspect a path through Python code, like this:

```
>>> import numpy as np
>>> class Tracer(np.lib.mixins.NDArrayOperatorsMixin):
...     def __init__(self):
...         self.functions = []
...     def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
...         self.functions.append(ufunc.__name__)
...         return self
...
>>> t = Tracer()
>>> np.sin(t)**2 + np.cos(t)**2
<__main__.Tracer object at 0x7ff2a0f33e80>
>>> t.functions
['sin', 'power', 'cos', 'power', 'add']
```

Tracers can see binary ops and functions, but calls out of Python are black boxes.

Issue #4: Algorithm visibility

The “tree structures” (mid-level) of Awkward Array are too coarse for these libraries because the third-party libraries don’t recognize arrays of trees.

They *do* recognize simple arrays of numbers (our “kernels,” or low-level).

See Anish Biswas’s presentation at https://indico.cern.ch/event/1033648 for a heroic attempt to integrate Awkward Array and JAX using PyTrees.

Issue #4: Algorithm visibility

The "tree structures" (mid-level) of Awkward Array are too coarse for these libraries because the third-party libraries don't recognize arrays of trees.

They *do* recognize simple arrays of numbers (our "kernels," or low-level).

See Anish Biswas's presentation at https://indico.cern.ch/event/1033648 for a heroic attempt to integrate Awkward Array and JAX using PyTrees.

This last problem with Awkward Array's C++ layer is the one that finally convinced us to change.

Awkward Array's Python-C++ codebase was a bad design.

Awkward Array's Python-C++ codebase was a bad design.

Does that mean you shouldn't mix Python and C++?

No!

Just watch out for these issues when you do.

1. Don't plan on sharing stdlib (e.g. `std::vector`) or pybind11 objects between Python modules. Share data as basic C types.
2. Python and C++ can share references, but ensure no reference cycles! 100% immutability would prevent cycles.
3. You can only release the GIL in blocks of code that do not touch Python in any way, including reference counts.
4. Think about interoperability plans: how much of your code will third-party libraries need to see?

Basically, keep Python and C++ an arm's length apart.

Basically, keep Python and C++ an arm's length apart.

A "loose coupling" avoids most of these issues.

Basically, keep Python and C++ an arm's length apart.

A "loose coupling" avoids most of these issues.

(This may be an advertisement for Julia.)