The Awkward World of Python and C++

Manasvi Goyal[1], Ianna Osborne[2], Jim Pivarski[2]

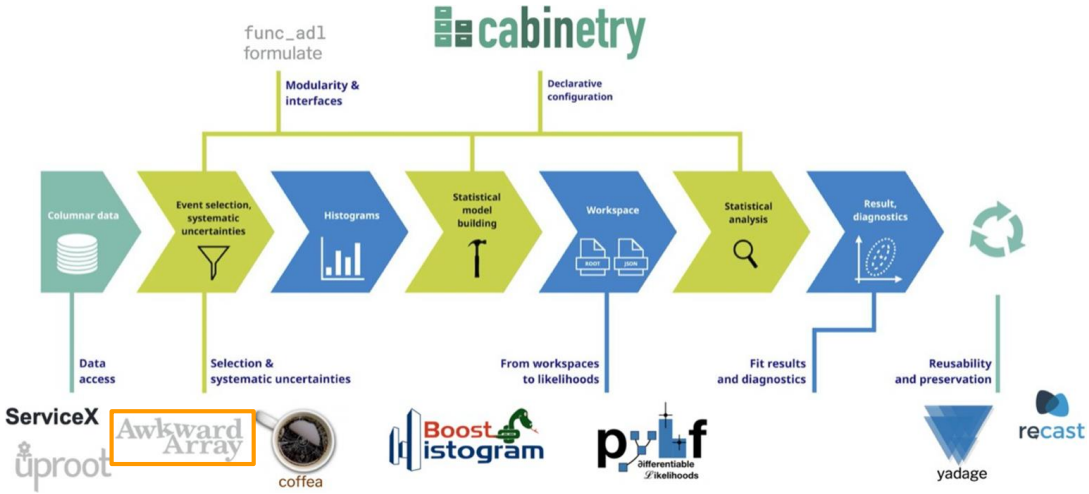
[1]*Delhi Technological University*

[2]*Princeton University*

Support for this work was provided by National Science Foundation cooperative agreement OAC-1836650.

Awkward Arrays

- Awkward Array is a library for nested, variable-sized data, including arbitrary-length lists, records, mixed types, and missing data, to manipulate JSON-like data using *NumPy-like idioms*.



DOI 10.5281/zenodo.4341376

How it works?

Example of an “awkward” array

```
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]}]
])
```

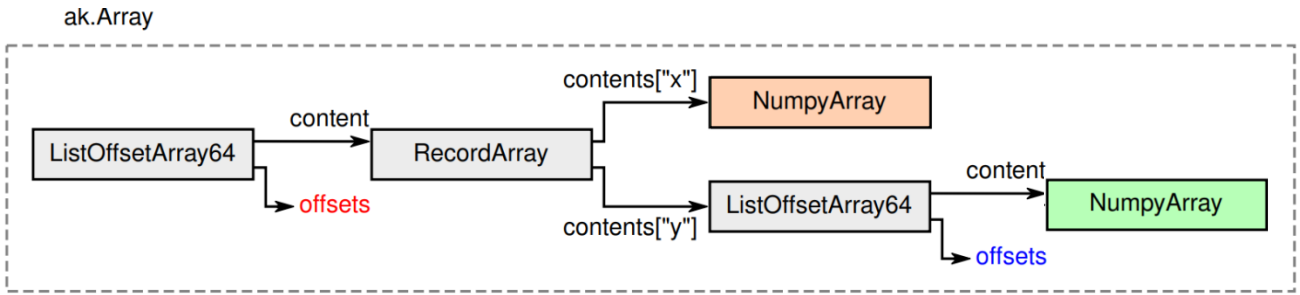
Record structures with differently typed fields

Array of variable-length lists (“ragged” or “jagged“ arrays)

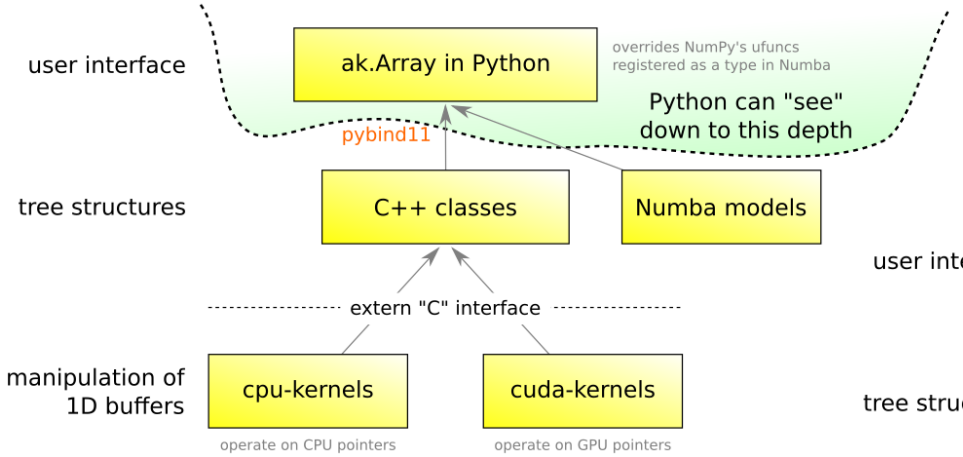
Nested variable-length lists

Missing data

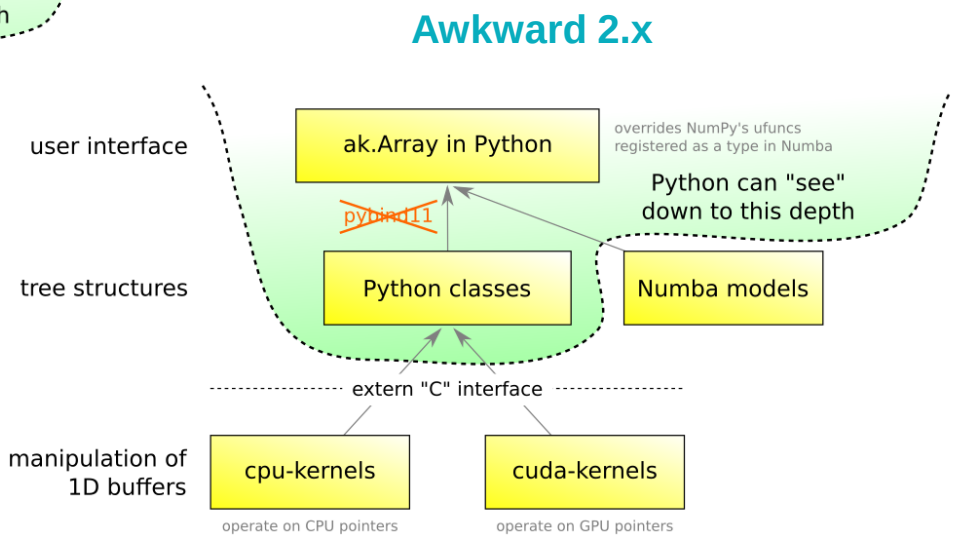
Heterogenous data (union/variant types)



Evolution of Architecture



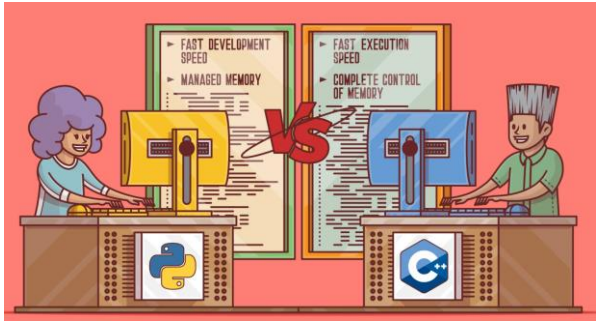
Awkward 1.x



Lessons learned in Python-C++ integration
Jim Pivarski, Princeton University, ACAT 2021

Python-C++ Integration

- Binding Python and C++ to get advantage of best features of both languages.



How to do it right?

- The header-only implementation allows using Awkward Arrays in an external project without linking to the awkward libraries.
- No specialised data types - only raw buffers, strings, and integers.
- The header only implementation allows for multiple applications.

Layout Builders

- “Layout” consists of composable elements that determine how an array is structured.
- Designed to build Awkward Arrays faster because it knows the type. (ArrayBuilder - discovers the type)
- Implementation details: C++14, header only, etc..
    - It uses header-only GrowableBuffer (see backup slides for details).
- *awkward::LayoutBuilder* specializes an Awkward data structure using C++ templates, which can then be filled and converted to a Python Awkward Array through *ak.from_buffers*.

Record Builder Example

```cpp
#include "awkward/LayoutBuilder.h"

enum Field : std::size_t {x, y};

UserDefinedMap fields_map({
  {Field::x, "x"},
  {Field::y, "y"}});
```

Constructing a Layout Builder from variadic templates!

```cpp
RecordBuilder<
    RecordField<Field::x, NumpyBuilder<double>>,
    RecordField<Field::y, ListOffsetBuilder<int64_t,
        NumpyBuilder<int32_t>>>
> builder;
```

```cpp
builder.set_field_names(fields_map);

auto& x_builder = builder.field<Field::x>();
auto& y_builder = builder.field<Field::y>();
```

Filling the Layout Builders

```cpp
x_builder.append(1.1);
auto& y_subbuilder =
  y_builder.begin_list();
y_subbuilder.append(1);
y_builder.end_list();
```

Record 1

```cpp
x_builder.append(2.2);
y_builder.begin_list();
y_builder.end_list();
```

Record 2

```cpp
x_builder.append(3.3);
y_builder.begin_list();
y_subbuilder.append(1);
y_subbuilder.append(2);
y_builder.end_list();
```

Record 3

Equivalent Array

```
[
  {"x": 1.1, "y": [1]},
  {"x": 2.2, "y": []},
  {"x": 3.3, "y": [1, 2]},
]
```

Record Builder User Interface

- Retrieve the set of buffer names and their sizes (as a no. of bytes):

```
std::map<std::string, size_t> names_nbytes = {};
builder.buffer_nbytes(names_nbytes);
```

- Allocate memory for these buffers in Python `np.empty(nbytes, dtype=np.uint8)` and get *void** pointers to these buffers by casting the output of `numpy_array.ctypes.data`.
- Let the LayoutBuilder fill these buffers:

```
std::map<std::string, void*> buffers;
builder.to_buffers(buffers);
```

- Finally, you get the JSON form with:

```
std::string form =
builder.form();
```

Layout Builder Form

```
{
"class": "RecordArray",
"contents": {
    "x": {
        "class": "NumpyArray",
        "primitive": "float64",
        "form_key": "node1"
    },
    "y": {
         class": "ListOffsetArray",
         "offsets": "i64",
         "content": {
             "class": "NumpyArray",
             "primitive": "int32",
             "form_key": "node3"
          },
          "form_key": "node2"
    } "
},
"form_key": "node0"
}
```

Awkward Arrays from RDataFrame

ak.from_rdataframe converts the selected columns as native Awkward Arrays.

Uses the C++ header-only implementation to simplify JIT compilation.

Awkward Arrays to RDataFrame and back

See Ianna Osborne's poster!

```
NumpyBuilder = cppyy.gbl.awkward.LayoutBuilder.Numpy[data_type]
builder = NumpyBuilder()
form = ak._v2.forms.from_json(form_str)
builder_type = type(builder).__cpp_name__

cpp_buffers_self.fill_from[builder_type](builder)
names_nbytes = cpp_buffers_self.names_nbytes[builder_type](builder)

buffers = empty_buffers(cpp_buffers_self, names_nbytes)
cpp_buffers_self.to_char_buffers[builder_type, data_type](builder)

array = ak._v2.from_buffers(
    form,
    builder.length(),
    buffers,
)
return _wrap_as_record_array(array)
```

Awkward Array in ctapipe

A framework for prototyping the low-level data processing algorithms for the Cherenkov Telescope Array.

Refactor their implementation to use Awkward Array

Array types are known in at compile time.

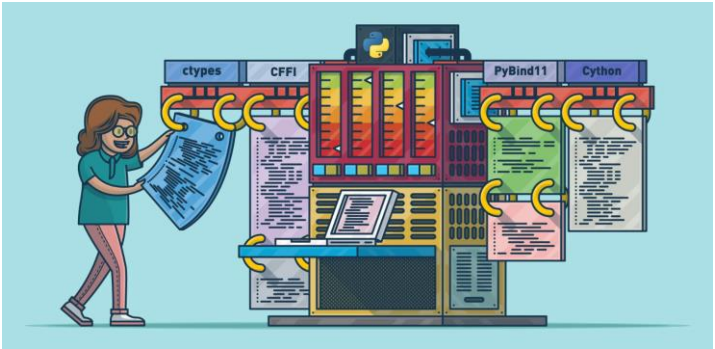
A EventIO format (a machine-independent hierarchical data format)

An event has the following attributes:

- `header`: a namedtuple containing the Corsika Event Header data
- `end_block`: a numpy array containing the Corsika Event End data
- `time_offset`, `x_offset`, `y_offset`: the offset of the array

Summary

- Header-only libraries that only fills buffers for downstream code to pass from C++ to Python using only C types.
- Opens up the door for users to analyse their data in Python by integrating Awkward Arrays with their projects easily without any hassle!
- Include awkward::LayoutBuilder directly without linking against platform-specific libraries or worrying about native dependencies



Additional Information

Have a look at these talks/posters for more information about Awkward Arrays, RDataFrame and more…

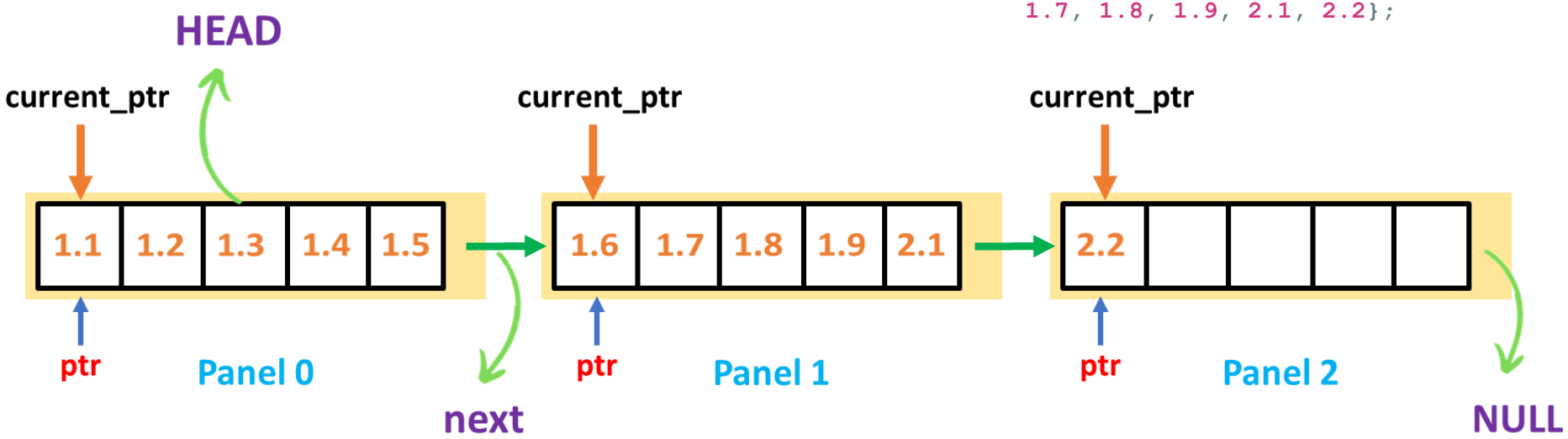
- Compiling Awkward Lorentz Vectors with Numba, Saransh Chopra
- Differentiating through Awkward Arrays using JAX and a new CUDA backend for Awkward Arrays, Anish Biswas
- High performance analysis with RDataFrame and the python ecosystem: Scaling and Interoperability, Josh Bendavid, Kenneth Long
- RDataFrame: a flexible and scalable analysis experience, Vincenzo Eduardo Padulano

Backup Slides

Growable Buffer with Panels

```
size_t panel_size = 5;

double data[11] = {1.1, 1.2, 1.3, 1.4, 1.5, 1.6,
                   1.7, 1.8, 1.9, 2.1, 2.2};
```



to_buffers in Layout & Array Builders

Float64 ArrayBuilder

(concatenates data, typecast, returns form)

```
const std::string
Float64Builder::to_buffers(BuffersContainer& container, int64_t& form_key_id) const {
  std::stringstream form_key;
  form_key << "node" << (form_key_id++);

  buffer_.concatenate(
    reinterpret_cast<double*>(
      container.empty_buffer(form_key.str() + "-data",
      buffer_.length() * (int64_t)sizeof(double))));

  return "{\"class\": \"NumpyArray\", \"primitive\": \"float64\", \"form_key\": \""
         + form_key.str() + "\"}";
}
```

```
void
to_buffers(std::map<std::string, void*>& buffers) const noexcept {
   offsets_.concatenate(static_cast<PRIMITIVE*>
     (buffers["node" + std::to_string(id_) + "-offsets"]));
   content_.to_buffers(buffers);
}
```

ListOffset LayoutBuilder

(concatenates data and fills it in a map)