PRINCETON UNIVERSITY

iris hep

An Awkward module

for round-tripping data structures between Python and Julia

Ianna Osborne, Jim Pivarski, Jerry Ling

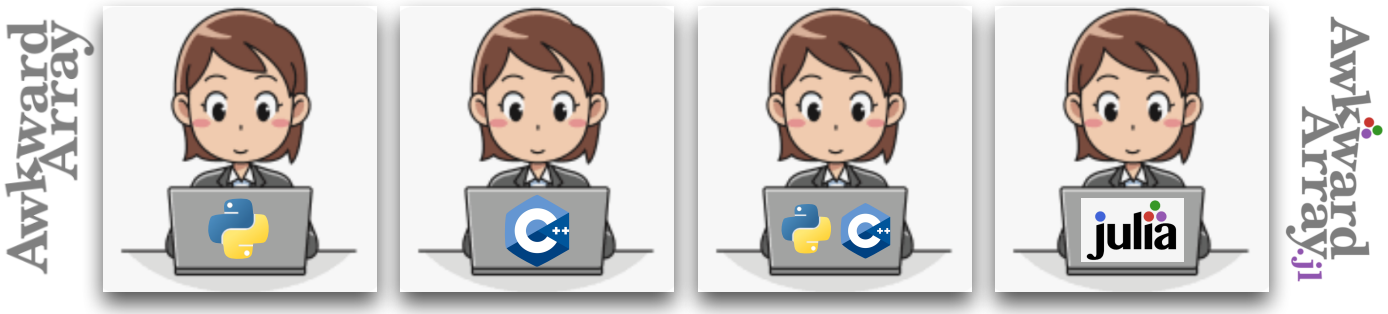
PRINCETON UNIVERSITY

iris hep

Introduction

Empowering Data Science and Scientific Computing

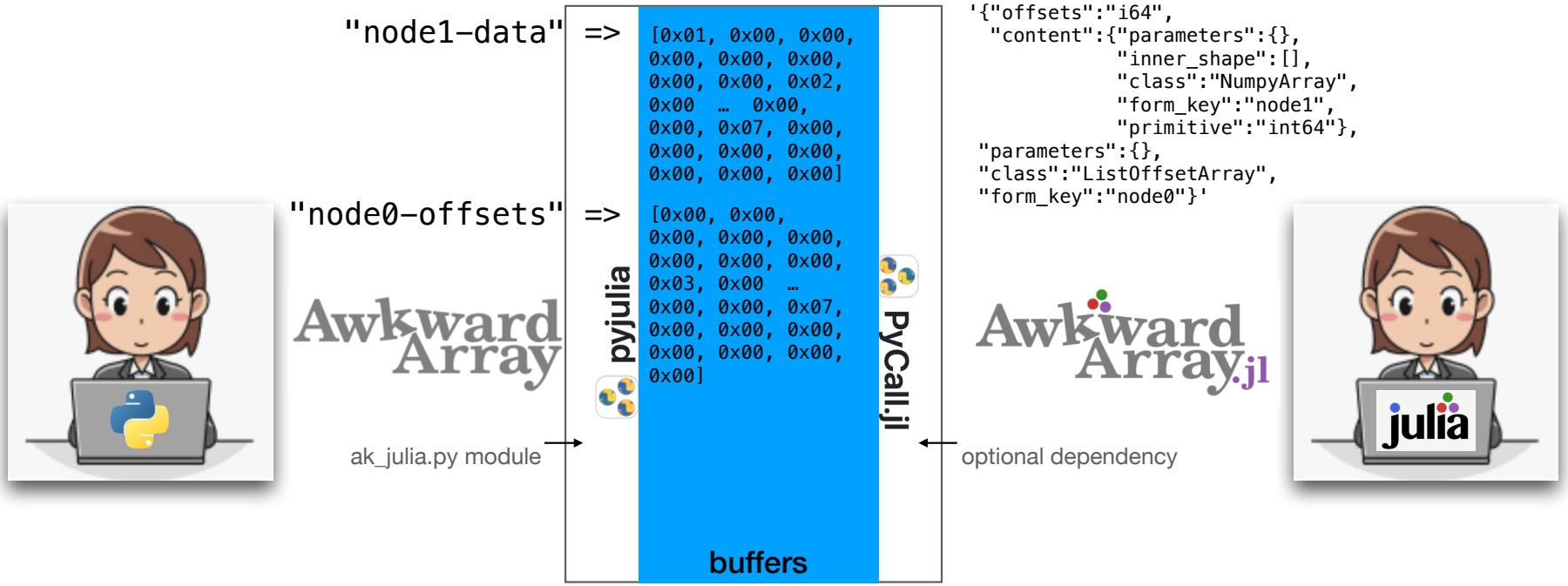
- Why now and why Julia? See Jim's talk Engaging the HEP community in Julia
- Sharing Awkward Array data structures between Python and Julia to encourage the Python users to run their analysis both in an eco-system of their choice and in Julia



PRINCETON UNIVERSITY

iris hep

PyJulia and PyCall facilitate seamless Awkward Array data exchange between Python and Julia



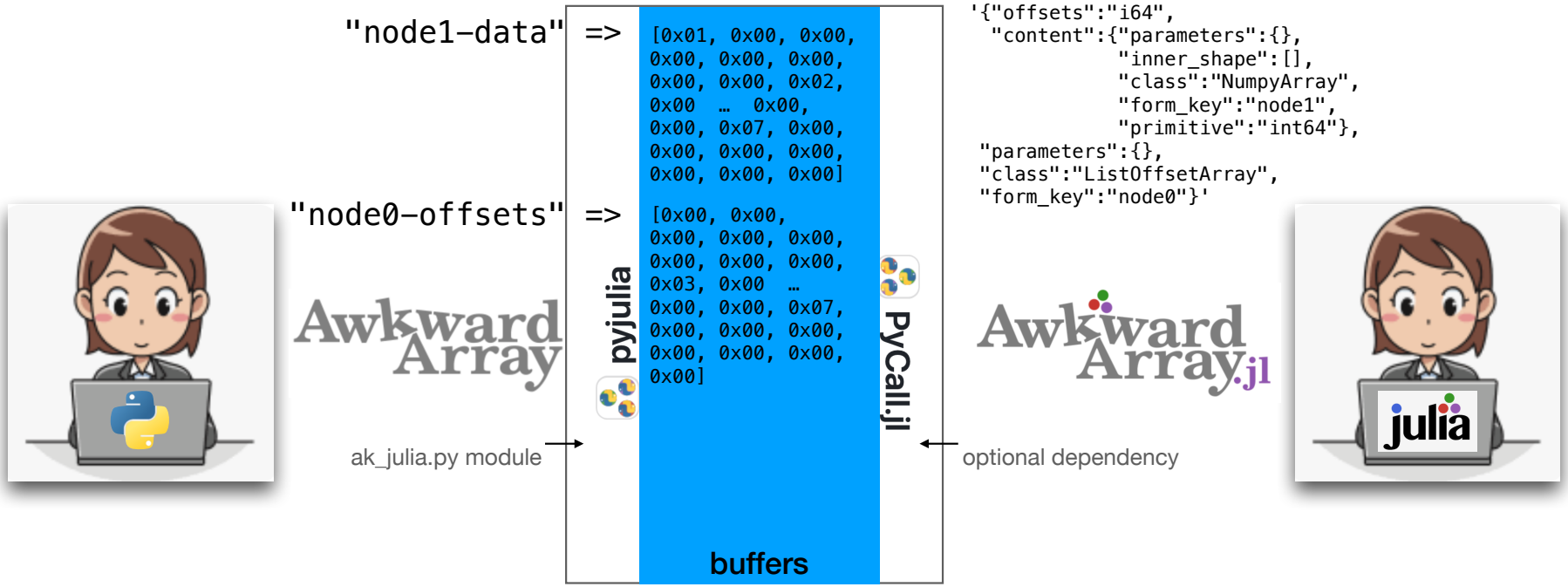

```
array = ak.from_buffers(form, len, containers)
form, len, containers = ak.to_buffers(array)
```

```
array = AwkwardArray.from_iter([[1, 2, 3], [4], [5, 6, 7]])
form, len, containers = AwkwardArray.to_buffers(array)
array = AwkwardArray.from_buffers(form, len, containers)
```

PRINCETON UNIVERSITY

iris hep

PyJulia and PyCall facilitate seamless Awkward Array data exchange between Python and Julia




```
array = ak.from_buffers(form, len, containers)
form, len, containers = ak.to_buffers(array)
```

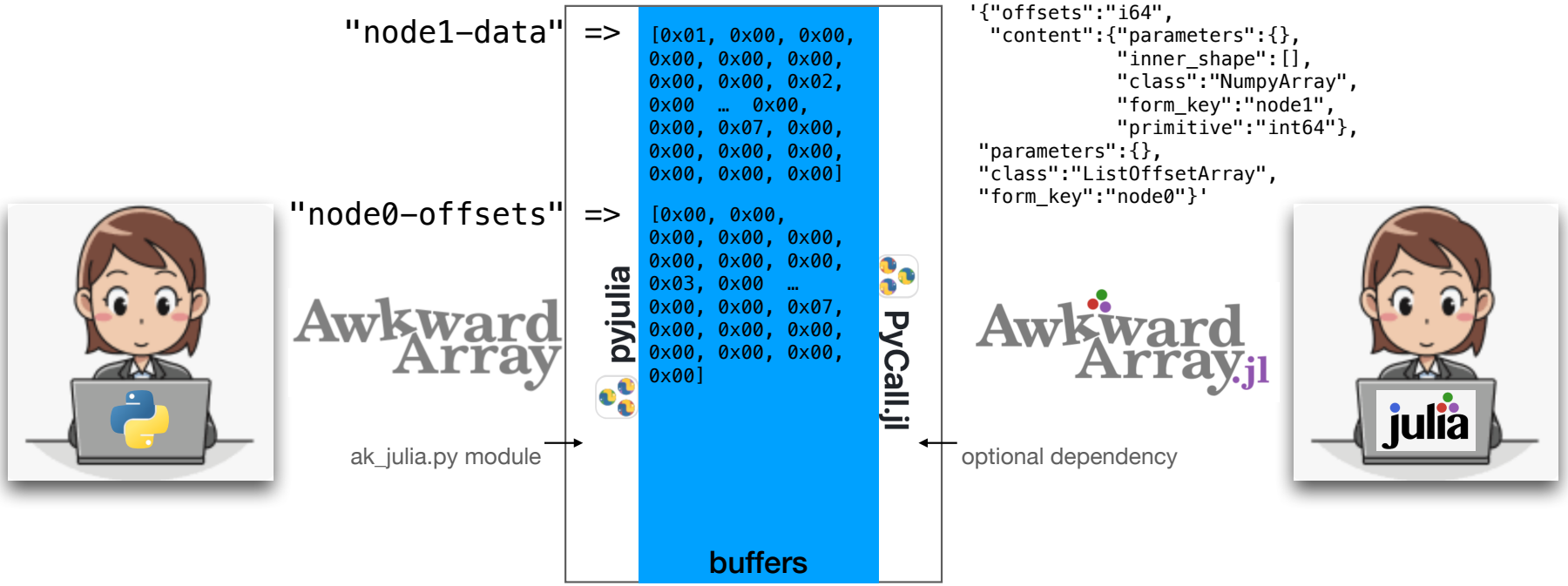
```
array = AwkwardArray.from_iter([[1, 2, 3], [4], [5, 6, 7]])
form, len, containers = AwkwardArray.to_buffers(array)
array = AwkwardArray.from_buffers(form, len, containers)
```

Q: PyJulia latest release 0.6.1? Is PyJulia and PyCall the right direction?

PRINCETON UNIVERSITY

iris hep

PyJulia and PyCall facilitate seamless Awkward Array data exchange between Python and Julia




```
'{"offsets":"i64",
  "content":{"parameters":{},
             "inner_shape":[],
             "class":"NumpyArray",
             "form_key":"node1",
             "primitive":"int64"},
 "parameters":{},
 "class":"ListOffsetArray",
 "form_key":"node0"}'
```

```
array = ak.from_buffers(form, len, containers)
form, len, containers = ak.to_buffers(array)
```

```
array = AwkwardArray.from_iter([[1, 2, 3], [4], [5, 6, 7]])
form, len, containers = AwkwardArray.to_buffers(array)
array = AwkwardArray.from_buffers(form, len, containers)
```

~~*Q: PyJulia latest release 0.6.1? Is PyJulia and PyCall the right direction?*~~

Thanks to Dr Chris Rackauckas

A: PythonCall & JuliaCall is a better choice

PRINCETON UNIVERSITY

iris hep

Awkward Arrays to buffers

example

```
julia> array = AwkwardArray.from_iter([[1, 2, 3], [4], [5, 6, 7]])
3-element AwkwardArray.ListOffsetArray{Vector{Int64}, AwkwardArray.PrimitiveArray{Int64,
Vector{Int64}, :default}, :default}:
 [1, 2, 3]
 [4]
 [5, 6, 7]

julia> form, len, containers = AwkwardArray.to_buffers(array)
("{\"offsets\":\"i64\",\"content\":{\"parameters\":{},\"inner_shape\":[],\"class\":
\"NumpyArray\",\"form_key\":\"node1\",\"primitive\":\"int64\"},\"parameters\":{},
\"class\":\"ListOffsetArray\",\"form_key\":\"node0\"}", 3,
Dict{String, AbstractVector{UInt8}}("node1-data" => [0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x02, 0x00  …  0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00],
"node0-offsets" => [0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x03, 0x00  …  0x00,
0x00, 0x07, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00]))
```

PRINCETON UNIVERSITY

iris hep

Awkward Arrays from buffers wrapper

user friendly

```
julia> array
3-element AwkwardArray.ListOffsetArray{Vector{Int64},
AwkwardArray.PrimitiveArray{Int64, Vector{Int64}, :default}, :default}:
 [1, 2, 3]
 [4]
 [5, 6, 7]

julia> ak_array = julia_array_to_python(array)
PyObject <Array [[1, 2, 3], [4], [5, 6, 7]] type='3 * var * int64'>

julia> ak.sum(ak_array)
28
```

PRINCETON UNIVERSITY

iris hep

Awkward Arrays from buffers

example

```
function julia_array_to_python(array)
    form, len, containers = AwkwardArray.to_buffers(array)

    numpy_arrays = Dict{String, Any}()

    for (key, value) in containers
        numpy_arrays[key] = np.asarray(value, dtype=np.uint8)
    end

  return ak.from_buffers(form, len, numpy_arrays)
end
```

PRINCETON UNIVERSITY

iris hep

Awkward Arrays from buffers

example

```
function julia_array_to_python(array)
    form, len, containers = AwkwardArray.to_buffers(array)

    numpy_arrays = Dict{String, Any}()

    for (key, value) in containers
        numpy_arrays[key] = np.asarray(value, dtype=np.uint8)
    end

  return ak.from_buffers(form, len, numpy_arrays)
end
```

Q: Who owns the memory?

PRINCETON UNIVERSITY

iris hep

Zero-Copy Array Passing: Julia to Python

example

- Zero-copy passing allows data to be transferred from Julia to Python without unnecessary data copying
- PyCall provides the means to interface with Python objects from Julia, ensuring efficient data transfer

```
>>> jl.eval("""
... using PyCall
... @pyimport numpy as np
... julia_array = [1, 2, 3]
... py_array = PyObject(julia_array)
... np.array(py_array)
... """)
array([1, 2, 3], dtype=int64)

>>> julia_array = jl.eval("julia_array")
>>> julia_array
array([1, 2, 3], dtype=int64)

>>> type(julia_array)
<class 'numpy.ndarray'>

>>> julia_array.ctypes.data
6422235712
>>> jl.eval("""
... pointer(julia_array)
... """)
c_void_p(6422235712)
```

PRINCETON UNIVERSITY

iris hep

Memory Usage

Reduced Memory Overhead

- Zero-copy array passing minimizes memory usage by eliminating the need for duplicate copies of data in both Python and Julia
- Zero-copy UnROOT.jl to Python Awkward Arrays? Who owns the data? Julia?
- User story when a copy is needed? Julia -> Python -> Julia

```
julia> function measure_memory_usage()
           julia_array = rand(10^7)
           array_size = Base.summarysize(julia_array)
           return array_size
       end
measure_memory_usage (generic function with 1 method)

julia> memory_usage = measure_memory_usage()
80000040

julia> println("Memory Usage (Julia): $memory_usage bytes")
Memory Usage (Julia): 80000040 bytes
```

```
>>> def measure_memory_usage():
...     large_array = np.random.rand(10**7)
...     array_size = large_array.nbytes
...     return array_size
...
>>> memory_usage = measure_memory_usage()
>>> print(f"Memory Usage (Python): {memory_usage} bytes")
Memory Usage (Python): 80000000 bytes
```

PRINCETON UNIVERSITY

Performance Acceleration

calling Julia from Python

```
In [65]: %%timeit
    ...: tmp_array = jl.eval("array")
    ...:
    ...:
423 ms ± 58.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)


In [66]: jl.eval(
    ...:     """
    ...: function path_length(array)
    ...:             total = 0.0
    ...:             for i in 1:length(array)
    ...:                 for j in 1:length(array[i])
    ...:                     total += array[i][j]
    ...:                 end
    ...:             end
    ...:             return total
    ...:         end
    ...: """
    ...: )
Out[66]: <PyCall.jlwrap path_length>

In [67]: %%timeit
    ...: jl.eval("path_length(array)")
    ...:
    ...:
133 ms ± 2.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [75]: array
Out[75]: <Array [[], [5.5, 24.2], ..., [...], [-22.1]] type='1048576 * var * float32'>

In [76]: ak.sum(array)
Out[76]: 53228.05

In [77]: %%timeit
    ...: ak.sum(array)
    ...:
    ...:
6.16 ms ± 56.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [47]: def path_length(array):
    ...:     total = 0.0
    ...:     for i in range(len(array)):
    ...:         for j in range(len(array[i])):
    ...:             total = total + array[i][j]
    ...:     return total

In [48]: %%timeit
    ...: total = path_length(array)
    ...:
    ...:
    ...:
2min 55s ± 1min 10s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

PRINCETON UNIVERSITY

iris hep

Julia Multithreading

easy to use

- Not all operations and libraries are automatically parallelized
- It's important to carefully design the code and choose appropriate algorithms to take full advantage of multithreading

```
julia> using Base.Threads

julia> num_threads = nthreads()
4

julia> println("Number of threads available: ", num_threads)
Number of threads available: 4

julia> using AwkwardArray

julia> data = AwkwardArray.PrimitiveArray([1, 2, 3])
3-element AwkwardArray.PrimitiveArray{Int64, Vector{Int64}, :default}:
 1
 2
 3

julia> @threads for i in 1:4
           push!(data, 100*i)
       end

julia> data
7-element AwkwardArray.PrimitiveArray{Int64, Vector{Int64}, :default}:
   1
   2
   3
 100
 300
 400
 200
```

PRINCETON UNIVERSITY

iris hep

Challenges

of integrating Python and Julia

- Learning Curve & Code Consistency: Migrating from Python to Julia may require team members to learn new syntax, libraries, and best practices, potentially leading to inconsistencies in the codebase.

- Dependency Management & Debugging: Ensuring the availability of necessary packages and debugging code that spans both languages can be challenging, especially for complex projects.

- Performance Optimization & Updates: Achieving performance gains and staying updated with the latest features and best practices requires expertise.

PRINCETON UNIVERSITY

iris hep

Conclusions

and questions

- Physicists are using Awkward Array in Python and data format conversion is the hardest part of language boundary-hopping.
  - Having access to this data structure will smooth the way for physicists to try Julia
- PyJulia and PyCall enable seamless data sharing between Python and Julia, eliminating the need for costly data copying operations.
  - We want to apply the zero-copy judiciously
- We welcome suggestions on managing version compatibility, dependencies, etc.
  - Your feedback is crucial for a smooth integration of Python and Julia

Thank you!

PRINCETON UNIVERSITY

iris hep

"Julia, with its high performance and flexibility, is poised to be the language of the future for scientific computing, data analysis, and beyond. Its ability to seamlessly integrate with other languages opens up a world of possibilities for developers and researchers alike."

Thank you, ChatGPT!