PRINCETON UNIVERSITY

iris hep

Bridging Worlds:

Achieving Language Interoperability between Julia and Python in Scientific Computing

Ianna Osborne, Jim Pivarski, Jerry Ling, 13 Mar 2024

ACAT 2024

PRINCETON UNIVERSITY

iris hep

The Landscape of Scientific Computing

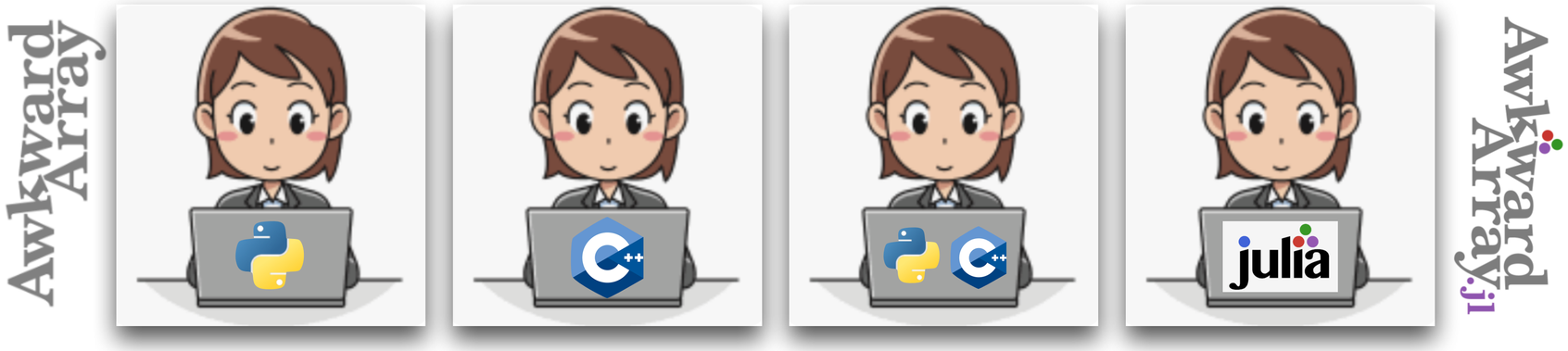
Python and Awkward Array at this workshop

- Python has long been a favorite for its simplicity and vast ecosystem:
  - 108. Awkward Family: expanding functionality through interrelated Python packages by Jim Pivarski
  - 31. Describe Data to get Science-Data-Ready Tooling: Awkward as a Target for Kaitai Struct YAML by Manasvi Goyal
  - 100. Using Legacy ATLAS C++ Calibration Tools in Modern Columnar Analysis Environments by Matthias Vigl
  - 122. Easy columnar file conversions with "hepconvert" by Zoë Bilodeau
  - 84. ServiceX, the novel data delivery system, for physics analysis by Kyungeon Choi
  - 104. dilax: Differentiable Binned Likelihoods in JAX by Manfred Peter Fackeldey

AwkwardArray.jl

Julia Introduction to Python Community

- Physicists are using Awkward Array in Python and data format conversion is the hardest part of language boundary-hopping
- 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
- See Jim's talk "Engaging the HEP community in Julia"

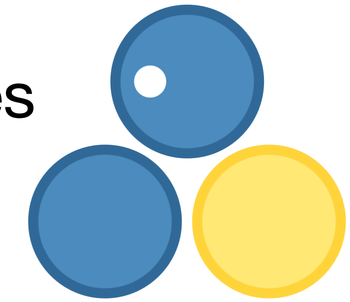


PRINCETON UNIVERSITY

iris hep

Interoperability
and GIL

- PythonCall and Julia Call allow to call Python code from Julia and Julia code from Python via a symmetric interface
  - Software that connects Julia and Python languages 14 repositories



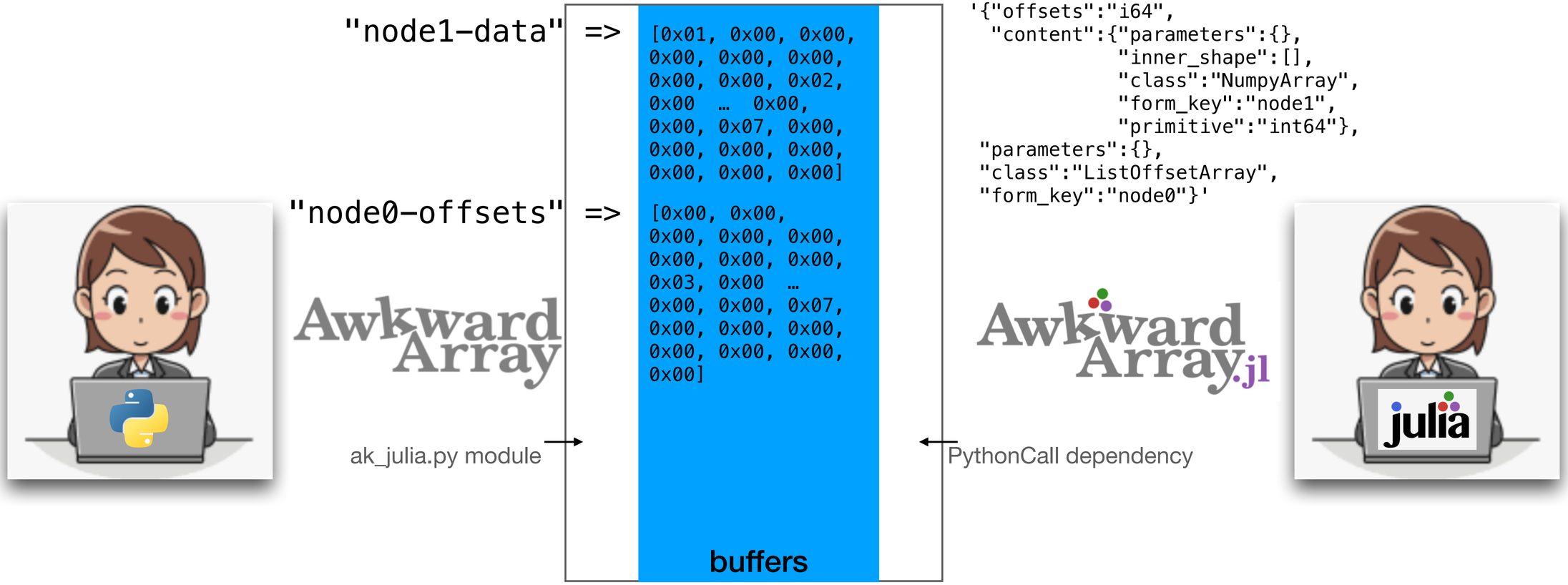
- Dropping GIL when calling Julia from Python (JuliaCall):
  - “If you can guarantee your Julia code doesn't call back into Python, you can release the GIL yourself. A future version of JuliaCall will allow something like *some_julia_function.jl_call_nogil(x, y, z)* to make this a bit easier.”

```
pythread = PythonCall.C.PyEval_SaveThread()
try
    # code which doesn't touch Python
finally
    PythonCall.C.PyEval_RestoreThread(pythread)
end
```

PRINCETON UNIVERSITY

iris hep

PythonCall and JuliaCall 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

Converting Awkward Arrays

from Julia to Python and from Python to Julia

```julia
using PythonCall
using AwkwardArray: convert

   array = AwkwardArray.ListOffsetArray(
        [0, 3, 3, 5],
        AwkwardArray.PrimitiveArray([1.1, 2.2, 3.3, 4.4, 5.5]),
    )
    py_array = convert(array)

    # Check if the function returns an awkward array
    py_array isa Py

    # Check if the awkward array has the correct layout
    typeof(py_array) == Py

    ak_array = pyconvert(Vector, pyimport("awkward").to_list(py_array))
    ak_array == [[1.1, 2.2, 3.3], [], [4.4, 5.5]]
```

```julia
   py_array = pyimport("awkward").Array([[1.1, 2.2, 3.3], [], [4.4, 5.5]])

   # Check if the function returns an awkward array
   array = convert(py_array)
   array isa AwkwardArray.ListOffsetArray

   array == [[1.1, 2.2, 3.3], [], [4.4, 5.5]]
```

PRINCETON UNIVERSITY

iris hep

Runtime Environment

and Julia package manager

- Conda.jl package allows one to use conda as a cross-platform binary provider for Julia for other Julia packages, especially to install binaries that have complicated dependencies like Python.

- The package dependencies are described in the Project.toml:

```
[deps]
Conda = "8f4d0f93-b110-5947-807f-2305c1781a2d"
JSON = "682c06a0-de6a-54ab-a142-c8b1cf79cde6"
PythonCall = "6099a3de-0909-46bc-b1f4-468b9a2dfc0d"
Tables = "bd369af6-aec1-5ad0-b16a-f7cc5008161c"
```

```
% julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.9.3 (2023-08-24)
 _/ |\__'_|_|_|\__'_|  |  https://github.com/conda-forge/julia-feedstock
|__/                   |

julia> using AwkwardArray

(@ak-julia) pkg> st
Status `~/anaconda3/envs/ak-julia/share/julia/environments/ak-julia/Project.toml`
  [7d259134] AwkwardArray v0.1.2
  [6e4b80f9] BenchmarkTools v1.4.0
  [992eb4ea] CondaPkg v0.2.22
  [a93c6f00] DataFrames v1.6.1
  [31a5f54b] Debugger v0.7.8
  [e30172f5] Documenter v1.2.1
  [682c06a0] JSON v0.21.4
  [98e50ef6] JuliaFormatter v1.0.45
  [6099a3de] PythonCall v0.9.15
  [295af30f] Revise v3.5.10
  [fd094767] Suppressor v0.2.6
  [bd369af6] Tables v1.11.1
  [3cd96dde] UnROOT v0.10.20
```

PRINCETON UNIVERSITY

iris hep

Python module JuliaCall

- The Python module JuliaCall can be installed with *pip install juliacall*

```
% python
Python 3.12.0 | packaged by conda-forge | (main, Oct  3 2023, 08:43:38) [Clang 15.0.7 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> from juliacall import Main as jl

[juliapkg] Locating Julia ^1.6.1
[juliapkg] Using Julia 1.9.3 at /Users/yana/anaconda3/envs/ak-julia/bin/julia
[juliapkg] Using Julia project at /Users/yana/anaconda3/envs/ak-julia/julia_env
[juliapkg] Installing packages:
           julia> import Pkg
           julia> Pkg.add([Pkg.PackageSpec(name="PythonCall", uuid="6099a3de-0909-46bc-b1f4-468b9a2dfc0d")])
           julia> Pkg.resolve()
           julia> Pkg.precompile()
   Resolving package versions...
    Updating `~/anaconda3/envs/ak-julia/julia_env/Project.toml`
  [6099a3de] + PythonCall v0.9.15
    Updating `~/anaconda3/envs/ak-julia/julia_env/Manifest.toml`
  [992eb4ea] + CondaPkg v0.2.22
  [9a962f9c] + DataAPI v1.15.0
  [e2d170a0] + DataValueInterfaces v1.0.0
  [82899510] + IteratorInterfaceExtensions v1.0.0
  [692b3bcd] + JLLWrappers v1.5.0
  [0f8b85d8] + JSON3 v1.14.0
  [1914dd2f] + MacroTools v0.5.11
  [0b3b1443] + MicroMamba v0.1.14
```

PRINCETON UNIVERSITY

iris hep

Runtime Environment

and Python package dependencies

- AwkwardArray installed in conda environment

```
% conda list
# packages in environment at /Users/y/anaconda3/envs/ak-julia:
#
# Name                    Version                   Build  Channel
awkward                   2.4.6              pyhd8ed1ab_0    conda-forge
awkward-cpp               24                 py312h49ebfd2_0    conda-forge
```

```
>>> import awkward as ak
>>> jl.seval("using AwkwardArray")
```

julia

```
julia> using PythonCall

julia> py_array = pyimport("awkward").Array([[1.1, 2.2, 3.3], [], [4.4, 5.5]])
Python: <Array [[1.1, 2.2, 3.3], [], [4.4, 5.5]] type='3 * var * float64'>
```

PRINCETON UNIVERSITY

iris hep

Calling Julia from Python

```
>>> import awkward as ak
>>> jl.seval("using AwkwardArray")
>>> array = ak.Array([[1,2,3],[],[4,5]])
>>> array
<Array [[1, 2, 3], [], [4, 5]] type='3 * var * int64'>
```

```
>>> jl.seval("""
... 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
... """
... )
Julia: path_length (generic function with 1 method)
```

```
>>> jl.path_length(array)
15.0
```

- Performance critical code should be inside a function
- Functions should take arguments, instead of operating on global variables

julia

```
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
```

PRINCETON UNIVERSITY

iris hep

Conversion to Julia

1. define custom conversion rule

- The conversion rules used whenever converting a Python object to a Julia object:

```
jl.seval("""
function pyconvert_rule_awkward_array_listoffset(
  ::Type{AwkwardArray.ListOffsetArray}, x::Py)
    array = AwkwardArray.convert(x)
    return PythonCall.pyconvert_return(array)
end
""")
```

PRINCETON UNIVERSITY

iris hep

Custom Conversion Rules

2. register it with PythonCall

```
jl.seval("""
    PythonCall.pyconvert_add_rule(
      "awkward.highlevel:Array",
      AwkwardArray.ListOffsetArray,
      pyconvert_rule_awkward_array_listoffset,
      PythonCall.PYCONVERT_PRIORITY_ARRAY
    )
""")
```

PRINCETON UNIVERSITY

iris hep

Potential Issues

Record Array example

- Create an Awkward Array in Julia and convert it to an Awkward Array in Python:

```
jl.seval("""
content_layout = AwkwardArray.RecordArray((
  a = AwkwardArray.PrimitiveArray([1, 2, 3, 4, 5]),
  b = AwkwardArray.PrimitiveArray([1.1, 2.2, 3.3, 4.4, 5.5])))
“””)
Julia:
5-element AwkwardArray.RecordArray{(:a, :b), Tuple{AwkwardArray.PrimitiveArray{Int64,
Vector{Int64}, :default}, AwkwardArray.PrimitiveArray{Float64,
Vector{Float64}, :default}}, :default}:
 {a: 1, b: 1.1}
 {a: 2, b: 2.2}
 {a: 3, b: 3.3}
 {a: 4, b: 4.4}
 {a: 5, b: 5.5}
>>> py_rec_arr = jl.convert(jl.content_layout)
>>> py_rec_arr
<Array [{a: 1, b: 1.1}, ..., {a: 5, ...}] type='5 * {a: int64, b: float64}'>
```

PRINCETON UNIVERSITY

iris hep

Potential Issues

Fields in Record Array

- One possible solution is to restrict the interface of the Record Array, for instance, by providing a 'field' attribute.

```
>>> jl.seval("""
content_layout[:a]
""")
Julia:
5-element AwkwardArray.PrimitiveArray{Int64, Vector{Int64}, :default}:
 1
 2
 3
 4
 5
>>> py_rec_arr = jl.convert(jl.content_layout)
>>> py_rec_arr["a"]
<Array [1, 2, 3, 4, 5] type='5 * int64'>
```

PRINCETON UNIVERSITY

iris hep

Potential Issues

first element indexing

- It's important to note that in Julia, container indexing starts with 1, unlike some other languages where indexing starts with 0
- One solution to ensure portability in code is to avoid hardcoding numbers for indexing and instead use the *firstindex* function, which adjusts based on the indexing conventions of the language.

```
>>> jl.content_layout[0]
Julia: {a: 1, b: 1.1}
>>> jl.content_layout[2]
Julia: {a: 3, b: 3.3}
>>> jl.content_layout[3]
Julia: {a: 4, b: 4.4}
>>> jl.seval("""
... content_layout[0]
... """)
juliacall.JuliaError: BoundsError: attempt to access 5-element Vector{Int64} at index [0]
```

```
>>> jl.seval("""
... content_layout[1]
... """)
Julia: {a: 1, b: 1.1}
```

```
>>> jl.seval("""
... firstindex(content_layout)
... """)
1
```

PRINCETON UNIVERSITY

iris hep

Use Cases

- Integrating high-performance Julia libraries into existing Python projects.
- Leveraging Python's extensive data analysis and visualization libraries in Julia workflows.
- Building hybrid applications that combine Python's ease of use with Julia's speed.

```
>>> cms_file = jl.ROOTFile("/PyHEP2023/Run2012BC_DoubleMuParked_Muons.root")
>>> cms_mumu_tree = jl.LazyTree(cms_file, "Events")
```

UnROOT.jl

```
>>> cms_mumu_tree
Julia:
```

Row	Muon_phi SubArray{Float3	nMuon UInt32	Muon_pt SubArray{Float3	Muon_eta SubArray{Float3	Muon_charge SubArray{Int	…
1	[-0.0343, 2.5	2	[10.8, 15.7]	[1.07, -0.564	[-1, -1]	…
2	[-0.275, 2.54	2	[10.5, 16.3]	[-0.428, 0.34	[1, -1]	…
3	[-1.22]	1	[3.28]	[2.21]	[1]	…
4	[-2.08, 0.251	4	[11.4, 17.6,	[-1.59, -1.75	[1, 1, 1, 1]	…
5	[-2.37, -2.31	4	[3.28, 3.64,	[-2.17, -2.18	[-1, -1, 1,	…
6	[-2.91, 2.46,	3	[3.57, 4.57,	[-1.37, -0.70	[-1, 1, -1]	…
7	[-0.0718, 3.0	2	[57.6, 53.0]	[-0.532, -1.0	[-1, 1]	…
8	[-2.25, -2.18	2	[11.3, 23.9]	[-0.772, -0.7	[1, -1]	…
9	[0.678, -2.03	2	[10.2, 14.2]	[0.442, 0.702	[-1, 1]	…
10	[3.13, 3.02]	2	[11.5, 3.47]	[2.34, 2.35]	[-1, 1]	…
11	[-3.07, -0.38	2	[8.82, 17.6]	[1.65, 0.715]	[1, -1]	…
12	[1.0, 0.024]	2	[14.6, 12.3]	[1.26, -1.29]	[1, 1]	…
13	[-2.3, 0.34,	6	[35.6, 15.1,	[-2.15, 0.176	[1, 1, 1, -1	…
14	[0.549, -2.27	3	[21.8, 9.55,	[-1.38, -0.77	[1, 1, 1]	…
⋮	⋮	⋮	⋮	⋮	⋮	⋱

2 columns and 61540399 rows omitted

PRINCETON UNIVERSITY

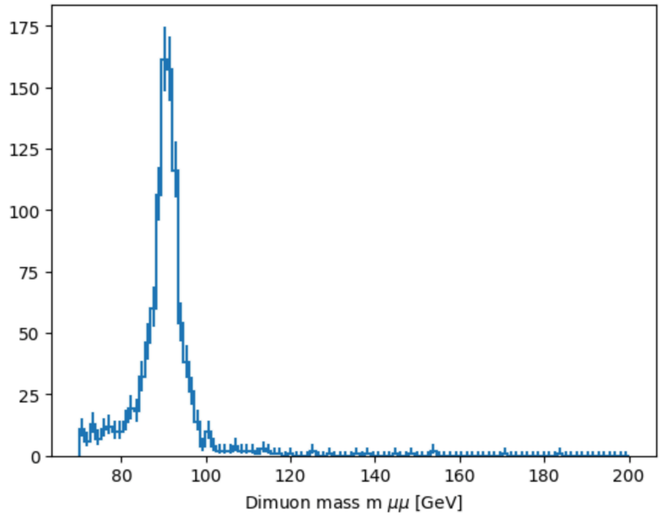
iris hep

CMS Data Analysis Example

from CERN Open Data portal DOI:10.7483/OPENDATA.CMS.LVG5.QT81

```
jl.seval("""
function invariant_mass(tree)
  layout = AwkwardArray.PrimitiveArray{Float64}()
  for event in tree
    if event.nMuon == 2
      if event.Muon_charge[1] != event.Muon_charge[2]
        result = sqrt(2 * event.Muon_pt[1] * event.Muon_pt[2] *
                  (cosh(event.Muon_eta[1] - event.Muon_eta[2]) -
                   cos(event.Muon_phi[1] - event.Muon_phi[2])))
        if result > 70
          push!(layout, result)
        end
      end
    end
  end
  layout
end
""")
Julia: invariant_mass (generic function with 1 method)
```

- Tutorial in Python with RDataFrame




```
>>> ak.mean(result[:1000])
91.43827227020263
```

```
>>> result = jl.invariant_mass(cms_mumu_tree)
5638140-element AwkwardArray.PrimitiveArray{Float32, Vector{Float32}, :default}:
```

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
- PythonCall and JuliaCall enable seamless data sharing between Python and Julia, eliminating the need for costly data copying operations.
  - We want to apply the zero-copy judiciously
  - If GIL is an issue it has to be handled manually
- Your feedback is welcome

Thank you!

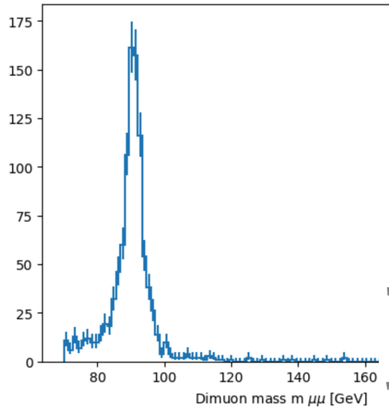
Support for this work was provided by NSF cooperative agreement OAC-1836650 (IRIS-HEP) and PHY-2323298

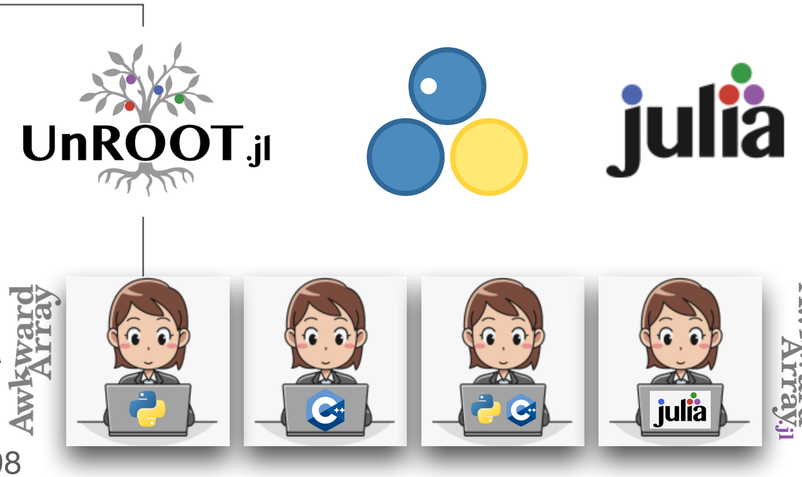
PRINCETON UNIVERSITY

Bridging Worlds:
Achieving Language Interoperability between Julia and Python in Scientific Computing

Ianna Osborne, Jim Pivarski, Jerry Ling, 13 Mar 2024

- AwkwardArray.jl in Julia serves as a bridge for Python users transitioning to Julia.
- The process of passing data structures from Python to Julia and utilizing Awkward Array establishes a bridge between the two languages, enabling bidirectional interoperability.
- Physicists are using Awkward Array in Python and **data format conversion is the hardest part of language boundary-hopping.**
- 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.
- PythonCall and JuliaCall allow to call Python code from Julia and Julia code from Python via a symmetric interface.
- Use Cases:
  - Adding fast Julia libraries to your Python projects
  - Using Python's data tools in Julia workflows
  - Creating apps that are easy like Python, but fast like Julia





Support for this work was provided by NSF cooperative agreement OAC-1836650 (IRIS-HEP) and PHY-2323298