

Engaging the HEP community in Julia

Jim Pivarski

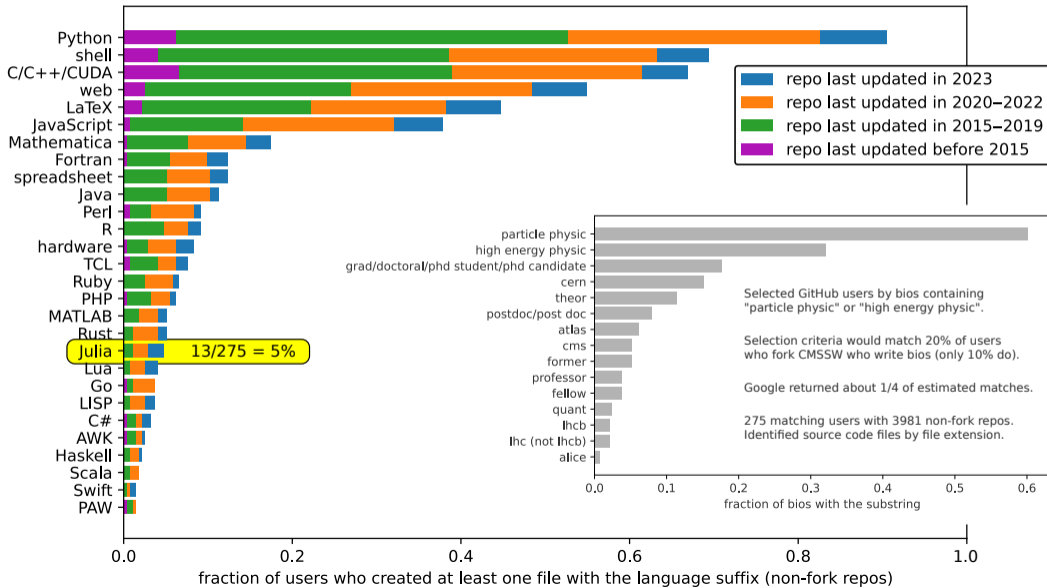
Princeton University – IRIS-HEP

November 6, 2023



Let's start with some numbers...

State of language use by particle physicists as of last Friday





Among “Materials” (PDFs and TXTs) in CERN’s Indico search since January 2022,

63 refer to Julia the programming language

324 refer to people named Julia

4 other/unclear

12 refer to Rust the programming language

(7 of those same documents also refer to Julia)

10 refer to oxidized metal

3 other/unclear

1 refers to Lua the programming language

(it’s used to configure the SIMION charged particle simulator)

4 refer to the LHC User’s Association

4 other/unclear



Similarly, it is increasingly a focus on ACAT and CHEP

ACAT 2022:

- ▶ Julia: 1 title and 1 abstract
- ▶ Python: 3 titles and 24 abstracts

CHEP 2023:

- ▶ Julia: 3 titles and 4 abstracts
- ▶ Python: 1 title and 35 abstracts

Only other programming languages mentioned: C++ (frequently) and Java (2 times).

And, it's the only language-based HSF group other than PyHEP



The HEP Software Foundation facilitates cooperation and **common efforts** in High Energy Physics software and computing internationally.

 [JuliaHEP 2023, 6-9 November 2023 \(more info\)](#)

Meetings

The HSF holds **regular meetings** in its activity areas and has bi-weekly coordination meetings as well. All of our meetings are open for everyone to join.

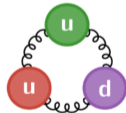
- [HSF Coordination Meeting #258, 12 October 2023](#)
- [HSF Coordination Meeting #257, 28 September 2023](#)
- [HSF Coordination Meeting #256, 14 September 2023](#)

[Upcoming HSF and community events »](#)

[Full list of past meetings »](#)

JuliaHEP Launches

After a lot of rising interest in Julia for HEP in the last few years, the HSF has started a new **JuliaHEP** working group.



We just published a new paper [Potential of the Julia programming language for high energy physics computing](#) and we're planning the first [JuliaHEP Workshop](#) in November. Keep an eye out for upcoming Julia events in the [calendar](#)!

Activities

We organise many activities, from our **working groups**, to organising **events**, to supporting projects as **HSF projects**, and helping communication within the community through our **discussion forums** and **technical notes**.

The HSF can also write **letters of collaboration and cooperation** to project proposals.

[How to get involved »](#)



Julia is not yet “adopted” in HEP, but it is getting more attention than any other rival to C++ and Python.



Julia is not yet “adopted” in HEP, but it is getting more attention than any other rival to C++ and Python.

From here, it could continue to rise in prominence or end up passing as a fad. *This is a critical time.*



As we've seen, Julia is a perfect fit for HEP, technologically.



As we've seen, Julia is a perfect fit for HEP, technologically.

- ▶ It allows for an exploratory phase, in which the data analyst focuses on *what* to compute, rather than how it will be accelerated.



As we've seen, Julia is a perfect fit for HEP, technologically.

- ▶ It allows for an exploratory phase, in which the data analyst focuses on *what* to compute, rather than how it will be accelerated.
- ▶ It allows the exploratory code to be tweaked to scale up to large datasets.



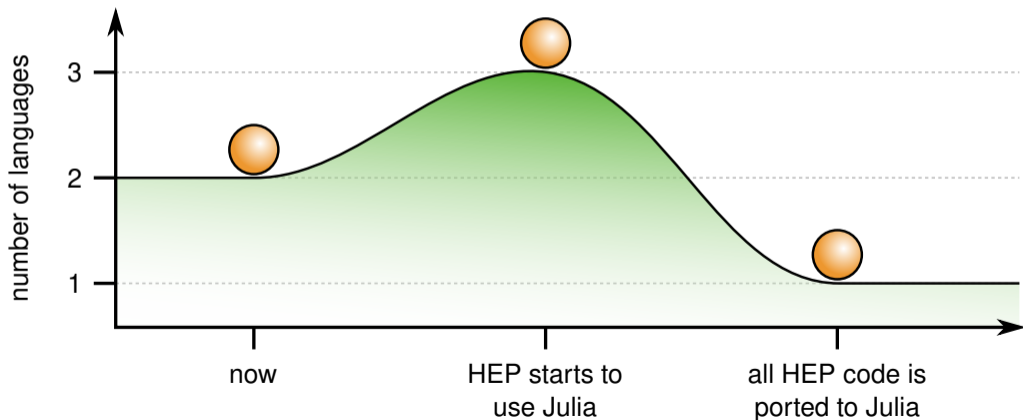
As we've seen, Julia is a perfect fit for HEP, technologically.

- ▶ It allows for an exploratory phase, in which the data analyst focuses on *what* to compute, rather than how it will be accelerated.
- ▶ It allows the exploratory code to be tweaked to scale up to large datasets.

There is a *gradual path* from brainstorming to optimized code, not a rewrite.



This argument focuses on Julia as a solution to the two-language problem, but we can't go from two languages to one language without going through three.



~~“If you build it, they will come.”~~



Preparing a complete stack of HEP tools in Julia will help adoption, but it will not eliminate the interim 3-language period.

~~“If you build it, they will come.”~~



Preparing a complete stack of HEP tools in Julia will help adoption, but it will not eliminate the interim 3-language period.

There will not be any clean break in which everyone is ready to set aside their old tools and take up new ones.

~~“If you build it, they will come.”~~



Preparing a complete stack of HEP tools in Julia will help adoption, but it will not eliminate the interim 3-language period.

There will not be any clean break in which everyone is ready to set aside their old tools and take up new ones.

(The closest approximation to that in HEP was the Fortran \rightarrow C++ transition, which was mandated top-down and lost a generation of HEP programmers.)



We need to give users *short-term reasons* to add Julia as a second or third language in their analysis work.



We need to give users *short-term reasons* to add Julia as a second or third language in their analysis work.

(“You’ll be able to replace all your Python and C++” is a long-term reason.)



☰ README.md

Awkward Array

pip package 2.4.8 conda-forge v2.4.6 python 3.8-3.12 license BSD 3-Clause Tests passing

Scikit-HEP Project NSF 1836650 DOI 10.5281/zenodo.4341376 docs online chat online

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

Arrays are **dynamically typed**, but operations on them are **compiled and fast**. Their behavior coincides with NumPy when array dimensions are regular and generalizes when they're not.

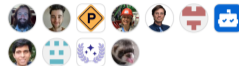
Motivating example [↗](#)

Given an array of lists of objects with `x`, `y` fields (with nested lists in the `y` field),

```
import awkward as ak

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]}]
])
```

Contributors 37



+ 26 contributors

Languages





☰ README.md

Awkward Array.jl

for Julia!






Why? [↗](#)

[Awkward Array](#) is a library for manipulating large-scale arrays of nested, variable-sized data in Python, using array-oriented idioms: like NumPy, but for any JSON-like data. In Python, using array-oriented idioms to avoid imperative for loops is necessary for fast computations. In Julia, imperative code is already fast, thanks to JIT-compilation, so you may be wondering why this package exists.

This package is a complete, one-to-one implementation of the Awkward Array data structures in Julia, which makes it possible to zero-copy share data between the two languages. Python scripts can sneak out to Julia to run a calculation at high speed. Julia programs can duck into Python to access some code that has been written in that language. [PyJulia](#) and [PyCall.jl](#) provide these capabilities (which this package uses) for ordinary data types; this package allows arrays of complex data to be shared as well.

Beyond communication with Python, columnar memory layouts have some advantages: data in an Awkward

Contributors 3

-  **jpivarski** Jim Pivarski
-  **Moelf** Jerry Ling
-  **ianna** Ianna Osborne

Languages

-  **Julia** 100.0%

This is what I was proposing in 2021



Julia for HEP Mini-workshop

Monday Sep 27, 2021, 3:00 PM → 6:45 PM Europe/Zurich

Benjamin Krikler (University of Bristol (GB)) , Eduardo Rodríguez (University of Liverpool (GB)) , Philippe Gras (Université Paris-Saclay (FR))

Description The PyHEP WG is launching a study of the potential Julia usage for HEP.

During this mini-workshop we will share the information which has already been collected, discuss the matters for which we need to share opinions, identify the questions that will require some work to be answered.

The workshop will focus on discussions and possibly practical work. The goal is the preparation of a report on the potential of Julia for HEP and recommendations on its usage.

The topics to be addressed are listed [here](#).

Because of the nature of the workshop, the agenda of the day will involve depending on the interests of the participants and the ideas that come out during the event and its preparation.

Some resource on Julia that can be consulted before the workshop

- Official site and documentation: <https://julialang.org/>
- Learning the language: <https://juliaacademy.com/>
- HEP represented at the Julia computing conference with the example of a [physics analysis](#) using Julia
- A github corner: <https://github.com/JuliaHEP>
- [Julia session at last PyHEP annual workshop](#)

Minutes (CodiMD d...) [Recording of works...](#)

Registration You are registered for this event.

99 [Modify registration](#)

Participants

Adam Lyon	Alberto Sanchez Hernandez	Alexander Held	Alexander Moreno Briceño	Alexei Strelchenko
Alexis Vallier	Amani Besma Bouasla	Andrea Valassi	Andreas Salzburger	Benedikt Hegner

12 How an Awkward Array/Julia bridge can introduce HEP to Julia.

Speaker: Jim Pivarski (Princeton University)

pivarski-awkward-ju...



Awkward Array has other JIT-compiled backends

- ▶ Numba: `ak.Arrays` can be arguments and return values of `@nb.njit`-compiled functions.



Awkward Array has other JIT-compiled backends

- ▶ Numba: `ak.Arrays` can be arguments and return values of `@nb.njit`-compiled functions.
- ▶ Numba-CUDA: `@nb.cuda.njit([extensions=ak.numba.cuda])`.



- ▶ Numba: `ak.Arrays` can be arguments and return values of `@nb.njit`-compiled functions.
- ▶ Numba-CUDA: `@nb.cuda.njit([extensions=ak.numba.cuda])`.
- ▶ ROOT RDataFrame: `ak.to_rdataframe/ak.from_rdataframe`.



Awkward Array has other JIT-compiled backends

- ▶ Numba: `ak.Arrays` can be arguments and return values of `@nb.njit`-compiled functions.
- ▶ Numba-CUDA: `@nb.cuda.njit([extensions=ak.numba.cuda])`.
- ▶ ROOT RDataFrame: `ak.to_rdataframe/ak.from_rdataframe`.
- ▶ cppy: `ak.Arrays` can be arguments and return values of functions defined by `cpyy.cppdef` (pass `ak.Array.cpp_type` as its C++ type).



- ▶ Numba: `ak.Arrays` can be arguments and return values of `@nb.njit`-compiled functions.
- ▶ Numba-CUDA: `@nb.cuda.njit([extensions=ak.numba.cuda])`.
- ▶ ROOT RDataFrame: `ak.to_rdataframe/ak.from_rdataframe`.
- ▶ cppy: `ak.Arrays` can be arguments and return values of functions defined by `cpyy.cppdef` (pass `ak.Array.cpp_type` as its C++ type).
- ▶ And now Julia.



In both Numba and C++, we define

- ▶ a Python `Lookup` object to hold a reference to the `ak.Array`, preventing it from going out of scope, and to present its tree-navigation metadata in a raw-byte format, and
- ▶ a Numba or C++ `ArrayView` object that points to a position in the structure, JIT-compiled to behave differently for each tree-node type.



In both Numba and C++, we define

- ▶ a Python `Lookup` object to hold a reference to the `ak.Array`, preventing it from going out of scope, and to present its tree-navigation metadata in a raw-byte format, and
- ▶ a Numba or C++ `ArrayView` object that points to a position in the structure, JIT-compiled to behave differently for each tree-node type.
- ▶ Numba and C++ do not own the array! It's a borrowed reference!



In both Numba and C++, we define

- ▶ a Python `Lookup` object to hold a reference to the `ak.Array`, preventing it from going out of scope, and to present its tree-navigation metadata in a raw-byte format, and
- ▶ a Numba or C++ `ArrayView` object that points to a position in the structure, JIT-compiled to behave differently for each tree-node type.
- ▶ Numba and C++ do not own the array! It's a borrowed reference!

In Julia, we define

- ▶ the whole layout tree in native Julia structures, and convert.



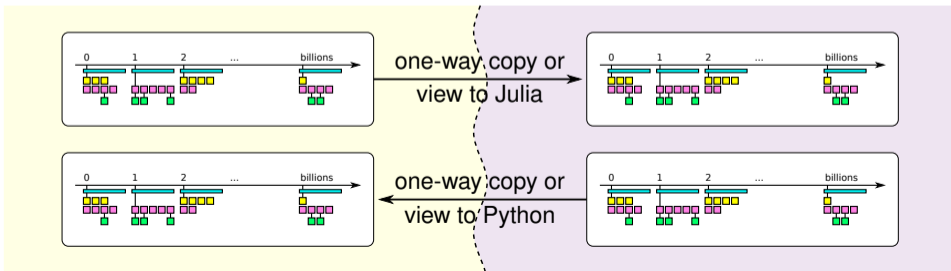
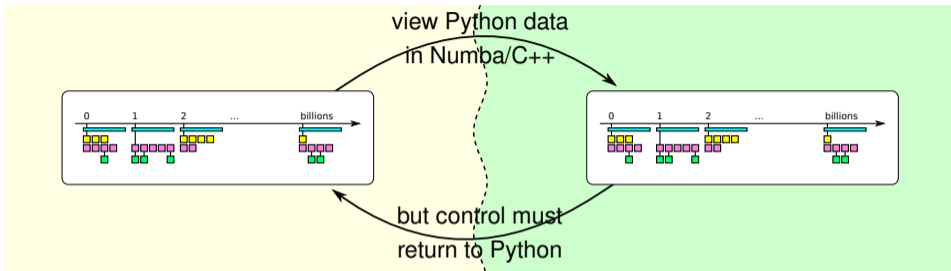
In both Numba and C++, we define

- ▶ a Python `Lookup` object to hold a reference to the `ak.Array`, preventing it from going out of scope, and to present its tree-navigation metadata in a raw-byte format, and
- ▶ a Numba or C++ `ArrayView` object that points to a position in the structure, JIT-compiled to behave differently for each tree-node type.
- ▶ Numba and C++ do not own the array! It's a borrowed reference!

In Julia, we define

- ▶ the whole layout tree in native Julia structures, and convert.
- ▶ Julia can own the array!

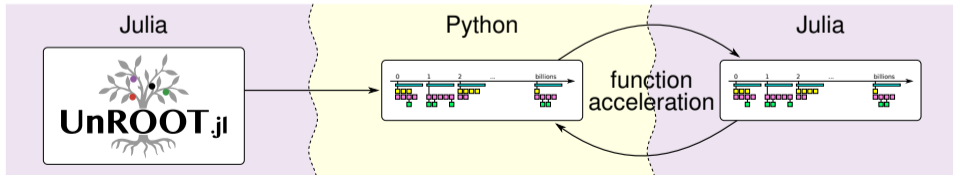
Python and Julia Awkward Arrays are symmetric, others are not





- ▶ In Numba, especially Numba-CUDA, only unowned views make sense. Control *will* return to Python.
- ▶ In RDataFrames created from Python, control will return to Python. We might need to reconsider this if we need to enable `RDataFrame::Snapshot` or distributed RDataFrames.
- ▶ Views make sense for cppy functions that will deconstruct the `ak.Array` before sending it on to other C++ libraries. If Awkward Arrays are to have a life in C++ beyond cppy, they'll need to be reimplemented as in Julia.
- ▶ In Julia, Awkward Arrays may be passed to other libraries as an opaque **Any**, **AbstractArray**, or a transparent `AwkwardArray.Content`.

This also opens the door to UnROOT.jl becoming a drop-in replacement for Uproot in Python workflows.





python bindings for rntuple, implementation of "uproot-cpp" #15

Igray opened this issue on Jul 4 · 14 comments



jpivarski commented on Jul 4

Member ...

I've been in favor of a compiled-but-Python-friendly Uproot for some time, but it's always been too large of a task—this will require dedicated effort and coordination (because I'm assuming more than one developer).

Some questions to ask about such a thing:

- Perhaps the compiled language should be Julia: [UnROOT.jl](#) already exists. Can its Python bindings be developed more?
- For common use-cases, precompiled is better, and [scientific-python/cookie](#) gives us the options of Scikit-Build/pybind11 for C++ and maturin for Rust.
- We also shouldn't disregard the possibility of doing it in Numba, since that can be partially compiled, partially not, and it has more affinity with Python types, as well as prior expertise among likely developers. In terms of JIT technology, it's no better or worse than the Julia option (it's all LLVM).

The main difference between these three options is what *people* you want to or are able to get together with this. Option 1 pulls Julia developers more into the Coffea world, option 2 is for people who like blank pages, starting from scratch, and option 3 is for pulling it together quickly with the Python + Numba expertise that's already in this area.



Composability: All buffers in a `AwkwardArray`.Content tree are **AbstractVector**, so it should be easy to swap in special features, like GPU-resident arrays, autodiff, units, etc.



Composability: All buffers in a `AwkwardArray.Content` tree are **AbstractVector**, so it should be easy to swap in special features, like GPU-resident arrays, autodiff, units, etc.

In Python, our “nplike” backends are complicated by the fact that we have to use array-oriented functions, which is a larger API, and not exactly the same among NumPy, CuPy, JAX, etc. Each new backend needs a shim.



Composability: All buffers in a `AwkwardArray.Content` tree are **AbstractVector**, so it should be easy to swap in special features, like GPU-resident arrays, autodiff, units, etc.

In Python, our “nplike” backends are complicated by the fact that we have to use array-oriented functions, which is a larger API, and not exactly the same among NumPy, CuPy, JAX, etc. Each new backend needs a shim.

Unification: Since `push!` and `append!` are implemented on **AbstractVector**, the functionality of `LayoutBuilder` (append-only array) and `ak.Array` (read-only array) are unified in the same object.



Composability: All buffers in a `AwkwardArray.Content` tree are **AbstractVector**, so it should be easy to swap in special features, like GPU-resident arrays, autodiff, units, etc.

In Python, our “nplike” backends are complicated by the fact that we have to use array-oriented functions, which is a larger API, and not exactly the same among NumPy, CuPy, JAX, etc. Each new backend needs a shim.

Unification: Since `push!` and `append!` are implemented on **AbstractVector**, the functionality of `LayoutBuilder` (append-only array) and `ak.Array` (read-only array) are unified in the same object.

In Python, these need to be two different objects because `LayoutBuilder` is only useful in Numba/C++, where arrays are view-only.



Some examples of AwkwardArray.jl

```
using AwkwardArray
using AwkwardArray: Index64, ListOffsetArray, PrimitiveArray

array = ListOffsetArray{Index64, PrimitiveArray{Float64}}{ }
push!(array, [1.1, 2.2, 3.3])
push!(array, [4.4])
append!(array, [[5.5, 6.6], [7.7, 8.8, 9.9]])

total = 0.0
for list in array
    for item in list
        total += item
    end
end

vector::Vector{Vector{Float64}} = AwkwardArray.to_vector(array)
array2 = AwkwardArray.from_iter(vector)
```

Still needs to be connected to Python and “play well” with Julia



Open

8 of 14 tasks

Checklist for the first phase of development #5

jpivarski opened this issue on Aug 9 · 0 comments

The first phase of development (targeting [JuliaHEP 2023](#)) will require the following.

- Depth of Julia-side functionality: data model, `is_valid`, int-getindex, range-getindex, iteration, equality (data equivalence, not layout equivalence), `length` / `firstindex` / `lastindex`, LayoutBuilder-style appending.
- [PrimitiveArray still needs multidimensional support to be one-to-one with NumpyArray.](#) #6
- [Array nodes must support `parameters`, which implies a strict dependence on JSON.jl.](#) #8
- [ak.from_iter equivalent to convert from various Julia types into AwkwardArray.](#) #10
- String representation for the Awkward type. (No need for the `Type` objects we have in Python.)
- [String representation for data \(following `src/awkward/_prettyprint.py`\).](#) #23
- [All of the Awkward layout types.](#) #12
- [Actually implement the `ak.to_buffers/ak.from_buffers` equivalents on the Julia side.](#) #24 (No need for the Form objects we have in Python; just navigate the JSON, since it only happens once. This might need to be a macro to customize output types.)

Nice to have:

- [A Python module for round-tripping data between Python and Julia](#) #14
- [ak.to_arrow/ak.from_arrow](#) equivalents on the Julia side, for better interop with the Julia packages that produce and consume Arrow data. (We don't want to round trip through Python for that.)
- Conversions to and from common Julia formats, such as [ArraysOfArrays.jl](#) and [VectorOfArrays](#).
- [add `from_table` that uses `Tables.jl` interface](#) #39
- Performance testing, probably using the jagged0/1/2/3 suite (synthetic) and the RNTuple suite (realistic analysis).
- Composition testing: can I swap in arrays with units? on GPUs? delayed processing? I'm using the `firstindex` / `lastindex` protocol to be offsets-safe—am I making any assumptions that will break naive composition? (Or are the other libraries?)



Overview

Timetable

Video connection

Registration

Participant List

Local participation

Social program

Contact

✉ tamas.gal@fau.de

✉ jutta.schnabel@fau.de

An Awkward module for round-tripping data structures between Python and Julia



📅 Nov 8, 2023, 11:00 AM

Light talk - 10min

🕒 15m

📍 ECAP (Erlangen Centre for Astroparticle Physics)

Speaker

👤 [Ianna Osborne](#) (Princeton University)

Description

Both Julia and Python have a strong presence in the sciences. In a typical HEP data analysis process Python is more common, however, there is an obvious advantage to transitioning legacy software to Julia. We discuss the sharing of Awkward Array data structures between the two worlds to encourage the Python users to run their analysis both in an ecosystem of their choice and in Julia.

We discuss how the memory, the data buffer copies, and the dependencies are managed. We analyse the performance acceleration calling Julia from Python and vice versa for the intensive array-oriented calculations on a large scale, but not very large dimension arrays of HEP data.



Open question: how does package management work in a Python + Julia environment?



Open question: how does package management work in a Python + Julia environment?

Is there a way to control PyCall/PyJulia's cross-language dependencies with conda?



We need stronger connections between HEP analysis tools in Python and HEP analysis tools in Julia.

- ▶ StatsBase.Histogram/FHist.jl generalization that is interchangeable with scikit-hep/boost-histogram, scikit-hep/hist?
- ▶ LorentzVectors.jl or LorentzVectorHEP.jl: interop with scikit-hep/vector?
- ▶ Corpuscles.jl: share data with scikit-hep/particle?
- ▶ IMinuit.jl ✓
- ▶ zfit, pyhf, cabinetry, Coffea, etc.?

Encourage Python users to use Julia *with* their Python/C++ code!
(Otherwise, they won't use it at all.)