



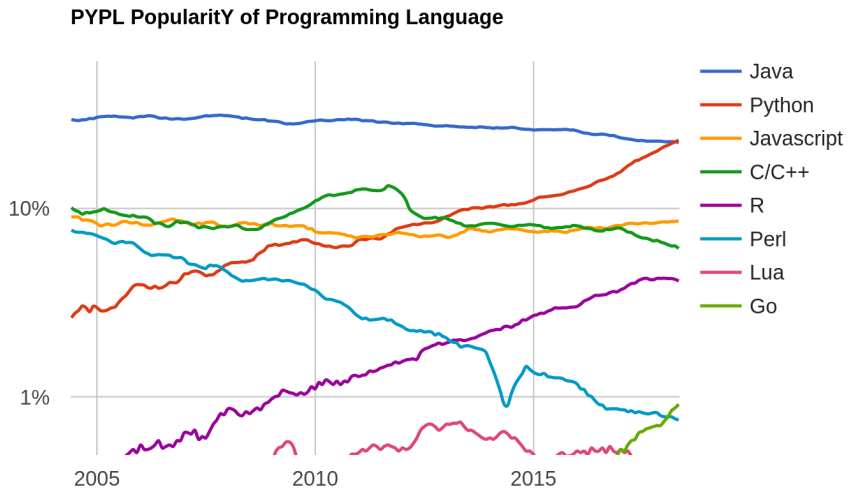
# HEP analysis in the Numpy ecosystem

Jim Pivarski

Princeton University – DIANA-HEP

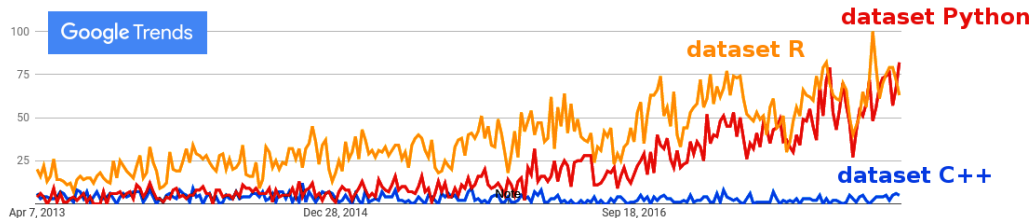
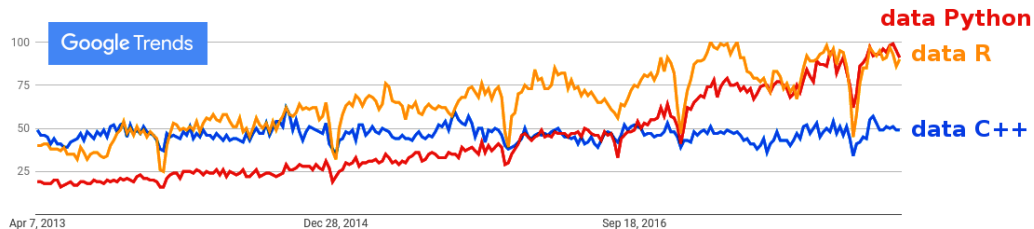
November 7, 2018

# In case we need a “Motivations” section

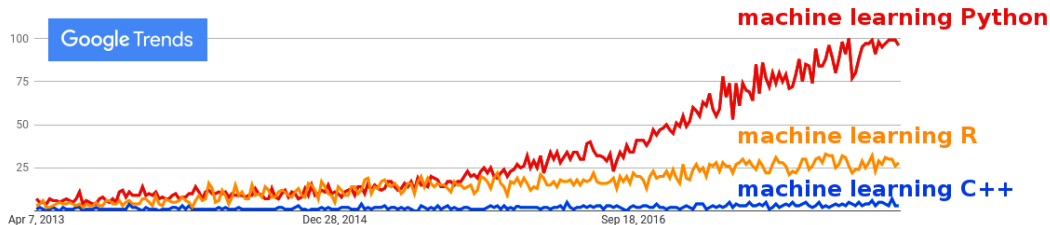
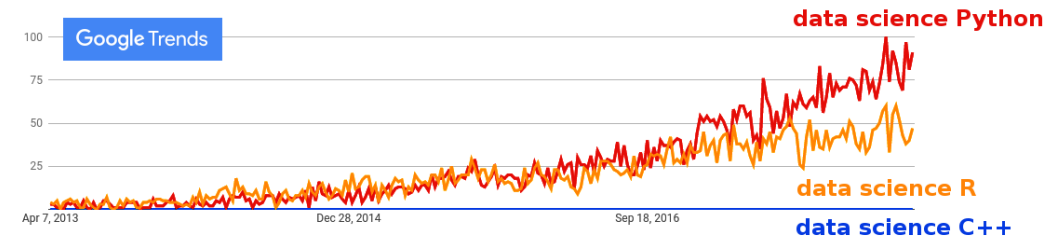


<http://pypl.github.io/PYPL.html>

# In case we need a “Motivations” section

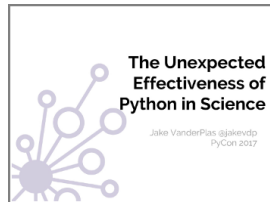


# In case we need a “Motivations” section



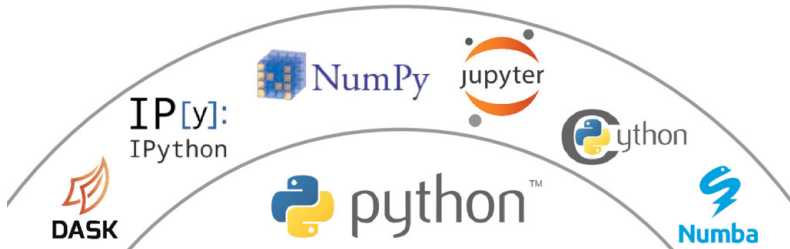
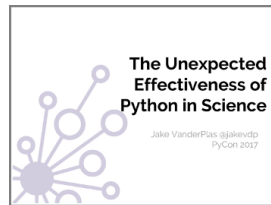


## Python's Scientific Stack



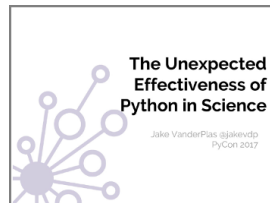
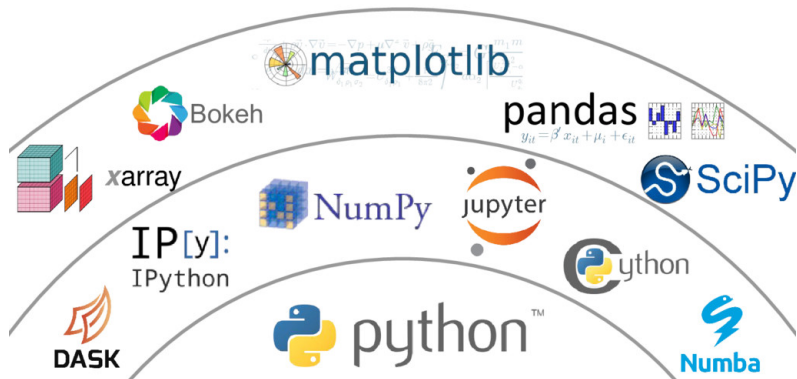


## Python's Scientific Stack



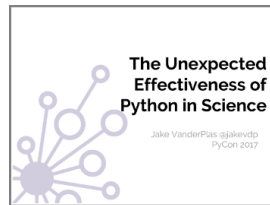
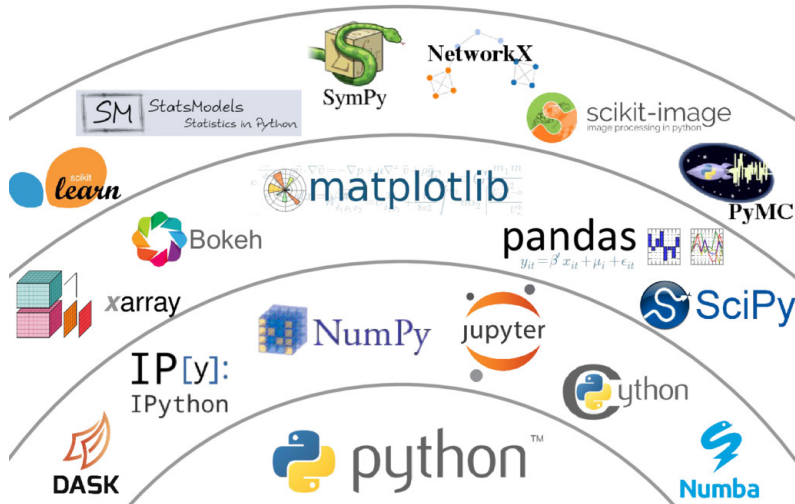


## Python's Scientific Stack



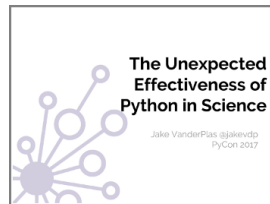
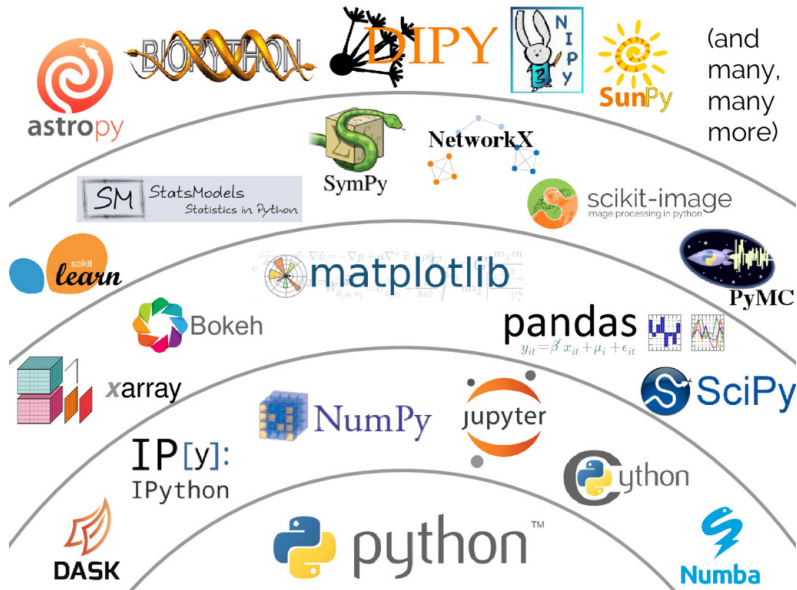


## Python's Scientific Stack

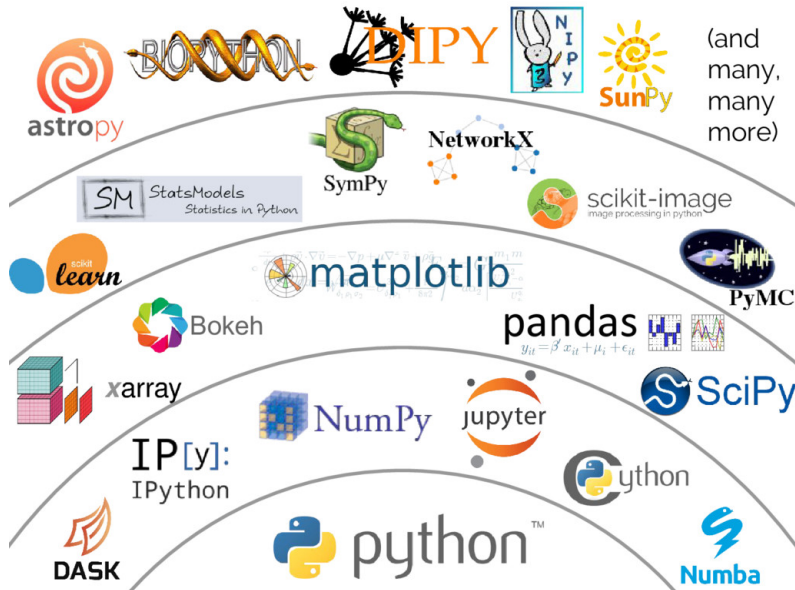




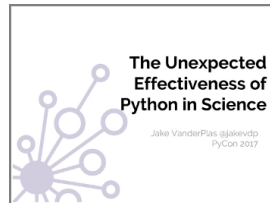
# Stealing from Jake VanderPlas's *Unexpected Effectiveness* talk



# Stealing from Jake VanderPlas's *Unexpected Effectiveness* talk

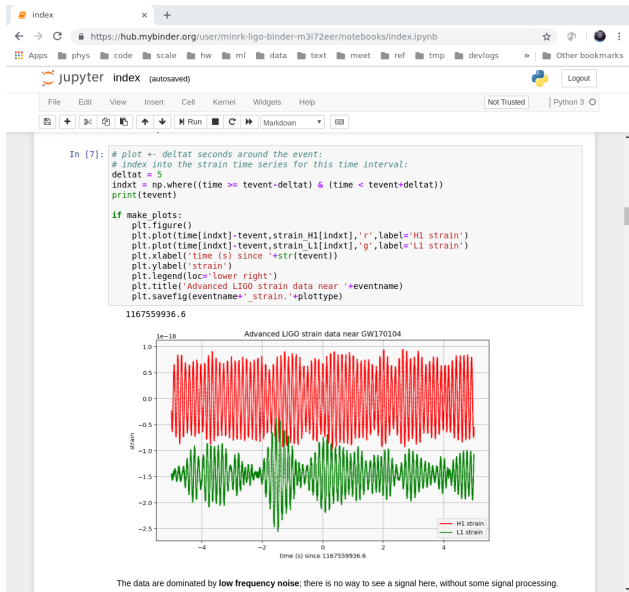


(and many, many more)



Searching for packages to do parts of a HEP analysis is eye-opening: you learn what is unique about what we do and what isn't.

# The other sciences are doing it, so why can't I?



- ▶ LIGO full analysis in Numpy + Jupyter + Binder → you can run all the signal processing yourself without downloading anything.
- ▶ LSST is adopting Python + Jupyter as the general astronomer's interface.
- ▶ XENON-nT is doing their full stack in Python + Numpy.
- ▶ (easy to find examples...)



Numpy allowed Python to fill a niche occupied by MATLAB and R: transforming, filtering, signal processing, and machine learning on GB's of rectangular array data.



Numpy allowed Python to fill a niche occupied by MATLAB and R: transforming, filtering, signal processing, and machine learning on GB's of rectangular array data.

That's not exactly HEP.

Numpy allowed Python to fill a niche occupied by MATLAB and R: transforming, filtering, signal processing, and machine learning on GB's of rectangular array data.

That's not exactly HEP.

**Python for loops are expressive but not fast.**

Good for small, complex processing that needs a fully nested object model.

**Numpy arrays are fast but not expressive.**

Good for biggish, more regular processing (including ML).

Astronomy, data science, etc. fit these cases better than HEP does.



1. HEP software stacks and conventions were developed *before* the Python/Numpy ecosystem and have to be retrofitted to communicate easily.
2. HEP analysis relies heavily on nested data: each event may have a different number of particles.
3. HEP analysis must be performed on TB of ntuples or PB of centrally produced data with thousands of users hitting the same system/files.

# 1. HEP software $\leftrightarrow$ Numpy





- ▶ The PyROOT project started in 2003, just before release 2.3 of **Python**. Every C++ class/function is accessible in Python, at a performance price.
- ▶ Many small HEP packages address specific access issues:
  - ▶ PyMinuit: extracted from my HEP analysis and open sourced in 2005
  - ▶ iminuit: modern version, used in astronomy
  - ▶ rootpy: more “Pythonic” interface overlaid on PyROOT
  - ▶ root\_numpy: faster bindings compiled into C++
  - ▶ **81** packages matching “HEP” in PyPI; **247** with “HEP” and Python in GitHub...
- ▶ The ROOT team is actively adding “Pythonizations” to PyROOT for a more Pythonic interface and also direct-to-Numpy access.
  - ▶ `ttree.AsMatrix()`: access TTree data in Python as Numpy
  - ▶ RDataFrame sources and sinks from and to Numpy arrays and Arrow



**Scikit-HEP:** an attempt to build community around a suite of interoperating Pythonic HEP libraries, like PyData or AstroPy.

**uproot:** reader and writer of the ROOT file format in Python.

**awkward-array:** extensions of array programming idioms to non-rectangular and deeply nested data.

**iminuit and proffit:** Pythonic interface to Minuit and likelihood builder that works with iminuit.

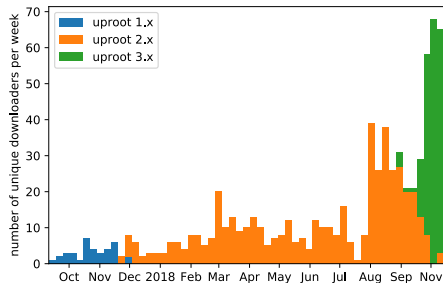
**numpythia and pyjet:** interfaces to Pythia and FastJet.

**decaylanguage, formulate, root\_numpy, vegascope...** packages with focused functionality, broad interfaces.

Originally, it was a two-week project to access ROOT data in a columnar form more quickly, intended as a backend for a future query service.

Then I got a lot of feedback from physicists trying to get their data into machine learning libraries.

So I pivoted, generalized it (2.x), and presented it as an end-user product.



Originally, it was a two-week project to access ROOT data in a columnar form more quickly, intended as a backend for a future query service.

Then I got a lot of feedback from physicists trying to get their data into machine learning libraries.

So I pivoted, generalized it (2.x), and presented it as an end-user product.

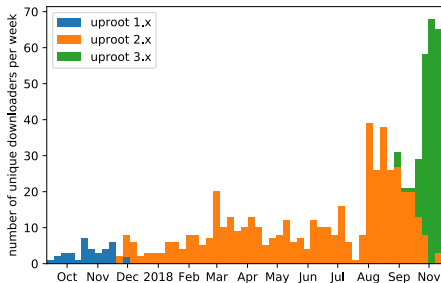
A lot of feedback is from users who are trying to do analysis without ROOT:

“Great package by the way, the ROOT-independency makes data analysis a lot more flexible.”

“Now I’ll go through the jagged array tutorial and after that I’ll write a wrapper library to make our data accessible without ROOT and any other huge, uncomfortable dependencies, halleluja!”

“My particular love of this package is driven by moving away from root as early in my processing as possible and enabling me to use tools I am more comfortable with. I’m not a HEP guy but a space physics guy using geant for instrument responses.”

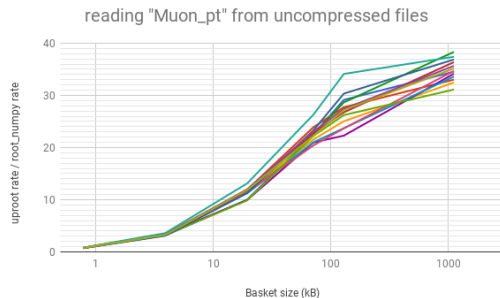
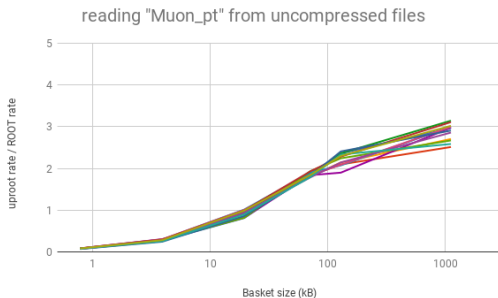
“Thank you for all of your work with this, being able to load root files without ROOT is wonderful!”

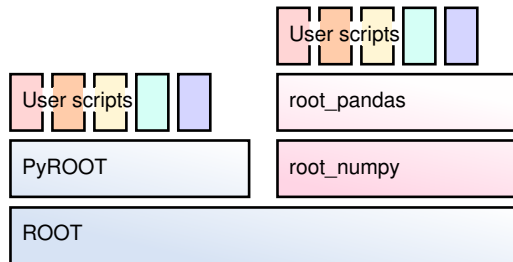
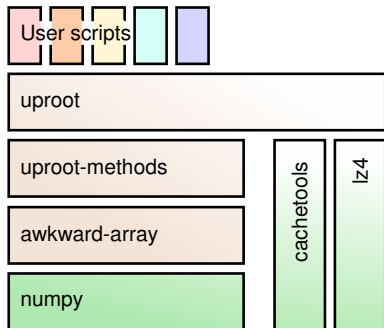


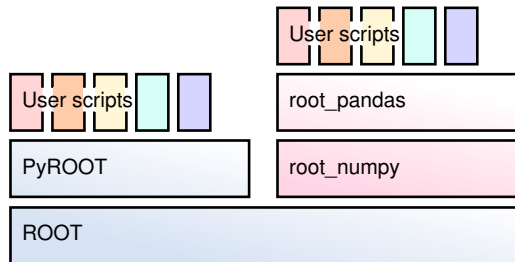
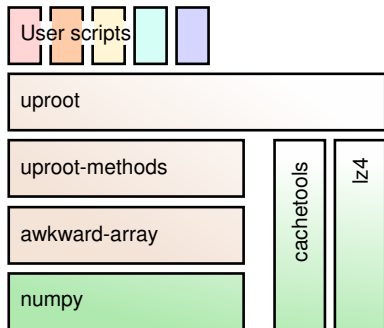
Based on the observation that:

- ▶ a Numpy analysis wants data in columnar arrays, and
- ▶ data in a ROOT file are already arranged as columns: find them and cast them!

Transforming columnar bytes on disk into event records back into columnar arrays is undesirable extra work. Dropping that step makes up for seeking to the positions of the columnar arrays using slow Python, *if the arrays (baskets) are large*.







uproot-methods defines behaviors for objects extracted from ROOT (e.g. methods for histograms and Lorentz vectors). C++ methods aren't defined in the file!

It's a separate library so that it can evolve on a different schedule and accept more user-contributed pull requests.



```
>>> import uproot
>>> tree = uproot.open("HZZ-objects.root")["events"]    # dict-like
>>> array = tree.array("muonp4")

>>> array                                                # all events
<JaggedArray [[TLorentzVector(-52.899, -11.655, -8.1608, 54.779)
               TLorentzVector(37.738, 0.69347, -11.308, 39.402)] ...]>
>>> array[0][1]                                         # second particle in first event
TLorentzVector(37.738, 0.69347, -11.308, 39.402)

>>> hastwo = (array.counts >= 2)                        # events with at least two muons
>>> leading = array[hastwo, 0]                          # mask and select first
>>> subleading = array[hastwo, 1]                      # mask and select second

>>> candidates = leading + subleading                  # Lorentz vector sum across all
>>> candidates.mass                                     # compute mass for all
array([90.22779777, 74.74654928, ..., 85.44384208, 75.96066262])
```

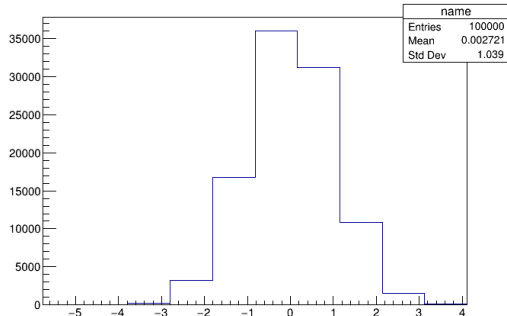


uproot version 3 has write support, but (currently) only for histograms.

```
>>> import uproot
>>> import numpy
>>> file = uproot.recreate("tmp.root")
>>> # give a Numpy histogram a name and save it using file as a dict
>>> file["name"] = numpy.histogram(numpy.random.normal(0, 1, 100000))
```

Read it back in ROOT:

```
>>> import ROOT
>>> file = ROOT.TFile("tmp.root")
>>> hist = file.Get("name")
>>> hist.Draw()
```



uproot does its one thing well: just I/O. No major updates forseen for *reading*, only bug-fixes and handling corner-case files that users send to me.

For *writing*, however, I plan to continue to work with Pratyush Das (who added histogram-writing as a DIANA-HEP fellow) to add support for writing simple TTrees on a 6–12 month timescale.



uproot does its one thing well: just I/O. No major updates forseen for *reading*, only bug-fixes and handling corner-case files that users send to me.

For *writing*, however, I plan to continue to work with Pratyush Das (who added histogram-writing as a DIANA-HEP fellow) to add support for writing simple TTrees on a 6–12 month timescale.



Small, interacting tools need common protocols. ROOT defines file and C++ protocols; the Scikit-HEP developers are designing in-memory and Pythonic protocols for fit distributions (following SciPy conventions), cost function builders, pseudoexperiments, and histograms (I'm testing Flatbuffers as a wire protocol).

## 2. HEP analysis and nested data structures

# HEP datasets are awkward



Most non-HEP data analysis tools (including ML) require rectangular data: a series of equal-sized records. HEP required nested data from the beginning (HYDRA, ZBOOK).

The nested  $\rightarrow$  tabular transition is lossy! It must be done as late as possible.

muons

p <sub>T</sub>	phi	eta
31.1	-0.481	0.882

p <sub>T</sub>	phi	eta
9.76	-1.24	0.924

p <sub>T</sub>	phi	eta
8.18	-0.119	0.923

mu1 p <sub>T</sub>	mu1 phi	mu1 eta	mu2 p <sub>T</sub>	mu2 phi	mu2 eta
31.1	-0.481	0.882	9.76	-0.124	0.924
5.27	1.246	-0.991	n/a	n/a	n/a
4.72	-0.207	0.953	n/a	n/a	n/a
8.59	-1.754	-0.264	8.714	0.185	0.629



uproot needed a minimally structured array just to represent data: jagged arrays.

uproot needed a minimally structured array just to represent data: jagged arrays.

Data must appear to be nested lists of variable-length lists but be managed efficiently by arrays. There are several ways to do this.

Logical structure:	[ [0, 1, 2], [], [3, 4], [5, 6, 7, 8], [] ]
Content:	[ 0, 1, 2, 3, 4, 5, 6, 7, 8 ]
Offsets:	[ 0, 3, 3, 5, 10, 10 ]
Parents:	[ 0, 0, 0, 2, 2, 3, 3, 3, 3 ]

uproot needed a minimally structured array just to represent data: jagged arrays.

Data must appear to be nested lists of variable-length lists but be managed efficiently by arrays. There are several ways to do this.

Logical structure:	[ [0, 1, 2], [], [3, 4], [5, 6, 7, 8], [] ]
Content:	[ 0, 1, 2, 3, 4, 5, 6, 7, 8 ]
Offsets:	[ 0, 3, 3, 5, 10, 10 ]
Parents:	[ 0, 0, 0, 2, 2, 3, 3, 3, 3 ]

Content + offsets is lossless, compact, and efficient for randomly accessing data.

Content + parents + total length is lossless, and it helps in reduction operations.



uproot needed a minimally structured array just to represent data: jagged arrays.

Data must appear to be nested lists of variable-length lists but be managed efficiently by arrays. There are several ways to do this.

Logical structure:	[ [0, 1, 2], [], [3, 4], [5, 6, 7, 8], [] ]
Content:	[ 0, 1, 2, 3, 4, 5, 6, 7, 8 ]
Offsets:	[ 0, 3, 3, 5, 10, 10 ]
Parents:	[ 0, 0, 0, 2, 2, 3, 3, 3, 3 ]

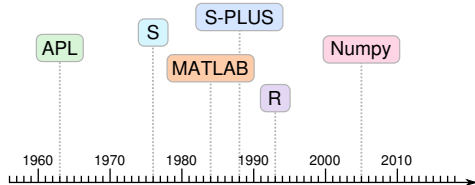
Content + offsets is lossless, compact, and efficient for randomly accessing data.

Content + parents + total length is lossless, and it helps in reduction operations.

Jagged arrays and other primitives can be composed to make any data structure: the jagged array of `TLorentzVector` from the uproot example was built out of jagged tables and mix-in methods.



Array programming expresses regular operations over rectangular data structures in a shorthand that implicitly loops over elements (i.e. hides the event loop).



- ▶ Multidimensional slices: `rgb_pixels[0, 50:100, ::3]`
- ▶ Elementwise operations: `all_pz = all_pt * sinh(all_eta)`
- ▶ Broadcasting: `all_phi - 2*pi`
- ▶ Masking (list compaction): `data[trigger & (pt > 40)]`
- ▶ Fancy indexing (gather/scatter): `all_eta[argsort(all_pt)]`
- ▶ Row/column commutativity (hides AoS  $\leftrightarrow$  SoA):  
`table["column"][7]` (row 7 of column array)  
`table[7]["column"]` (field of row tuple 7)
- ▶ Array reduction: `array.sum()  $\rightarrow$  scalar`



- ▶ Multidimensional slices: `events["jets"][:, 0] → first jet per event`
- ▶ Elementwise operations: `jetpt * sinh(jeteta) → keep jagged structure`
- ▶ Broadcasting: `jetphi - metphi → expand metphi from one-per-event to one-per-jet before operation`
- ▶ Masking (list compaction):  
`data[trigger] → drop whole events`  
`data[jetpt > 40] → drop jets from events`
- ▶ Fancy indexing (gather/scatter):  
`a = argmax(jetpt) → [[2], [], [1], [4]]`  
`jeteta[a] → [[3.6], [], [-1.2], [0.4]]`
- ▶ Row/column commutativity (project jagged tables to jagged arrays before indexing):  
`events["jets"]["pt"][7, 1] (all the same)`  
`events["jets"][7]["pt"][1]`  
`events[7]["jets"]["pt"][1]`  
`events["jets"][7, 1]["pt"]`  
`events[7]["jets"][1]["pt"]`
- ▶ Jagged array reduction: `jetpt.max() → array of max jet  $p_T$  per event`

# Examples from a recent tutorial



```
>>> import awkward
>>> a = awkward.JaggedArray.fromiter([[ 1,    2,    3], [], [ 4,    5]])
>>> b = awkward.JaggedArray.fromiter([[ 10,   20,   30], [], [ 40,   50]])
>>> m = awkward.JaggedArray.fromiter([[True, False, True], [], [False, True]])
>>> flat  = numpy.array([ 100, 200, 300])
>>> mflat = numpy.array([False, True, True])
>>> i = awkward.JaggedArray.fromiter([[2, 1], [], [1, 1, 0, 1]])

>>> a + b                                     # Elementwise operations
<JaggedArray [[11 22 33] [] [44 55]] at 7b229f329908>

>>> a + flat                                  # Broadcasting
<JaggedArray [[101 102 103] [] [304 305]] at 7b229f7105c0>

>>> a[m]                                      # Masking by jagged (selects particles)
<JaggedArray [[1 3] [] [5]] at 7b229f3290f0>

>>> a[mflat]                                 # Masking by flat (selects events)
<JaggedArray [[] [4 5]] at 7b229f3295c0>

>>> a[i]                                      # Fancy indexing
<JaggedArray [[3 2] [] [5 5 4 5]] at 7b229f3299b0>
```



## Procedural HEP: selecting the “best” leptoquark candidate

```
leptoquarks = []
for event in dataset:
    best = None
    for jet in event.jets:
        for lepton in event.leptons:
            if cut(jet, lepton):
                leptoquark = jet + lepton          # Lorentz vector + operator
                if best is None or quality(leptoquark) > quality(best):
                    best = leptoquark
if best is not None:
    leptoquarks.append(best)
```

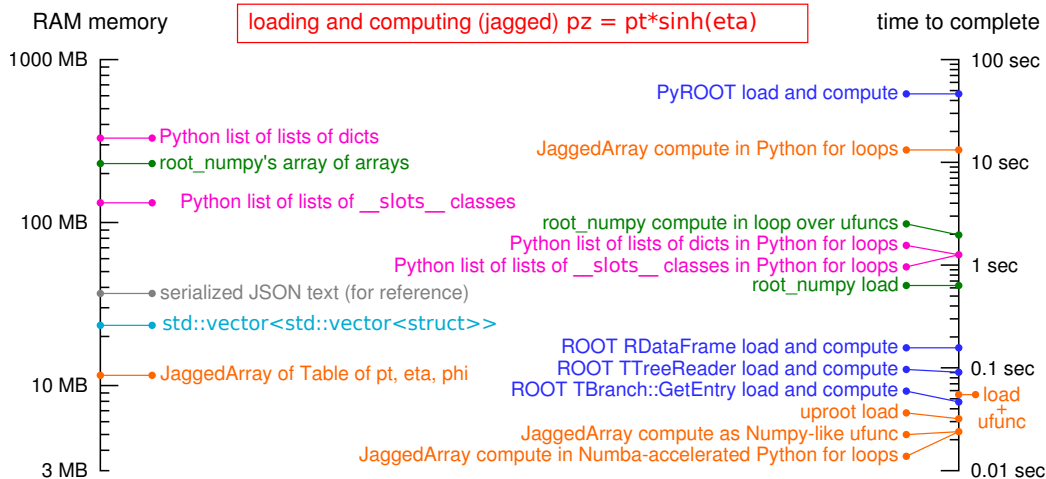
## Jagged array solution:

```
pairs = events["jets"].cross(events["leptons"]) # cross-join within events
goodpairs = pairs[cut(pairs["0"], pairs["1"])] # select from pairs["0"] and pairs["1"]
candidates = goodpairs["0"] + goodpairs["1"] # Lorentz vector + operator across all
best = candidates[quality(candidates).argmax()] # get objects with the best indexes
leptoquarks = best.flatten() # reduce to flat array
```

# Array programming has competitive performance



It isn't just a crutch for slow Python. Efficiency is from SIMD-like data organization.



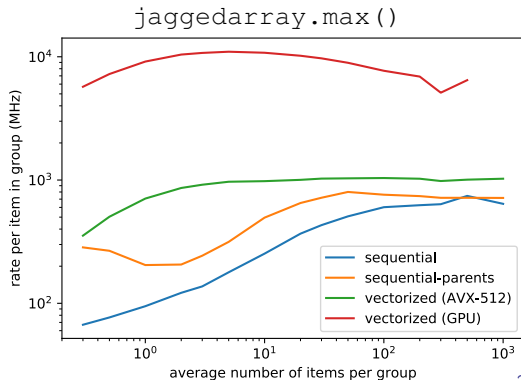
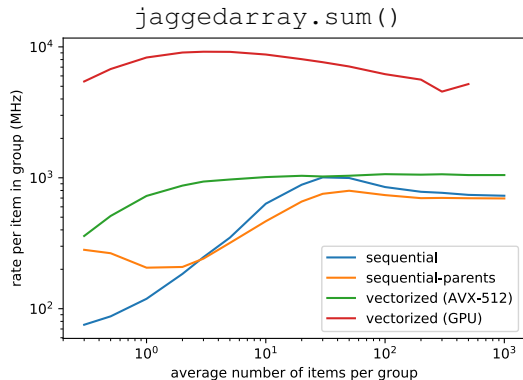
# Once expressed this way, array operations can be highly optimized



Jaydeep Nandi (GSoC student) developed vectorized implementations of jagged array primitives.



Below, vectorized-CPU (green) and GPU (red) outperform conventional for-loop versions of `jaggedarray.sum()` and `jaggedarray.max()`.



# awkward-array: suite of special arrays to build any data structure



class	purpose	type	class	purpose	type
JaggedArray	representing an array of different-length subarrays, indexed as elements	$[0, \infty) \rightarrow T$	IndexedArray	indexes elements of an array as a pointer by index position	$T$
ByteJagged-Array	representing an array of different-length subarrays, indexed as bytes	$[0, \text{inf}) \rightarrow T$	ByteIndexed-Array	indexes elements of an array as a pointer by byte position	$T$
Table	representing a table of different-typed columns; <b>mutable: columns can be added or removed</b>	$\text{Table}(T \dots)$	SparseArray	represents an array storing only elements whose values is not a default value (such as zero); in a sense, the opposite of IndexedArray	$T$
UnionArray	tagging and indexing elements in arrays to simulate a heterogeneous array	$\text{Union}(T \dots)$	ChunkedArray	logically concatenates discontinuous chunks into one big array; chunk sizes might be known or unknown; appendable	$T$
MaskedArray	byte-masking elements of an array as N/A	$\text{Option}(T)$	Appendable-Array	allocates array chunks to append or extend a Numpy array; <b>mutable: number of rows can grow; restricted: Numpy content only</b>	$T$
BitMasked-Array	bit-masking elements of an array as N/A	$\text{Option}(T)$			
IndexedMasked-Array	indexes elements of an array as N/A or as a pointer (content is sparse)	$\text{Option}(T)$			
ObjectArray	generates objects in an array upon access	opaque	VirtualArray	generates an array upon first access, then caches	$T$



### 3. Scaling up to PB and thousands of users



In one way, we don't need anything new: Pythonic array operations can run in a batch job like anything else.



In one way, we don't need anything new: Pythonic array operations can run in a batch job like anything else.

However, the implicit loop presents an opportunity for transparent multiprocessing: `candidates.mass` may be computed for a single particle candidate, an array of millions or a jagged array of billions.

- ▶ **Dask:** accumulates delayed operations and runs them across all cores or across a cluster of “thousands.” LSST is considering it for astronomer’s jobs.
- ▶ **Joblib:** less fine-grained jobs, dispatching Python functions, not expressions.
- ▶ **Parsl:** also less fine-grained, integrates with schedulers (Condor, Slurm, GRID); academic project from Chicago, Argonne, and Illinois.
- ▶ **PySpark:** if a high-speed bridge from Java to Python (via Arrow) is added.


I'm still hoping we can use off-the-shelf parts; there's a lot still to test.

Much of my upcoming work will involve integrating awkward-array with industry tools, to make these tools usable for general HEP data:

- ▶ **Dask** for distributed processing,
- ▶ **Pandas** for statistics and plotting,
- ▶ **Numba** for JIT-compilation,
- ▶ **blosc/bcolz** for transparent compression,
- ▶ **CuPy** for GPU processing. . .


Much of my upcoming work will involve integrating awkward-array with industry tools, to make these tools usable for general HEP data:

- ▶ **Dask** for distributed processing,
- ▶ **Pandas** for statistics and plotting,
- ▶ **Numba** for JIT-compilation,
- ▶ **blosc/bcolz** for transparent compression,
- ▶ **CuPy** for GPU processing. . .

I didn't mention  in this talk because I'm rethinking how histogramming ought to be done, such as a protocol to enable projects with different strengths to share histogram objects in memory. This gets generally to common protocols and code reuse, which many of us in Scikit-HEP are considering together.

Much of my upcoming work will involve integrating awkward-array with industry tools, to make these tools usable for general HEP data:

- ▶ **Dask** for distributed processing,
- ▶ **Pandas** for statistics and plotting,
- ▶ **Numba** for JIT-compilation,
- ▶ **blosc/bcolz** for transparent compression,
- ▶ **CuPy** for GPU processing. . .

I didn't mention  in this talk because I'm rethinking how histogramming ought to be done, such as a protocol to enable projects with different strengths to share histogram objects in memory. This gets generally to common protocols and code reuse, which many of us in Scikit-HEP are considering together.

My eventual goal is the query system I've described elsewhere; I see these as prerequisite pieces that are usable even before analysis-as-a-service is a reality.