Analysis 2.0:
New approaches to particle physics analysis

Ben Krikler
Lund University
8th March 2020

b.krikler@cern.ch
@benkrikler

A post-doc looking for Dark Matter at:

University of BRISTOL

CMS

LZ

A fellow of:

Software Sustainability Institute

A founder of:

FAST HEP

A convener of:

PyHEP

Goals

1. **Showcase recent developments** for analysis of last couple of years
2. **Throw ideas out there** that might give you a lead for things to look at

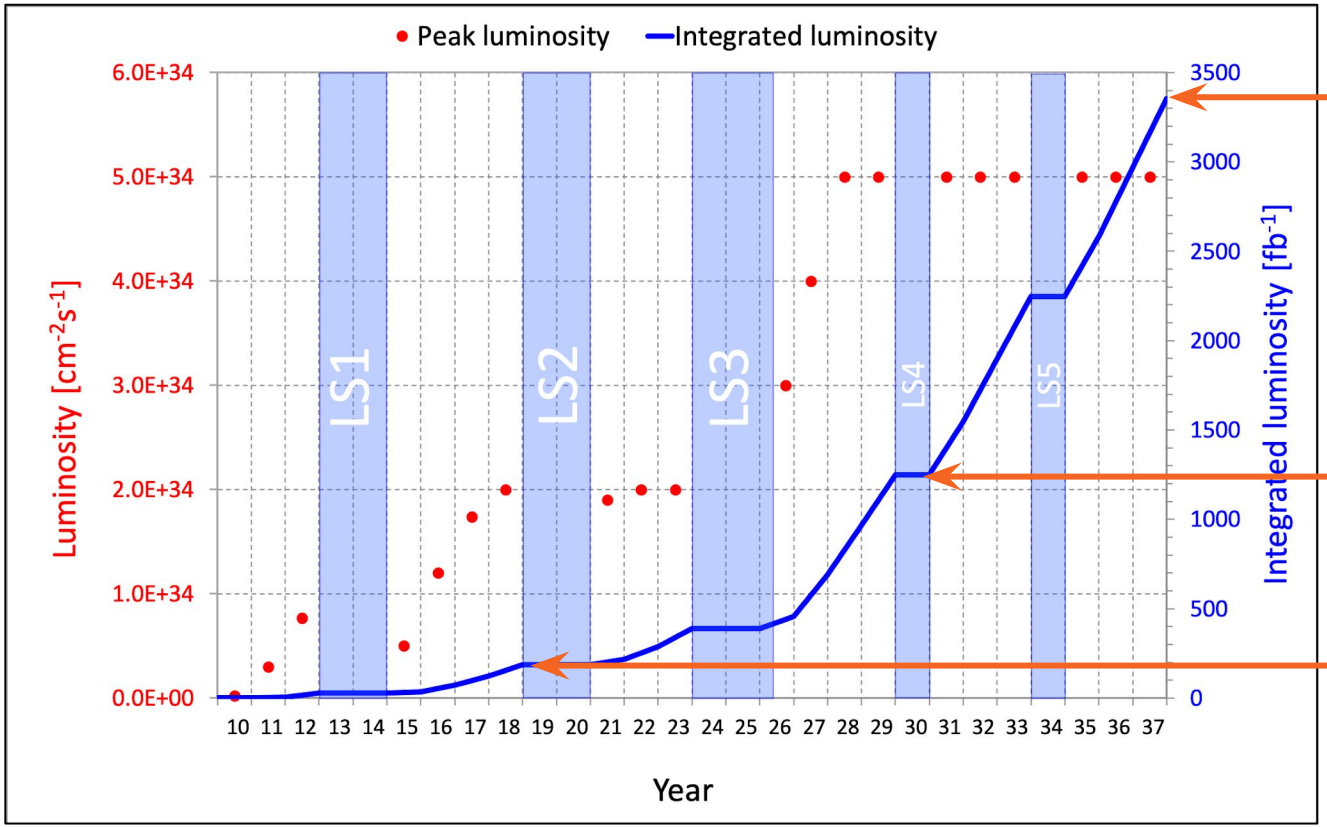
Outline

1. **Challenges facing particle physics**
2. **Python as a solution**
3. **Columnar Analysis**
4. **The FAST-HEP project**

- High-level analysis = very final stages of processing
- This is a very broad topic: need a whole conference
- Some personal opinions: I welcome any counter-opinions!

Three challenges facing our field

Future data volumes: HL-LHC



+18 years: x22

+11 years: x8

Now ~150 fb^{-1}

https://lhc-commissioning.web.cern.ch/lhc-commissioning/schedule/images/optimistic-nominal-19.png

Square Kilometer Array

Minh Huynh, CHEP 2019
The Square Kilometre Array Computing

SDP headline design numbers*

Input
- ~800 GByte/s INGEST (in total), from Central Signal Processor

Temporarily store
- Data set up to 15 PBytes
- 100PBytes total distributed buffer

Process
- 250 PFLOPS total peak
  - 10% efficiency assumed

Preserve and ship
- (up to) 2 PetaByte per day of Science Data Products

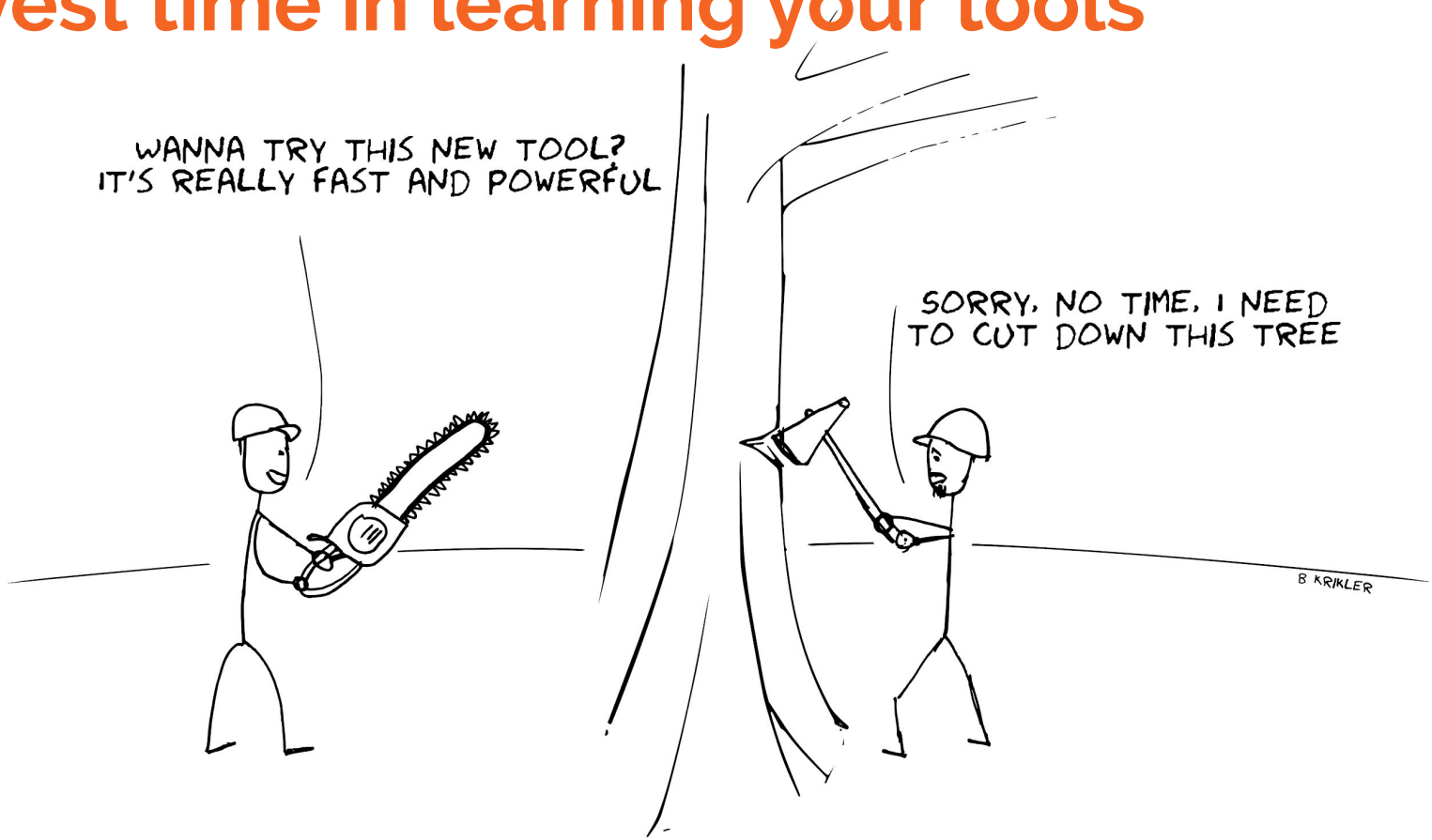
> 600 PB per year for around 50 years
⇒ 30 Exabytes of data
⇒ "Exascale computing"

*all numbers subject to change (Totals for both SKA-Low and SKA-Mid SDP)

Challenge #1

Our data will grow massively unlike our resources

Invest time in learning your tools



A hypothesis:

Time to **learn** $\propto$ Time to **code** $\propto$ Time to **insight**

Challenge #1

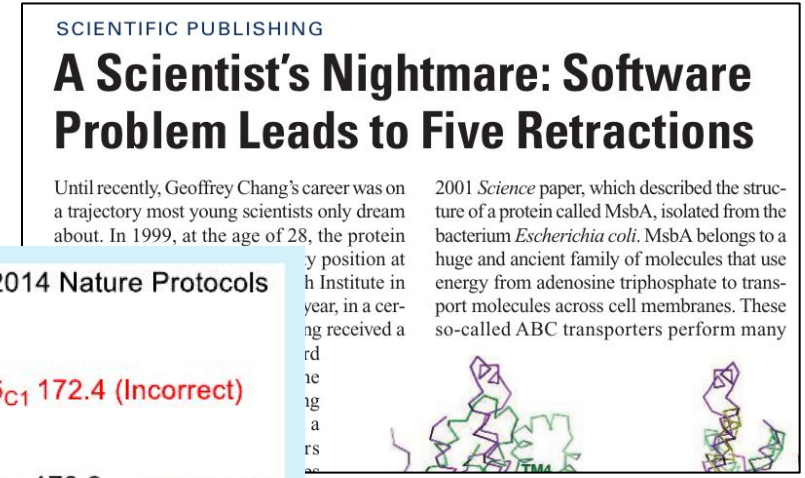
Our data will grow massively unlike our resources

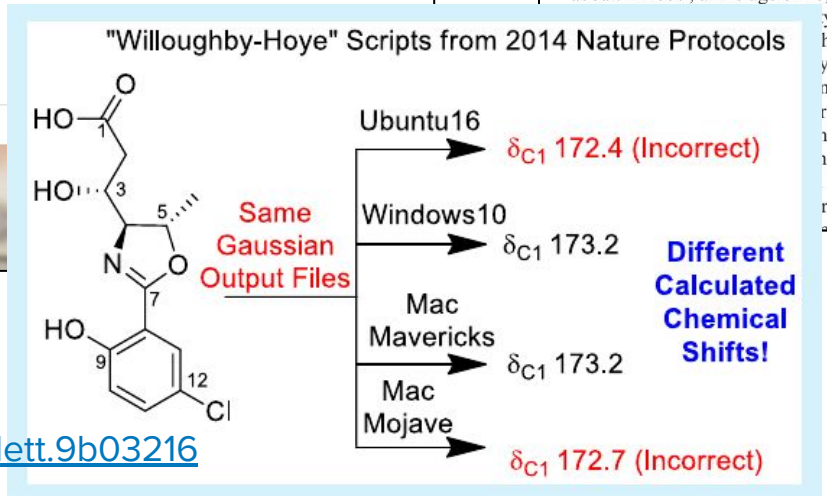
Challenge #2

Scientists first, developers second: code is slow to write & run and often error-prone

Bugs and reproducibility

Dec. 2006 [DOI: 10.1126/science.314.5807.1856](https://doi.org/10.1126/science.314.5807.1856)





Oct. 2019
[DOI:10.1021/acs.orglett.9b03216](https://doi.org/10.1021/acs.orglett.9b03216)

Challenge #1

Our data will grow massively unlike our resources

Challenge #2

Scientists first, developers second: code is slow to write & run and often error-prone

Challenge #3

A paper is not enough to describe an analysis in a reproducible way


Challenge #1
Challenge #2
Challenge #3
Our data
grow ma
unl
reso
paper is not
ough to
cribe an
ysis in a
producible
way
Rethink our approach

Solution #1

Too much data:

What does “Big data” do?
Use resources more intelligently

Solution #2

Good code is tough:

Adopt easier languages and open source practices

Solution #3

Irreproducibility:

Reduce gap between paper and actual analysis code

Python for Particle Physics

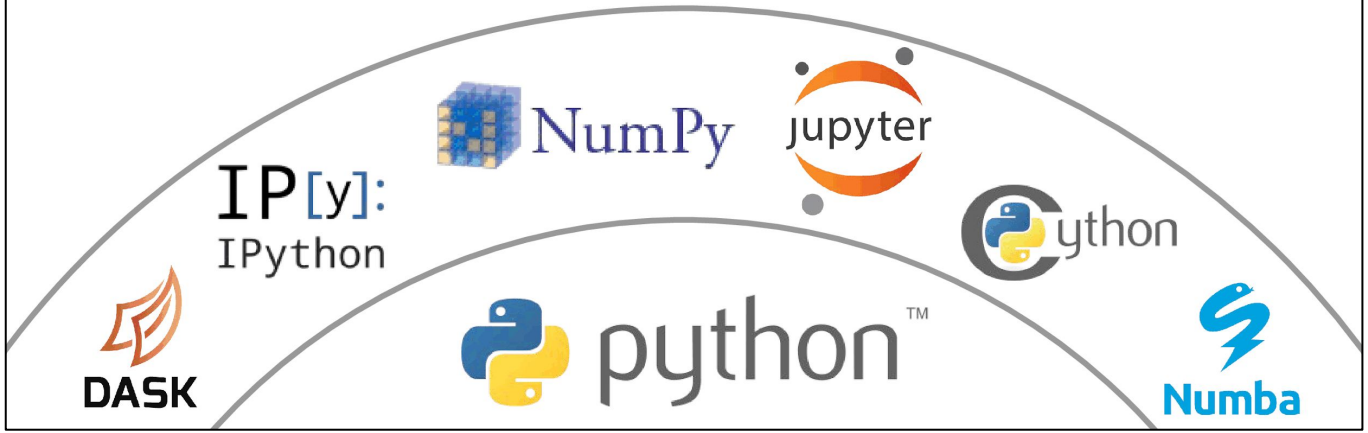
Why Python for scientific research?

Adapted from Jake Vander Plas' The unexpected effectiveness of Python in Scientific Research

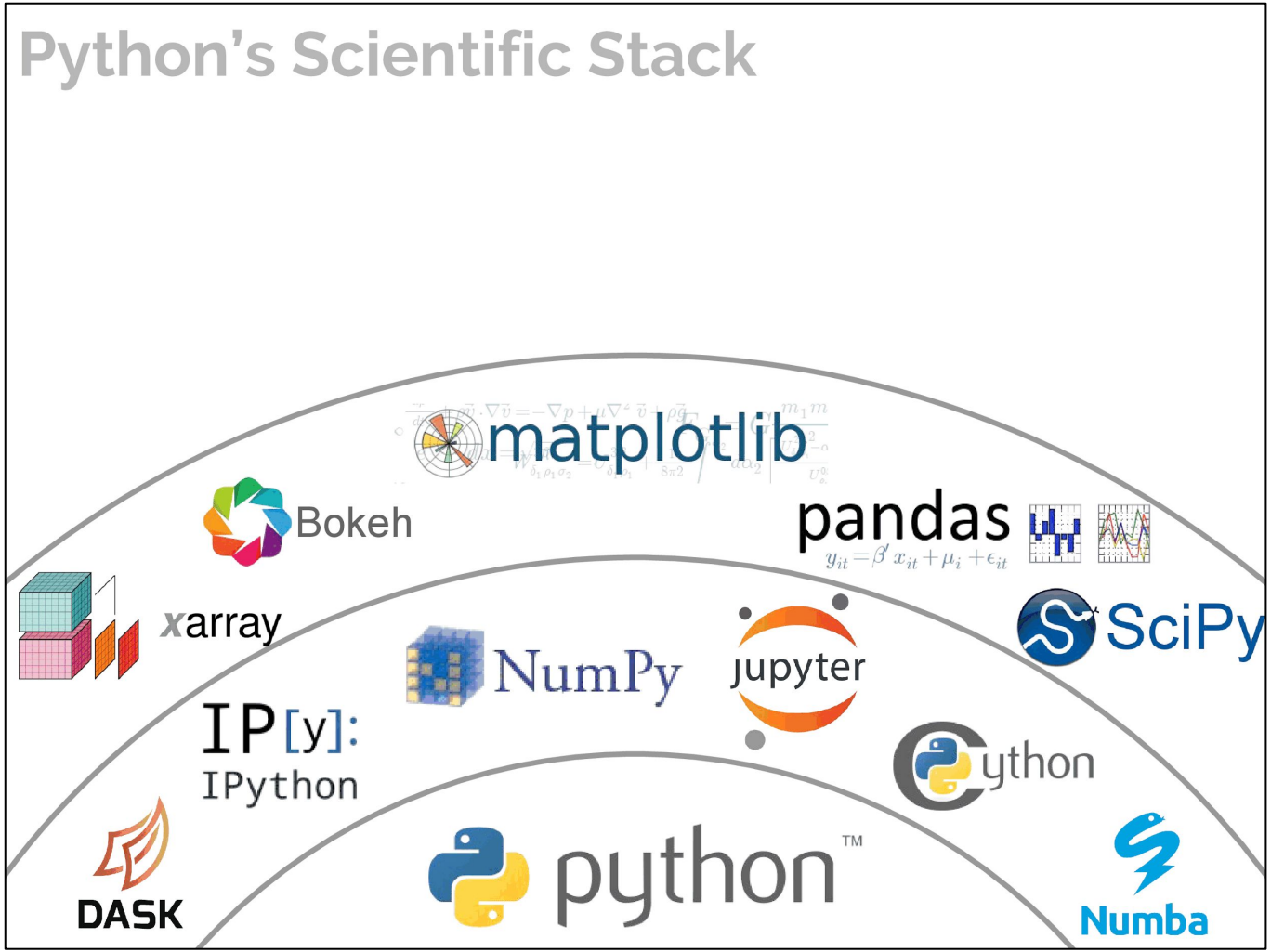
- Interoperability with other languages
  - Bindings to C++, fortran, etc
  - We can continue using existing tools (if wanted)
- Perfect for exploratory work
  - Free, open-source and no required IDE
  - No compiling
  - Little boilerplate code
  - E.g. Jupyter notebooks (though this is no longer python-only)
- Package ecosystem
  - "Batteries included" so standard library provides many functions: argparse, globbing, regular expressions, URL requests, math
  - Package manager gives access to huge community-driven ecosystem
  - "Open-source" by default

Jake VanderPlas: PyCon 2017

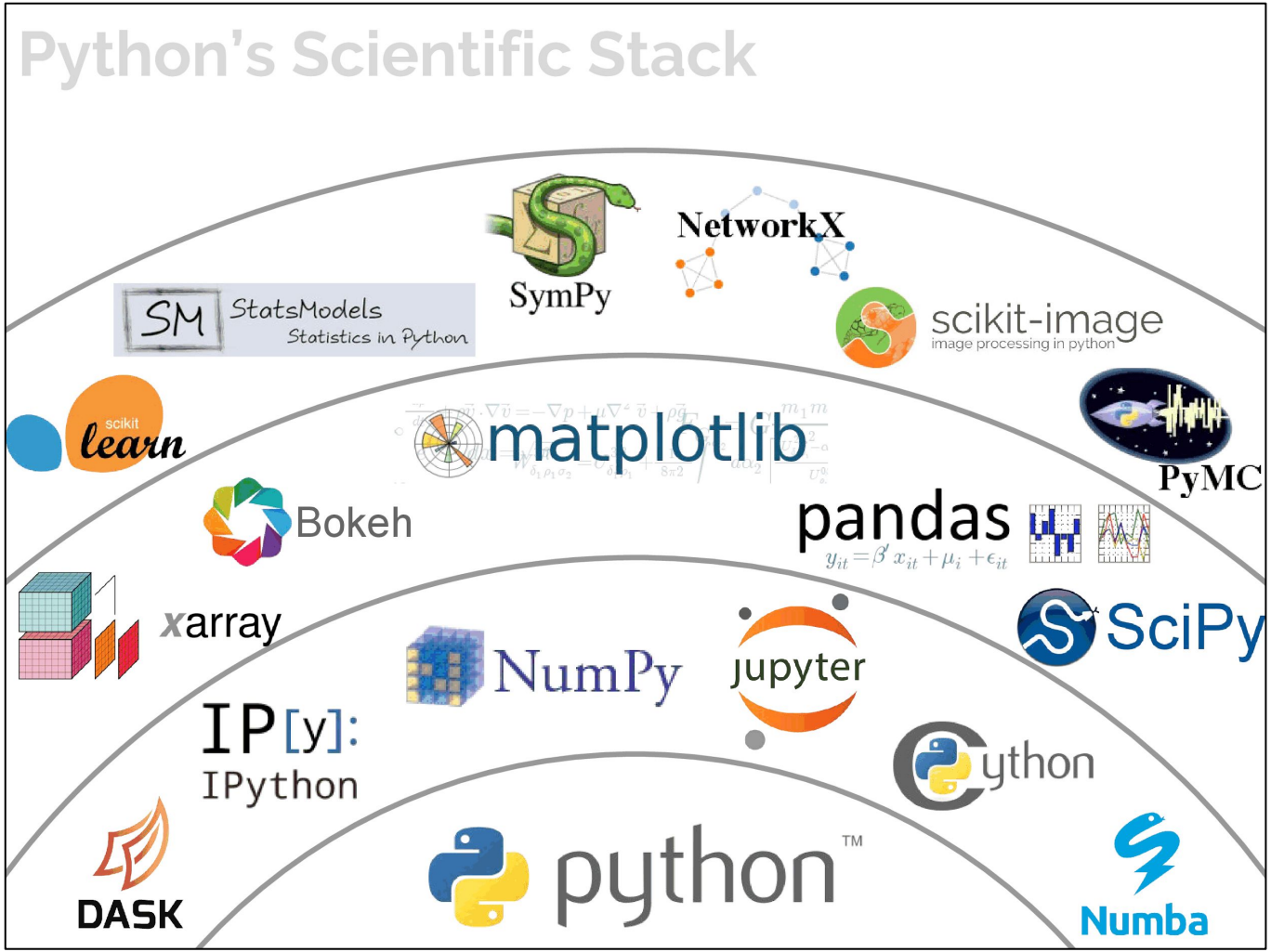
Python's Scientific Stack



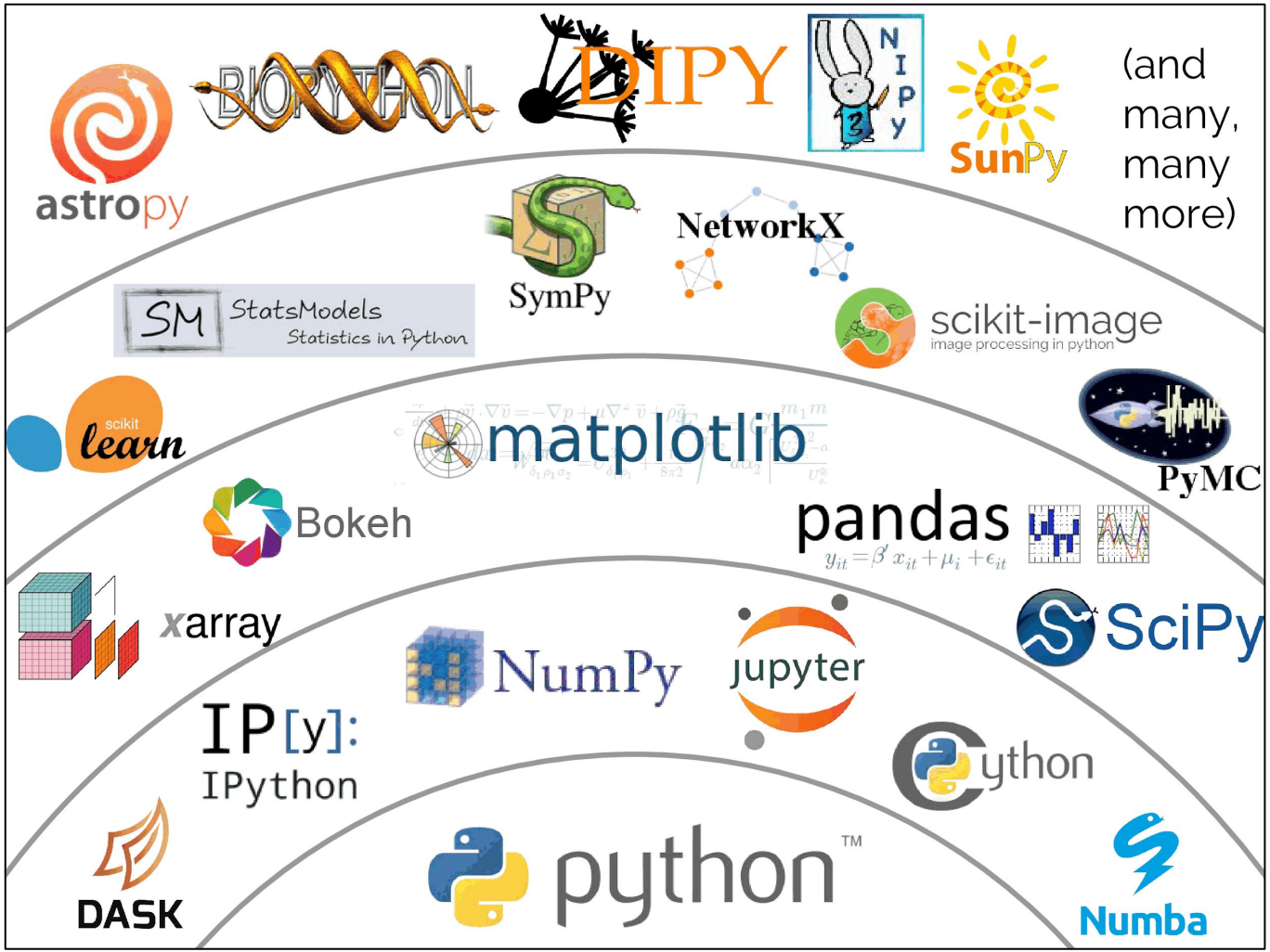
Jake VanderPlas: PyCon 2017



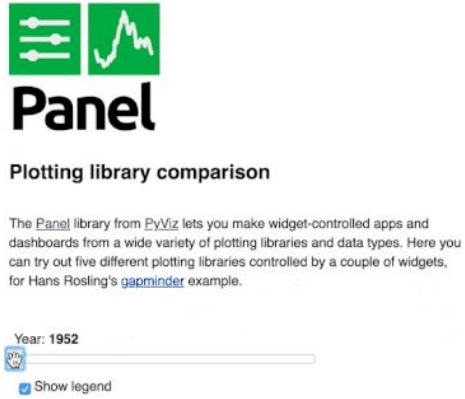
Jake VanderPlas: PyCon 2017

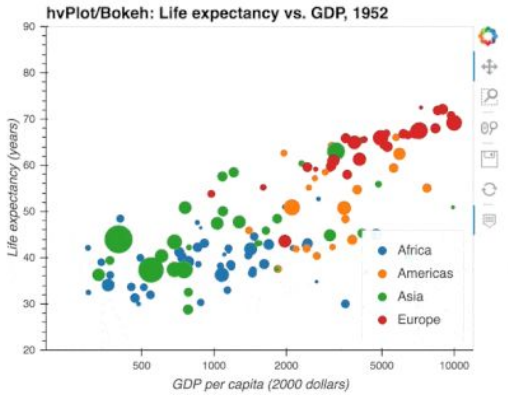


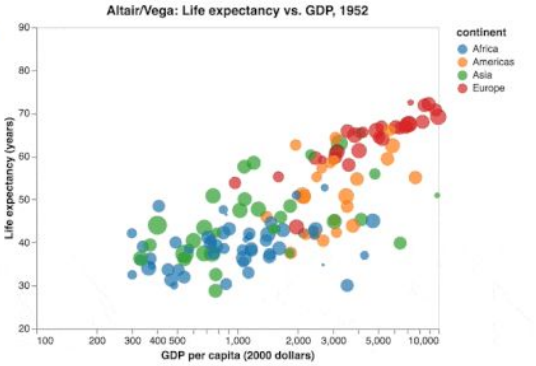
Jake VanderPlas: PyCon 2017

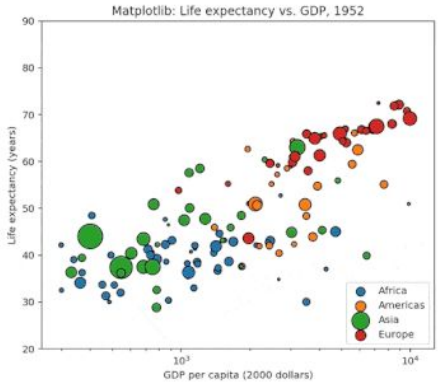


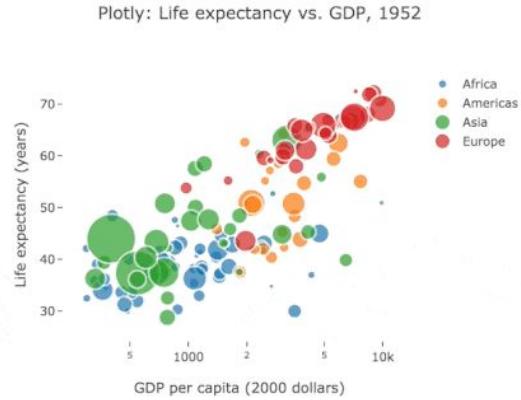
Panel and PyViz







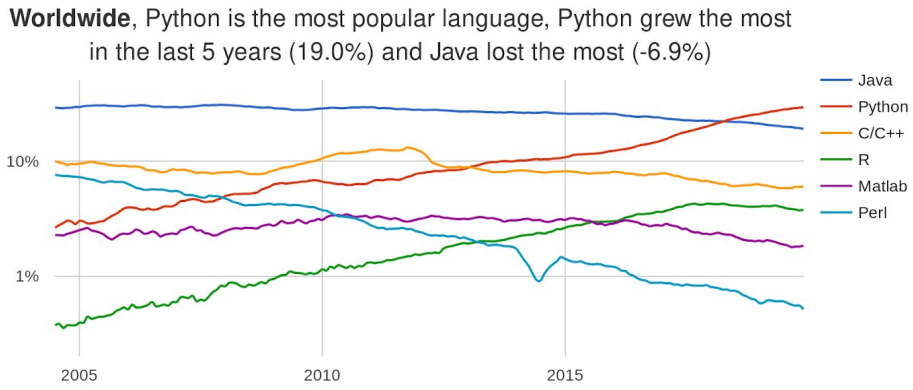




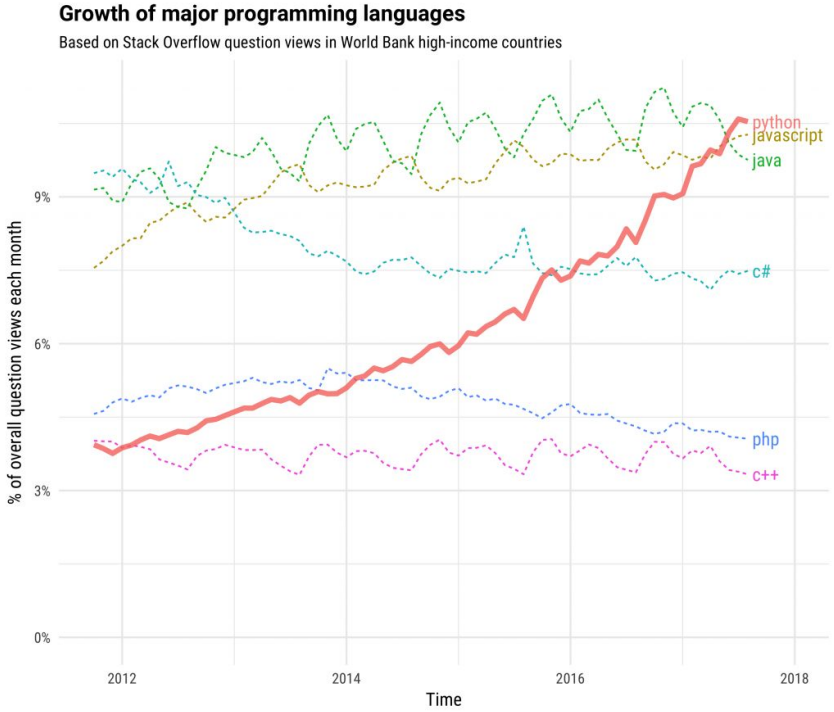
Keynote on interactive data exploration using Panel

- https://medium.com/@philipp.jfr/panel-announcement-2107c2b15f52

As a result: Python world's most popular language



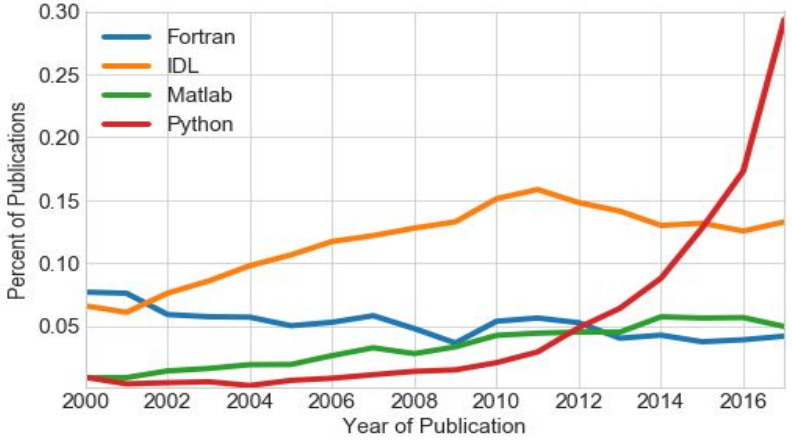
PYPL index, Dec. 2019: based on web searches for tutorials on a given language



Stack Overflow queries: Since 2017 Python has been most popular

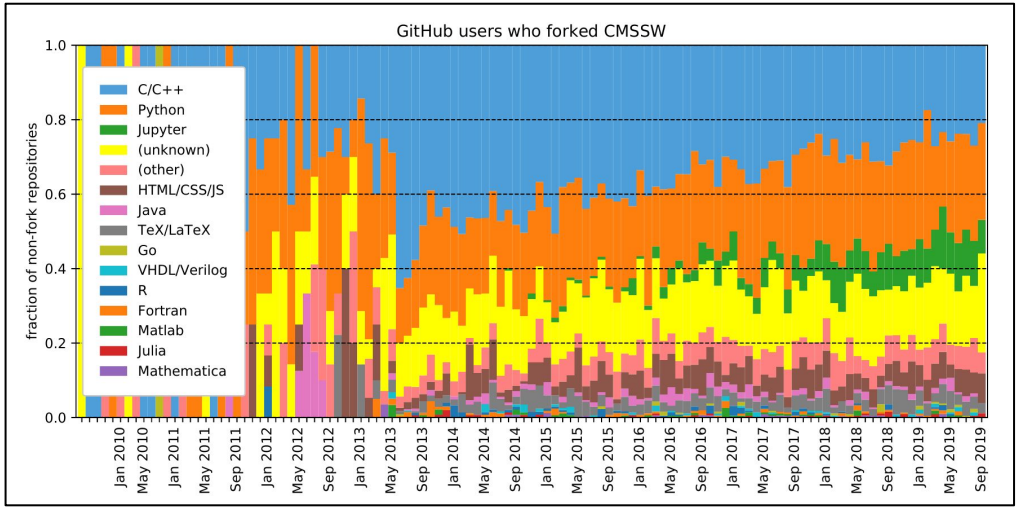
The trend is clear in physics too

Easily the dominant language in Astrophysics



https://gist.github.com/jakevdp/f75c09e43320290ffbedbca43f9fd917

On CMS: most users' code outside is now Python



Analysis by Jim Pivarski

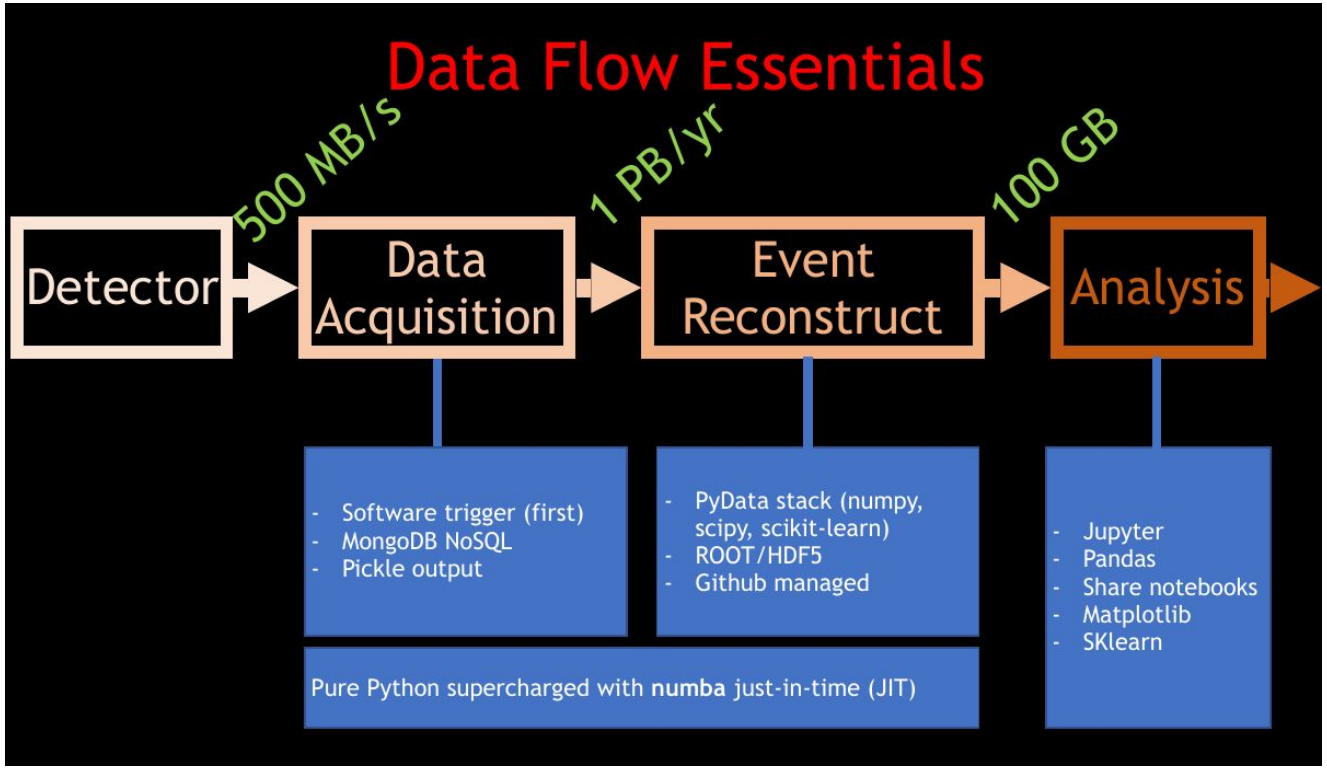
Why Python for high-level particle physics analysis?

- Data analysis outside of particle physics not really in C++ these days:
  - It's primarily in Python
  - ⇒ guidance and tutorials already online
  - ⇒ more useful for students after a PhD
  - ⇒ use industry-standard tools with little extra work ⇒ free personpower
- For example: machine learning
  - https://github.com/josephmisiti/awesome-machine-learning
  - 291 libraries in Python
  - 59 tools in C++

Full experiment stack: Xenon1T

DAQ, trigger, reco and analysis code all in python

Chris Tunnel for Xenon1T, PyHEP2018 https://zenodo.org/record/1418513



Point 1:

Python as a 1st class analysis language: many examples in HEP, much to be gained

But: “isn’t Python slow?”

Sort of:

- Interpreted not compiled
- Global Interpreter Lock: standard interpreted not multi-threaded
- Dynamically typed: attribute look-up more involved
- Primitive types use relatively large

Although:

- Python can now be Just in time compiled (e.g. Numba)
- Other interpreters maturing (e.g. PyPy)

And, crucially, there are other ways of doing things….

Columnar Analysis

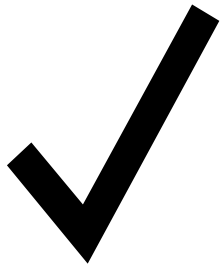
How do I say:

“He’s as cool as a cucumber”

in french ?

“Il a froid comme un concombre”

“Il est d'un calme olympien”



“He is calmly Olympian”

which is a long way to say:

to get good results need to

learn how to think in that language, not just the words

Numpy

Manipulate arrays of data in one go using high-level interface


```python
import numpy

px = numpy.random.normal(0, 100, size=1_000_000)
py = numpy.random.normal(0, 100, size=1_000_000)
```

Pure python loop over px and py pairs:

```python
pt = []
for i in range(len(px)):
    pt.append(numpy.sqrt(px[i]**2 + py[i]**2))
```

O(N) python instructions

Numpy

Manipulate arrays of data in one go using high-level interface


```python
import numpy

px = numpy.random.normal(0, 100, size=1_000_000)
py = numpy.random.normal(0, 100, size=1_000_000)
```

Pure python loop over px and py pairs:

```python
pt = []
for i in range(len(px)):
    pt.append(numpy.sqrt(px[i]**2 + py[i]**2))
```

O(N) python instructions

Using numpy array operations:

```python
pt = numpy.sqrt(px**2 + py**2)
```

O(1) python instructions

O(N) heavily optimised instructions

Numpy

Manipulate arrays of data in one go using high-level interface


```
1  import numpy
2
3  px = numpy.random.normal(0, 100, size=1_000_000)
4  py = numpy.random.normal(0, 100, size=1_000_000)
```

Pure python loop over px and py pairs:

```
6  pt = []
7  for i in range(len(px)):
8      pt.append(numpy.sqrt(px[i]**2 + py[i]**2))
```

O(N) python instructions

Using numpy array operations:

```
6  pt = numpy.sqrt(px**2 + py**2)
```

O(1) python instructions

O(N) heavily optimised instructions

Numpy operations are: Single Instruction Multiple Data (SIMD)

```
8  selected = mass[(pt > 1000) & (2 < eta) & (eta < 5)]
```


*Summary shamelessly ripped from Chris Burr, CERN

Numpy (2)

A high-level interface to low-level routines:

- Uses vectorized programming in CPU for efficiency
- Supports multi-dimensional arrays

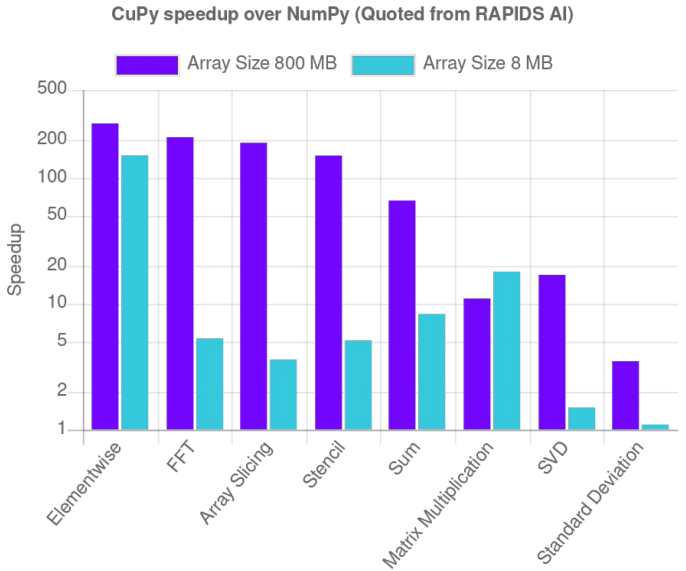
Numpy (2)

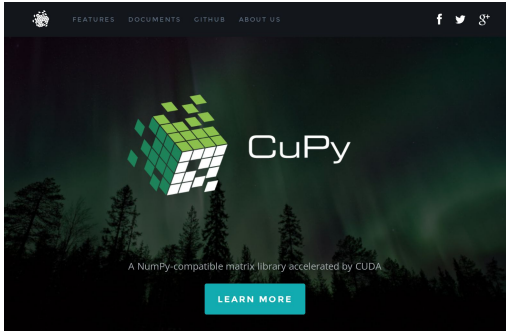
A high-level interface to low-level routines:

- Uses vectorized programming in CPU for efficiency
- Supports multi-dimensional arrays

But this is python:

- Dynamic nature of language
- Package ecosystem
- ⇒ Cupy: Same user code can run on GPUs
- See also PyHEADTAIL





Numpy (2)

A high-level interface to low-level routines:

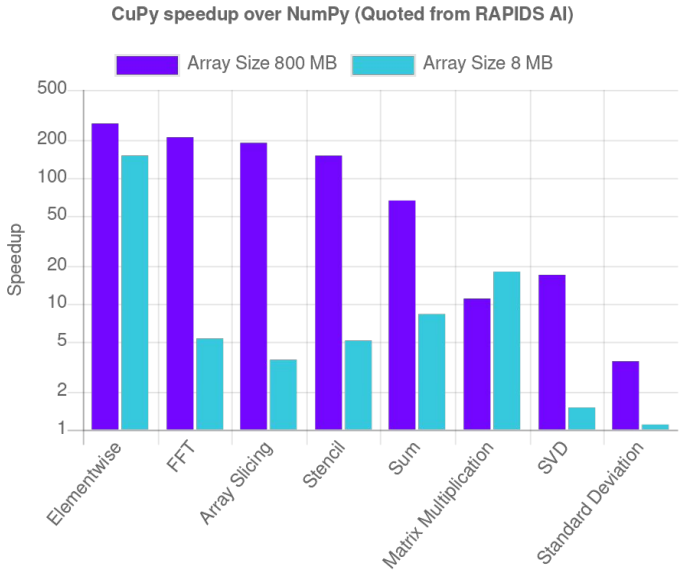
- Uses vectorized programming in CPU for efficiency
- Supports multi-dimensional arrays

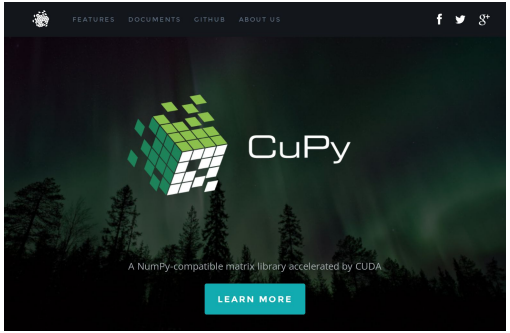
But this is python:

- Dynamic nature of language
- Package ecosystem
- ⇒ Cupy: Same user code can run on GPUs
- See also PyHEADTAIL

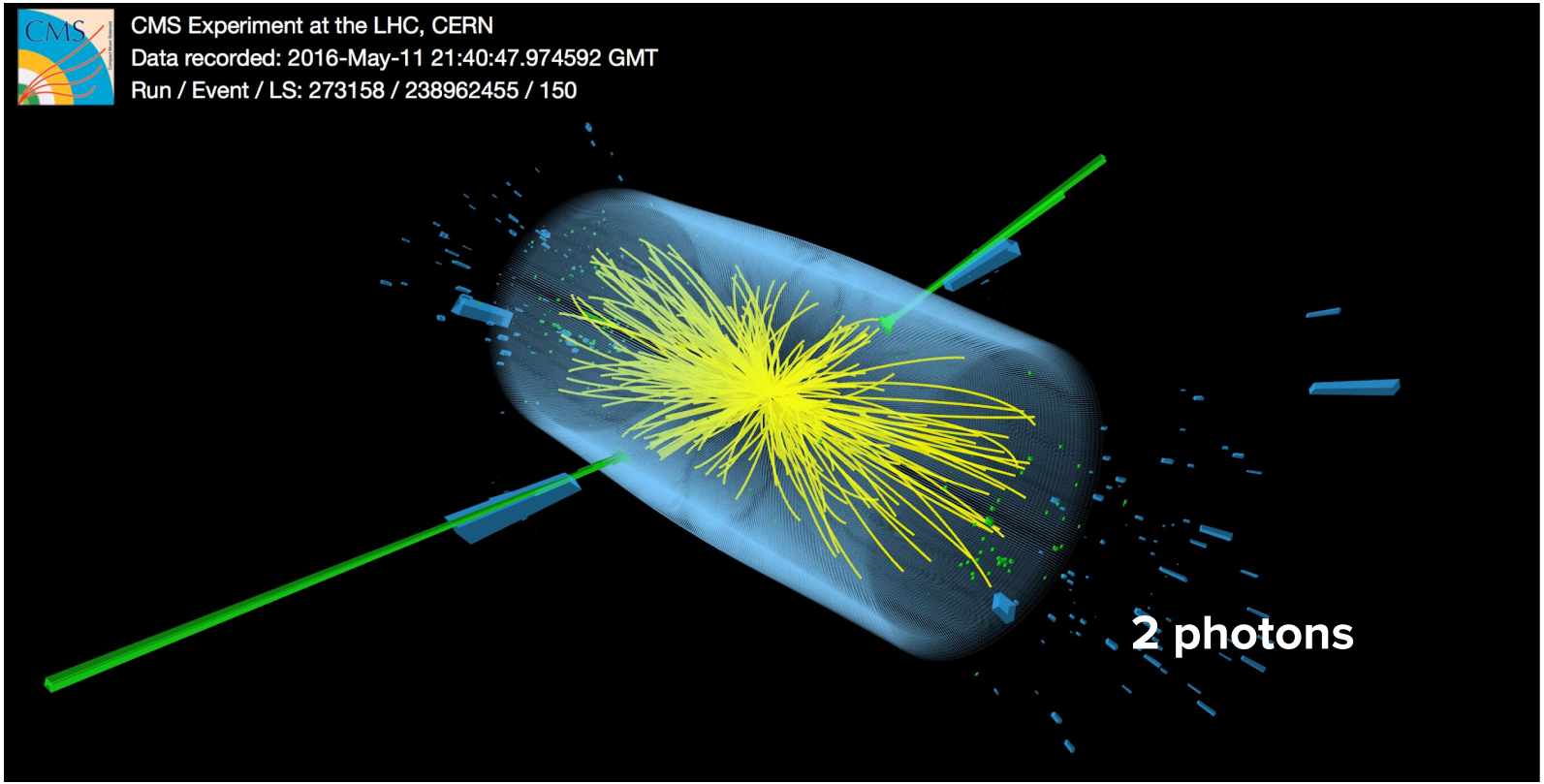
Difficulties for particle physics:

- Getting data from ROOT files into such arrays without a for-loop
- Our data is often more structured than simple arrays

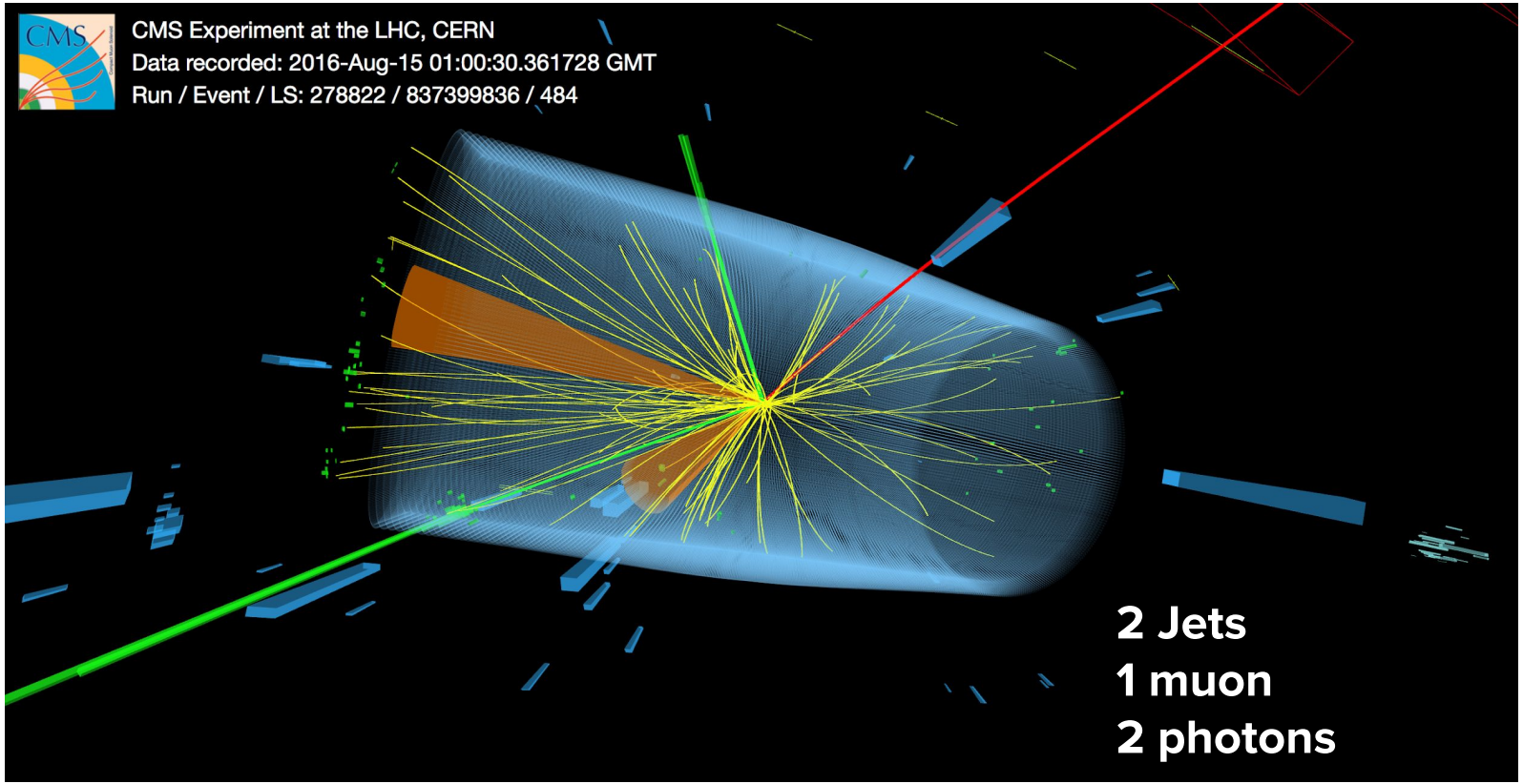




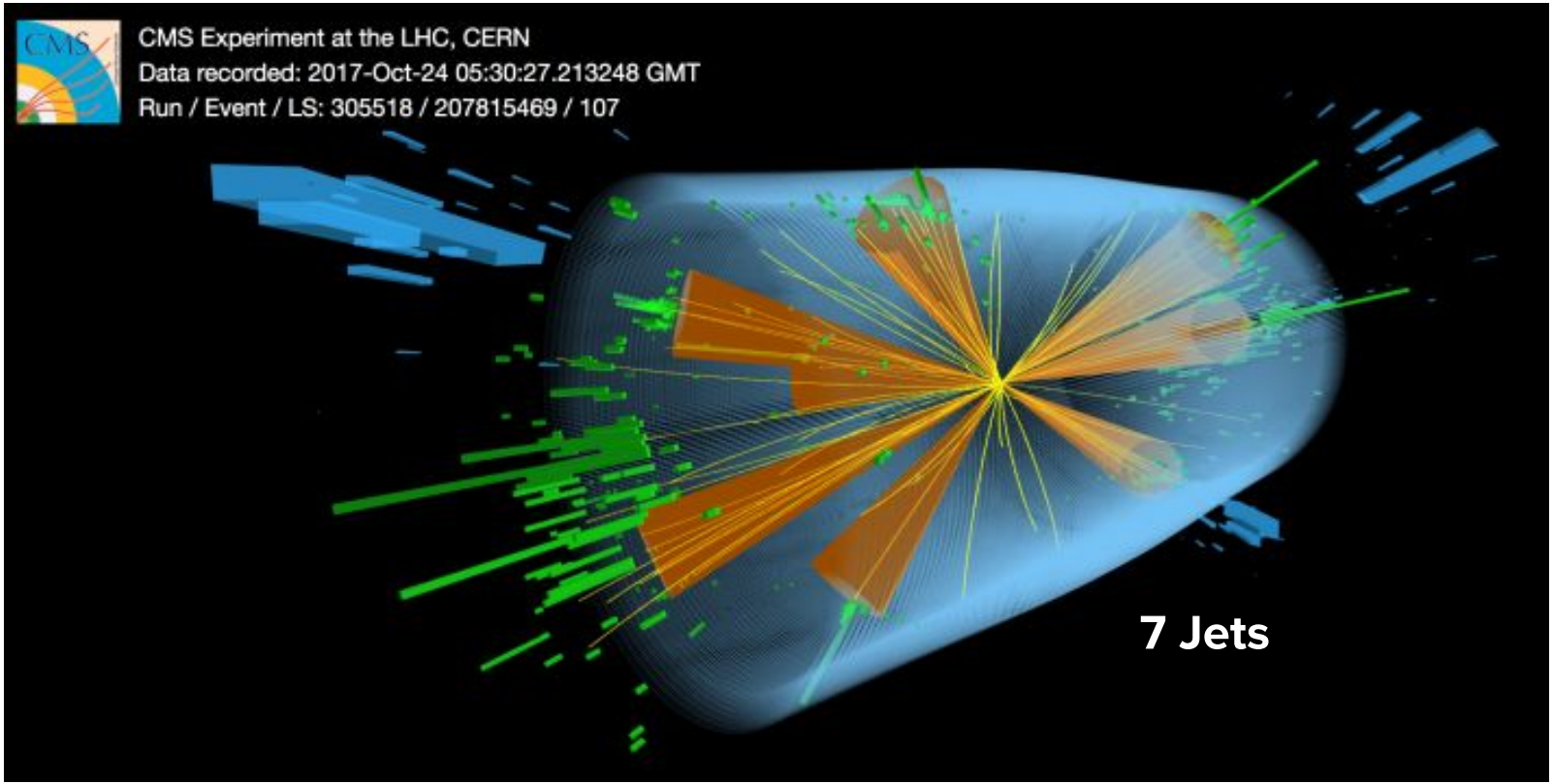
Structured “events”



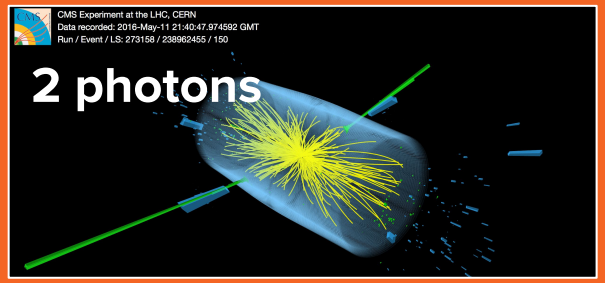
Structured “events”

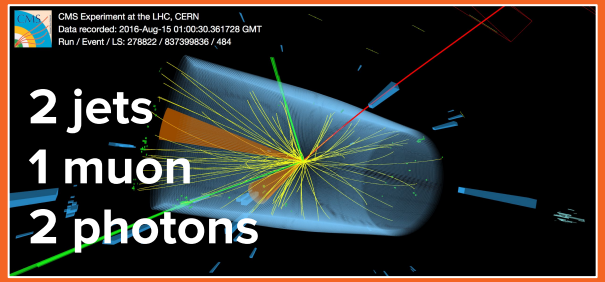


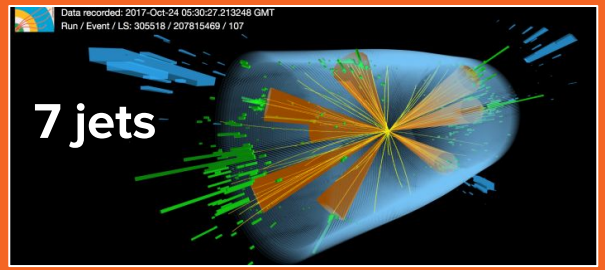
Structured “events”



Traditional HEP data structure





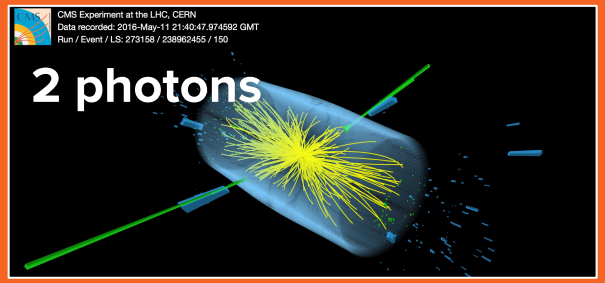


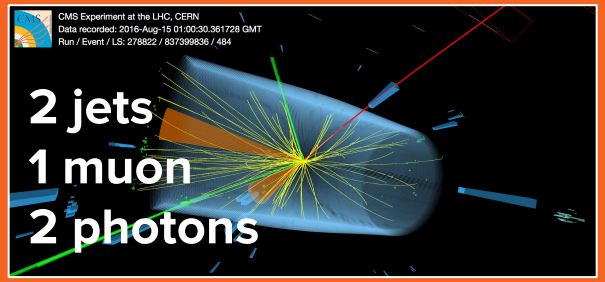
Event #1
time
1:01

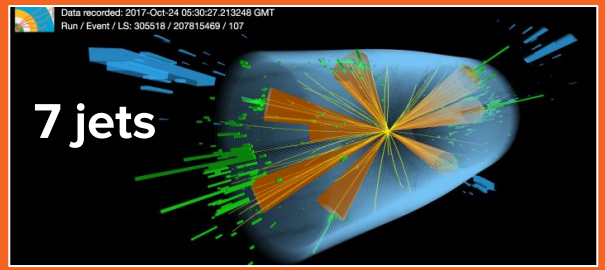
Event #2
time
1:03

Event #3
time
1:05

Traditional HEP data structure







Event #1

time: 1:01

photons — mom.: 80 | 60

Event #2

time: 1:03

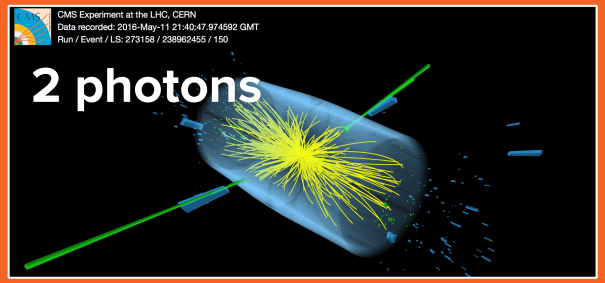
photons — mom.: 90 | 87

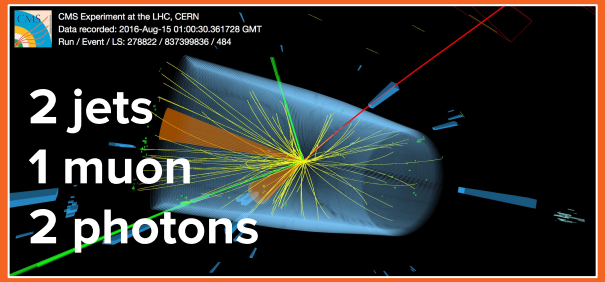
Event #3

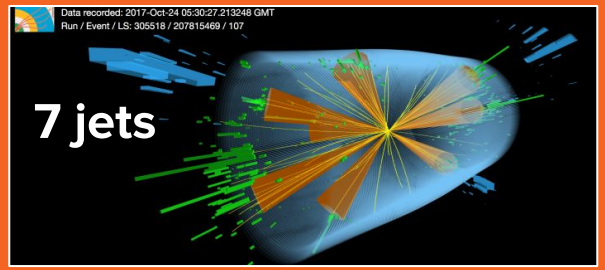
time: 1:05

photons — mom.:

Traditional HEP data structure







Event #1

time	photons	
	mom.	80 60
1:01	eta	-2 1.1

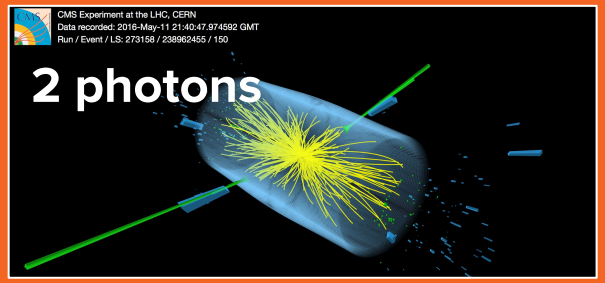
Event #2

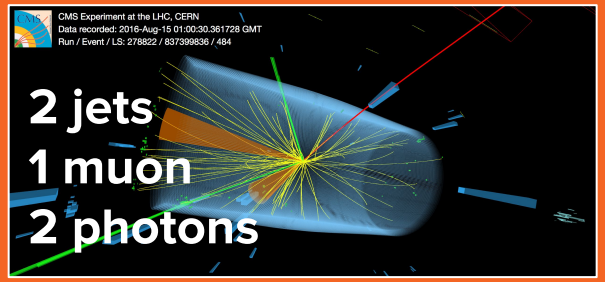
time	photons	
	mom.	90 87
1:03	eta	3 2.2

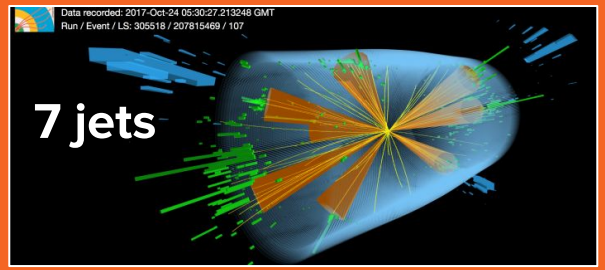
Event #3

time	photons	
	mom.	
1:05	eta	

Traditional HEP data structure







Event #1

time	photons		jets
	mom. 80	60	mom.
1:01	eta -2	1.1	eta

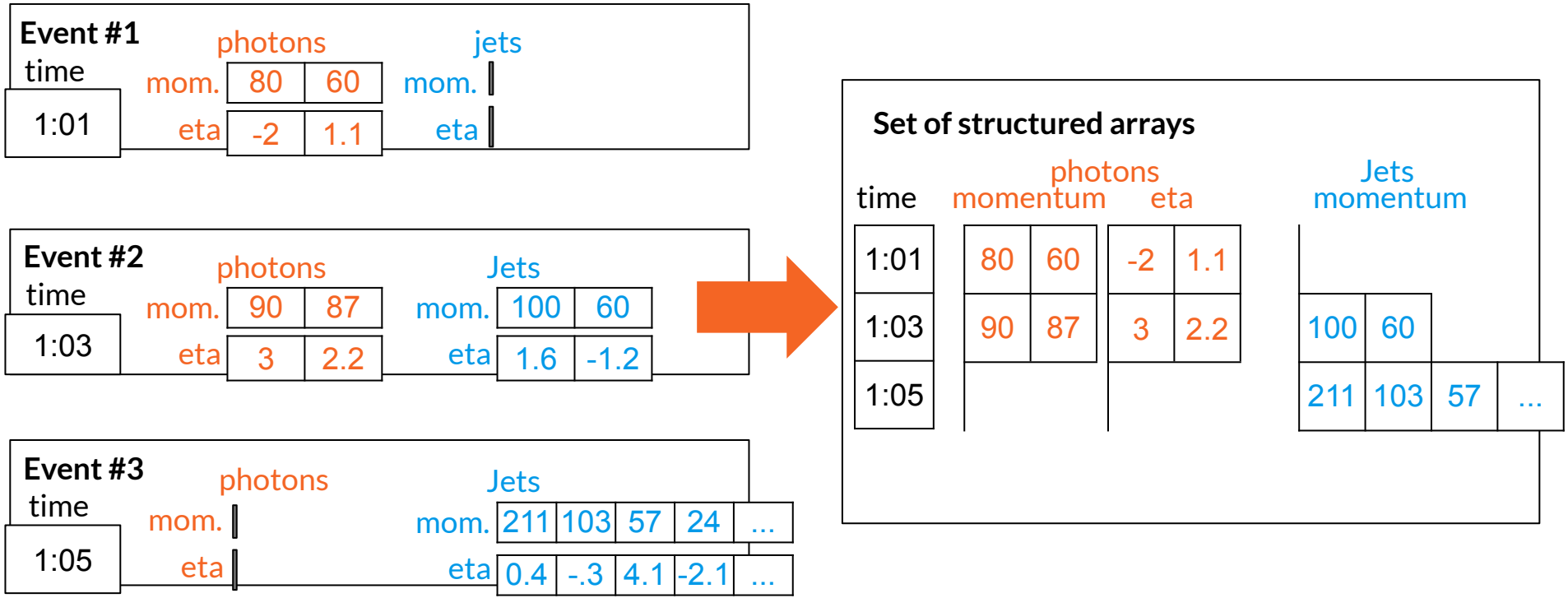
Event #2

time	photons		Jets		
	mom. 90	87	mom. 100	60	
1:03	eta 3	2.2	eta 1.6	-1.2	

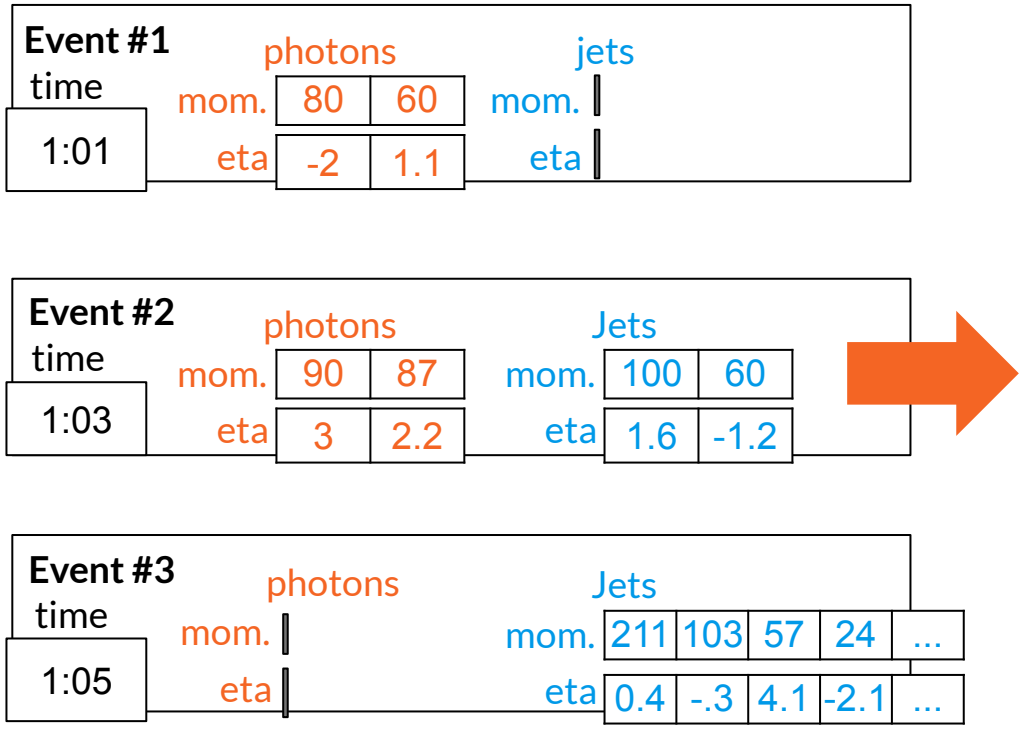
Event #3

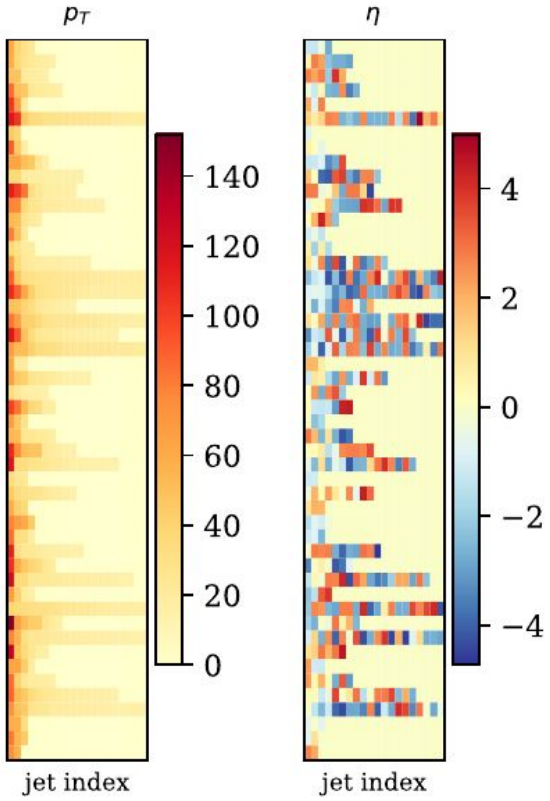
time	photons		Jets				
	mom.		mom. 211	103	57	24	..
1:05	eta		eta 0.4	-.3	4.1	-2.1	..

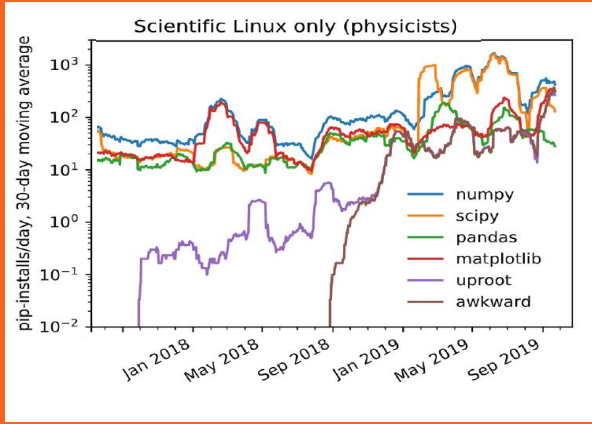
How can we turn this into numpy arrays?



How can we turn this into numpy arrays?







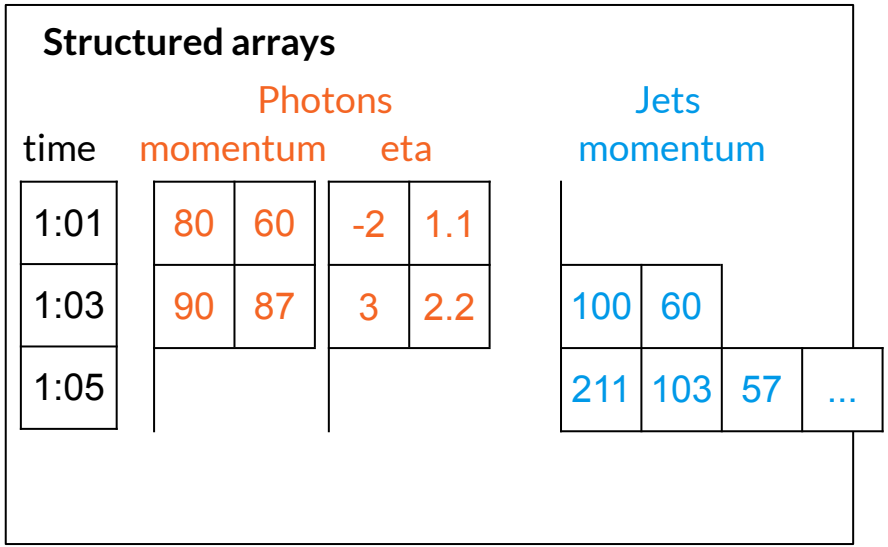
- For particle physics: our existing file format made this tricky to do efficiently until…

- Uproot = micro pythonic ROOT
  - Does one thing: IO of ROOT files in python
  - Efficient memory handling: baskets of data on disk copied into numpy array directly
  - About 2 years old -- one of the most important packages for particle physics with python

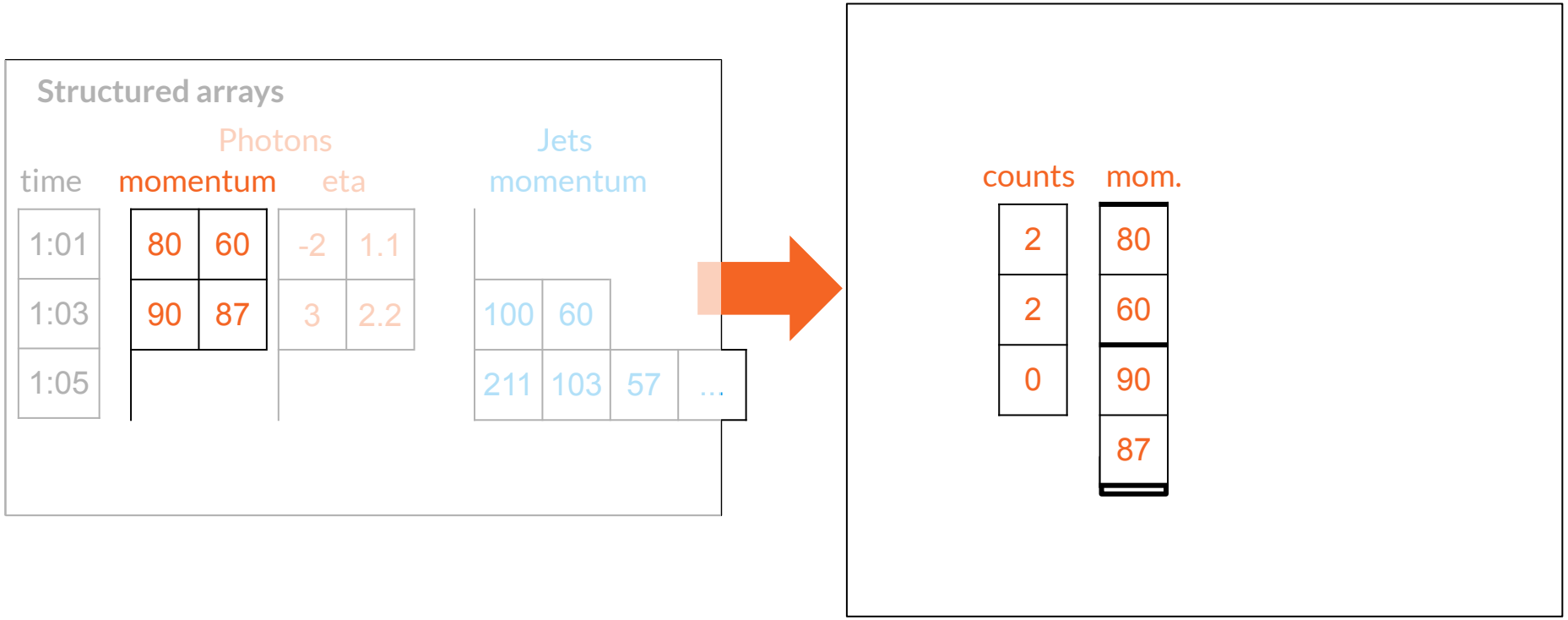
- After this: uproot will be maintenance only, no other major developments planned

But how to make “numpy arrays” for variables with different lengths in each event?

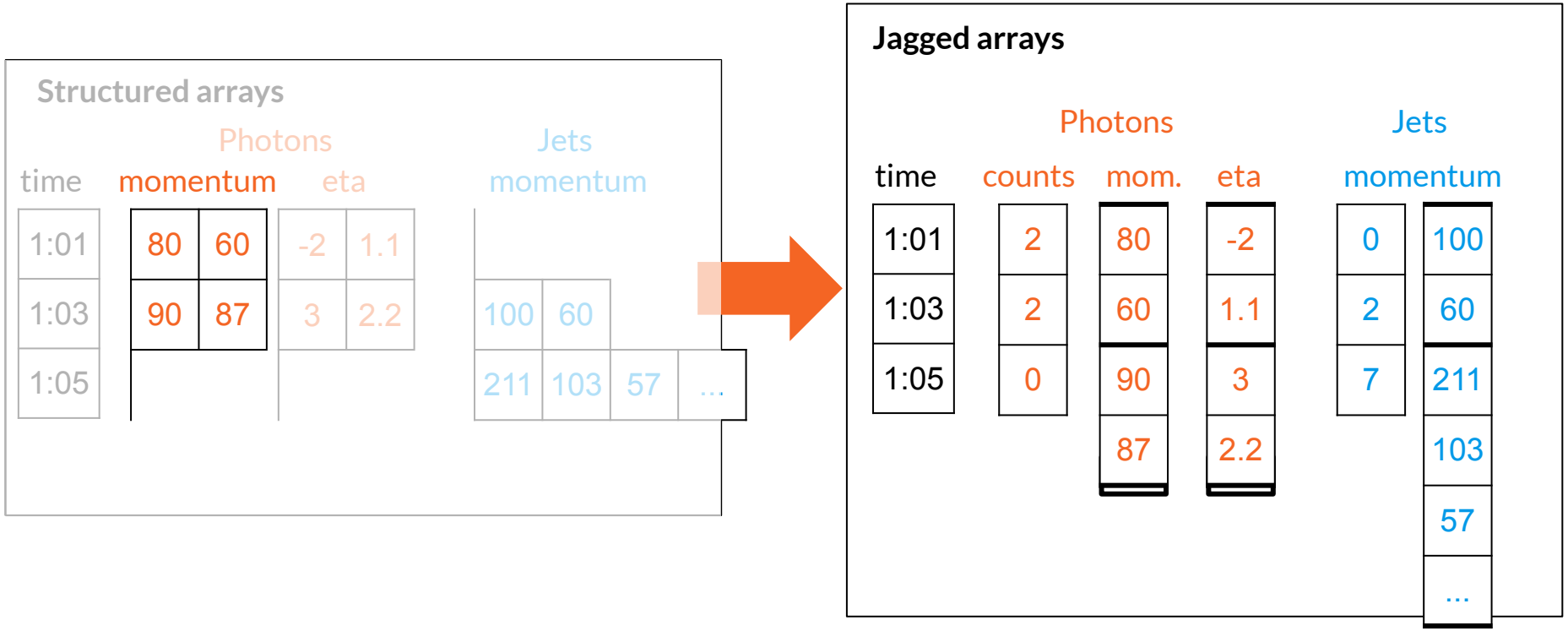
How can we turn this into numpy arrays?



How can we turn this into numpy arrays?



How can we turn this into numpy arrays?



Awkward Array

- Implements the concept of jagged arrays
  - Broadcasting, masking, reducing
- Methods to manipulate these without a python for loop: very quick operations
  - Internally using numpy
- Version 1.0 should be released soon:
  - Rewrite the internals
  - Tidy up the interface
  - Let other packages interpret awkward arrays easily (numba, numexpr)

Jagged Arrays

array1

#1	2	3	1	88
#2	7	13		
#3	17	1	34	

largest value in each event

array1.max()

#1	88
#2	13
#3	34

number of objects in each event

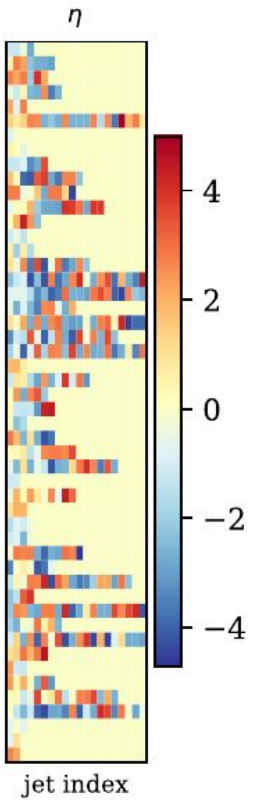
array1.count()

#1	4
#2	2
#3	3

Position of smallest value in event

array1.argmin()

#1	2
#2	0
#3	2

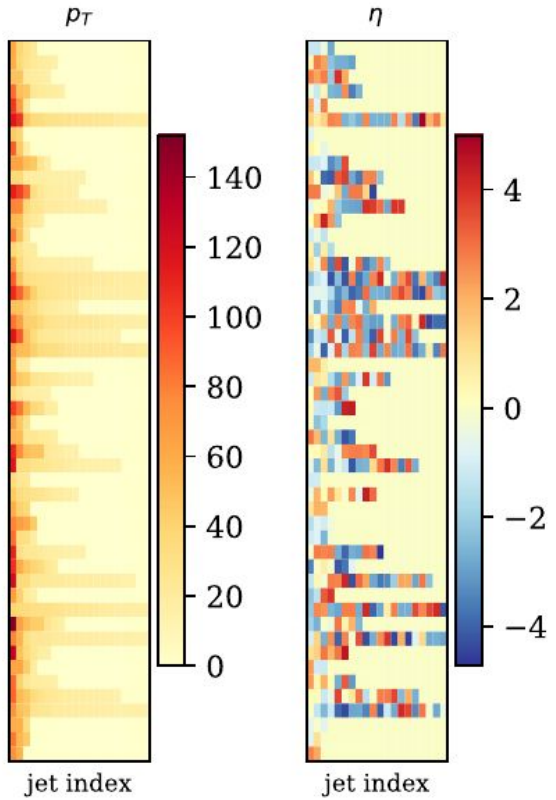


Jagged Arrays

For example, find the momentum of the jet with the largest magnitude of eta in each event:

Break it down:

- `numpy.abs(Jet_eta)`= absolute eta of every jet in every event

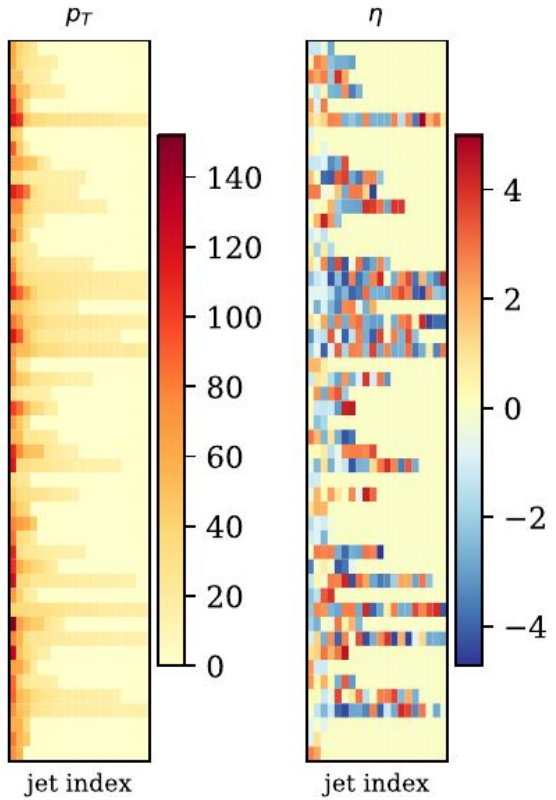


Jagged Arrays

For example, find the momentum of the jet with the largest magnitude of eta in each event:

Break it down:

- `numpy.abs(Jet_eta)`= absolute eta of every jet in every event
- `numpy.abs(Jet_eta).argmax()`= index of jet with largest absolute eta for each event. Number between 0 and Njet

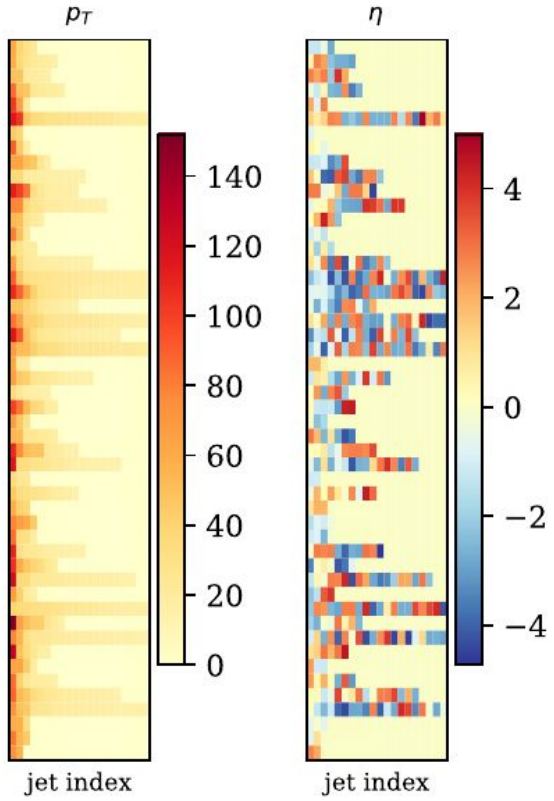


Jagged Arrays

For example, find the momentum of the jet with the largest magnitude of eta in each event:

Break it down:

- `numpy.abs(Jet_eta)`= absolute eta of every jet in every event
- `numpy.abs(Jet_eta).argmax()`= index of jet with largest absolute eta for each event. Number between 0 and Njet
- `Jet_mom[numpy.abs(Jet_eta).argmax()]`= pt of the jet with the largest absolute eta for each event, now a simple 1D array




```
mom_largest_eta = Jet_mom[numpy.abs(Jet_eta).argmax()]
```

Some non-particle physics uses?

Credits: NASA

Letters in words in sentences in paragraphs

enormous quantities of diverse, disconnected data. These data sets present substantial analytic challenges, but can also illuminate new avenues of inquiry that yield unprecedented improvements in global health. Roam is realizing this potential by combining our proprietary data platform with advanced machine learning, empowering life sciences companies, hospital systems, insurers, and

Coffea - Column Object Framework for Effective Analysis

Fermilab project to build an analysis framework on top of awkward array and uproot

Separation of “user code” and “executors”

- User writes a Processor to do the analysis
- Executor runs this on different distributed job systems, e.g.:
  - Local multiprocessing, Parsl or Dask (batch systems), **Spark** cluster

Coffea ***achieved 1 to 3 MHz*** event processing rates

- Using Spark cluster on same site as data at Fermilab

Point 2:

Particle physics working hard to connect with “big data” tools: MHz event processing possible

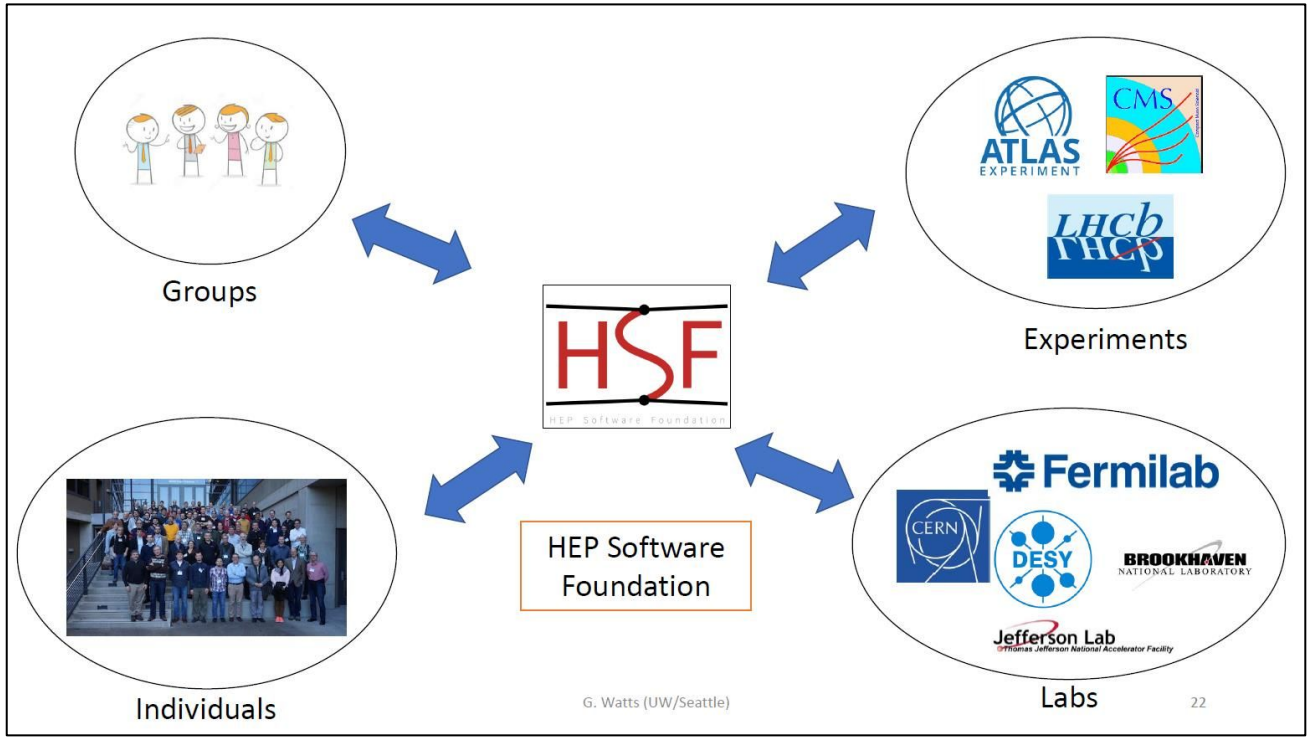
Building a community for particle physics analysis software

HEP Software Foundation

Connect stakeholders in particle physics code

Facilitate coordination and common efforts in software and computing across HEP in general

A very bottom up approach
“Do-ocracy”



PyHEP: Python for Particle Physics

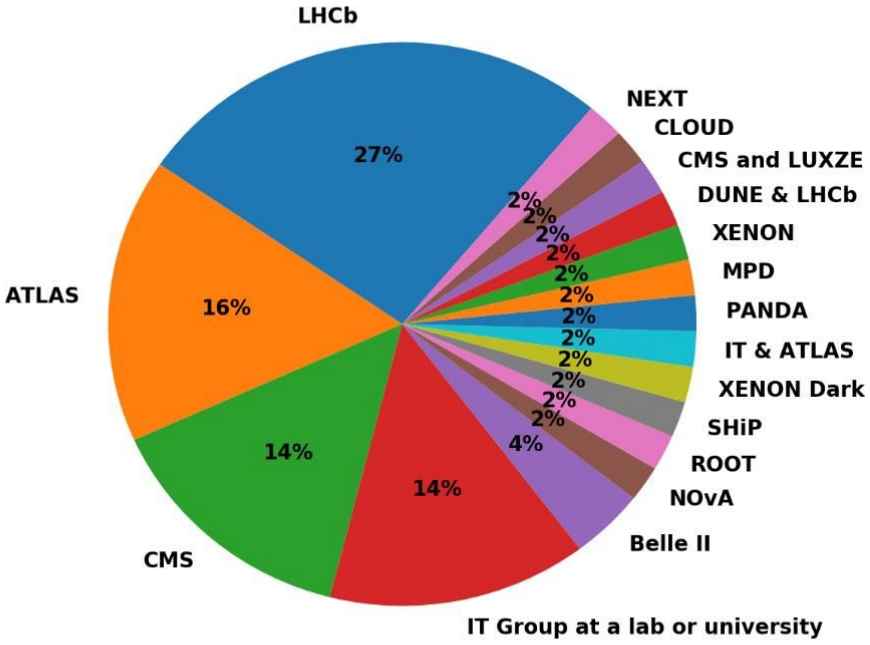
Working group of the HSF

Building a community of users and developers

Training, exploring, and developing

In-person workshop roughly once a year

Join in: gitter.im/PyHEP



PyHEP workshops

PyHEP 2018: Sofia, Bulgaria

PyHEP 2019: Abingdon, UK

python SOFTWARE FOUNDATION

iris hep Institute for Research & Innovation in Software for High Energy Physics

Science & Technology Facilities Council

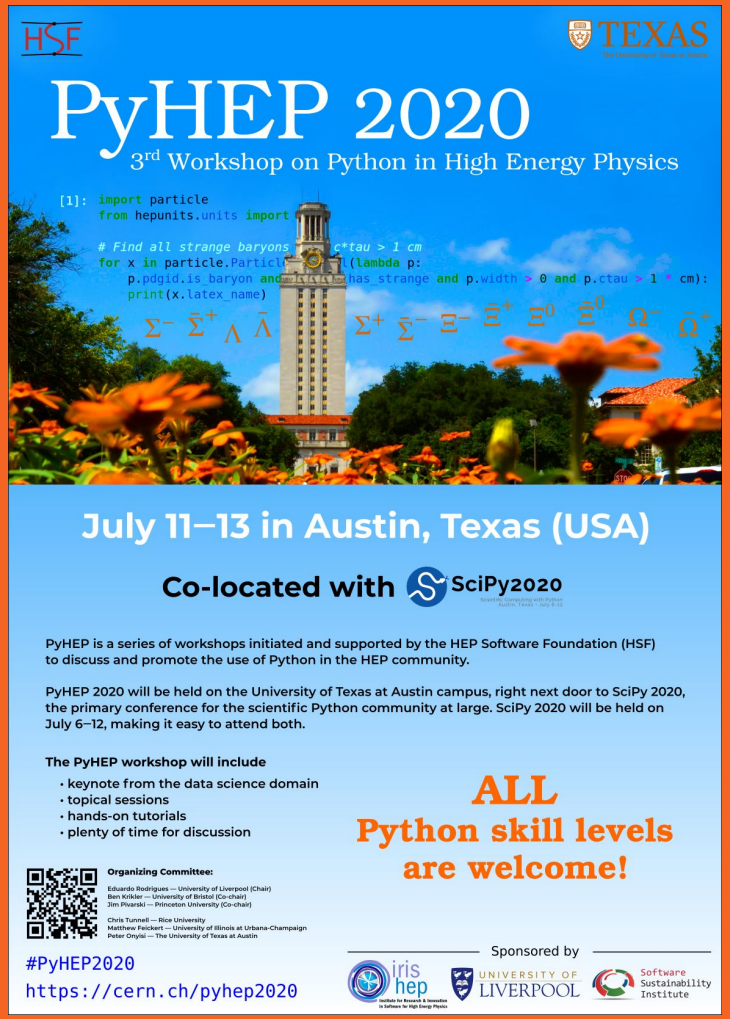
Software Sustainability Institute

PyHEP2020

indico.cern.ch/e/PyHEP2020

11 to 13th July in Austin, Texas

Co-located with SciPy (6 - 12th)



scikit-hep

http://scikit-hep.org/
https://github.com/scikit-hep/

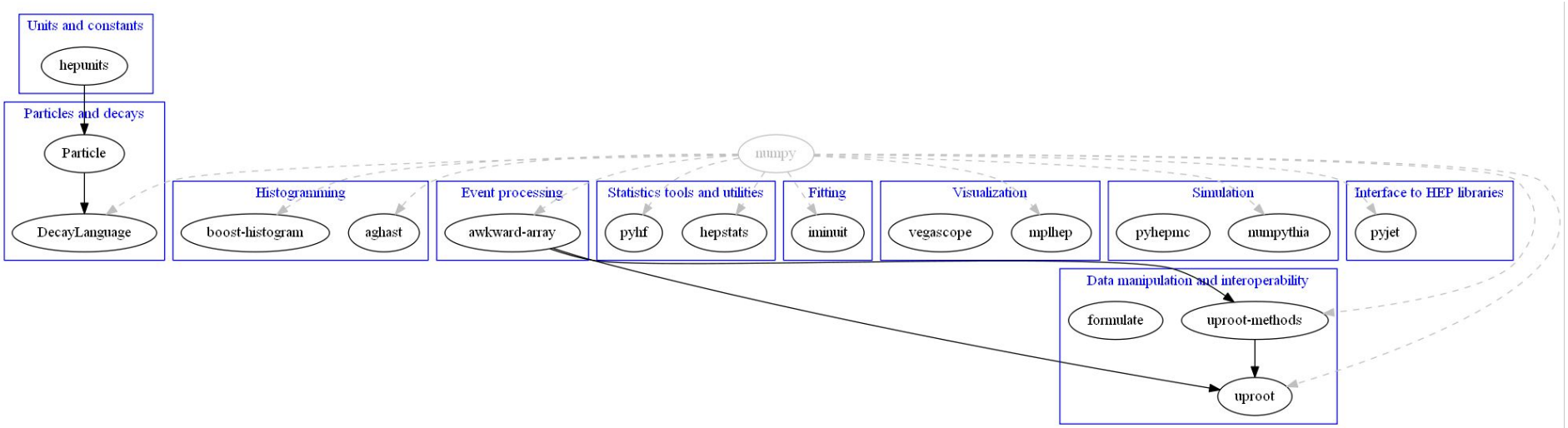
The success of Python for astronomy is partly due to the Astropy project

PyHEP brings us the community

Scikit-hep allows us to collect packages together

- e.g. Uproot and Awkward-array exist within scikit-hep project

The full scikit-hep ecosystem



+ many “affiliated packages” which are closely connected

DecayLanguage

Programmatic interface to:

- Parametrise
- Visualise
- And generate from

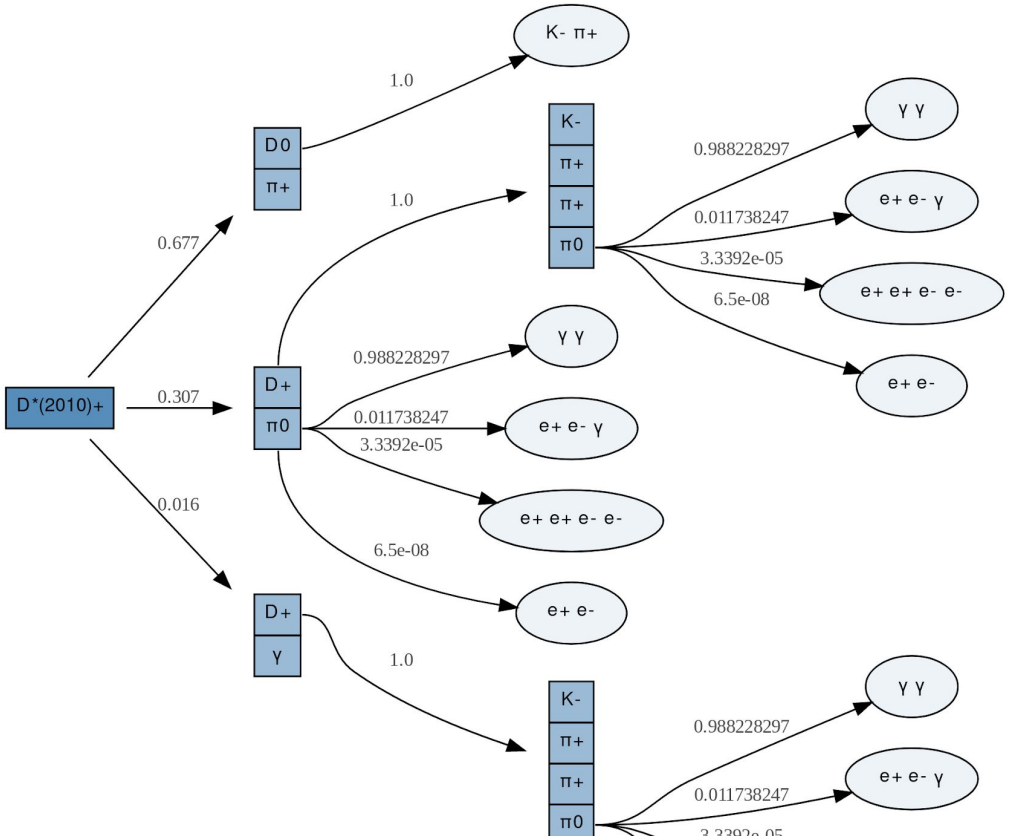
Particle decay chains

Mainly used on LHCb so far

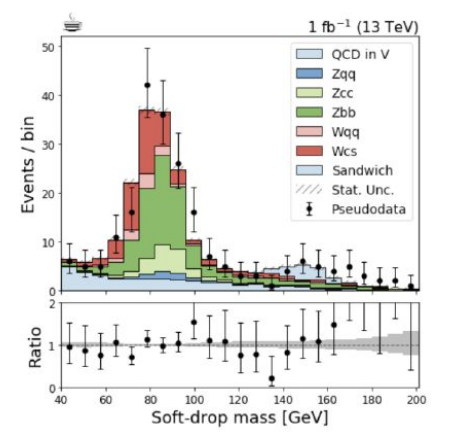
Helpful for our background tables?

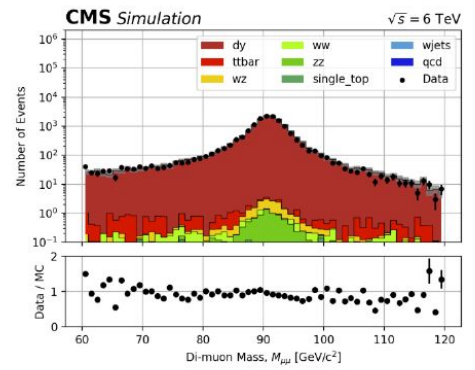
- Can extend particle data with isot

```
dc = dfp_Dst.build_decay_chains('D*+')
DecayChainViewer(dc)
```



mpl-hep





Particle Physics loves histograms!

But matplotlib is a little tricky with pre-binned data

Survey on plotting needs:

- Stacked histograms
- Good error bars
- Ratios of 1D plots
- Simple “COLZ” option
- Consistent plot styling

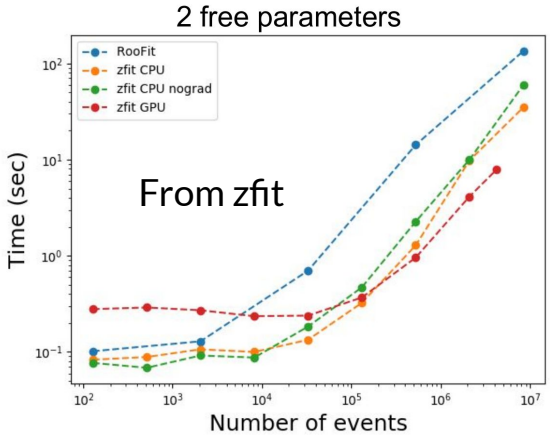
mpl-hep package should become associated with matplotlib (spoken with matplotlib devs)

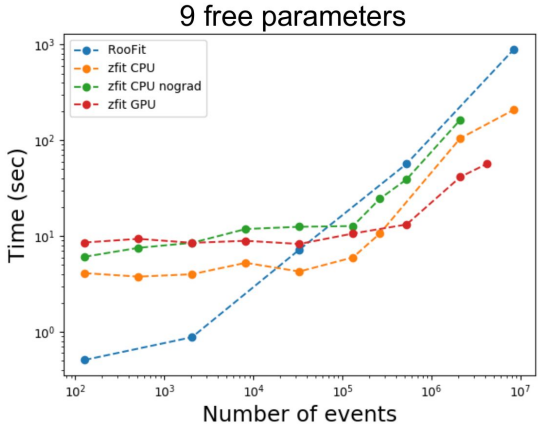
Fitting

Many presentations on fitting and statistics

Using TensorFlow as a backend:

- Zfit -- focussed on unbinned fits, adapting deep learning techniques for model fitting
- PyHF -- store the entire likelihood on HEPData





Point 3:
Building a community of users from specific domains important for tools and training

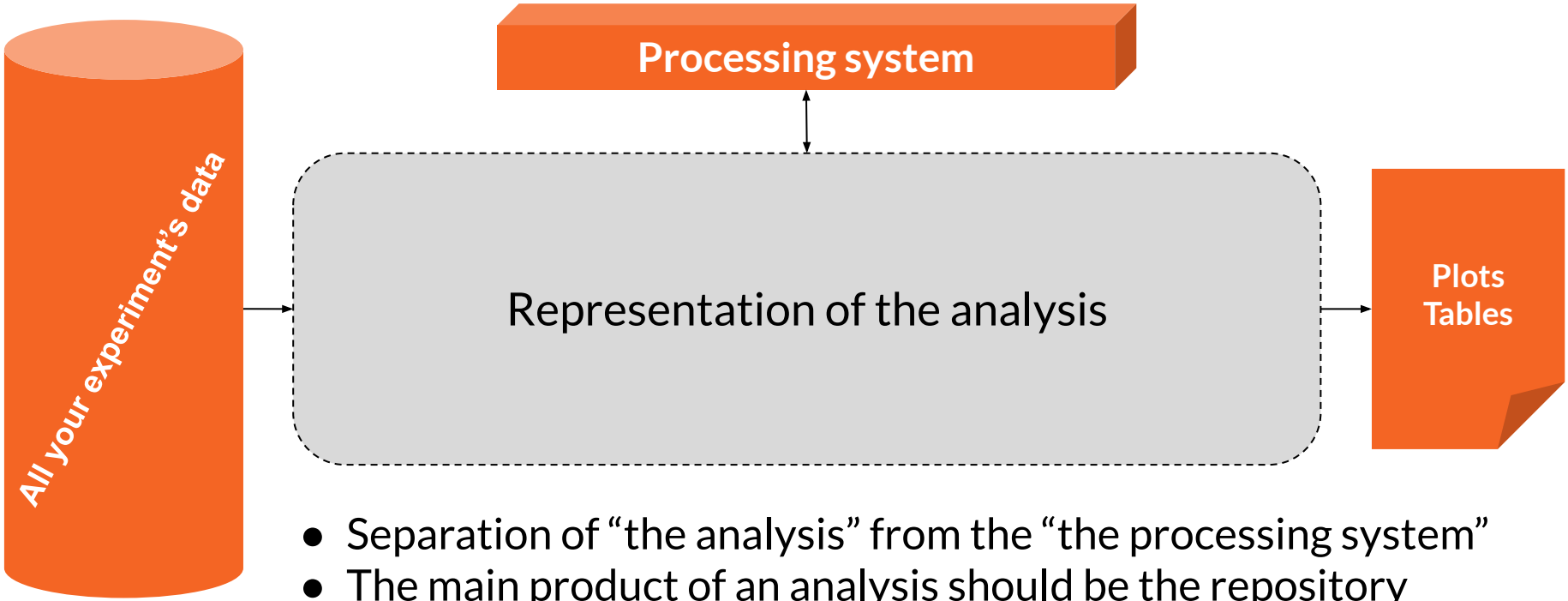
But: Is Python “high-level” enough?

User decides flow control

Writing full jagged array manipulations can be tough (e.g. object matching)

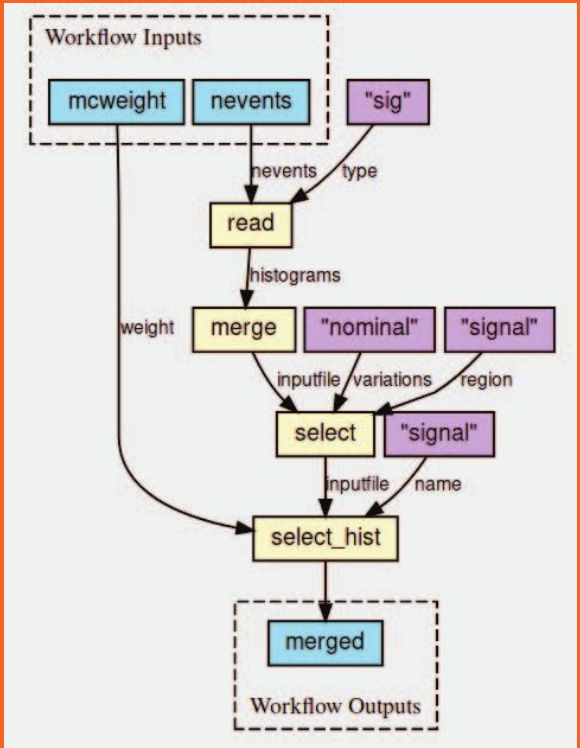
Analysis description languages

Analysis *versus* analysis tools



- Separation of "the analysis" from the "the processing system"
- The main product of an analysis should be the repository

Workflow managers



Let another system take care of your workflow

Many tools:

- SnakeMake, Parsl, Airflow
- More integrated: Spark, DASK

Define your analysis as a Directed Acyclic Graph (DAG)

- Each task is a node
- Data flow is an arrow connecting two nodes

For free:

- Caching of repeated tasks at each node
- Can optimise the DAG: “elide” (remove) nodes if result is never used

Analysis description languages

A large fraction of LHC analyses involve only a few steps

Can we encapsulate these into a “Domain Specific Language”?

Several different attempts to build an ADL:

- LINQ (Gordon Watts et al)
- NAIL (Andrew Rizzi)
- FAST-HEP (this talk)
- Dedicated workshop at Fermilab last May: https://indico.cern.ch/event/769263/

The

toolkit

F.A.S.T = Faster Analysis Software Taskforce

- UK-based particle physicists
- Started around May 2017
- Explore ways to accelerate and improve our analysis code
- Use of 1 to 3-day “hack-shops” to test new ideas

How we have worked

Design principles:

- Write as little code as possible: act as glue
- Contribute first to other projects
- Value modularity

Goals:

a. Reproducibility
b. Simplicity
c. Speed
d. Documentation
e. Automation

Current FAST-HEP codebase

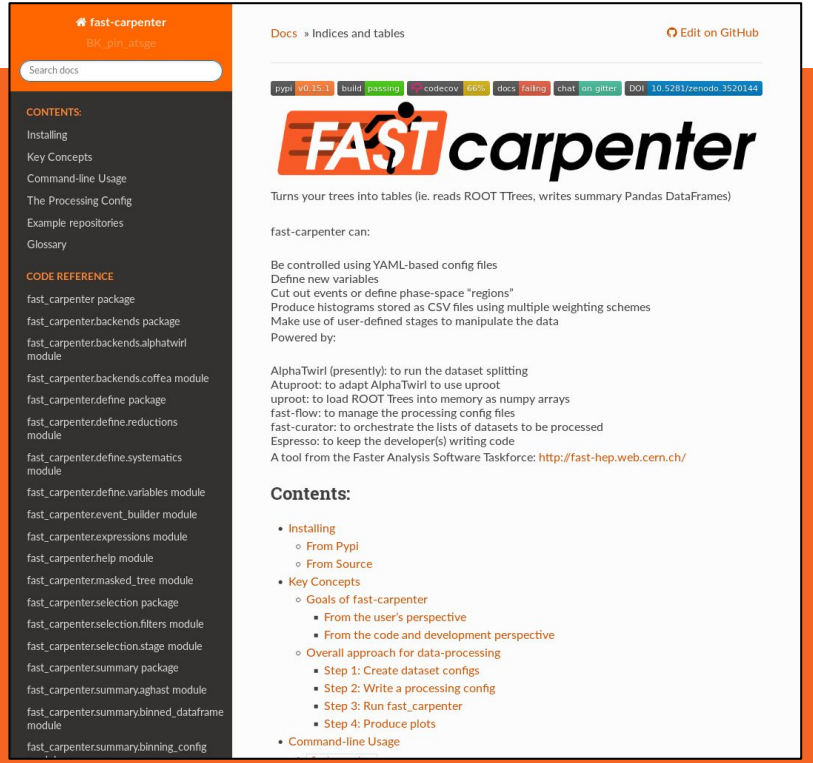
Being used for **CMS analyses**, **LUX-ZEPLIN** and **ATLAS** investigated, used for design studies of **DUNE,** and **FCC** experiments

New features being fed back to core packages from analysis-specific repositories

- Direct use in Jupyter notebooks
- Writing skimmed / slimmed outputs
- Persistency outside of CSV formats
- Docker container for running at NERSC, etc

Making it easy to find and use

- All public on github:
  - github.com/fast-hep/
  - Main package: github.com/fast-hep/fast-carpenter
- On PyPI, e.g. fast-carpenter
- Docker image with all tools: fasthep/fast-hep-docker
- Docs: fast-carpenter.readthedocs.io/
- Clonable demo analysis repository:
  - gitlab.cern.ch/fast-hep/public/fast_cms_public_tutorial
- Chat: gitter.im/FAST-HEP



The FAST toolkit

For internals:

use Python

uproot

Awkward Array

NumExpr

at 

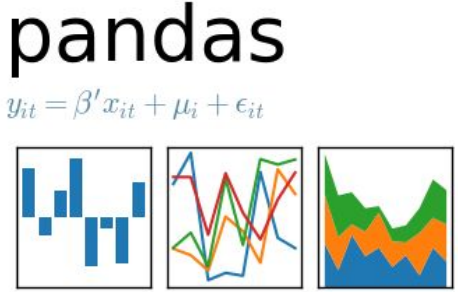
The FAST toolkit

For internals:
use Python

NumExpr

at 

For data:
use Pandas



The FAST toolkit

For internals:
use Python

uproot

Awkward Array

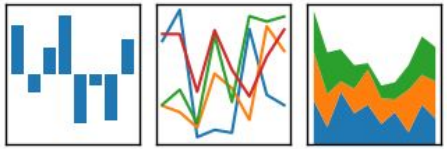
NumExpr

at (☕)

For data:
use Pandas

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



For descriptions:
use YAML...

Describing analysis with YAML

- A superset of JSON
  - Easier to read
- Naturally declarative:
  - No “control flow” (e.g. no for loops)
- Widely used to describe pipeline configuration:
  - gitlab-CI, travis-CI, Azure CI/CD, Ansible, Kubernetes, etc
  - HEPData: YAML for reproducible Data

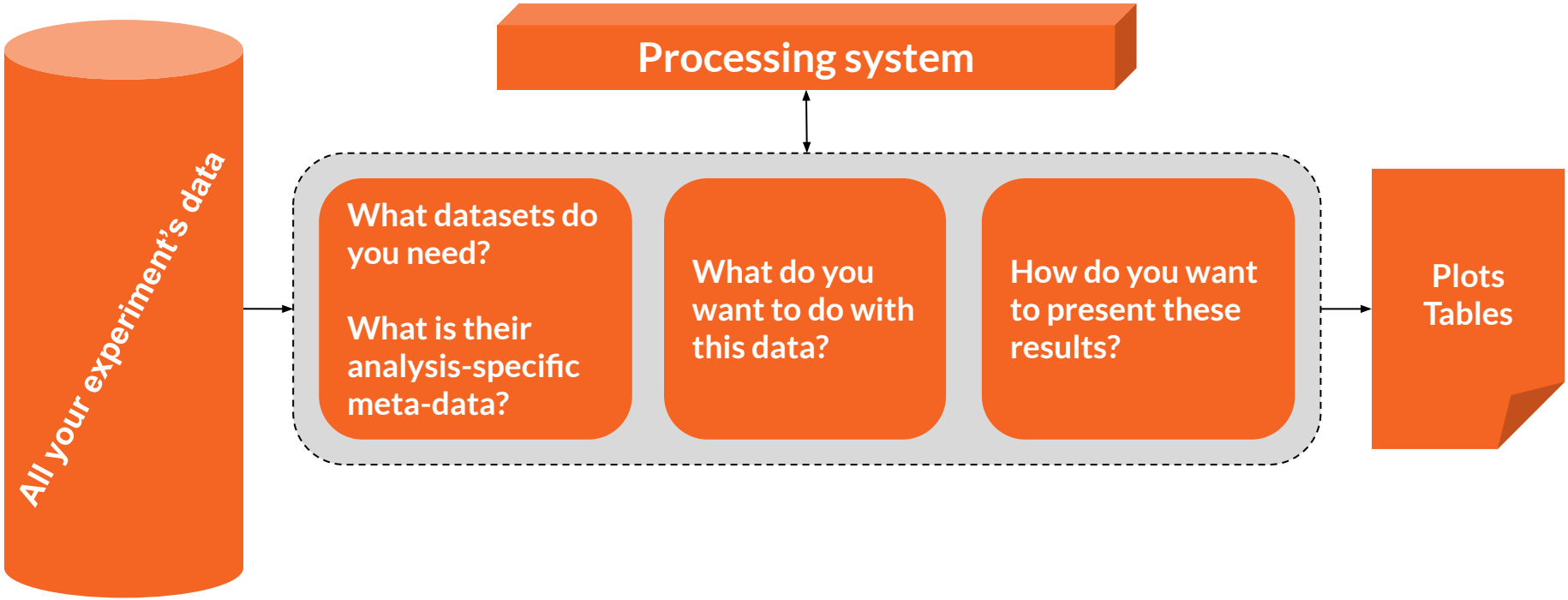
JSON

```
[{"martin":{"name": "Martin Devloper",
    "job": "Developer",`
    "Skills": ["python", "perl", "pascal"]}
,{"tabitha":{"name": "Tabitha Bitumen", "job":
"Developer", "Skills": ["lisp", "fortran",
"erlang"]}}]
```

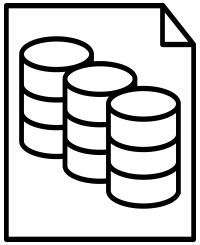
YAML

```
-  martin:
    name: Martin Devloper
    job: Developer
    skills:
      - python
      - perl
      - pascal
-  tabitha:
    name: Tabitha Bitumen
    job: Developer
    skills:
      - lisp
      - fortran
      - erlang
```

Analysis *versus* analysis tools

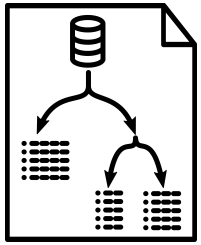


Step 1:
`fast_curator`



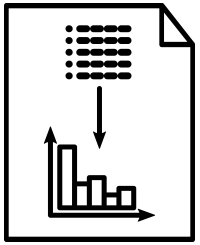
Dataset description

Step 2:
`fast_carpenter`
`(using fast-flow)`



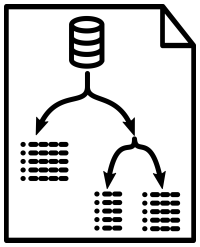
Analysis description

Step 3:
`fast_plotter`
`fast_datacard`



Plotting and postprocessing

Step 2:
`fast_carpenter`



Analysis description

Take your trees and make them into tables

- Just like a carpenter

Table = Pandas DataFrame

Two main types of table for now:

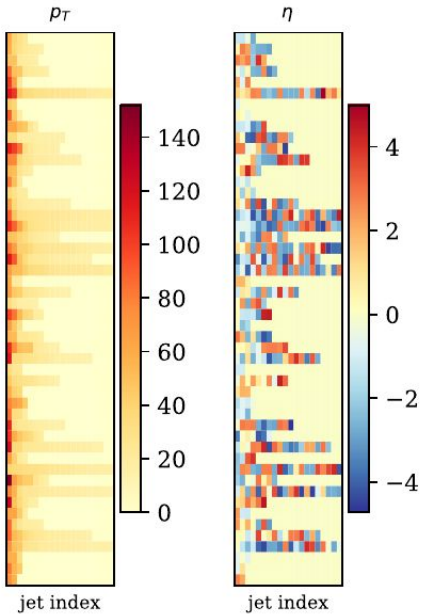
- Histogram
- Cutflow

Cover most typical particle physics analyses

- BUT: very easy to extend

Command-line switch between different work-flow managers / batch systems

Define Stage:
fast_carpenter.Define

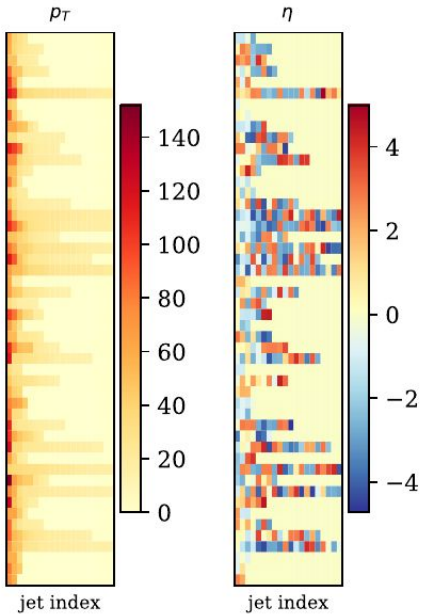


From Joosep Pata's talk at PyHEP19

```
- Muon_Pt: "sqrt(Muon_Px ** 2 + Muon_Py ** 2)"
- IsoMuon_Idx: (Muon_Iso / Muon_Pt) < 0.10
- HasTwoMuons: NIsoMuon >= 2
```

- Simple operations
- Preserve the "jaggedness"

Define Stage:
fast_carpenter.Define



From Joosep Pata's talk at PyHEP19

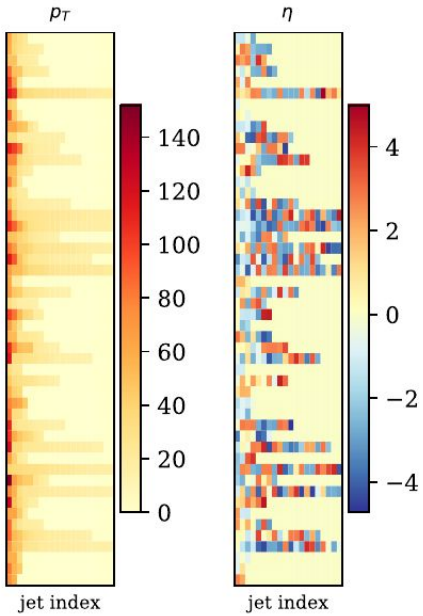
```
- Muon_Pt: "sqrt(Muon_Px ** 2 + Muon_Py ** 2)"
- IsoMuon_Idx: (Muon_Iso / Muon_Pt) < 0.10
- HasTwoMuons: NIsoMuon >= 2
```

- Simple operations
- Preserve the "jaggedness"

Take the Nth object
(on the deepest dimension)

```
- Muon_lead_Pt: {reduce: 0, formula: Muon_Pt}
- Muon_sublead_Pt: {reduce: 1, formula: Muon_Pt}
```

Define Stage:
fast_carpenter.Define



From Joosep Pata's talk at PyHEP19

```
- Muon_Pt: "sqrt(Muon_Px ** 2 + Muon_Py ** 2)"
- IsoMuon_Idx: (Muon_Iso / Muon_Pt) < 0.10
- HasTwoMuons: NIsoMuon >= 2
```

- Simple operations
- Preserve the "jaggedness"

Take the Nth object
(on the deepest dimension)

```
- Muon_lead_Pt: {reduce: 0, formula: Muon_Pt}
- Muon_sublead_Pt: {reduce: 1, formula: Muon_Pt}
```

```
- NIsoMuon:
    formula: IsoMuon_Idx
    reduce: count_nonzero

- IsoMuPtSum:
    formula: Muon_Pt
    reduce: sum
    mask: IsoMuon_Idx
```

- Reduce dimensionality with a function
- Mask out objects in the event

Select events

fast_carpenter.CutFlow

```
DiMu_controlRegion:
    weights: {nominal: weight}
    selection:
        All:
          - {reduce: 0, formula: Muon_pt > 30}
          - leadJet_pt > 100
          - DiMuon_mass > 60
          - DiMuon_mass < 120
          - Any:
            - nCleanedJet == 1
            - DiJet_mass < 500
            - DiJet_deta < 2
```

Remove events from subsequent stages

Produces a cut-flow summary table

- Weighted / raw counts

Selection is specified as nested dictionaries of **`All`**, **`Any`** and a list of expressions

Individual cuts use same scheme as variable definition

Fill a histogram

fast_carpenter.BinnedDataFrame

```
NumberMuons:
  binning:
      - {in: NMuon}
      - {in: NIsoMuon}
  weights: [EventWeight, EventWeight_NLO_up]

DiMuonMass:
  binning:
      - in: DiMuon_Mass
        bins: {low: 60, high: 120, nbins: 60}
  weights: {weighted: EventWeight}
```

- How to bin the data
  - Arbitrary dimensions
- Event weights
- Output written to disk as Pandas dataframe

Output of BinnedDataframe stage

```
>>> import pandas as pd
>>> df = pd.read_csv('tbl_dataset.dimu_mass--weighted.csv')
>>> print(df.groupby('dataset').nth([0, 1, 2]).set_index('dimu_mass', append=True))
                              n  weighted:sumw  weighted:sumw2
dataset    dimu_mass
data       (-inf, 60.0]  993.0            NaN             NaN
           (60.0, 61.0]   38.0            NaN             NaN
           (61.0, 62.0]   25.0            NaN             NaN
dy         (-inf, 60.0]  821.0     655.570801     1017.549133
           (60.0, 61.0]   56.0      23.963226       12.091142
           (61.0, 62.0]   56.0      25.572840       13.094129
qcd        (-inf, 60.0]    0.0       0.000000        0.000000
           (60.0, 61.0]    0.0       0.000000        0.000000
           (61.0, 62.0]    0.0       0.000000        0.000000
single_top (-inf, 60.0]   32.0       1.741041        0.100682
           (60.0, 61.0]    1.0       0.065288        0.004263
           (61.0, 62.0]    1.0       0.005831        0.000034
ttbar      (-inf, 60.0]   49.0      11.392980        3.072051
           (60.0, 61.0]    3.0       0.840432        0.236490
           (61.0, 62.0]    2.0       0.319709        0.075986
wjets      (-inf, 60.0]    1.0       0.311917        0.097292
           (60.0, 61.0]    0.0       0.000000        0.000000
           (61.0, 62.0]    0.0       0.000000        0.000000
ww         (-inf, 60.0]   61.0       3.600221        0.221474
           (60.0, 61.0]    1.0       0.063284        0.004005
           (61.0, 62.0]    2.0       0.102053        0.005617
wz         (-inf, 60.0]   15.0       0.320914        0.007842
           (60.0, 61.0]    2.0       0.053328        0.001424
           (61.0, 62.0]    0.0       0.000000        0.000000
zz         (-inf, 60.0]   47.0       0.360053        0.002981
           (60.0, 61.0]    0.0       0.000000        0.000000
           (61.0, 62.0]    0.0       0.000000        0.000000
```

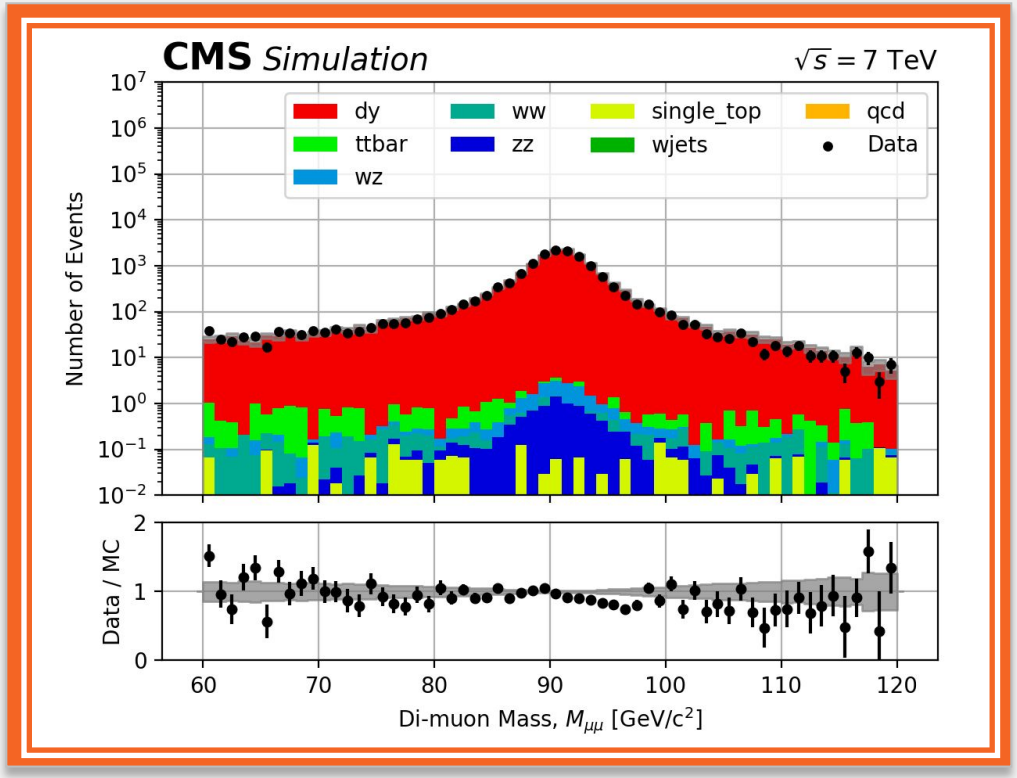
Showing only first three rows for each dataset (using groupby operation)

BinnedDataframes into plots

- Plot on the right with:

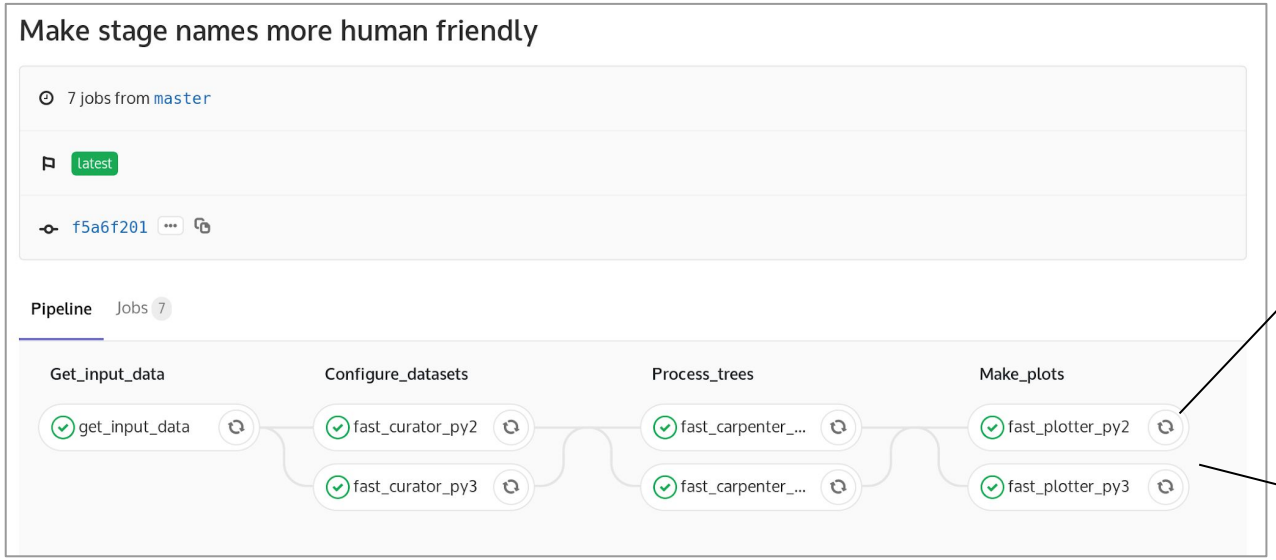
```
fast_plotter -y log \
-c plot_config.yml \
-o tbl_*.csv
```

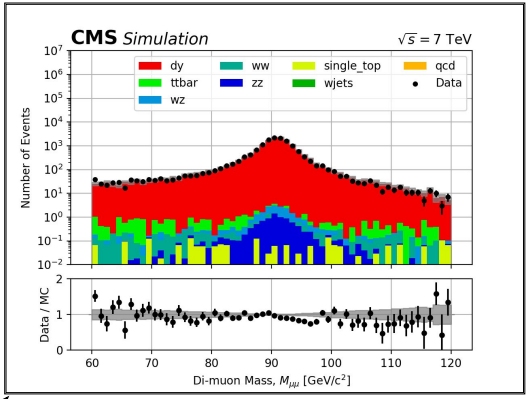
- YAML config:
  - Colour scheme, axis labels
  - Dataset definition
  - Annotations
  - Legend

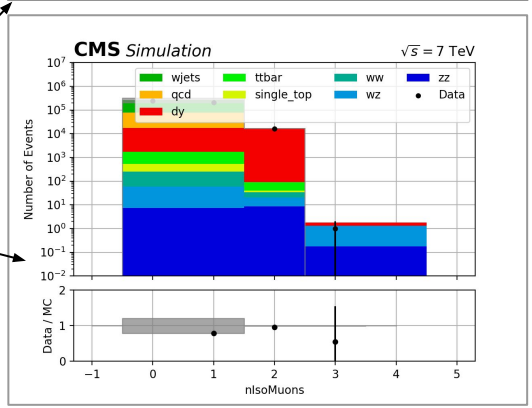


Plot of DiMuonMass using binned dataframe from fast-carpenter stage

“Analysis in a CI pipeline”







- To run this:
  - Demo analysis in a pipeline
  - The gitlab-ci config
  - Script tying the commands together
- Feasibility for huge datasets unclear, but can happily manage subsets of data for testing

Point 4:

It’s easy to build and share new packages: a bunch of people in a room can do it!

Wrapping up

Summary

Particle physics faces major computing challenges

- Lots of data
- Fewer relative resources

Python is a first class analysis language

- E.g. industry, astrophysics
- We seem to be at a tipping point within particle physics

Many new approaches to integrate HEP analyses with other tools

- PyHEP and scikit-hep projects
- Columnar Data Analysis

Exploring new options together in a group is the best way to learn, e.g. FAST-HEP

- And our resulting tools are now helping several experiments

Links to talks that inspired this

Andrea Rizzi: CHEP 2019
https://indico.cern.ch/event/773049/contributions/3581369/attachments/1940586/3217540/Rizzi_CHEP.pdf

Jim Pivarski: CHEP 2018 plenary:
https://indico.cern.ch/event/587955/contributions/3012337/attachments/1683637/2706186/pivarski-chep-analysistools.pdf

Jim Pivarski: CHEP 2018 parallel:
https://indico.cern.ch/event/587955/contributions/2937525/attachments/1678398/2695563/pivarski-chep-columnardata.pdf

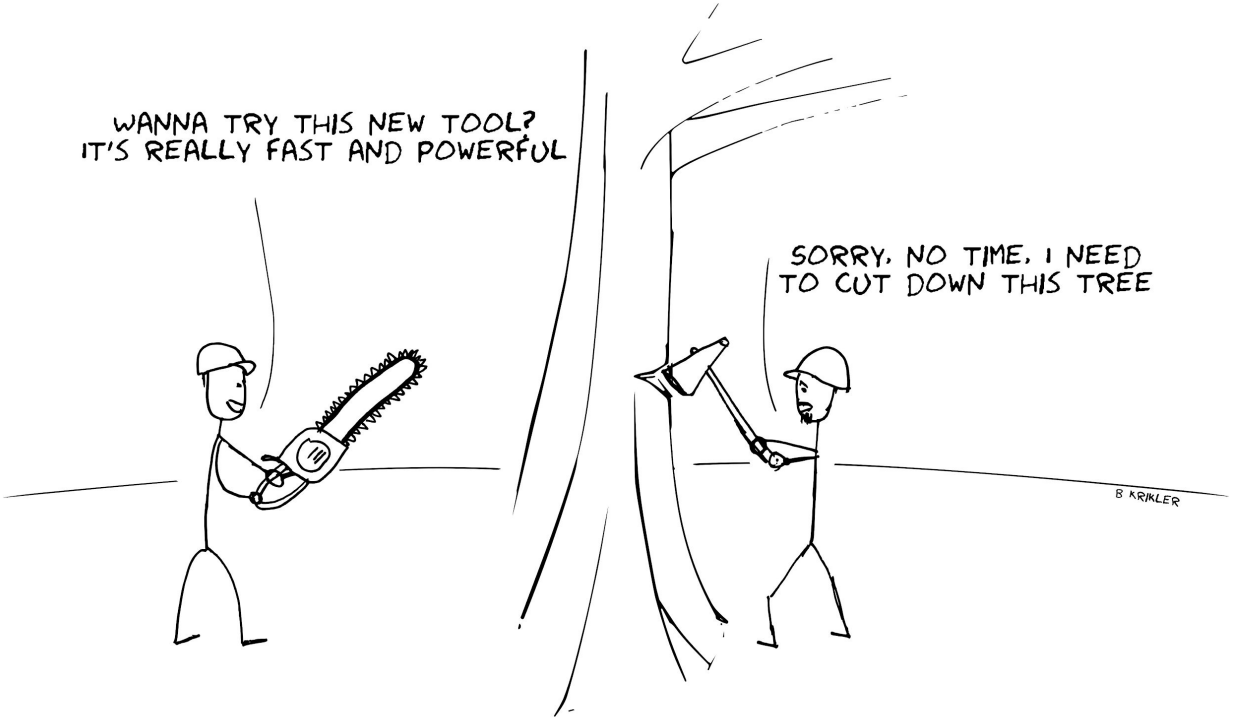
Jake VanderPlas: PyCon 2017
https://speakerdeck.com/jakevdp/the-unexpected-effectiveness-of-python-in-science

Jake VanderPlas: PyCon 2018
https://speakerdeck.com/jakevdp/seven-strategies-for-optimizing-numerical-code

Thank You

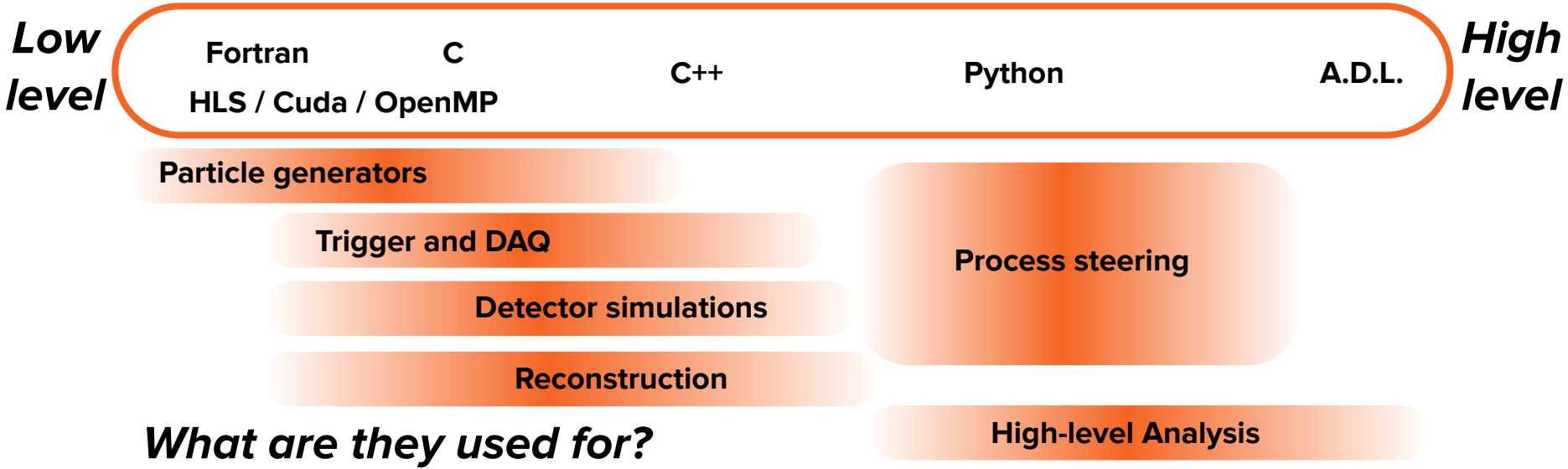
b.krikler@cern.ch

@benkrikler



The future HEP code landscape (?)

The future HEP code landscape (?)



The future HEP code landscape (?)

Who needs to know them?

1st year HEP PhD student

Finishing HEP PhD student

Applied / detector PhD student

Analysis teams

Sims / reconstruction experts

Low level

Fortran C

HLS / Cuda / OpenMP

C++

Python

A.D.L.

High level

Particle generators

Trigger and DAQ

Process steering

Detector simulations

Reconstruction

What are they used for?

High-level Analysis

Jupyter Notebook?

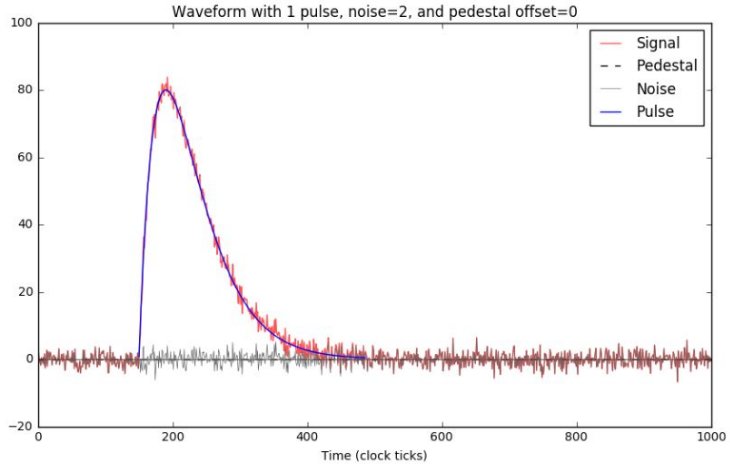
Waveforms will contain multiple components:

- Noise
- Pedestal
- One or more actual signal pulses

Here we assume that the shape of a signal pulse is given by the expression: $f(x;\tau) = xe^{1-x/\tau}$

```
In [3]: wave=Waveform([[150,80]],noise=2,pedestal=0)
        wave.plot_all(show_noise=True)
        plt.legend()
```

```
Out[3]: <matplotlib.legend.Legend at 0x7fb5b6ff8860>
```



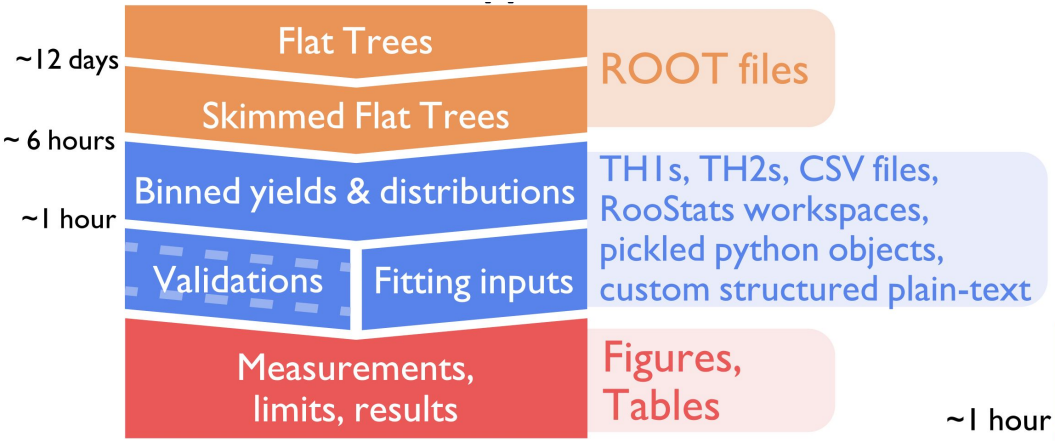
Template pulse

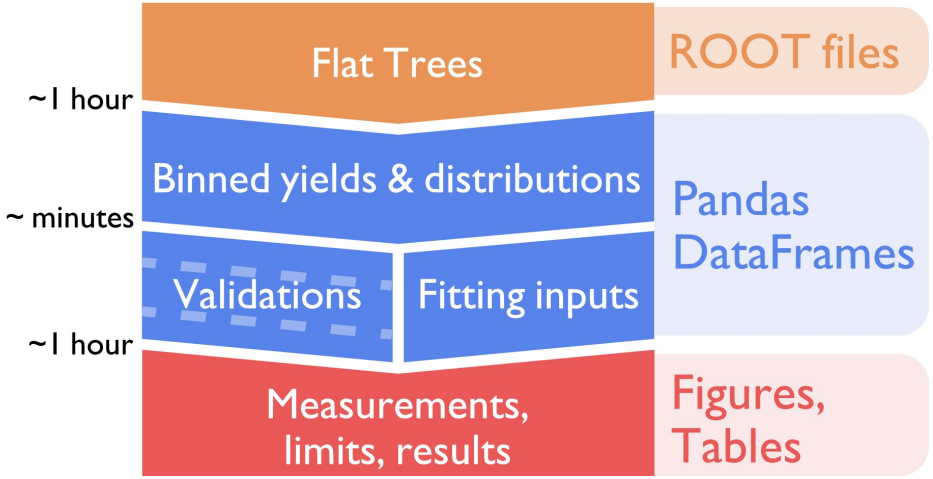
Now we set up our template pulse. We cheat here and use the analytic expression that we know is being used to generate the pulses, but in a real situation this would be a sizeable task, involving pulse registration and averaging.

We also fix all pulse shaping times from here on, to 50 ticks.

- Great:
  - Mixing code, documentation, and results
- Bad:
  - Code can still be dense
  - Scaling to full analysis?
  - Connecting to batch system tricky
  - Version control
- Carpenter can be used via Python API: provide python dicts instead of YAML
  - Addresses some of bad points above

Streamlining analysis

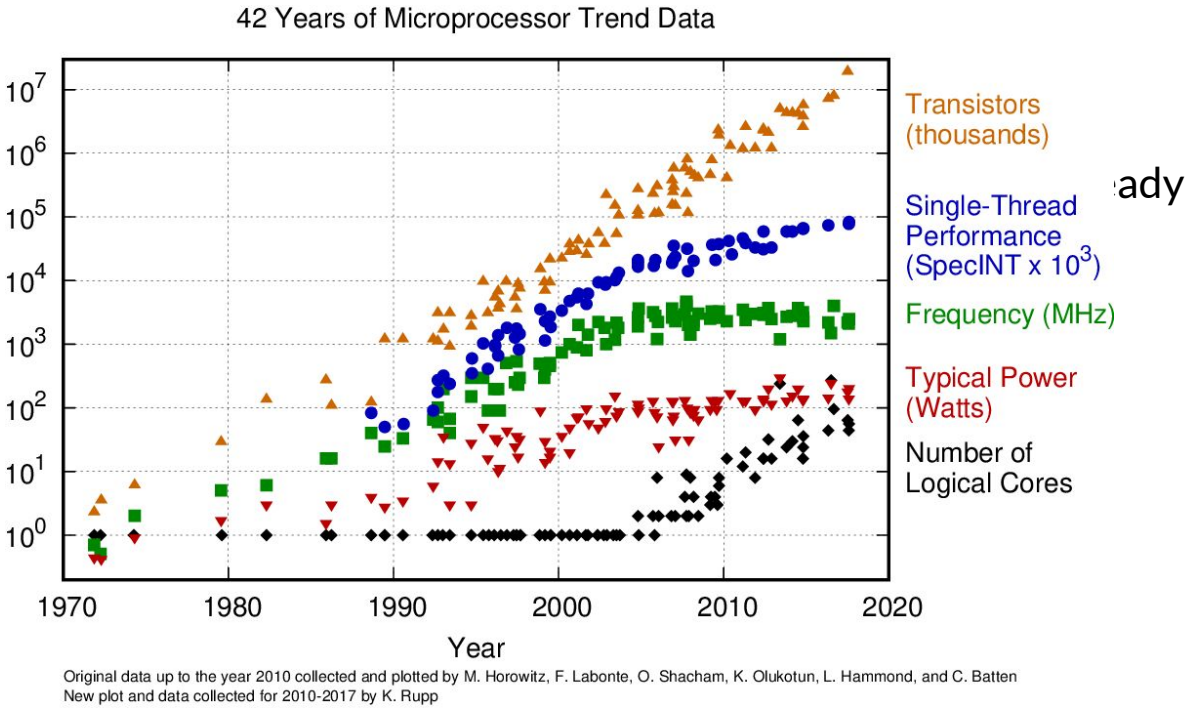




Declarative programming

- Declarative languages the **user says WHAT**, the **interpretation decides HOW**
- User gives up flow control:
  - Cannot do: “Loop over each event, add this to that if something is true, etc”
- Allows:
  - More concise description
  - Fewer bugs
  - Easier to reproduce and share
  - Optimisation behind the scenes

Processing trends



ady

https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

What is Pandas?

- Programmatic tables, built on numpy
- A staple of data science
- https://pandas.pydata.org/

```
A = ['foo', 'bar', 'foo', 'bar']
B = ['one', 'one', 'two', 'three']
C = np.random.randn(4)
D = np.random.randn(4)

df = pd.DataFrame({"A": A, "B": B,
                   "C": C, "D": D})
```

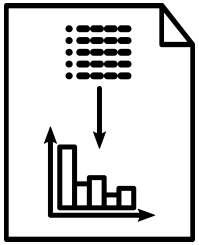
```
df
```

	A	B	C	D
0	foo	one	-0.678386	0.072926
1	bar	one	-0.338564	-1.038362
2	foo	two	0.527912	-0.478806
3	bar	three	-0.237991	-1.296666

```
df.set_index(["A", "B"])
```

		C	D
A	B		
foo	one	-0.678386	0.072926
bar	one	-0.338564	-1.038362
foo	two	0.527912	-0.478806
bar	three	-0.237991	-1.296666

Step 3:
fast_plotter
fast_datacard



Plotting and postprocessing

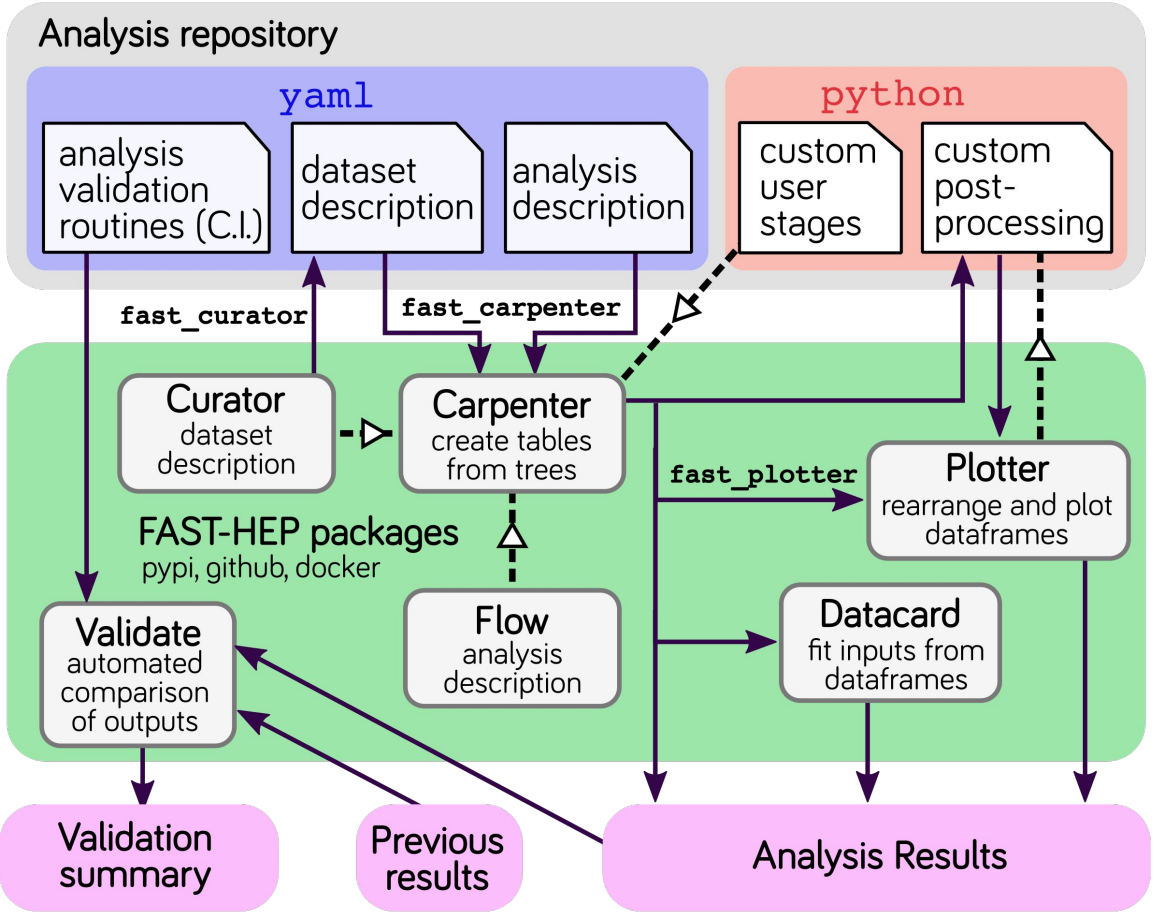
fast-plotter:

- Easy to produce basic plots, tools to support final publication-quality
- Command-line tool with reasonable defaults and simple configuration

fast-datacard:

- Bring resulting DataFrames into CMS' Combine fitting procedures

Interplay in a typical user's analysis repo



Describe what to do with the data

What type of action to take at each step:

- Stage1 = A built-in stage of fast-carpenter
- Stage2 = A stage imported from a python module
- IMPORT = Import a list of stages and their descriptions from another YAML file

Configure each named stage above

```yaml
stages:
  - Stage1: StageFromBackend
  - Stage2: module.that.provides.some.Stage
  - IMPORT: "{this_dir}/another_description.yaml"


Stage1:
  keyword: value
  another_keyword: [a, list, of, values]


Stage2:
  arg1:
      takes: ["a", "dict"]
      with: 3
      different: keys
```

Scikit-validate

- Luke's package grown out of FAST hack-shops
- Predominantly used on LZ so far
- Interested from various people in the room to use it

User-defined stages

```
stages:
  - BasicVars: fast_carpenter.Define
  - DiMuons: cms_hep_tutorial.DiObjectMass
  - Histogram: BinnedDataframe

...

DiMuons:
    mask: IsoMuon_Idx
```

- Carpenter should provide most commonly needed stages
- But if it doesn't: can define your own
  - Break out of declarative YAML to full, imperative python
- Any importable python class with the correct interface
- Keep separation of analysis decision from data-flow

User-defined stages

```
def event(self, chunk):
    # Get the data as a pandas dataframe
    px, py, pz, energy = chunk.tree.arrays(self.branches, outputtype=tuple)

    # Rename the branches so they're easier to work with here
    if self.mask:
        mask = chunk.tree.array(self.mask)
        px = px[mask]
        py = py[mask]
        pz = pz[mask]
        energy = energy[mask]

    # Find the second object in the event (which are sorted by Pt)
    has_two_obj = px.counts > 1

    # Calculate the invariant mass
    p4_0 = TLorentzVectorArray(px[has_two_obj, 0], py[has_two_obj, 0],
                               pz[has_two_obj, 0], energy[has_two_obj, 0])
    p4_1 = TLorentzVectorArray(px[has_two_obj, 1], py[has_two_obj, 1],
                               pz[has_two_obj, 1], energy[has_two_obj, 1])
    di_object = p4_0 + p4_1

    # insert nans for events that have fewer than 2 objects
    masses = np.full(len(chunk.tree), np.nan)
    masses[has_two_obj] = di_object.mass

    # Add this variable to the tree
    chunk.tree.new_variable(self.out_var, masses)
    return True
```

Hack-shop=

½ ***hack***athon + ½ work***shop***

- Talks to set the scene, get everyone up to speed, layout goals
  - Given newcomers: Today will also be walkthrough / tutorial
- Focussed hacking: people “in a room” for a couple of days
  - e.g. “play” with setting up an analysis using these tools
- Feel free to ask questions at any time
  - Collaborative not competitive like traditional hackathon
  - Slack or Zoom

Output of CutFlow stage

```
>>> import pandas as pd
>>> pd.read_csv("cuts_EventSelection-weighted.csv", header=[0, 1], index_col=[0, 1, 2])
                                                  passed_incl             passed_excl             totals_excl
                                                   unweighted    EventWeight  unweighted    EventWeight  unweighted    EventWeight
dataset    depth cut
data       0     All                                  15995.0   15995.000000     15995.0   15995.000000    469384.0   469384.000000
           1     NIsoMuon >= 2                        16208.0   16208.000000     16208.0   16208.000000    469384.0   469384.000000
                 triggerIsoMu24 == 1                 469384.0  469384.000000     16208.0   16208.000000     16208.0    16208.000000
                 {'formula': 'Muon_Pt > 25', 'reduce': 0}    229710.0  229710.000000     15995.0   15995.000000     16208.0    16208.000000
dy         0     All                                  37263.0   16628.843750     37263.0   16628.843750     77729.0    34115.511719
           1     NIsoMuon >= 2                        37559.0   16829.451172     37559.0   16829.451172     77729.0    34115.511719
                 triggerIsoMu24 == 1                  77729.0   34115.511719     37559.0   16829.451172     37559.0    16829.451172
                 {'formula': 'Muon_Pt > 25', 'reduce': 0}     73374.0   32168.121094     37263.0   16628.843750     37559.0    16829.451172
qcd        0     All                                      0.0       0.000000         0.0       0.000000       142.0    79160.507812
           1     NIsoMuon >= 2                            0.0       0.000000         0.0       0.000000       142.0    79160.507812
                 triggerIsoMu24 == 1                    142.0   79160.507812         0.0       0.000000         0.0        0.000000
                 {'formula': 'Muon_Pt > 25', 'reduce': 0}      16.0    6014.819336         0.0       0.000000         0.0        0.000000
single_top 0     All                                    110.0       5.676235       110.0       5.676235      5684.0      311.622986
           1     NIsoMuon >= 2                          111.0       5.748312       111.0       5.748312      5684.0      311.622986
                 triggerIsoMu24 == 1                   5684.0     311.622986       111.0       5.748312       111.0        5.748312
                 {'formula': 'Muon_Pt > 25', 'reduce': 0}     5278.0     290.494965       110.0       5.676235       111.0        5.748312
ttbar      0     All                                    206.0      47.293686       206.0      47.293686     36941.0     7929.475586
           1     NIsoMuon >= 2                          226.0      51.629749       226.0      51.629749     36941.0     7929.475586
                 triggerIsoMu24 == 1                   4515.0    1001.804932       206.0      47.293686       226.0       51.629749
                 {'formula': 'Muon_Pt > 25', 'reduce': 0}     5067.0    1109.433960       206.0      47.293686       206.0       47.293686
wjets      0     All                                      1.0       0.311917         1.0       0.311917    109737.0   209603.531250
           1     NIsoMuon >= 2                            1.0       0.311917         1.0       0.311917    109737.0   209603.531250
                 triggerIsoMu24 == 1                 109737.0  209603.531250         1.0       0.311917         1.0        0.311917
                 {'formula': 'Muon_Pt > 25', 'reduce': 0}     99016.0  191354.781250         1.0       0.311917         1.0        0.311917
ww         0     All                                    243.0      12.577849       243.0      12.577849      4580.0      229.949570
           1     NIsoMuon >= 2                          244.0      12.639496       244.0      12.639496      4580.0      229.949570
                 triggerIsoMu24 == 1                   4580.0     229.949570       244.0      12.639496       244.0       12.639496
                 {'formula': 'Muon_Pt > 25', 'reduce': 0}     4214.0     212.997131       243.0      12.577849       244.0       12.639496
wz         0     All                                    623.0      13.157759       623.0      13.157759      3367.0      69.927917
           1     NIsoMuon >= 2                          623.0      13.157759       623.0      13.157759      3367.0      69.927917
                 triggerIsoMu24 == 1                   3367.0      69.927917       623.0      13.157759       623.0       13.157759
                 {'formula': 'Muon_Pt > 25', 'reduce': 0}     3125.0      65.436157       623.0      13.157759       623.0       13.157759
zz         0     All                                   1232.0       8.985804      1232.0       8.985804      2421.0      16.922522
           1     NIsoMuon >= 2                         1235.0       8.998816      1235.0       8.998816      2421.0      16.922522
                 triggerIsoMu24 == 1                   2421.0      16.922522      1235.0       8.998816      1235.0       8.998816
                 {'formula': 'Muon_Pt > 25', 'reduce': 0}     2325.0      16.362473      1232.0       8.985804      1235.0       8.998816
```

Resulting cut-flow outputs from EventSelection config on earlier slide

Step 1:
fast_curator

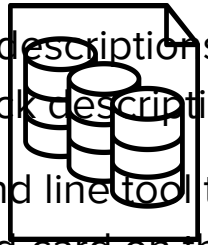
Curator: what files do you want to work on?

Dataset descriptions don’t change often

- Track descriptions in repo, easy to review

Command line tool to help write YAML

- Wild-card on the command line
- Hooks ready for experiment-specific catalogues, e.g. CMS DAS
- Integrate with Rucio (?)



Dataset description

Just how “fast” is this?

On a laptop: as quick as a C++ equivalent

For example, the demo repo:

- fast-carpenter: 6 seconds
- C++ example: 4 seconds

More benchmarks and examples on their way

Many optimisations possible

- caching, DAG optimisation, etc
- started working with Coffea to use them under the hood

Dataset description

```
datasets:
  - eventtype: data
    Files: [input_files/HEPTutorial/files/data.root]
    name: data
    nevents: 469384
  - files:
     - input_files/HEPTutorial/files/dy.root
     - input_files/HEPTutorial/files/dy_2.root
    name: dy
    nevents: 77729
    nfiles: 2

defaults:
  eventtype: mc
  nfiles: 1
  tree: events

import:
  - "{this_dir}/WW.yml"
  - "{this_dir}/WZ.yml"
```

- Each dataset has a list of files
- A unique dataset name

- Default metadata

- Can Import other dataset files
- Build complex nested dataset descriptions

An example set of stages

```
stages:
  # Just defines new variables
  - BasicVars: Define
  # A custom class to form the invariant mass of a
  # two-object system
  - DiMuons: cms_hep_tutorial.DiObjectMass
  # Filled a binned dataframe
  - NumberMuons: fast_carpenter.BinnedDataframe
  # Select events by applying cuts
  - EventSelection: CutFlow
  # Fill another binned dataframe
  - DiMuonMass: BinnedDataframe
```

Depreciation of Python 2

- Python 2.7 support will be withdrawn on 1st January 2020 (it was released 3rd July 2010)
- Key packages have dropped support:
  IPython, Jupyter, matplotlib, numpy, pandas, scikit-learn, XGboost, dask, …
  For LHCb: Ganga

Guido van Rossum - Python Language - PyCon 2016

https://python3statement.org/

Other reasons to use Python 3

- Dictionaries are ordered (CPython 3.6+, Python 3.7+)
- * and ** behave sensibly `test(**dict_1, **dict_2)`
- In my experience, it's been faster!
- print is actually function with kwargs like `sep`, `end` and `flush`
- Separate str/bytes types
- Exception chaining
- Keyword only arguments
- Many little standard library improvements:
  - Recursive globbing, LRU cache, secrets module, Enum

Overall: It's not any one feature, it's just makes everything **quicker, easier and less buggy!**

f-strings

- My number one feature is f-strings (Python 3.6+)

```
1 mass_low = 1890
2 mass_high = 2050
3 cut = f'({mass_low} < D_Mass) & (D_Mass < {mass_high})'
```

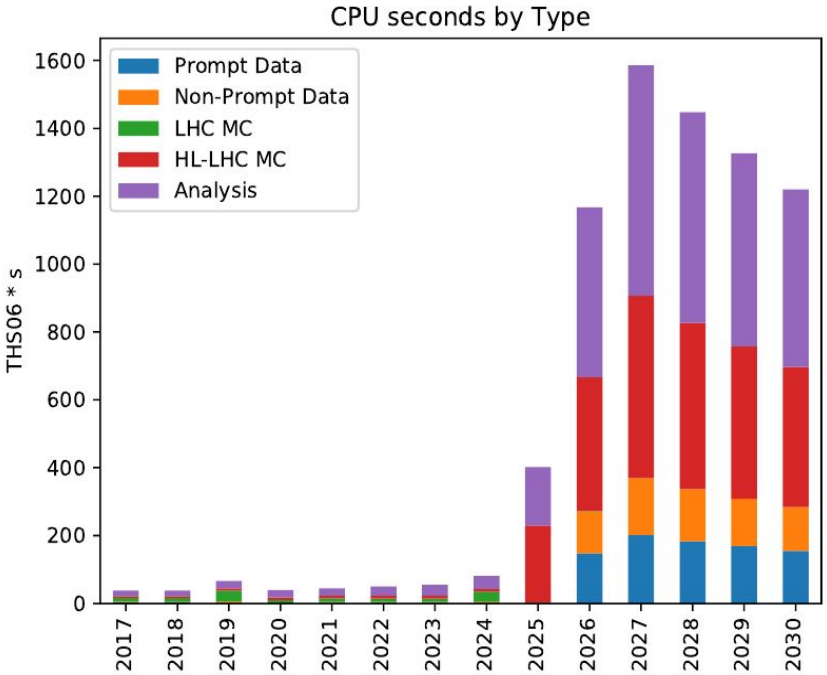
- Why are they better?
  - Compact and easy to read
  - Bugs are generally easier to see
  - Plays nicely with linters

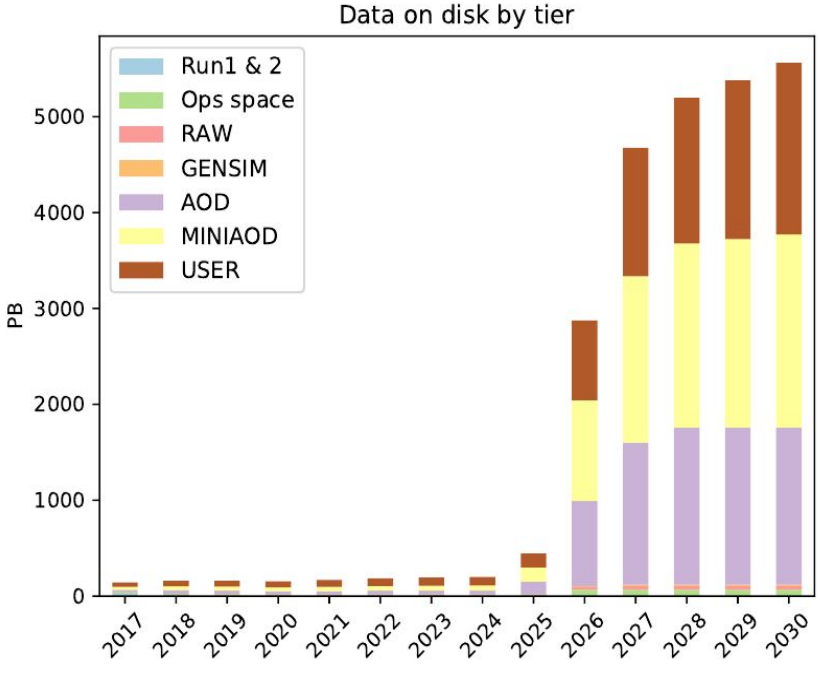
```
5 cut = '(%f < D_Mass) & (D_Mass < %f)' % mass_low, mass_high
6
7 cut = '({0} < D_Mass) & (D_Mass < {1})'.format(mass_low, mass_high)
8
9 cut = '({mass_low} < D_Mass) & (D_Mass < {mass_high})'.format(mass_low, m
```

Python 2 will still work so why care?

- You'll be stuck using old versions of libraries
  - No bug fixes
  - No new features
  - No support: some libraries not automatically close issues that mention Python 2
- You can't use new libraries
  - No new shiny machine learning tools
- Wastes the time of library developers who support both
  - Time can be better spent on support, bugfixes or new features
- If you're ever forced to move, it will only get harder
  - Minor incompatible changes to libraries add up over time
  - It's easier to do many minor updates instead of a few massive ones

Future data volumes: HL-LHC





HSF Roadmap: [DOI: 10.1007/s41781-018-0018-8](https://doi.org/10.1007/s41781-018-0018-8)

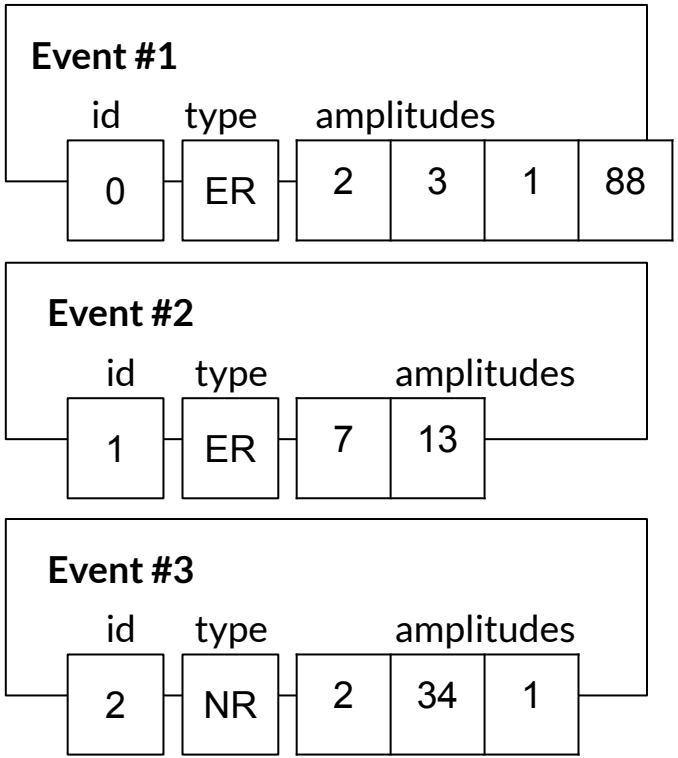
From CMS: “User data” 30% of disk space, “Analysis” 40% of CPU

Traditional HEP data structure

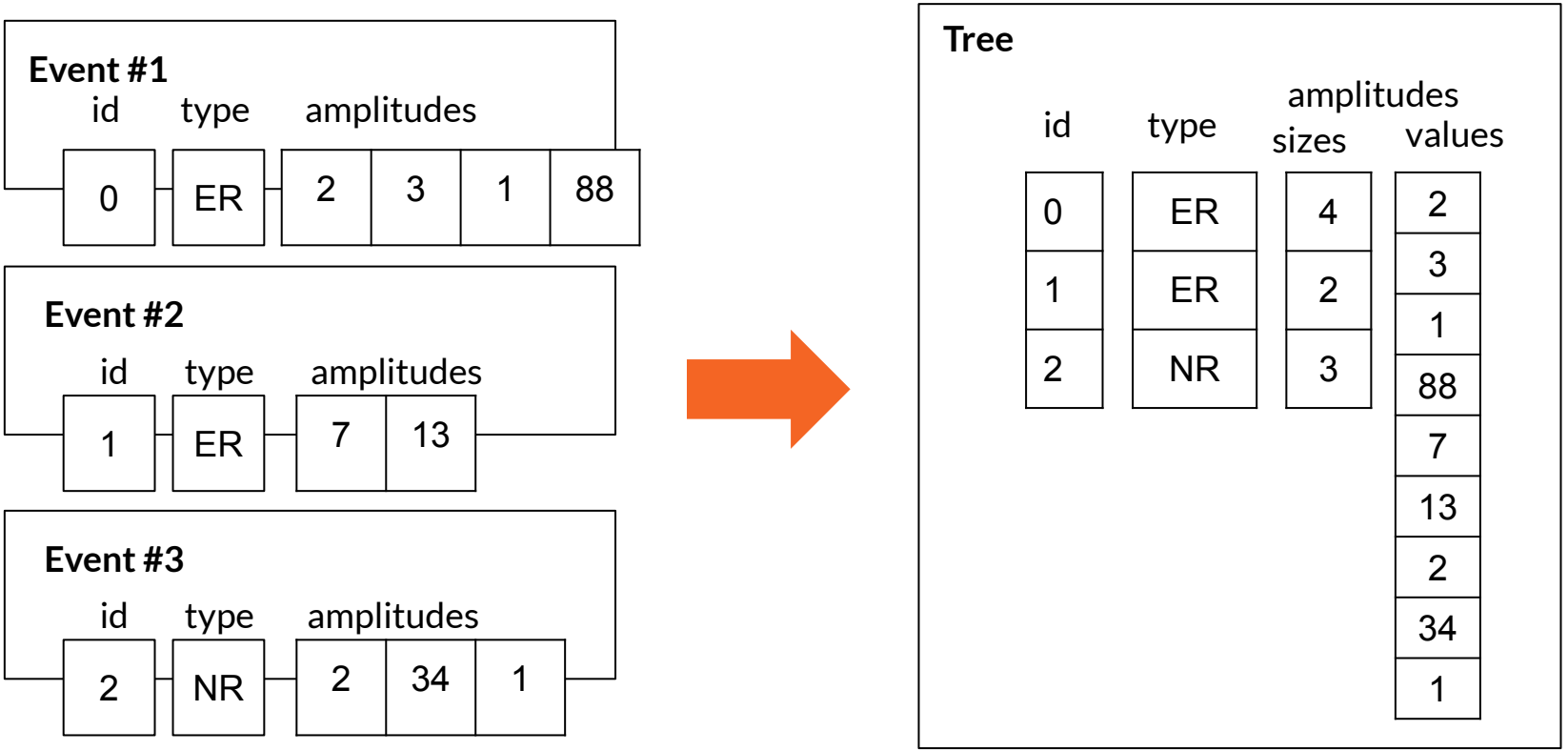
Pseudo-code (not python or c++)

```
Class Event:
       Int id
       Enum type
       Vector<Float> pulse_amplitudes

Function WriteTree():
       TFile file("outfile")
       TTree tree(...)
       Event an_event
       tree.Branch("event", &an_event)

       For each event:
              an_event.id = event number
              an_event.type = some event type
              For each pulse:
                     an_event.pulse_amplitudes.append(some value)

              tree.Fill()
       tree.Write()
```

Builds events that look like:



… which on disk ROOT's split mode makes



ROOT file splitting

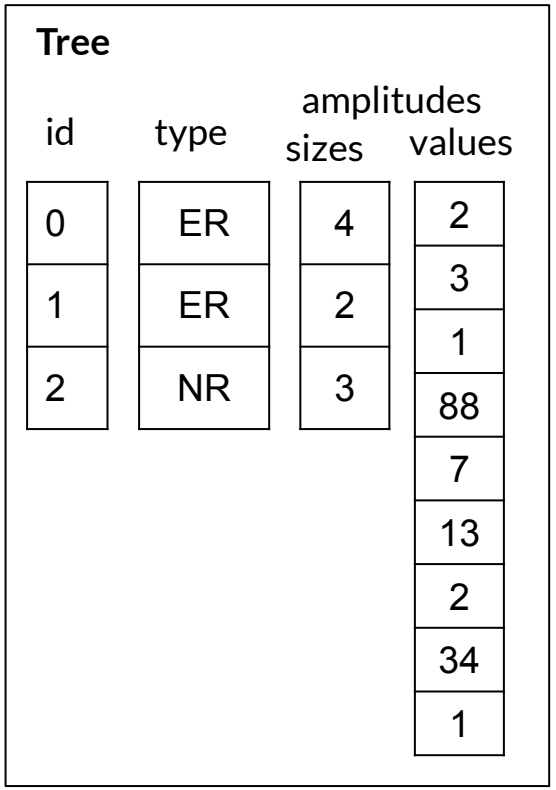
Fails for complex objects e.g. vectors of vectors of floats in each event

Improves compression on disk

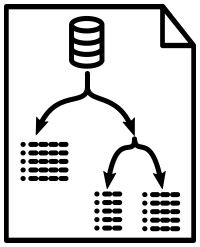
Is why SetBranchStatus speeds up reading back data: only read the branches you want

The on disk layout of split branches is a set of contiguous arrays

- Read all data for a branch directly into a numpy array



Step 2:
`fast_carpenter`



Analysis description

Take your trees and make them into tables

- Just like a carpenter

Table = Pa

Two main

- Histo
- Cutfl

Cover mo

- BUT:

Command

work-flow