School of Physics

हैदराबाद विश्वविद्यालय
University of Hyderabad

columnar analysis - I

Prayag Yadav
CMSLab

15 November 2024

Agenda

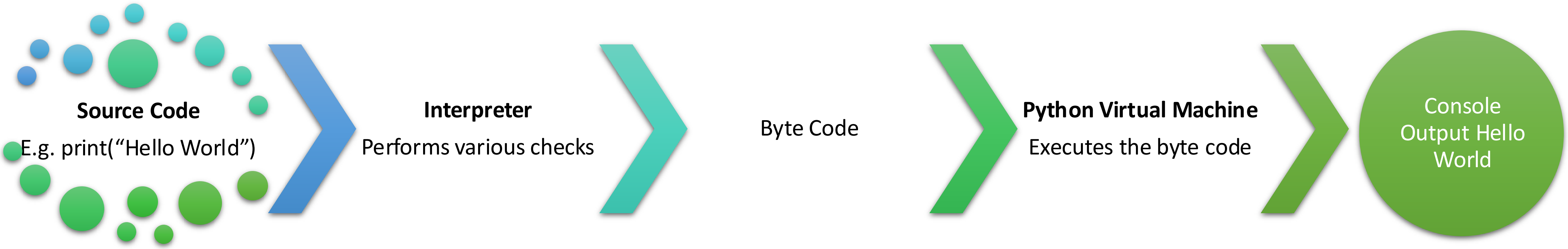
Agenda:

- To understand the tools needed for columnar analysis in HEP.
- Review data structures used in HEP and current analysis paradigms.
- To realize the need for a new way to handle data from HEP.
- Get introduced to libraries like Uproot and Awkward array.

Some general comments

The ABCs of "running" a code in python

- Python is an interpreted language



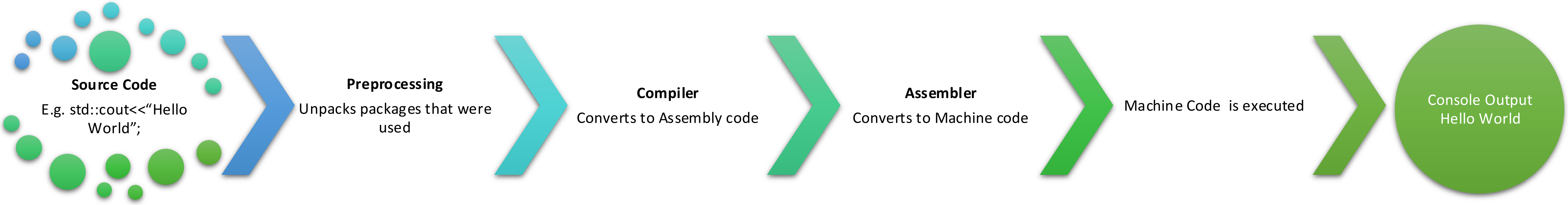
The ABCs of “running” a code in python

- Python is dynamically typed

```
>>> x = 0
>>> type(x)
<class 'int'>
>>> y = 0.2
>>> type(y)
<class 'float'>
>>> z = x+y
>>> type(z)
<class 'float'>
>>> k = "Python"
>>> type(k)
<class 'str'>
```

The ABCs of “running” a code in C++

- C++ is a compiled language



The ABCs of “running” a code in C++

- C++ is statically typed

Test.C:

```
#include<iostream>
int main(){
  x = 2;
  std::cout<<x<<std::endl;
}
```

Output:

```
test.C: In function ‘int main()’:
test.C:3:2: error: ‘x’ was not declared in this scope
    3 |   x = 2;
      |   ^
```

The ABCs of "running" a code in C++

- C++ is statically typed

Test.C:

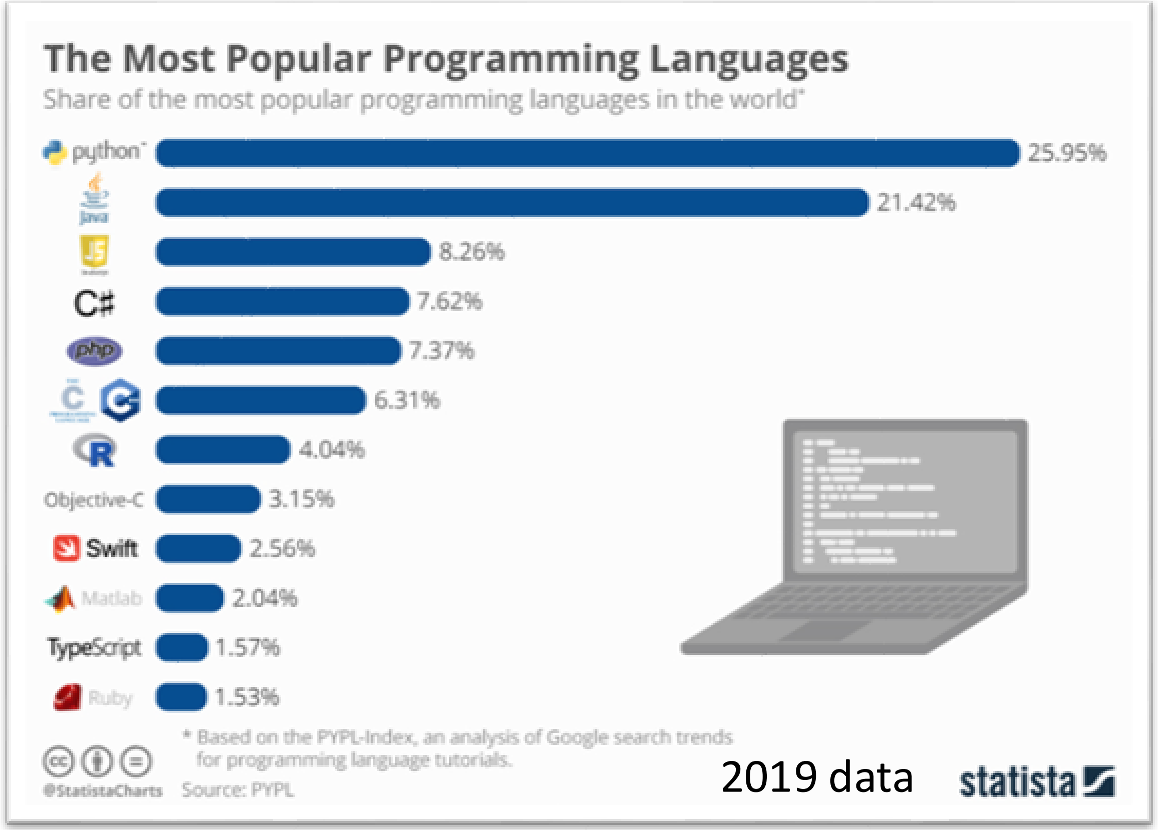
```
#include<iostream>

int main(){

  float x = 2;

  std::cout<<x<<std::endl;

}
```

Output:

2.0

Why should we use python?

- Python continues to be the fastest growing language in the world.
- Python has a high-level syntax which is easy to learn.
- Python has extensive community support with thousands of libraries and modules.
- Strong support for machine leaning applications.
- Good implementation for parallel computing and big data scenarios.



2019 data

Why should we use python?: Libraries

Awkward Array

uproot

NumPy

pandas

scikit learn

PyTorch

TensorFlow

jupyter

Numpy

- Numpy is short for Numerical Python.
- It provides support to handle multidimensional array formated data and process such data fast.
- Numpy also has a lot of functions needed for scientific computing.

```
>>> import numpy as np
>>> a = np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> b = np.arange(1,10).reshape((3,3))
>>> b
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> c = np.arange(10)
>>> c
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.sqrt(c)
array([0. , 1. , 1.41421356, 1.73205081, 2. ,2.23606798, 2.44948974,
2.64575131, 2.82842712, 3.])
```

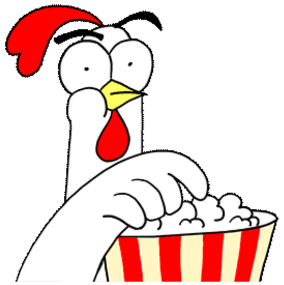
Data in HEP (especially from CERN)

Let’s look inside a root file

```
prayag@PureBook-S14:~/Coffea/data$ ls
ZZTo4e.root
prayag@PureBook-S14:~/Coffea/data$ root –l ZZTo4e.root
>>> root [0]
Attaching file ZZTo4e.root as _file0...
(TFile *) 0x55a01b484850
>>> root [1] .ls
TFile**         ZZTo4e.root
 TFile*         ZZTo4e.root
  KEY: TTree    Events;1        Events
```

Let’s look inside a root file

```
>>> root [2] Events->Print()
******************************************************************************
*Tree    :Events    : Events                                    *
*Entries :  1499093 : Total =      324940562 bytes  File  Size =  161655422 *
*        :          : Tree compression factor =   2.01                  *
******************************************************************************
*Br    0 :run       : run/I                                    *
*Entries :  1499093 : Total  Size=    5998601 bytes  File Size  =     31892 *
*Baskets :       21 : Basket Size=    485888 bytes  Compression= 188.07     *
*............................................................................*
*Br    1 :luminosityBlock : luminosityBlock/i                  *
*Entries :  1499093 : Total  Size=    5998901 bytes  File Size  =     57934 *
*Baskets :       21 : Basket Size=    485888 bytes  Compression= 103.53     *
*............................................................................*
...                    ...                    ...                    ...
...                    ...                    ...                    ...
...                    ...                    ...                    ...
```

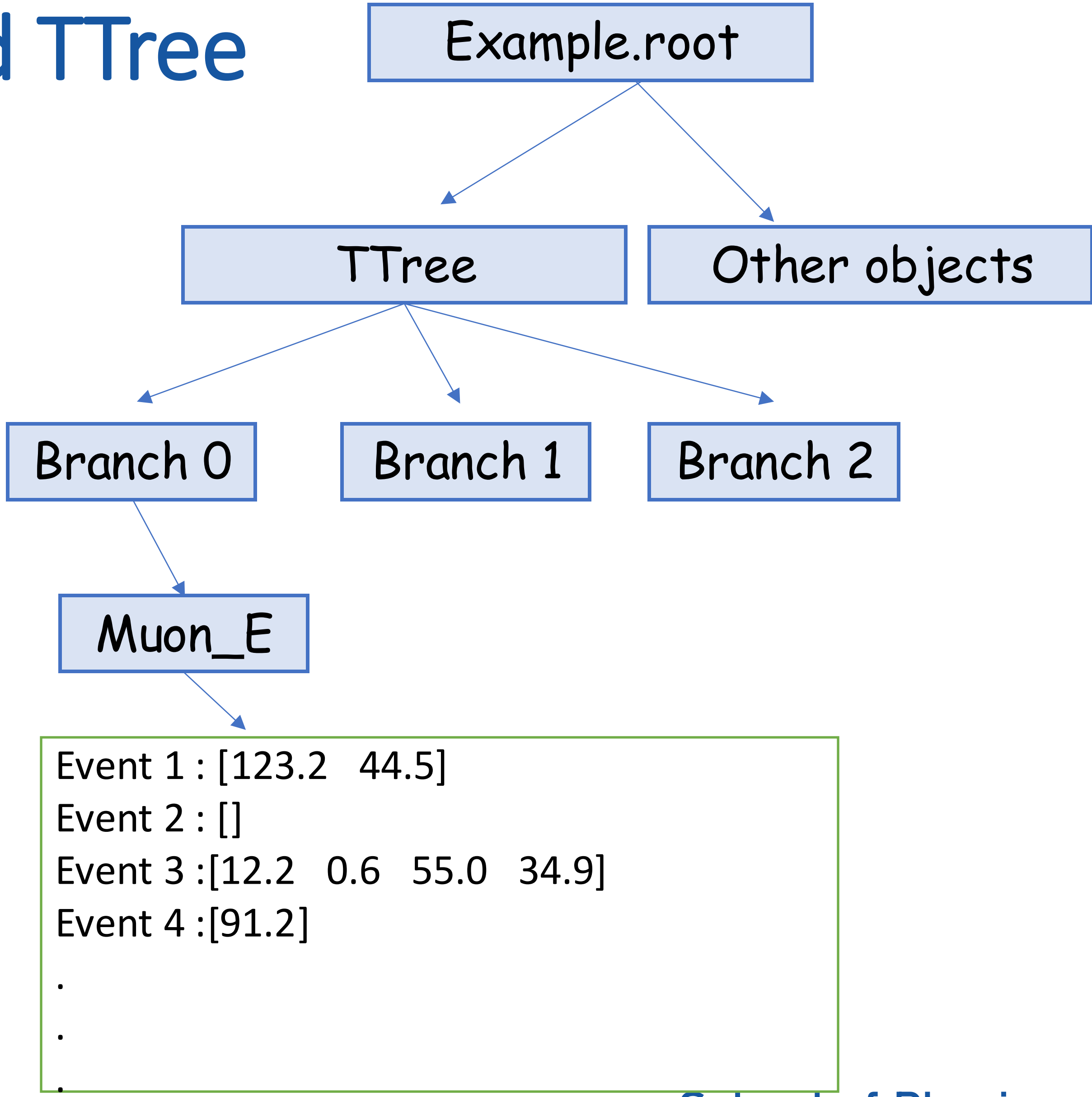


What’s going on?

[Answer in upcoming slides]

The Anatomy of .root files and TTree

- Root files are made specifically for data compression and to handle objects called TTree .
- A tree may contain several Branches.
- A Branch usually contains a long list of events(Record type row wise).
- Each event has a constant list of variables(leaves) stored in their individual columns.
- Each variable contains a data type ,for e.g., 1D array.(Basket)
- These 1D array usually have variable lengths.



Jaggedness and Compression

Jagged Structure

J	A	G	G	E	D			
A	R	R	A	Y	S			
A	R	E						
C	O	N	F	U	S	I	N	G

.
.
.

Columnar structure with offset array for event reference

Muon_E	Offset
123.2	0
44.5	0
12.2	2
0.6	2
55.0	2
34.9	2
91.2	3
.	.
.	.
.	.

Compression in the root files: Demystified

Offset		Muon_E		Muon_eta		Muon_phi		
0	→	123.2	→	123.2	→	123.2	→	…
0	→	44.5	→	44.5	→	44.5	→	…
2	→	12.2	→	12.2	→	12.2	→	…
2	→	0.6	→	0.6	→	0.6	→	…
2	→	55.0	→	55.0	→	55.0	→	…
2	→	34.9	→	34.9	→	34.9	→	…
3	→	91.2	→	91.2	→	91.2	→	…
.		.		.		.		.
.		.		.		.		.
.		.		.		.		.

Same offset for many variables

Each column is a list and has the same data type for all elements.

Saving one single list is cheaper than storing a million lists.

This type of transformation produces highly optimized compression.

The problem with Numpy and other current tools

- Numpy can't handle jagged arrays. Numpy can handle rectangular arrays only.
- Using a different numpy array for each event leads to huge computational penalty.
- Pandas too can't handle jagged arrays.
- When using C++ we expand the compressed root file on RAM and use for loops which might not give us the best efficiency.

Muon_E

Event 1 : [123.2 44.5]	create ndarray
Event 2 : []	create ndarray
Event 3 :[12.2 0.6 55.0 34.9]	create ndarray
Event 4 :[91.2]	create ndrray
.	Have to create millions of arrays!
.	

School of Physics CMSLab

A short speed test

Good old for-loops

```
[1]: import random
     import math

     px = [random.gauss(0, 10) for i in range(100000)]
     py = [random.gauss(0, 10) for i in range(100000)]
```

```
[2]: %%timeit

     pt = []
     for px_i, py_i in zip(px, py):
         pt.append(math.sqrt(px_i**2 + py_i**2))
```

```
23.1 ms ± 1.31 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Numpy

```
[3]: import numpy as np

     px = np.array(px)
     py = np.array(py)
     px, py
```

```
[3]: (array([-5.32934667, 18.72374772,  6.96034087, ..., -6.88514995,
             11.72751268, -5.08110035]),
      array([-5.78379212,  3.70989121, -2.01736772, ..., -0.79598757,
              8.70246878,  4.30970604]))
```

```
[4]: %%timeit

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

```
197 µs ± 3.27 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

pyROOT

```
[5]: import ROOT
```

```
Welcome to JupyROOT 6.28/04
```

```
[6]: ROOT.gInterpreter.Declare('''
void compute_pt(int32_t N, double* px, double* py, double* pt) {
    for (int32_t i = 0; i < N; i++) {
        pt[i] = sqrt(px[i]*px[i] + py[i]* py[i]);
    }
}
''')
```

```
[6]: True
```

Just-In-Time (JIT) compilation

```
[7]: pt = np.empty_like(px)
ROOT.compute_pt(len(px), px, py, pt)
pt
```

```
[7]: array([ 7.8647433 , 19.08774533,  7.24680051, ...,  6.93100902,
        14.60368161,  6.66266816])
```

```
[8]: %%timeit

ROOT.compute_pt(len(px), px, py, pt)
```

```
209 µs ± 5.39 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Numba

Numba

```
[13]: import numba as nb

      @nb.jit(nopython=True)
      def compute_pt(px, py):
          pt = np.empty_like(px)
          for i, (px_i, py_i) in enumerate(zip(px, py)):
              pt[i] = np.sqrt(px_i**2 + py_i**2)
          return pt
```

```
[14]: compute_pt(px, py)
```

```
[14]: array([ 7.8647433 , 19.08774533,  7.24680051, ...,  6.93100902,
             14.60368161,  6.66266816])
```

```
[15]: %%timeit

      compute_pt(px, py)
```

```
163 µs ± 1.09 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Just-In-Time (JIT) compilation

Summarizing our needs

Summarizing our needs

- Need a framework to handle root files in python without the need to install root
- Need a framework to handle jaggedness in arrays
- framework should be intuitive to implement .i.e. should have easily understandable code
- framework should allow a good computational advantage over the already existing tools

Uproot and Awkward Array

Uproot and Awkward Array

- Uproot is a I/O library which facilitates reading and writing of root files without the need to install root C++.

- Awkward array is a tool specifically designed to handle variable sized nested arrays, as is common in HEP.

Uproot and Awkward Array

- Awkward array and Uproot work together to fix all our previous problems.
- Awkward array understands numpy-like idioms. Hence, we should have no problems provided we already know how to use numpy.

Fig: Jai and Veeru

Uproot

A brief demonstration of uproot

```python
import uproot as ur
```

Jagged array with 4bit float elements

```
[4]: upfile = ur.open('data/ZZTo4e.root')
     uptree = upfile['Events']
     uptree.show()
```

```
name                 | typename                 | interpretation
---------------------+--------------------------+-------------------------------
run                  | int32_t                  | AsDtype('>i4')
luminosityBlock      | uint32_t                 | AsDtype('>u4')
event                | uint64_t                 | AsDtype('>u8')
PV_npvs              | int32_t                  | AsDtype('>i4')
PV_x                 | float                    | AsDtype('>f4')
PV_y                 | float                    | AsDtype('>f4')
PV_z                 | float                    | AsDtype('>f4')
nMuon                | uint32_t                 | AsDtype('>u4')
Muon_pt              | float[]                  | AsJagged(AsDtype('>f4'))
Muon_eta             | float[]                  | AsJagged(AsDtype('>f4'))
Muon_phi             | float[]                  | AsJagged(AsDtype('>f4'))
Muon_mass            | float[]                  | AsJagged(AsDtype('>f4'))
Muon_charge          | int32_t[]                | AsJagged(AsDtype('>i4'))
Muon_pfRelIso03_all  | float[]                  | AsJagged(AsDtype('>f4'))
Muon_pfRelIso04_all  | float[]                  | AsJagged(AsDtype('>f4'))
Muon_dxy             | float[]                  | AsJagged(AsDtype('>f4'))
```

A brief demonstration of uproot

```
elec = uptree.arrays(
            [ 'nElectron', 'Electron_pt', 'Electron_eta', 'Electron_phi’, 'Electron_charge' ],
            cut="nElectron >= 2",
            how="zip",
            entry_stop=1000000
            )
```

An Awkward Array

```
<Array [{nElectron: 3, Electron: [, ... ] type='638197 * {"nElectron": uint32, "..."'>
```

A brief demonstration of uproot

Info about the first event

```
elec[0]
```

<Record ... phi: 1.15, charge: 1}]} type='{"nElectron": uint32, "Electron": var ...'>

```
elec[0]["Electron"]
```

<Array [{pt: 11.9, eta: 0.335, ... charge: 1}] type='3 * {"pt": float32, "eta": ...'>

```
elec[0]["Electron"].fields
```

['pt', 'eta', 'phi', 'charge']

```
elec[0]["Electron"]["pt"]
```

<Array [11.9, 26.7, 40.2] type='3 * float32'>

Awkward Array

A brief demonstration of awkward array

```
[127]: events = ur.open("2022-08-01-uproot-awkward-columnar-hats/data/HZZ.root:events") #directly point to the tree
events.show()
```

```
name                 | typename                 | interpretation
---------------------+--------------------------+-------------------------------
NJet                 | int32_t                  | AsDtype('>i4')
Jet_Px               | float[]                  | AsJagged(AsDtype('>f4'))
Jet_Py               | float[]                  | AsJagged(AsDtype('>f4'))
Jet_Pz               | float[]                  | AsJagged(AsDtype('>f4'))
Jet_E                | float[]                  | AsJagged(AsDtype('>f4'))
Jet_btag             | float[]                  | AsJagged(AsDtype('>f4'))
Jet_ID               | bool[]                   | AsJagged(AsDtype('bool'))
NMuon                | int32_t                  | AsDtype('>i4')
Muon_Px              | float[]                  | AsJagged(AsDtype('>f4'))
Muon_Py              | float[]                  | AsJagged(AsDtype('>f4'))
Muon_Pz              | float[]                  | AsJagged(AsDtype('>f4'))
Muon_E               | float[]                  | AsJagged(AsDtype('>f4'))
Muon_Charge          | int32_t[]                | AsJagged(AsDtype('>i4'))
Muon_Iso             | float[]                  | AsJagged(AsDtype('>f4'))
NElectron            | int32_t                  | AsDtype('>i4')
Electron_Px          | float[]                  | AsJagged(AsDtype('>f4'))
```

A brief demonstration of awkward array

Numpy's equivalent is cumbersome and inefficient:

[130]:
```
jagged_numpy = events["Muon_Px"].array(entry_stop = 20, library = "np")
jagged_numpy
```

[130]:
```
array([array([-52.899456,  37.73778 ], dtype=float32),
       array([-0.81645936], dtype=float32),
       array([48.98783  ,  0.8275667], dtype=float32),
       array([22.088331, 76.69192 ], dtype=float32),
       array([45.17132 , 39.750957], dtype=float32),
       array([ 9.22811 , -5.793715], dtype=float32),
       array([12.538717, 29.54184 ], dtype=float32),
       array([34.88376], dtype=float32),
       array([-53.166973,  11.49187 ], dtype=float32),
       array([-67.014854, -18.118755], dtype=float32),
       array([15.983028, 34.684406], dtype=float32),
       array([-70.51191 , -38.028744], dtype=float32),
       array([58.943813], dtype=float32),
       array([-15.587871], dtype=float32),
       array([-122.33012  ,   -1.0597527], dtype=float32),
       array([-46.704155,  39.020023], dtype=float32),
       array([51.29466, 17.45092], dtype=float32),
       array([43.2812], dtype=float32),
       array([-45.923935,  22.549767], dtype=float32),
       array([ 43.293606, -33.28158 ,  -4.376191], dtype=float32)],
      dtype=object)
```

A brief demonstration of awkward array

What if I want the first item in each list as an array?

```
np.array([x[0] for x in jagged_numpy]) #computationally poor since we are iterating here
```

```
array([ -52.899456  ,   -0.81645936,   48.98783   ,   22.088331  ,
         45.17132   ,    9.22811   ,   12.538717  ,   34.88376   ,
        -53.166973  ,  -67.014854  ,   15.983028  ,  -70.51191   ,
         58.943813  ,  -15.587871  , -122.33012   ,  -46.704155  ,
         51.29466   ,   43.2812    ,  -45.923935  ,   43.293606  ],
      dtype=float32)
```

```
jagged_numpy[:,0] #cant even slice since its a dtype = object
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[132], line 1
----> 1 jagged_numpy[:,0] #cant even slice since its a dtype = object

IndexError: too many indices for array: array is 1-dimensional, but 2 were indexed
```

That’s where awkward helps us

A brief demonstration of awkward array

```
jagged_awkward = events["Muon_Px"].array(entry_stop = 20, library = "ak")
jagged_awkward
```

```
<Array [[-52.9, 37.7], ... 43.3, -33.3, -4.38]] type='20 * var * float32'>
```

```
LeadingMuons = jagged_awkward[:,0]
LeadingMuons
```

```
<Array [-52.9, -0.816, 49, ... -45.9, 43.3] type='20 * float32'>
```

```
LeadingMuons.to_numpy()
```

```
array([ -52.899456  ,   -0.81645936,   48.98783   ,   22.088331  ,
         45.17132   ,    9.22811   ,   12.538717  ,   34.88376   ,
        -53.166973  ,  -67.014854  ,   15.983028  ,  -70.51191   ,
         58.943813  ,  -15.587871  , -122.33012   ,  -46.704155  ,
         51.29466   ,   43.2812   ,  -45.923935  ,   43.293606  ],
      dtype=float32)
```

A brief demonstration of awkward array

Why is awkward array more efficient?

- Slicing through data is more computationally inexpensive than modifying any large buffers over and over again.
- Can use numpy-like idioms on json-like data structure(dictionary/hash-map)
- Numpy-like functional broadcasting operations(ufuncs) are valid in awkward array

And they have many more features….

1. To learn more about the uproot and awkward array, please find the link my jupyter notebook : ColumnarHEP
2. To get a deeper understanding of the concepts involved, follow the LPC HATS on Uproot and Awkward array.

Thank you for listening!