hep_tables

G. Watts (UW/Seattle)

Prototype!

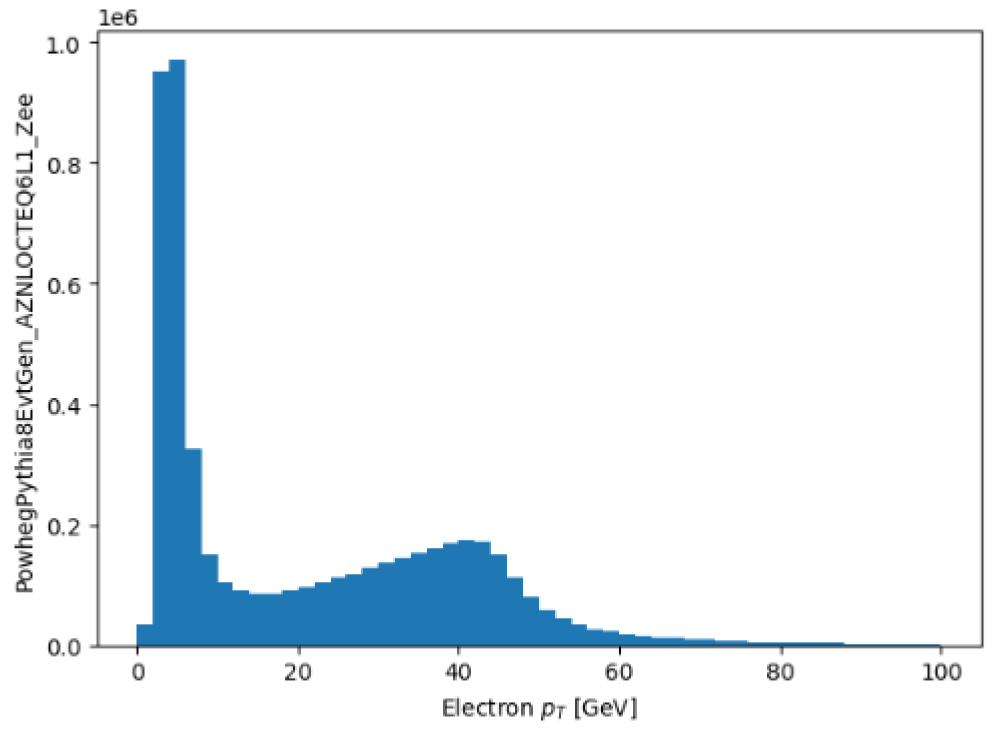
- Been building this to explore some highlevel ideas connected with Analysis Systems, interactive analysis, and…

Probably closer to this…

- Comments on the ideas obviously welcome
- I've made a few design decisions that are wrong
  - Some of them I should have known from my LINQ work!

Goal 1: Easy things should be easy

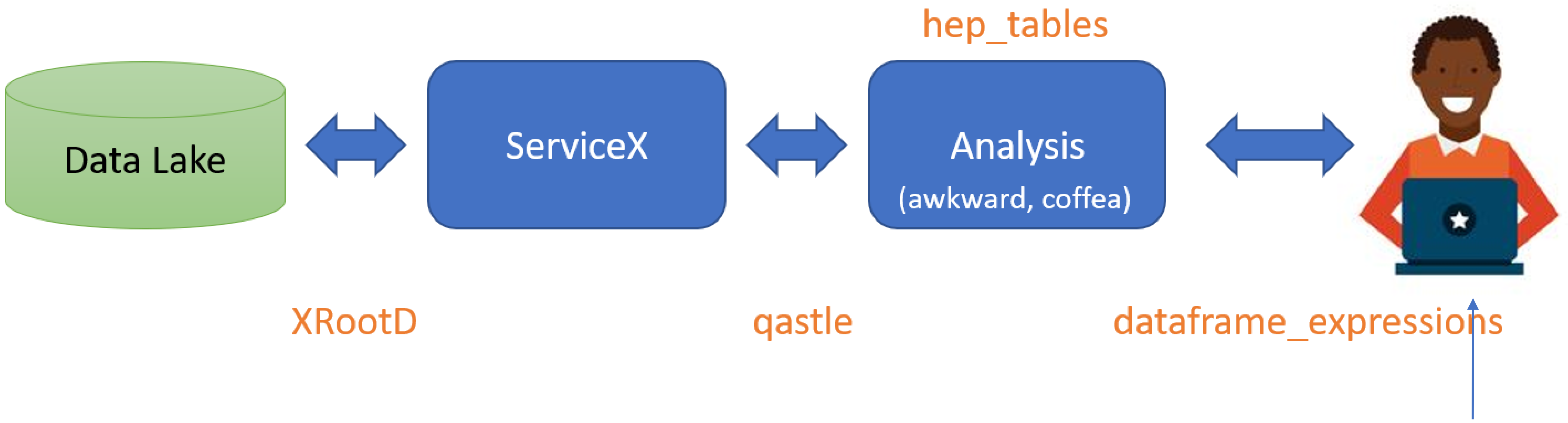
```
1 dataset = EventDataset('localds://mc15_13TeV:mc15_13TeV.361106.PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee.merge.
  df = xaod_table(dataset)
2 pts = df.Electrons('Electrons').pt/1000.0
3 np_pts = make_local(pts)

4 plt.hist(np_pts.flatten(), range=(0, 100), bins=50)
  plt.xlabel('Electron $p_T$ [GeV]')
  _ = plt.ylabel('PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee')
```



1. Onetime boilerplate to indicate the dataset you want
2. Extract the electrons from the "Electrons" xAOD bank, access the p_T property of the object, and turn it from MeV to GeV
3. Get the column of electron p_T from the dataset using ServiceX.
4. Plot it. np_pts is an JaggedArray when it comes back.

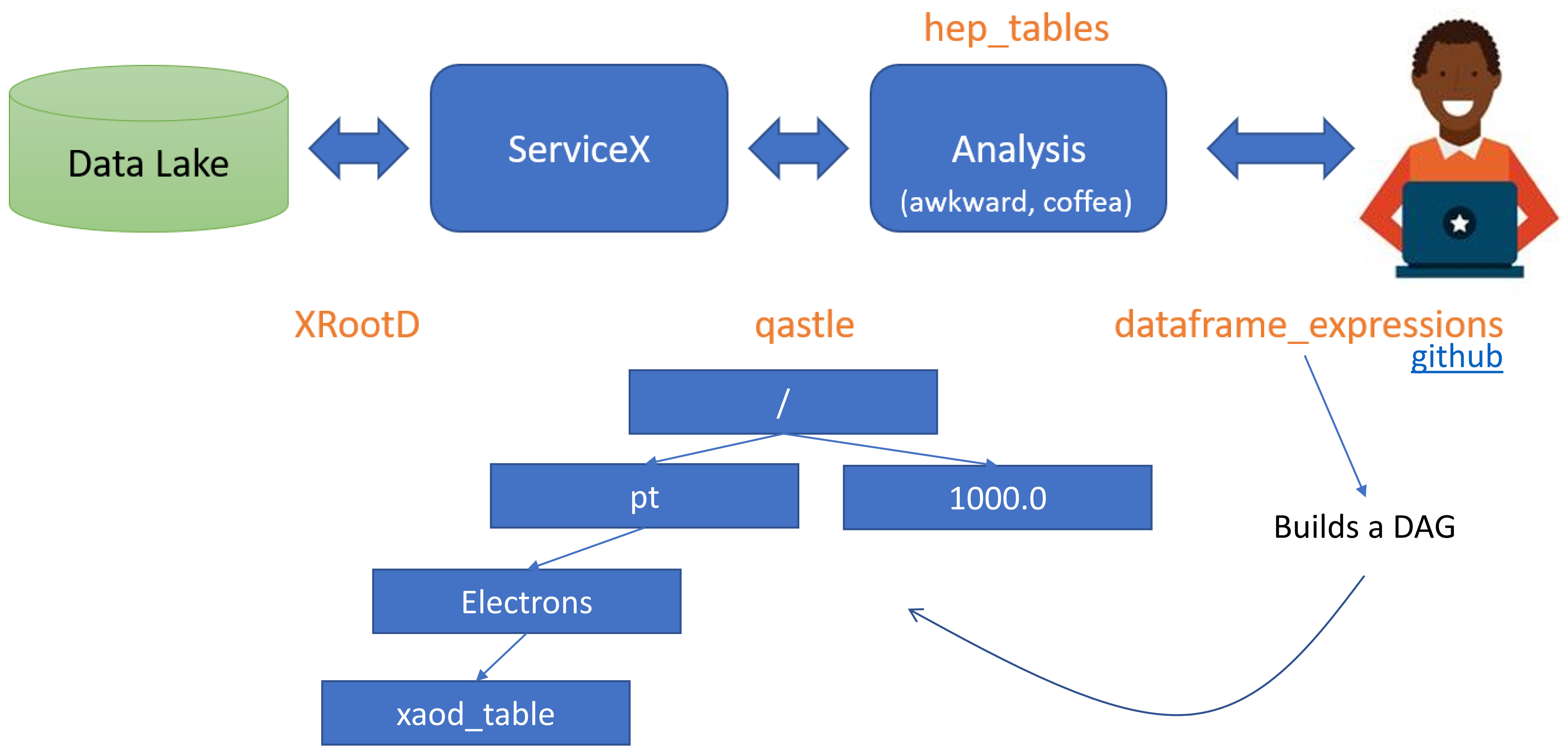
Jupyter Notebook Demo (not binder!)

Where does this fit?

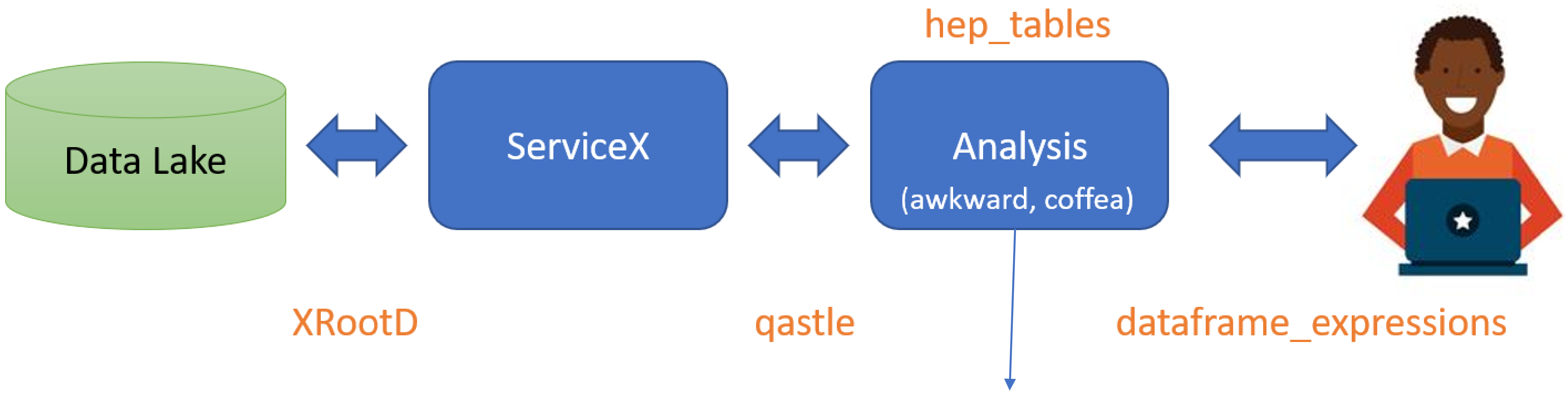


User enters expressions in Jupyter notebook or in python source files.

Where does this fit?



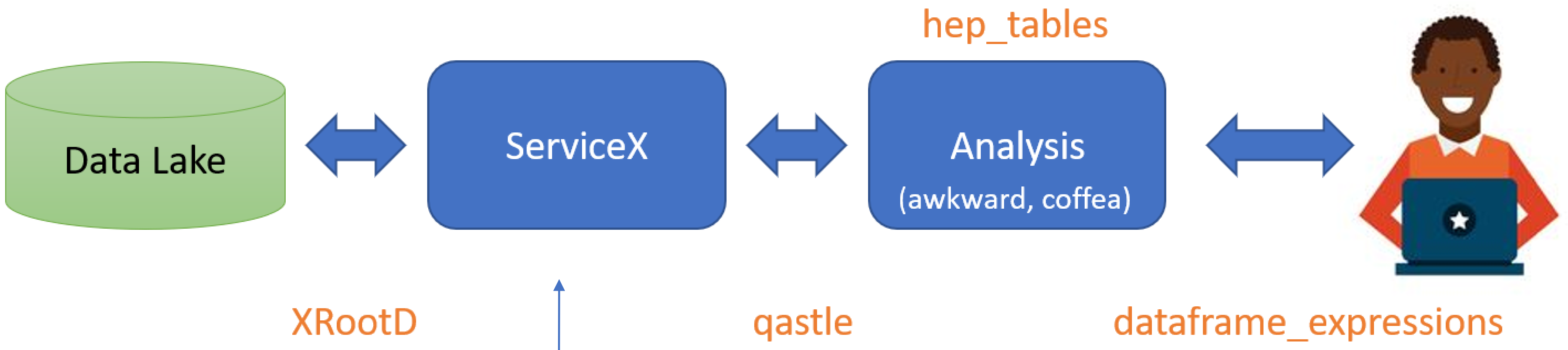
Where does this fit?



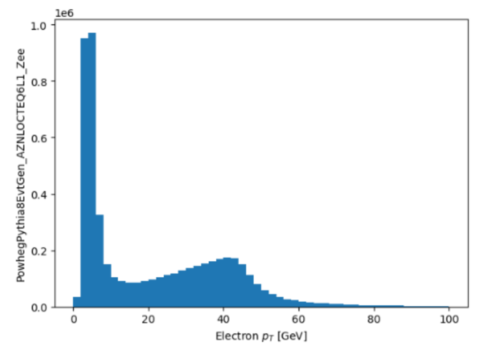
Converts the DAG to [qastle]

```
(Select data_column_source
        (lambda (list Event)
                (list (call (attr (attr Event 'Electrons') 'pt'))
                      (call (attr (attr Event 'Electrons') 'eta'))
                      (call (attr (attr Event 'Electrons') 'phi'))
                      (call (attr (attr Event 'Electrons') 'e'))
                      (call (attr (attr Event 'Muons') 'pt'))
                      (call (attr (attr Event 'Muons') 'eta'))
                      (call (attr (attr Event 'Muons') 'phi'))
                      (call (attr (attr Event 'Muons') 'e')))))
```

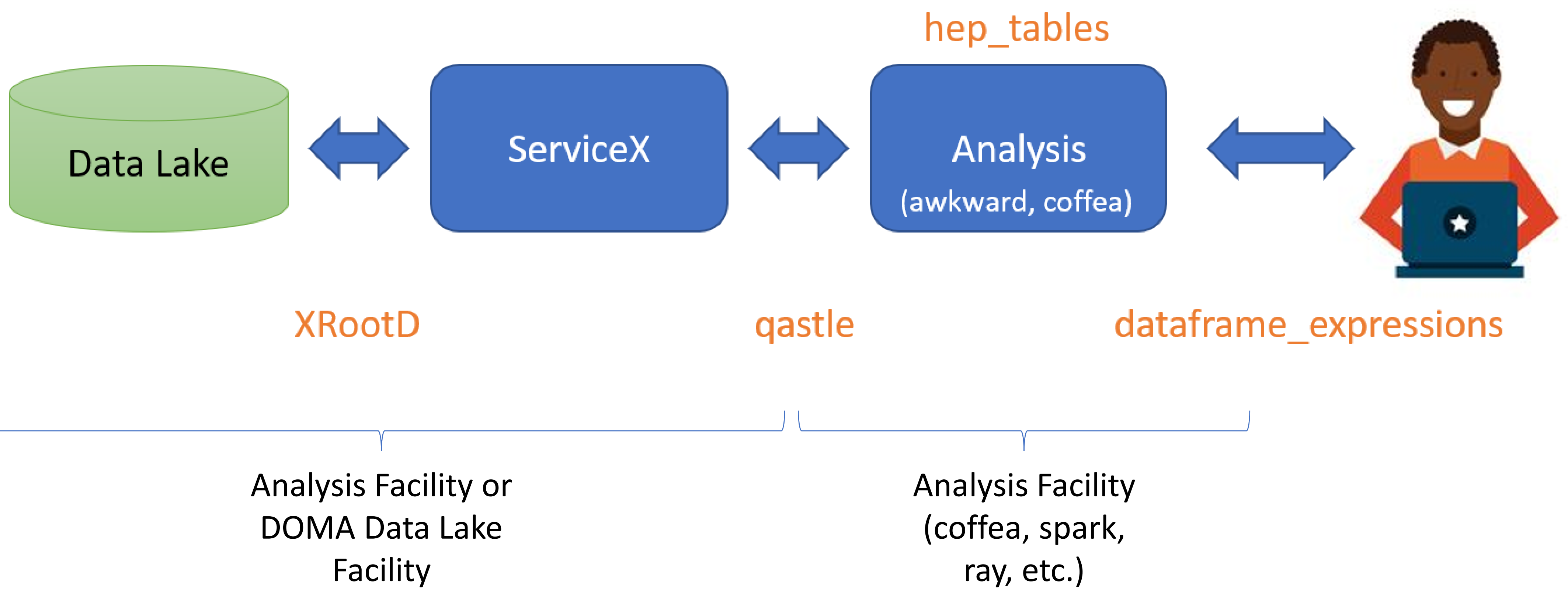
Where does this fit?



ServiceX fetches the datasets, files, etc., extracts requested data, makes some simple cuts, and ships it back.



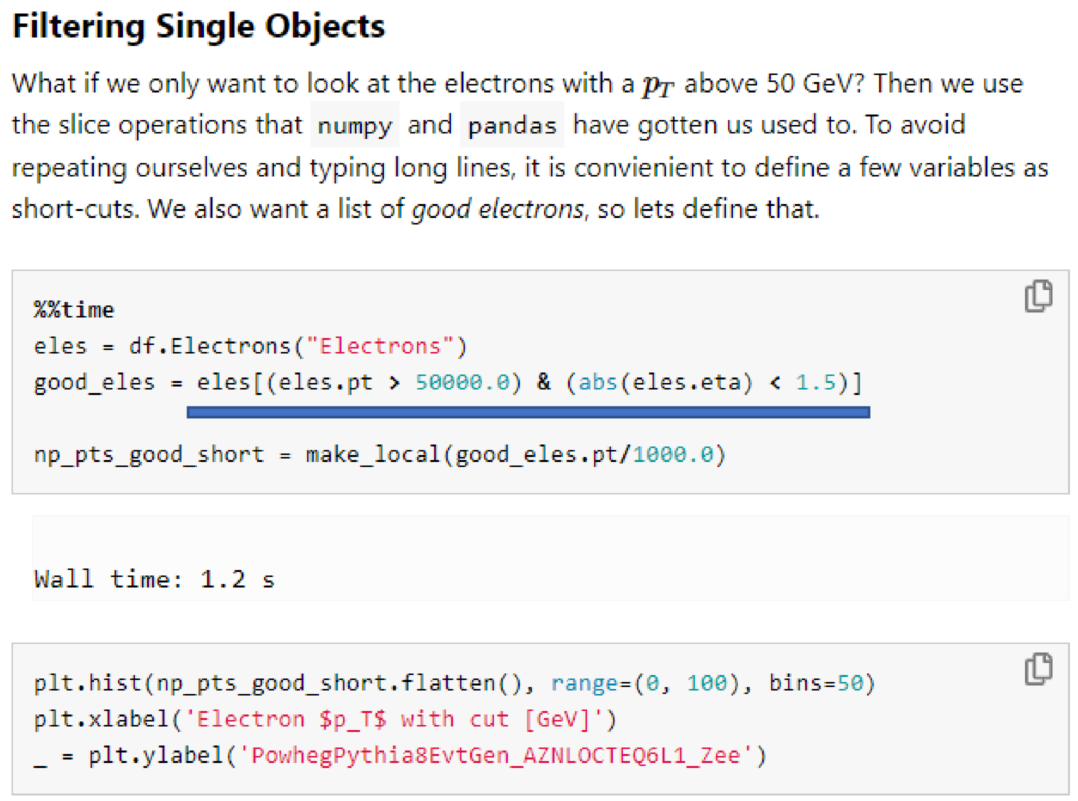
Where does this fit?



Filtering

Pretty much as you'd expect from your experience of using array slicing with pandas and numpy

notebook



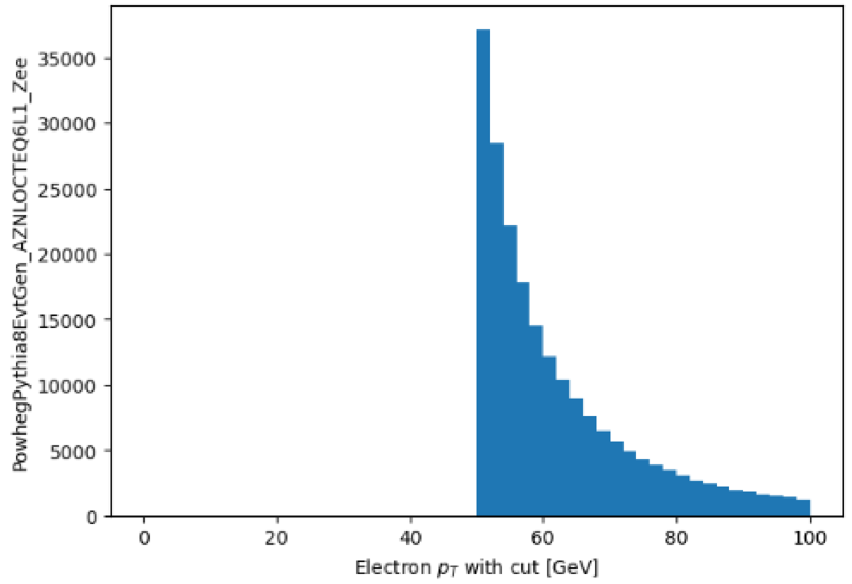
Filtering Single Objects

What if we only want to look at the electrons with a p_T above 50 GeV? Then we use the slice operations that `numpy` and `pandas` have gotten us used to. To avoid repeating ourselves and typing long lines, it is convienient to define a few variables as short-cuts. We also want a list of *good electrons*, so lets define that.

```
%%time
eles = df.Electrons("Electrons")
good_eles = eles[(eles.pt > 50000.0) & (abs(eles.eta) < 1.5)]

np_pts_good_short = make_local(good_eles.pt/1000.0)
```

```
Wall time: 1.2 s
```

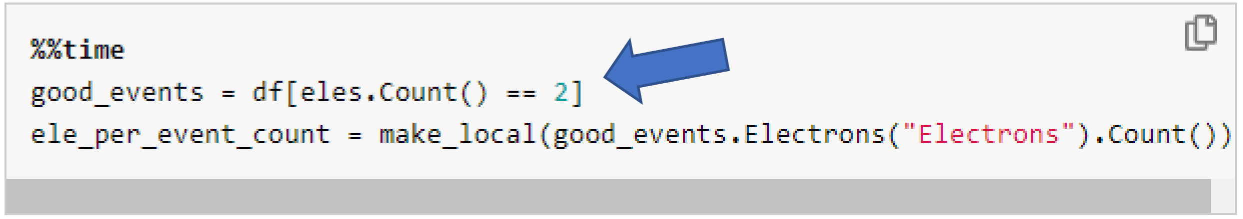
```
plt.hist(np_pts_good_short.flatten(), range=(0, 100), bins=50)
plt.xlabel('Electron $p_T$ with cut [GeV]')
_ = plt.ylabel('PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee')
```



Filter on # of Objects

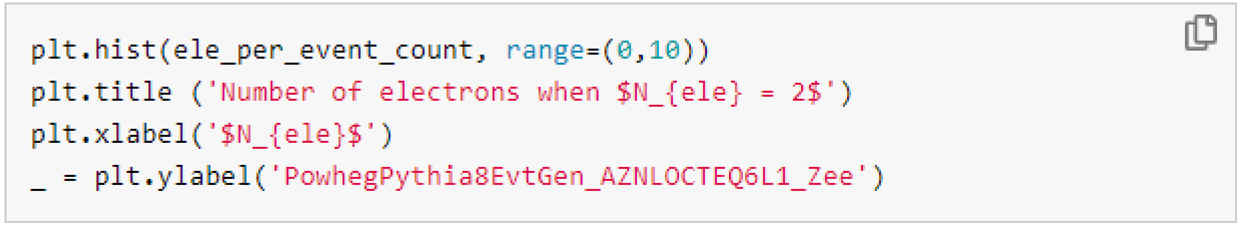
I've been using the "Count()" method to count, but if python allows a non-integer return from `len`, that should work just fine too.

notebook

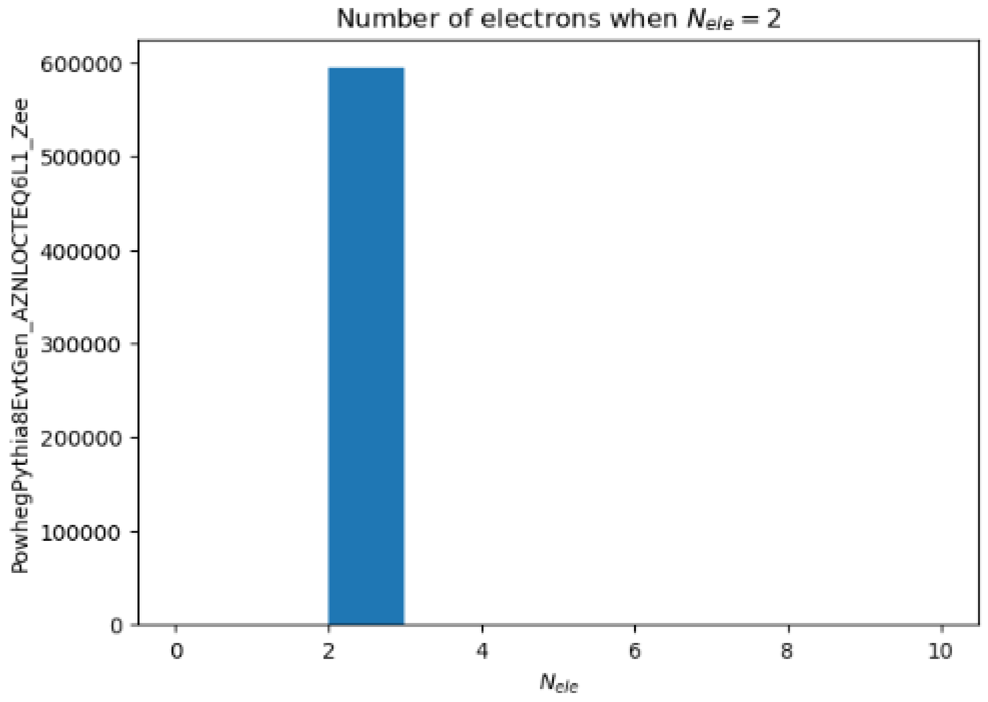


```
%%time
good_events = df[eles.Count() == 2]
ele_per_event_count = make_local(good_events.Electrons("Electrons").Count())
```

```
Wall time: 686 ms
```



```
plt.hist(ele_per_event_count, range=(0,10))
plt.title ('Number of electrons when $N_{ele} = 2$')
plt.xlabel('$N_{ele}$')
_ = plt.ylabel('PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee')
```

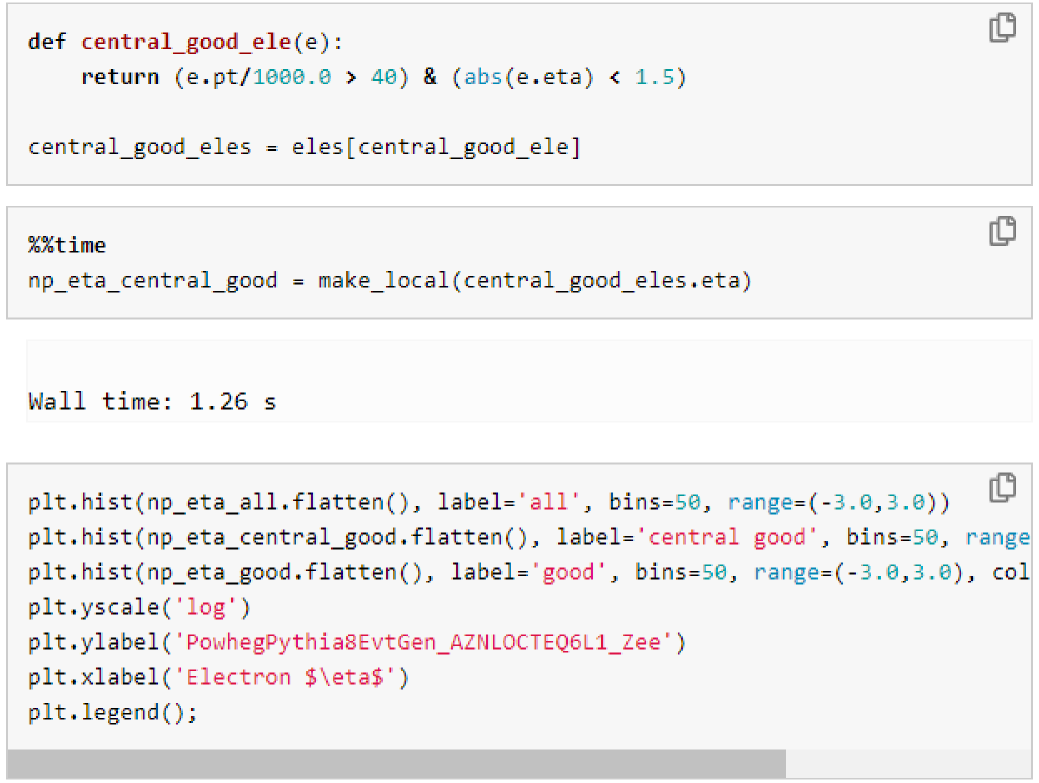


Filter Functions

For when thinking about arrays can get a little messy…

- Single object-like semantics
- No loops, if statements, etc.
- Has to render a DAG, not an algorithm that gets translated.
- The "? :" operator is valid (or should be).

notebook

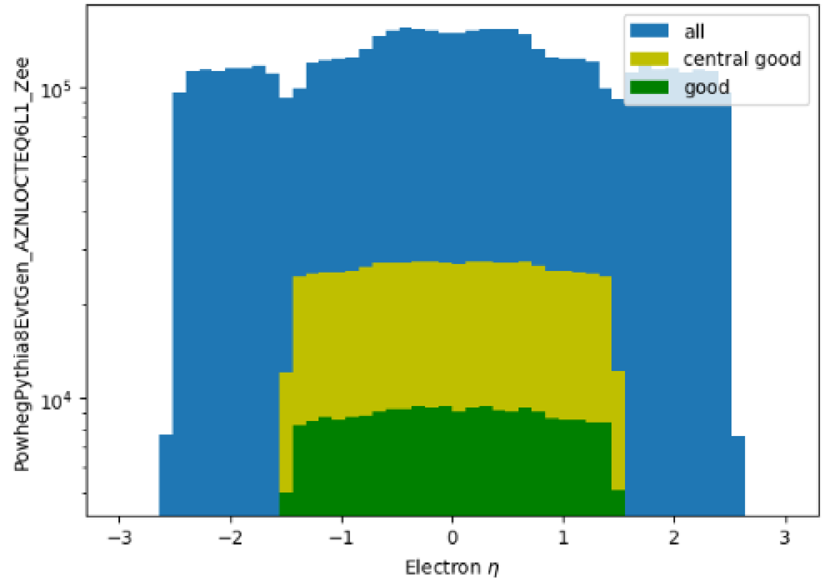


```
def central_good_ele(e):
    return (e.pt/1000.0 > 40) & (abs(e.eta) < 1.5)

central_good_eles = eles[central_good_ele]
```

```
%%time
np_eta_central_good = make_local(central_good_eles.eta)
```

```
Wall time: 1.26 s
```

```
plt.hist(np_eta_all.flatten(), label='all', bins=50, range=(-3.0,3.0))
plt.hist(np_eta_central_good.flatten(), label='central good', bins=50, range
plt.hist(np_eta_good.flatten(), label='good', bins=50, range=(-3.0,3.0), col
plt.yscale('log')
plt.ylabel('PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee')
plt.xlabel('Electron $\eta$')
plt.legend();
```



Goal 2: Separate “parsing” of python use from implementation

`dataframe_expressions`

- Knows no semantics!!
- Turns what you write into a DAG
- Understands lambda functions an the like
- Can extend the data model (see comming slides) with computed columns, a regular expression syntax, and by declaring whole new objects.
- Knows what to do with numpy functions (array ufunc)
- Leaky abstractions in the form of backend custom functions can be declared

See readme for a description

`hep_tables`

- Turn a dataframe_expressions into a qastle query
- Has a type system (heuristics mostly, with some explicit decl)

Documentation

Not really

Design Goals

There are two approaches to analyzing large datasets. One has the physicist dealing directly with coordinating the backend: scheduling, numbers of parallel processors, etc. The other has the physicist manipulate the dataset as a whole, and then let the backend decide how to split the work up.

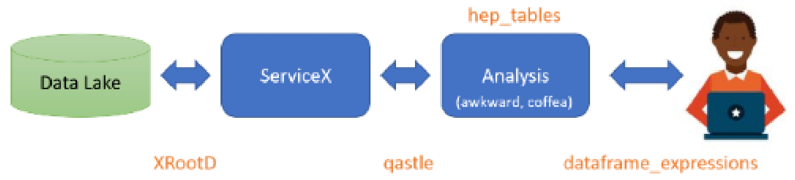
Each approach has advantages and disadvantages. The first approach means that the user needs to know details that have nothing to with physics, OTOH, they have complete control and can really take advantage of their knowledge of the shape of the data, the layout of the backend they are running on, etc. The latter approach trades these two - the user manipulates the data, and thinks very little about how the data is processed, but then it is very hard to really take advantage of the layout of the system.

Further, it should be noted that the second approach will use the tools developed for the first approach! And any actual solution will almost certainly not be purely one approach or a second approach.

The combination of `hep_tables` and `dataframe_expressions` leans more towards the second approach.

Where might a prototype end up?

IRL, one would expect the backend to split the query up, the first part would send to `servicex` and get the data, and then the second part would run on that returned data and produce results, something like this:



A quick description of the various bits:

- *Data Lake*: The experiment's data store. Usually backed by *rucio*.
- *XRootD*: The wire-level protocol used to move files
- *ServiceX*: distributed cloud application that extracts columns of data quickly from experiment's data. Capable of windowing rendered columns with simplified cuts.

Goal 3: More complex things possible…

Attempting to use a few simple concepts that are composable…

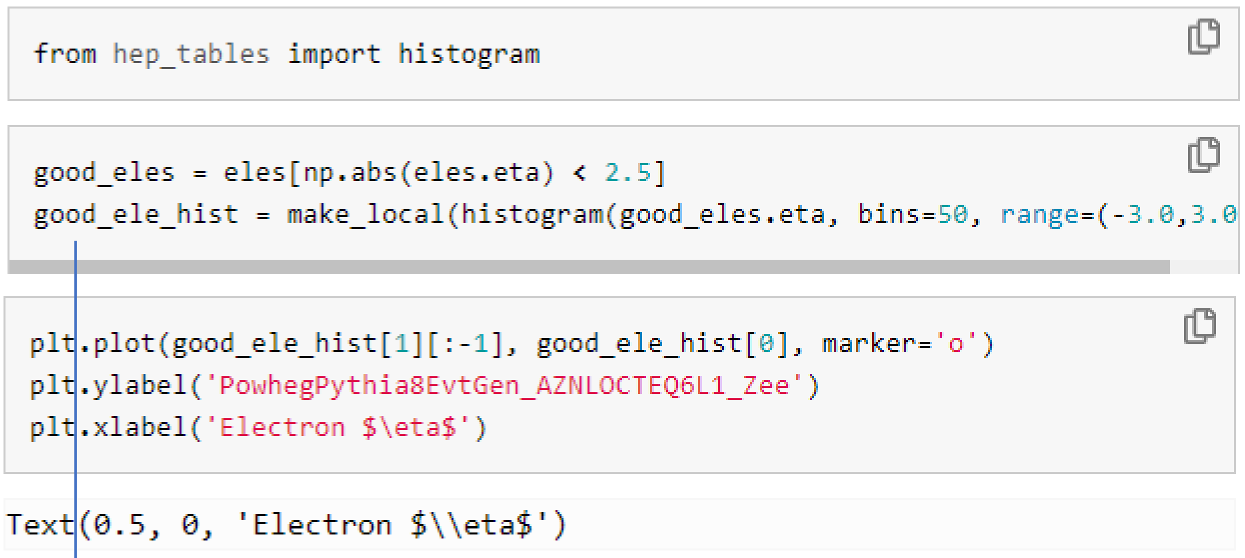
And if you read the code 6 months from now you understand WTF you did…

Everything on the backend

In Run 4 we will have histograms that can't be built on your laptop.

Can we implement a histogram on the backend in this system?

This is a cheat, but it does demonstrate the point…

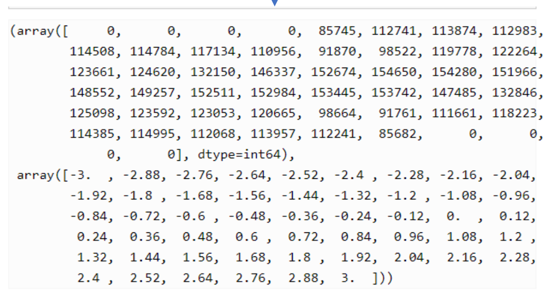


```
from hep_tables import histogram
```

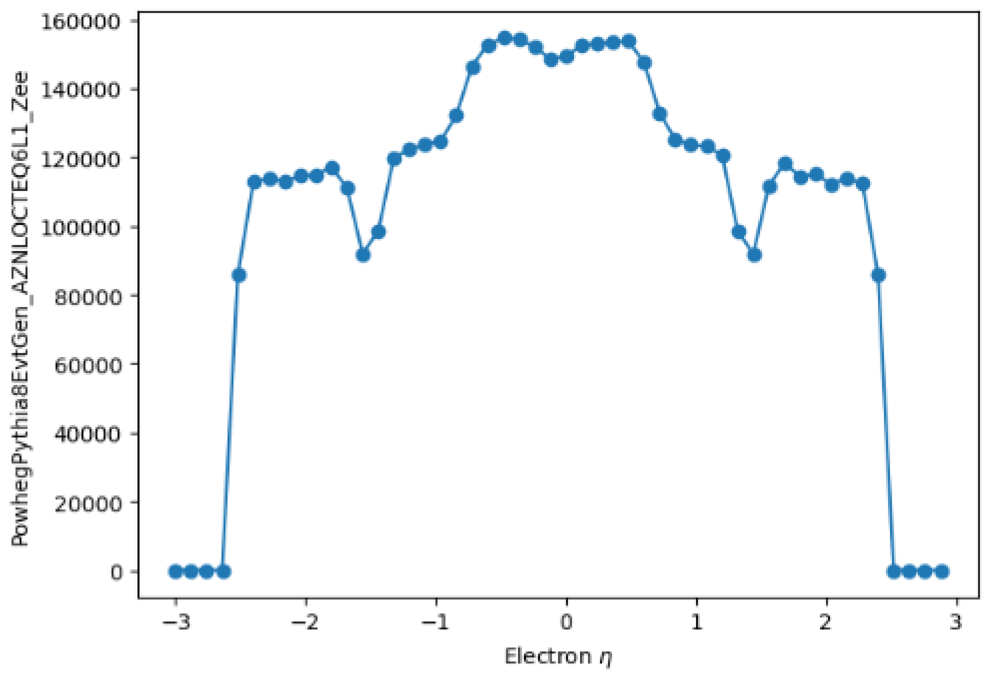
```
good_eles = eles[np.abs(eles.eta) < 2.5]
good_ele_hist = make_local(histogram(good_eles.eta, bins=50, range=(-3.0,3.0
```

```
plt.plot(good_ele_hist[1][:-1], good_ele_hist[0], marker='o')
plt.ylabel('PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee')
plt.xlabel('Electron $\eta$')
```

```
Text(0.5, 0, 'Electron $\\eta$')
```



```
(array([     0,      0,      0,      0,  85745, 112741, 113874, 112983,
        114508, 114784, 117134, 110956,  91870,  98522, 119778, 122264,
        123661, 124620, 132150, 146337, 152674, 154650, 154280, 151966,
        148552, 149257, 152511, 152984, 153445, 153742, 147485, 132846,
        125098, 123592, 123053, 120665,  98664,  91761, 111661, 118223,
        114385, 114995, 112068, 113957, 112241,  85682,      0,      0,
             0,      0], dtype=int64),
 array([-3.  , -2.88, -2.76, -2.64, -2.52, -2.4 , -2.28, -2.16, -2.04,
        -1.92, -1.8 , -1.68, -1.56, -1.44, -1.32, -1.2 , -1.08, -0.96,
        -0.84, -0.72, -0.6 , -0.48, -0.36, -0.24, -0.12,  0.  ,  0.12,
         0.24,  0.36,  0.48,  0.6 ,  0.72,  0.84,  0.96,  1.08,  1.2 ,
         1.32,  1.44,  1.56,  1.68,  1.8 ,  1.92,  2.04,  2.16,  2.28,
         2.4 ,  2.52,  2.64,  2.76,  2.88,  3.  ]))
```



notebook

Map Function

This ends up being the same as `eles.pt/1000.0`

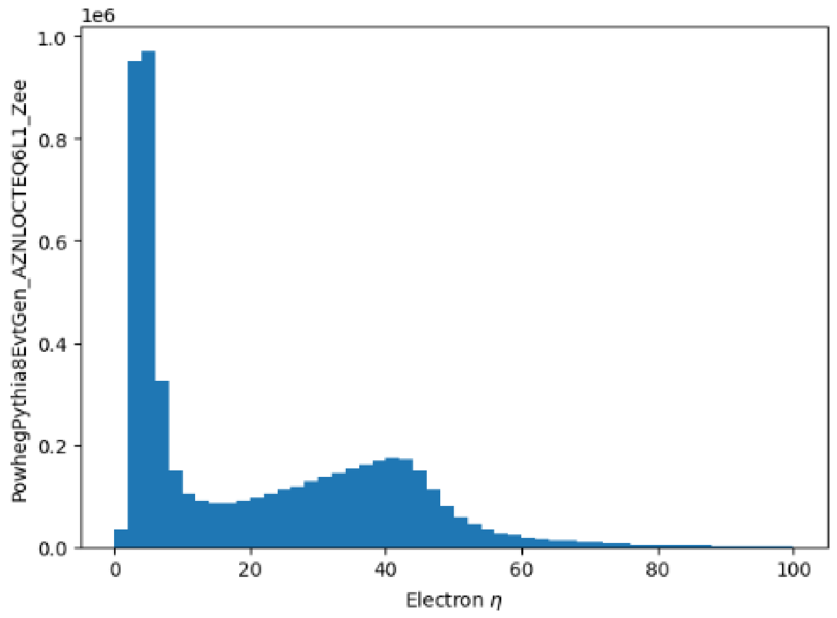
The power comes when you use the ability of a python lambda function to capture other variables.

```
mapped_eles = eles.map(lambda e: e.pt/1000.0)
```

```
%%time
mapped_eles_pt = make_local(mapped_eles)
```

```
Wall time: 1.83 s
```

```
plt.hist(mapped_eles_pt.flatten(), bins=50, range=(0, 100.0))
plt.ylabel('PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee')
_ = plt.xlabel('Electron $\eta$')
```



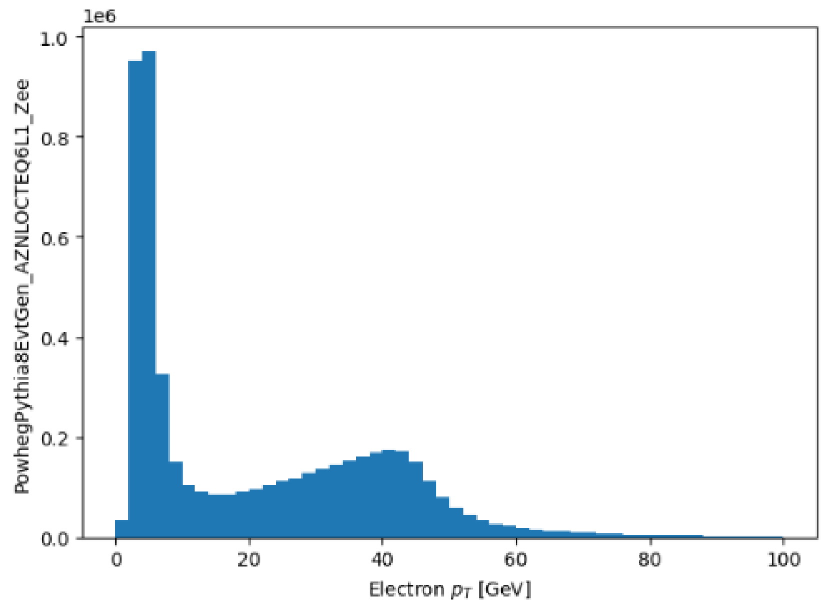
notebook

Adding computed columns

```
df['all_ele'] = df.Electrons("Electrons")
df.all_ele['mypt'] = df.all_ele.pt / 1000.0
```

We can then use them as if we had the leaves or collections as reulgar parts of the event:

```
%%time
pts = make_local(df.all_ele.mypt)
```



You could also write:

```
df.all_ele['mypt'] = lambda e: e.pt/1000.0
```

Single object semantics makes this way easier to read!

notebook

Object Associations

Let's associate all electrons with their MC particles

Setup:

```
mc_part = df.TruthParticles('TruthParticles')
mc_ele = mc_part[(mc_part.pdgId == 11) | (mc_part.pdgId == -11)]

eles = df.Electrons('Electrons')

def good_e(e):
    'Good electron particle'
    return (e.ptgev > 20) & (abs(e.eta) < 1.4)

good_eles = eles[good_e]
good_mc_ele = mc_ele[good_e]
```

notebook

Object Associations

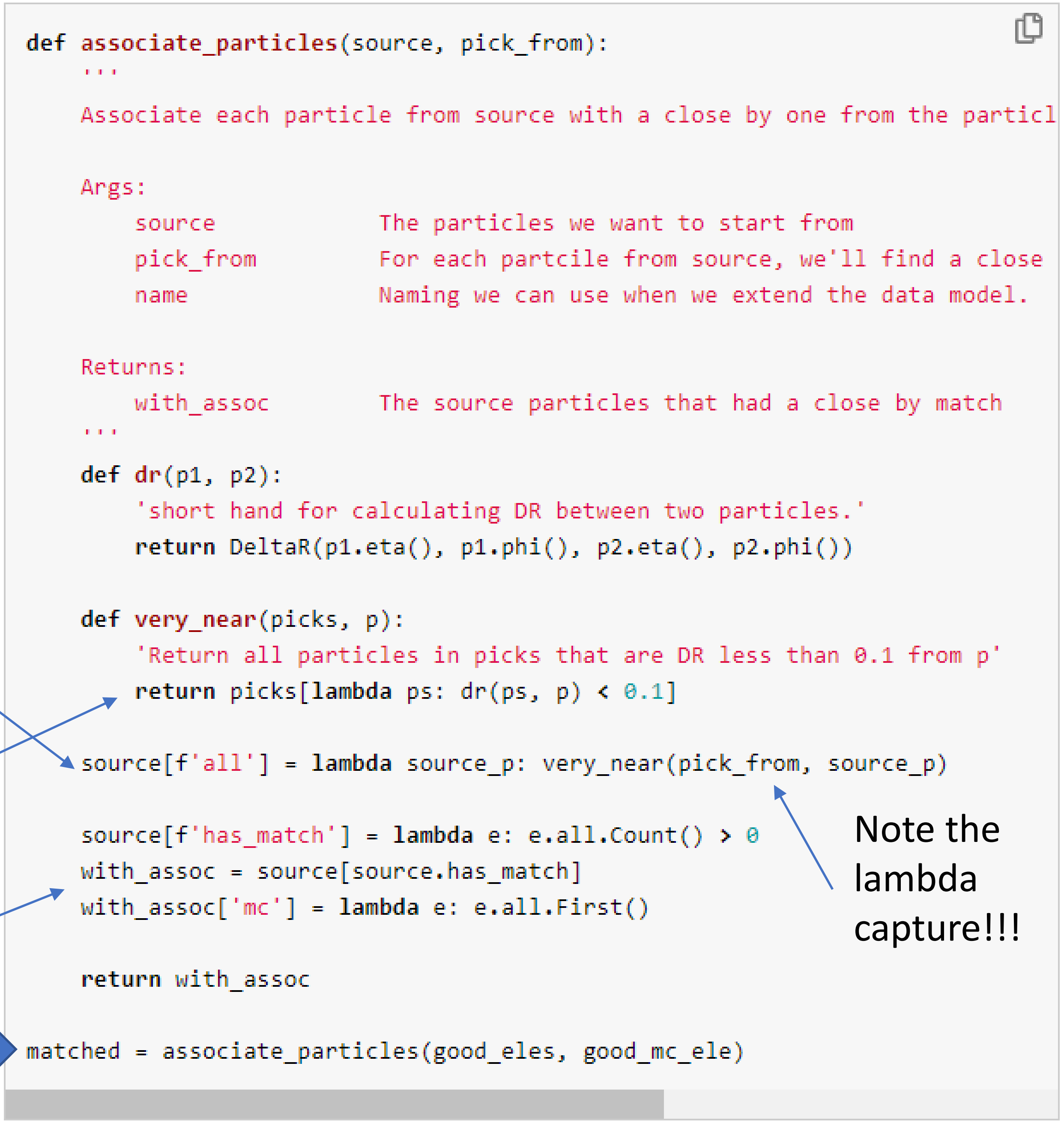
Matching:

Key line: for each good electron, select only the very near good MC electrons.

Filter by all that are near by for each electron we are “looking” at!

Build the data model to make life easy…

notebook



```
def associate_particles(source, pick_from):
    '''
    Associate each particle from source with a close by one from the particl

    Args:
        source              The particles we want to start from
        pick_from           For each partcile from source, we'll find a close
        name                Naming we can use when we extend the data model.

    Returns:
        with_assoc          The source particles that had a close by match
    '''
    def dr(p1, p2):
        'short hand for calculating DR between two particles.'
        return DeltaR(p1.eta(), p1.phi(), p2.eta(), p2.phi())

    def very_near(picks, p):
        'Return all particles in picks that are DR less than 0.1 from p'
        return picks[lambda ps: dr(ps, p) < 0.1]

    source[f'all'] = lambda source_p: very_near(pick_from, source_p)

    source[f'has_match'] = lambda e: e.all.Count() > 0
    with_assoc = source[source.has_match]
    with_assoc['mc'] = lambda e: e.all.First()

    return with_assoc

matched = associate_particles(good_eles, good_mc_ele)
```

Note the lambda capture!!!

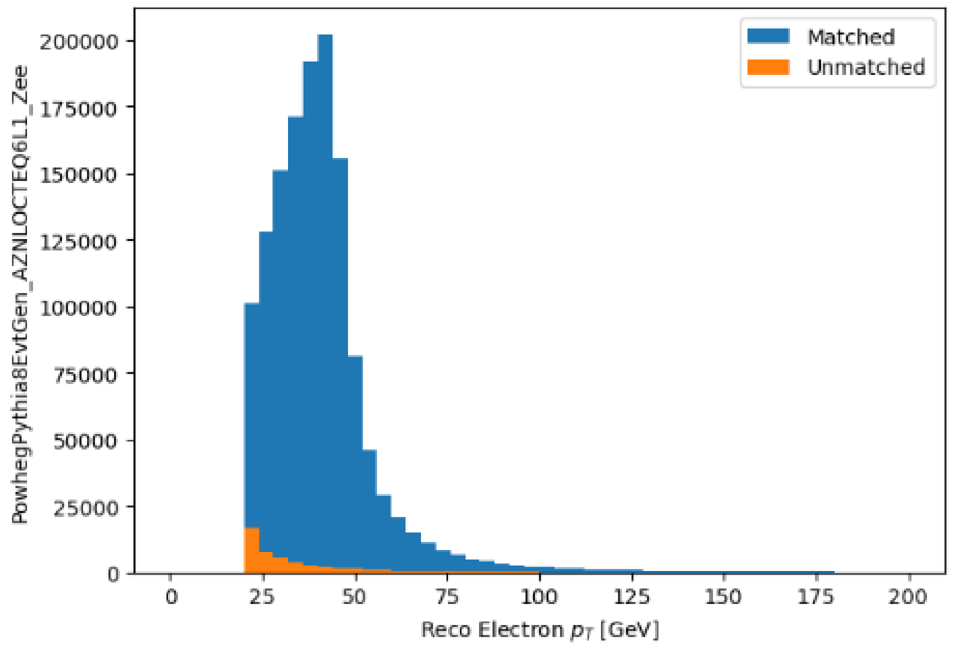
Object Associations

Let's compare at the p_T of the matched and unmatched electrons:

```
matched_ele_pt = make_local(good_eles[good_eles.has_match].ptgev)
unmatched_ele_pt = make_local(good_eles[~good_eles.has_match].ptgev)
```

```
plt.hist(matched_ele_pt.flatten(), bins=50, range=(0,200), label='Matched')
plt.hist(unmatched_ele_pt.flatten(), bins=50, range=(0,200), label='Unmatche
plt.xlabel('Reco Electron $p_T$ [GeV]')
plt.ylabel('PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x21367e67b08>
```



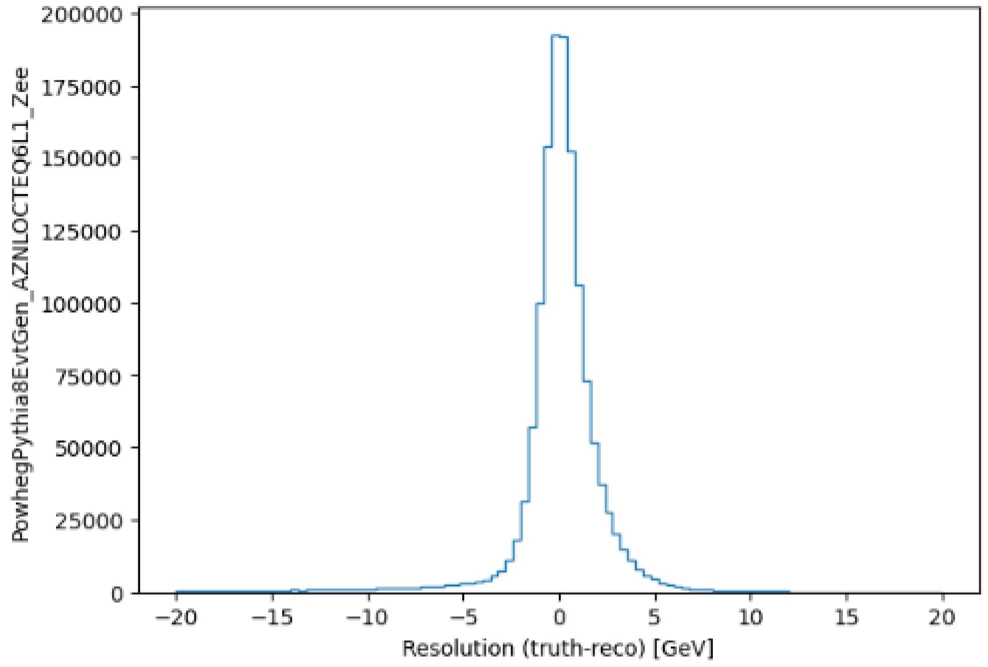
notebook

Object Associations

A resolution plot…

```
pt_matched_mc = make_local(matched.mc.ptgev)
pt_matched_reco = make_local(matched.ptgev)
```

```
plt.hist((pt_matched_mc-pt_matched_reco).flatten(), bins=100, range=(-20, 20
plt.ylabel('PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee')
_ = plt.xlabel("Resolution (truth-reco) [GeV]")
```



notebook

Object Associations

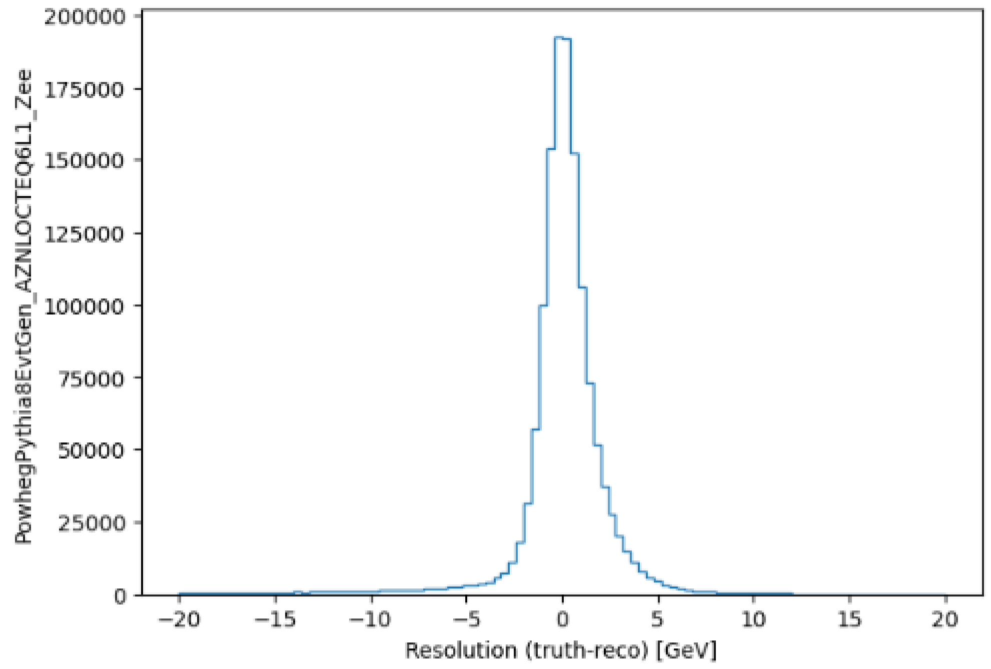
A resolution plot…

This is what I'm currently working on…

(note I had to bring the data local to make this plot)

```
pt_matched_mc = make_local(matched.mc.ptgev)
pt_matched_reco = make_local(matched.ptgev)
```

```
plt.hist((pt_matched_mc-pt_matched_reco).flatten(), bins=100, range=(-20, 20
plt.ylabel('PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee')
_ = plt.xlabel("Resolution (truth-reco) [GeV]")
```



notebook

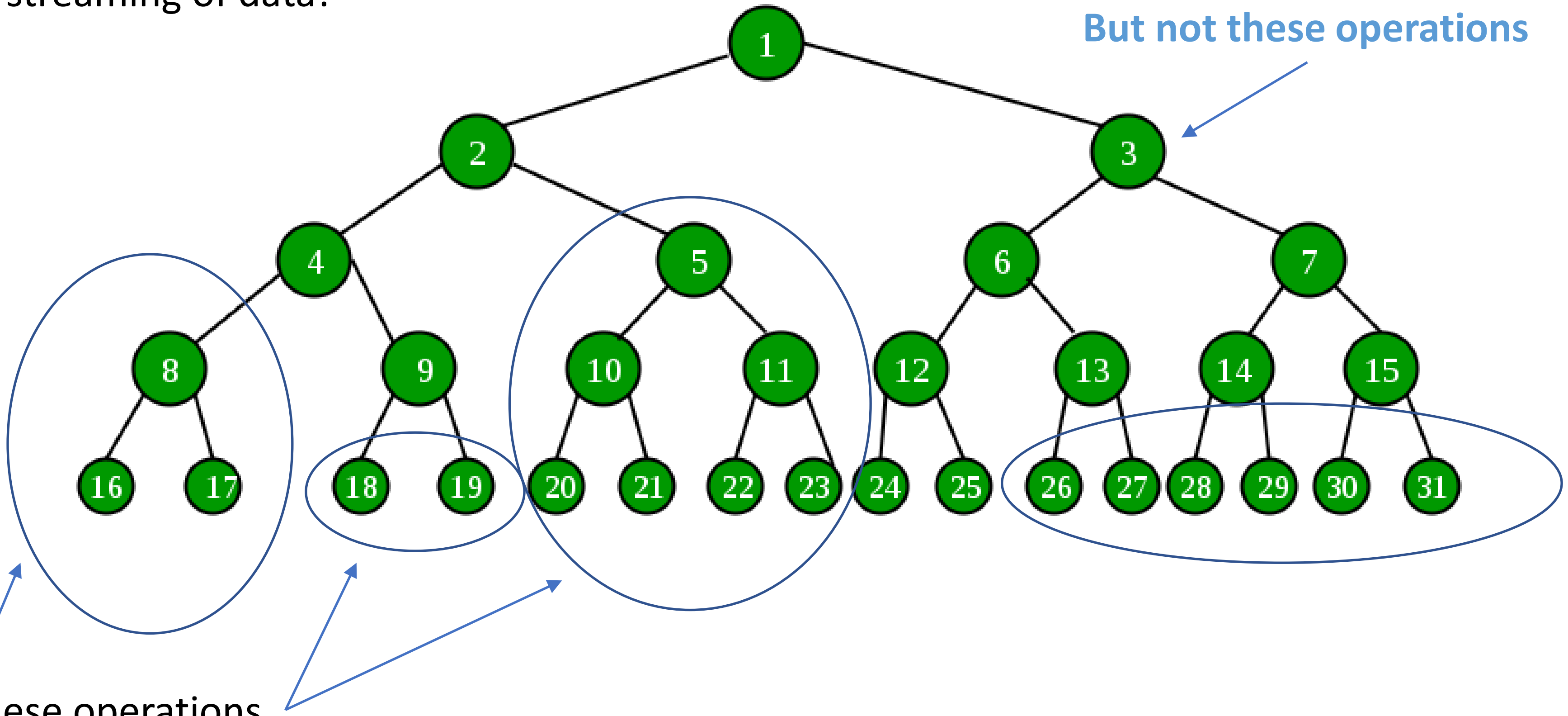
Object Associations

Summary:

hl_tables

ServiceX is meant to do simple streaming of data!

Our DAG:



But not these operations

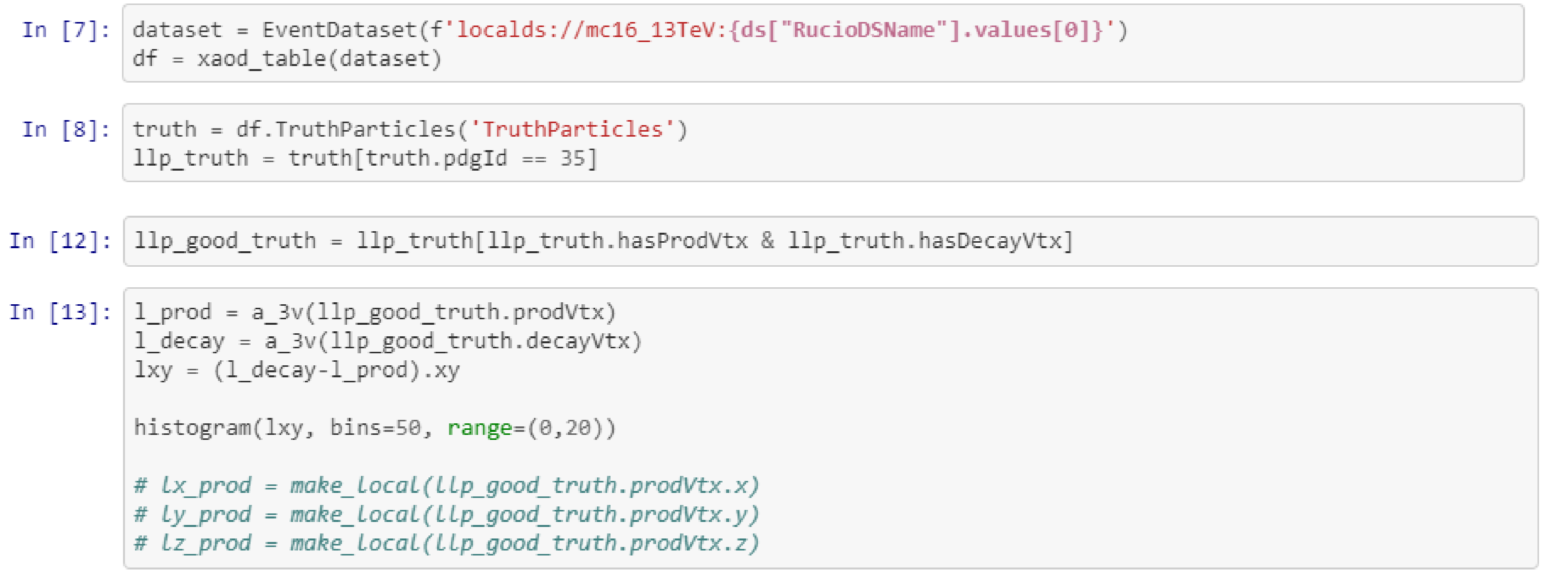
ServiceX can do these operations

Want to write…

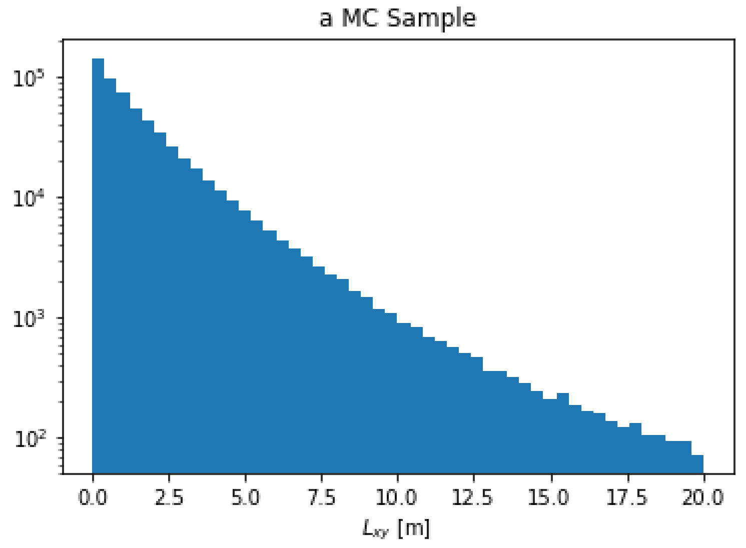
1. The histogram function caused all the data to be fetched.
2. Operations will occur on servicex as well as directly with awkward array, depending on their capabilities

Design is extensible…

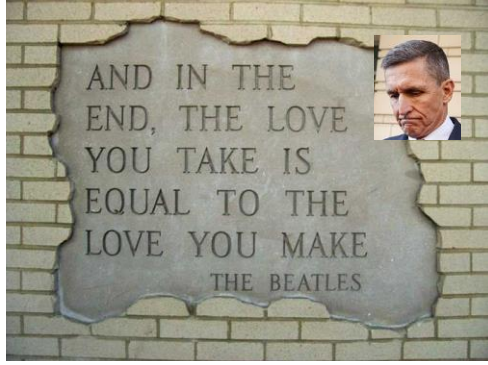
Once a simple version works…



```
In [7]:  dataset = EventDataset(f'localds://mc16_13TeV:{ds["RucioDSName"].values[0]}')
         df = xaod_table(dataset)

In [8]:  truth = df.TruthParticles('TruthParticles')
         llp_truth = truth[truth.pdgId == 35]

In [12]: llp_good_truth = llp_truth[llp_truth.hasProdVtx & llp_truth.hasDecayVtx]

In [13]: l_prod = a_3v(llp_good_truth.prodVtx)
         l_decay = a_3v(llp_good_truth.decayVtx)
         lxy = (l_decay-l_prod).xy

         histogram(lxy, bins=50, range=(0,20))

         # lx_prod = make_local(llp_good_truth.prodVtx.x)
         # ly_prod = make_local(llp_good_truth.prodVtx.y)
         # lz_prod = make_local(llp_good_truth.prodVtx.z)
```



You Made It To The End!


AND IN THE END, THE LOVE YOU TAKE IS EQUAL TO THE LOVE YOU MAKE
THE BEATLES

- I’m using a new MC analysis idea I’m developing to drive the features here.
  - There are lots of gaps
  - And you can’t try it out yet (well, you could, but servicex setup, etc.)
- It should work for ntuples and for xAOD’s (nanoAODs) just fine
  - But semantics are a little different and it hasn’t been tested… yet.
- Challenges for expressing data, etc.
- Goals:
  - Feel reasonable about goal 1, goal 2
  - Goal 3 – need to use it in more than contrived examples
- Prototype: There are some definite architecture and design decisions I made which are not correct
  - Many of them I figured them out only when trying to put together more complex operations
  - So want to continue to “break” this to get a better idea.
- What is next?
  - See if I can use it to make an analysis on 57 MC rucio datasets
  - Add coffea idioms in as hl_tables gets more sophisticated (if my structure can deal with it).
  - Distributed processing to simulate analysis facility?
- What else?