Aghast: communication among histogram implementations

Jim Pivarski

Princeton University – IRIS-HEP

April 15, 2019

Context: slide from last summer (8 rows commented out)

pip?	name	last release	interface style	depends on	integrates with
	PyROOT	2018	HEP	ROOT	numpy
	YODA	2018	HEP	*compiled*	matplotlib, yaml
√	physt	2018	HEP + data science	numpy	pandas, xarray, dask, protobuf, matplotlib, vega (plotting), folium (maps)
√	fast-histogram	2018	simple (astronomy)	numpy	
√	qhist	2018	HEP	ROOT	
√	rootpy	2017	HEP	ROOT	pytables, matplotlib, stats
√	Vaex (vaex.io)	2017	all-in-one GUI for big data, fast heatmaps	*many!*	Jupyter, matplotlib, HDF5, pandas, C++
√	hdrhistogram	2017	"high dynamic range"	*compiled*	Java, C++
√	multihist	2017	numpy wrapper	numpy	matplotlib
√	matplotlib-hep	2016	HEP	matplotlib	numpy, scipy
√	pyhistogram	2014	HEP	numpy	matplotlib, datetime
√	histogram	2011	HEP	numpy	matplotlib, HDF5
√	SimpleHist	2011	HEP	numpy, matplotlib	ROOT
√	paida	2007	HEP		AIDA!
	theodoregoetz	never	HEP	scipy, uncertainties	numpy, matplotlib

Over the years, I've written five:

pip?	name	last release	interface style	depends on	integrates with
	Plothon	2006	HEP	ROOT	SVG
	SVGFig	2008	HEP	*nothing!*	SVG
	Cassius	2013	HEP + data science	numpy	SVG, Augustus (Open Data Group)
√	Histogrammar	2016	combinational library	numpy	Spark, Julia, CUDA, ROOT (Cling), matplotlib, Bokeh, Vega
√	histbook	2018	HEP	numpy	Spark, Pandas, Vega

And today, you've heard about four more:

pip?	name	last release	interface style	depends on	integrates with
√	fcat.hist	2019	HEP	Matplotlib	numpy, awkward
	mpl-hep	2019	HEP	Matplotlib	numpy, pandas
	boost-histogram	2019	HEP	*compiled*	C++, numpy
√	hist	2019	HEP	the above	

We're not lacking for options, but perhaps for unity.

Moreover, these libraries are specializing:

- **boost-histogram** provides fast and flexible *filling*,
- **mpl-hep** provides many *plotting* options.
- Other tools focus on *fitting*.

The **hist** package will wrap that up behind a single "`import hist`" command, but to do so, we need to get histogram objects from one library to another.

Tying it all together

The **aghast** library (**ag**gregated, **h**istogram-like **st**atistics) is an *in-memory, serializable ontology* of histogram-like objects.

aghast

Aghast is a histogramming library that does not fill histograms and does not plot them. Its role is behind the scenes, to provide better communication between histogramming libraries.

in-memory: The purpose is to move histogram-like data from one library to another, invisibly as part of a function call. It is not primarily intended for long-term storage in files. In this respect, it is like Apache Arrow.

in-memory: The purpose is to move histogram-like data from one library to another, invisibly as part of a function call. It is not primarily intended for long-term storage in files. In this respect, it is like Apache Arrow.

serializable: We use Flatbuffers as a wire protocol to move data over a network or between languages, if necessary. Flatbuffers supports partial reading/deserialization. (Flatbuffers is for network games; designed for speed.)

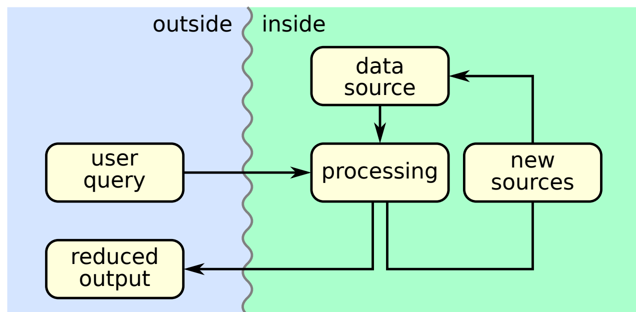
in-memory: The purpose is to move histogram-like data from one library to another, invisibly as part of a function call. It is not primarily intended for long-term storage in files. In this respect, it is like Apache Arrow.

serializable: We use Flatbuffers as a wire protocol to move data over a network or between languages, if necessary. Flatbuffers supports partial reading/deserialization. (Flatbuffers is for network games; designed for speed.)

ontology: Aghast focuses exclusively on representing the data and converting histograms into histograms. It does not fill or plot them. In this respect, it is like Predictive Model Markup Language (PMML), which neither trains nor scores machine learning models.

Another reason we want library-neutral histograms

In IRIS-HEP Analysis Systems, we're developing a **Query Service** to keep large datasets inside a system while being responsive to user requests from outside.



The "**reduced output**" needs to be in a form that can be transmitted, merged, and converted into any format. Such an object could be called "**a ghast**."

What does it look like?

```
>>> import numpy, aghast
>>> h_numpy = numpy.histogram(numpy.random.normal(0, 1, 100000),
...                           bins=20, range=(-5, 5))
```

What does it look like?

```
>>> import numpy, aghast
>>> h_numpy = numpy.histogram(numpy.random.normal(0, 1, 100000),
...                           bins=20, range=(-5, 5))
>>> aghast.from_numpy(h_numpy)
```

What does it look like?

```
>>> import numpy, aghast
>>> h_numpy = numpy.histogram(numpy.random.normal(0, 1, 100000),
...                           bins=20, range=(-5, 5))
>>> aghast.from_numpy(h_numpy).dump()
```

```
Histogram(
  axis=[
    Axis(binning=RegularBinning(num=20, interval=RealInterval(low=-5.0, high=5.0)))
  ],
  counts=
    UnweightedCounts(
      counts=
        InterpretedInlineInt64Buffer(
          buffer=
              [    2     5    21   107   477  1696  4378  9146 15084 19273 18933 14960
                9216  4425  1648   480   122    23     4     0])))
```

What does it look like?

```
>>> import numpy, aghast
>>> h_numpy = numpy.histogram(numpy.random.normal(0, 1, 100000),
...                            bins=20, range=(-5, 5))
>>> aghast.from_numpy(h_numpy).dump()
```

```
Histogram(
  axis=[
    Axis(binning=RegularBinning(num=20, interval=RealInterval(low=-5.0, high=5.0)))
  ],
  counts=
    UnweightedCounts(
      counts=
        InterpretedInlineInt64Buffer(
          buffer=
              [    2     5    21   107   477  1696  4378  9146 15084 19273 18933 14960
                9216  4425  1648   480   122    23     4     0]))))
```

Long names needed to construct these objects → not for end-users!

What does it look like?

```
>>> import ROOT, aghast
>>> h_root = ROOT.TH1F("name", "title", 20, -5, 5)
>>> h_root.FillRandom("gaus", 100000)
>>> aghast.from_root(h_root).dump()
```

```
Histogram(
  axis=[
    Axis(
      binning=
        RegularBinning(
          num=20,
          interval=RealInterval(low=-5.0, high=5.0),
          overflow=RealOverflow(loc_underflow=BinLocation.below1, loc_overflow=BinLocation.above1)),
      statistics=[
        Statistics(
          moments=[
            Moments(sumwxn=InterpretedInlineInt64Buffer(buffer=[100000]), n=0),
            Moments(sumwxn=InterpretedInlineFloat64Buffer(buffer=[100000]), n=0, weightpower=1),
            Moments(sumwxn=InterpretedInlineFloat64Buffer(buffer=[100000]), n=0, weightpower=2),
            Moments(sumwxn=InterpretedInlineFloat64Buffer(buffer=[-102.751]), n=1, weightpower=1),
            Moments(sumwxn=InterpretedInlineFloat64Buffer(buffer=[104061]), n=2, weightpower=1)
          ])
      ])
  ],
  counts=
    UnweightedCounts(
      counts=
```

What does it look like?

```
>>> import ROOT, aghast
>>> h_root = ROOT.TH1F("name", "title", 20, -5, 5)
>>> h_root.FillRandom("gaus", 100000)
>>> aghast.from_root(h_root).dump()
```

(The ROOT histogram has `moments` to preserve unbinned mean and standard deviation.)

```
Histogram(
  axis=[
    Axis(
      binning=
        RegularBinning(
          num=20,
          interval=RealInterval(low=-5.0, high=5.0),
          overflow=RealOverflow(loc_underflow=BinLocation.below1, loc_overflow=BinLocation.above1)),
      statistics=[
        Statistics(
          moments=[
            Moments(sumwxn=InterpretedInlineInt64Buffer(buffer=[100000]), n=0),
            Moments(sumwxn=InterpretedInlineFloat64Buffer(buffer=[100000]), n=0, weightpower=1),
            Moments(sumwxn=InterpretedInlineFloat64Buffer(buffer=[100000]), n=0, weightpower=2),
            Moments(sumwxn=InterpretedInlineFloat64Buffer(buffer=[-102.751]), n=1, weightpower=1),
            Moments(sumwxn=InterpretedInlineFloat64Buffer(buffer=[104061]), n=2, weightpower=1)
          ])
      ])
  ],
  counts=
    UnweightedCounts(
      counts=
```

What does it look like?

ROOT → Aghast → Numpy:

```
>>> aghast.to_numpy(aghast.from_root(h_root))
```

```
(array([0.0000e+00, 0.0000e+00, 4.0000e+00, 1.8000e+01, 1.0000e+02,
        4.7900e+02, 1.6950e+03, 4.3380e+03, 9.2950e+03, 1.4974e+04,
        1.9195e+04, 1.9289e+04, 1.4781e+04, 9.1450e+03, 4.3850e+03,
        1.6680e+03, 4.9200e+02, 1.1600e+02, 2.3000e+01, 3.0000e+00,
        0.0000e+00, 0.0000e+00], dtype=float32),
 array([-inf, -5. , -4.5, -4. , -3.5, -3. , -2.5, -2. , -1.5, -1. , -0.5,
         0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
         inf]))
```

What does it look like?

ROOT $\rightarrow$ Aghast $\rightarrow$ Numpy:

```
>>> aghast.to_numpy(aghast.from_root(h_root))
```

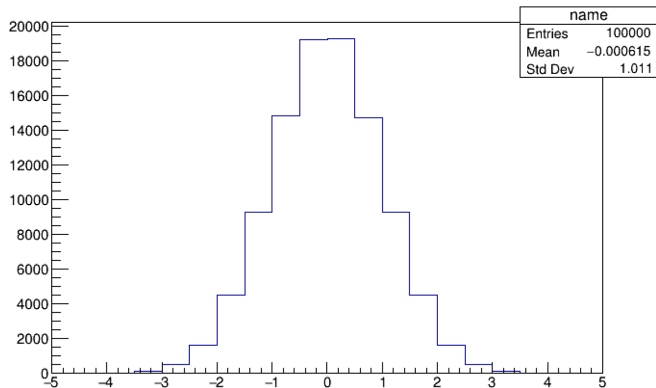
```
(array([0.0000e+00, 0.0000e+00, 4.0000e+00, 1.8000e+01, 1.0000e+02,
        4.7900e+02, 1.6950e+03, 4.3380e+03, 9.2950e+03, 1.4974e+04,
        1.9195e+04, 1.9289e+04, 1.4781e+04, 9.1450e+03, 4.3850e+03,
        1.6680e+03, 4.9200e+02, 1.1600e+02, 2.3000e+01, 3.0000e+00,
        0.0000e+00, 0.0000e+00], dtype=float32),
 array([-inf, -5. , -4.5, -4. , -3.5, -3. , -2.5, -2. , -1.5, -1. , -0.5,
         0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
         inf]))
```

Even though Numpy doesn't have a concept of regularly spaced bins or overflow bins, we can shoehorn the ROOT histogram into this form.

What does it look like?

Numpy → Aghast → ROOT:

```
>>> aghast.to_root(aghast.from_numpy(h_numpy), "name").Draw()
```



What does it look like?

Merging histograms ("hadd") with different structures:

```
>>> (aghast.from_numpy(h_numpy) + aghast.from_root(h_root)).dump()
```

```
Histogram(
  axis=[
    Axis(
      binning=
        RegularBinning(
          num=20,
          interval=RealInterval(low=-5.0, high=5.0),
          overflow=RealOverflow(loc_underflow=BinLocation.above1,
                                loc_overflow=BinLocation.above2)))
  ],
  counts=
    UnweightedCounts(
      counts=
        InterpretedInlineFloat64Buffer(
          buffer=
              [0.0000e+00 5.0000e+00 3.6000e+01 2.1700e+02 9.6800e+02 3.2320e+03
               8.7780e+03 1.8562e+04 2.9957e+04 3.8380e+04 3.8551e+04 2.9654e+04
               1.8311e+04 8.7890e+03 3.2790e+03 1.0010e+03 2.4000e+02 3.4000e+01
               6.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00])))
```

What does it look like?

In Pandas, binnings become indexes and per-bin statistics (only one here) become columns.

```
>>> aghast.to_pandas(aghast.from_root(h_root))
```

```
                unweighted
[-inf, -5.0)           0.0
[-5.0, -4.5)           0.0
[-4.5, -4.0)           4.0
[-4.0, -3.5)          18.0
[-3.5, -3.0)         100.0
[-3.0, -2.5)         479.0
[-2.5, -2.0)        1695.0
[-2.0, -1.5)        4338.0
[-1.5, -1.0)        9295.0
[-1.0, -0.5)       14974.0
[-0.5, 0.0)        19195.0
[0.0, 0.5)         19289.0
[0.5, 1.0)         14781.0
[1.0, 1.5)          9145.0
[1.5, 2.0)          4385.0
[2.0, 2.5)          1668.0
[2.5, 3.0)           492.0
[3.0, 3.5)           116.0
[3.5, 4.0)            23.0
[4.0, 4.5)             3.0
[4.5, 5.0)             0.0
[5.0, inf)             0.0
```

Common way of describing features of all histogram libraries

- Collection
- Histogram
- Axis
- IntegerBinning
- RegularBinning
- RealInterval
- RealOverflow
- HexagonalBinning
- EdgesBinning
- IrregularBinning
- CategoryBinning
- SparseRegularBinning
- FractionBinning
- PredicateBinning
- VariationBinning
- Variation
- Assignment
- UnweightedCounts
- WeightedCounts
- InterpretedInlineBuffer
- InterpretedInlineInt64Buffer
- InterpretedInlineFloat64Buffer
- InterpretedExternalBuffer
- Profile
- Statistics
- Moments
- Quantiles
- Modes
- Extremes
- StatisticFilter
- Covariance
- ParameterizedFunction
- Parameter
- EvaluatedFunction
- BinnedEvaluatedFunction
- Ntuple
- Column
- NtupleInstance
- Chunk
- ColumnChunk
- Page
- RawInlineBuffer
- RawExternalBuffer
- Metadata
- Decoration

Common way of describing features of all histogram libraries

- Only one **Histogram** type; it's *n*-dimensional and slicable like a Numpy array.
- Regular/irregular/string-labeled/sparse axis types are all **Binnings** (and they become an index or multi-index in Pandas).
- Efficiency plots are plots that have a **FractionBinning** in some axis.
- **Collections** of plots representing the same observables with different cuts share a **PredicateBinning**.
- **Collections** of plots representing different systematic variations share a **VariationBinning**.
- An **Axis** may also have **Moments** and **Correlations**.
- **Profile** plots are those that have binned **Moments**. Many **Profiles** can share the same binning (and they become columns in Pandas).
- **Functions** may be attached to **Histograms** or may be unattached in **Collections**.
- **Collections** may also contain simple **Ntuples** for unbinned fitting.

Extracting bin values

Numpy-like slicing with overflow bins

```
>>> allcounts = numpy.ones((7, 7), int)
>>> allcounts[5, :] = allcounts[6, :] = allcounts[:, 0] = allcounts[:, 1] = 999

>>> h = aghast.Histogram(
...     [aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.above1,
...                                      loc_overflow=aghast.RealOverflow.above2))),
...      aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.below2,
...                                      loc_overflow=aghast.RealOverflow.below1)))],
...     aghast.UnweightedCounts(
...         aghast.InterpretedInlineBuffer.fromarray(allcounts)))
```

Numpy-like slicing with overflow bins

```
>>> allcounts = numpy.ones((7, 7), int)
>>> allcounts[5, :] = allcounts[6, :] = allcounts[:, 0] = allcounts[:, 1] = 999

>>> h = aghast.Histogram(
...     [aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                      aghast.RealOverflow(loc_underflow=aghast.RealOverflow.above1,
...                                          loc_overflow=aghast.RealOverflow.above2))),
...      aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                      aghast.RealOverflow(loc_underflow=aghast.RealOverflow.below2,
...                                          loc_overflow=aghast.RealOverflow.below1)))],
...     aghast.UnweightedCounts(
...         aghast.InterpretedInlineBuffer.fromarray(allcounts)))

>>> h.counts[:, 0:5]                        # normal slice gets the non-overflow bins

array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])
```

Numpy-like slicing with overflow bins

```
>>> allcounts = numpy.ones((7, 7), int)
>>> allcounts[5, :] = allcounts[6, :] = allcounts[:, 0] = allcounts[:, 1] = 999

>>> h = aghast.Histogram(
...     [aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.above1,
...                                      loc_overflow=aghast.RealOverflow.above2))),
...      aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.below2,
...                                      loc_overflow=aghast.RealOverflow.below1)))],
...     aghast.UnweightedCounts(
...         aghast.InterpretedInlineBuffer.fromarray(allcounts)))

>>> h.counts[-numpy.inf:numpy.inf, :]      # infinity at one end means under/overflow

array([[999, 999, 999, 999, 999],
       [  1,   1,   1,   1,   1],
       [  1,   1,   1,   1,   1],
       [  1,   1,   1,   1,   1],
       [  1,   1,   1,   1,   1],
       [  1,   1,   1,   1,   1],
       [999, 999, 999, 999, 999]])
```

Numpy-like slicing with overflow bins

```
>>> allcounts = numpy.ones((7, 7), int)
>>> allcounts[5, :] = allcounts[6, :] = allcounts[:, 0] = allcounts[:, 1] = 999

>>> h = aghast.Histogram(
...     [aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                   aghast.RealOverflow(loc_underflow=aghast.RealOverflow.above1,
...                                       loc_overflow=aghast.RealOverflow.above2))),
...      aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                   aghast.RealOverflow(loc_underflow=aghast.RealOverflow.below2,
...                                       loc_overflow=aghast.RealOverflow.below1)))],
...     aghast.UnweightedCounts(
...         aghast.InterpretedInlineBuffer.fromarray(allcounts)))

>>> h.counts[:numpy.inf, -numpy.inf:3]     # in each dimension

array([[999,   1,   1,   1],
       [999,   1,   1,   1],
       [999,   1,   1,   1],
       [999,   1,   1,   1],
       [999,   1,   1,   1],
       [999, 999, 999, 999]])
```

Histogram-to-histogram transformations: slicing, projecting, and rebinning

Pandas-like loc/iloc slicing

```
>>> allcounts = numpy.ones((7, 7), int)
>>> allcounts[5, :] = allcounts[6, :] = allcounts[:, 0] = allcounts[:, 1] = 999

>>> h = aghast.Histogram(
...     [aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.above1,
...                                      loc_overflow=aghast.RealOverflow.above2))),
...      aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.below2,
...                                      loc_overflow=aghast.RealOverflow.below1)))],
...     aghast.UnweightedCounts(
...         aghast.InterpretedInlineBuffer.fromarray(allcounts)))

>>> h.loc[0.0:, :].axis[0].dump()          # loc slices to make a new histogram

Axis(
  binning=
    RegularBinning(
      num=3,
      interval=RealInterval(low=-1.0, high=5.0),
      overflow=RealOverflow(loc_underflow=BinLocation.above1,
                            loc_overflow=BinLocation.above2)))
```

Pandas-like loc/iloc slicing

```
>>> allcounts = numpy.ones((7, 7), int)
>>> allcounts[5, :] = allcounts[6, :] = allcounts[:, 0] = allcounts[:, 1] = 999

>>> h = aghast.Histogram(
...     [aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.above1,
...                                      loc_overflow=aghast.RealOverflow.above2))),
...      aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.below2,
...                                      loc_overflow=aghast.RealOverflow.below1)))],
...     aghast.UnweightedCounts(
...         aghast.InterpretedInlineBuffer.fromarray(allcounts)))

>>> h.iloc[2:, :].axis[0].dump()            # iloc slices with index coordinates

Axis(
  binning=
    RegularBinning(
      num=3,
      interval=RealInterval(low=-1.0, high=5.0),
      overflow=RealOverflow(loc_underflow=BinLocation.above1,
                            loc_overflow=BinLocation.above2)))
```

Pandas-like loc/iloc slicing

```
>>> allcounts = numpy.ones((7, 7), int)
>>> allcounts[5, :] = allcounts[6, :] = allcounts[:, 0] = allcounts[:, 1] = 999

>>> h = aghast.Histogram(
...     [aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.above1,
...                                      loc_overflow=aghast.RealOverflow.above2))),
...      aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.below2,
...                                      loc_overflow=aghast.RealOverflow.below1)))],
...     aghast.UnweightedCounts(
...         aghast.InterpretedInlineBuffer.fromarray(allcounts)))

>>> h.iloc[::2, ::2].axis[0].dump()        # ::2 rebins by a factor of 2

Axis(
  binning=
    RegularBinning(
      num=2,
      interval=RealInterval(low=-5.0, high=3.0),
      overflow=RealOverflow(loc_underflow=BinLocation.above1,
                            loc_overflow=BinLocation.above2)))
```

Pandas-like loc/iloc slicing

```
>>> allcounts = numpy.ones((7, 7), int)
>>> allcounts[5, :] = allcounts[6, :] = allcounts[:, 0] = allcounts[:, 1] = 999

>>> h = aghast.Histogram(
...     [aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.above1,
...                                      loc_overflow=aghast.RealOverflow.above2))),
...      aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.below2,
...                                      loc_overflow=aghast.RealOverflow.below1)))],
...     aghast.UnweightedCounts(
...         aghast.InterpretedInlineBuffer.fromarray(allcounts)))

>>> h.iloc[::2, ::2].counts[:, :]          # ::2 rebins by a factor of 2

array([[4, 4],
       [4, 4]])
```

Pandas-like loc/iloc slicing

```
>>> allcounts = numpy.ones((7, 7), int)
>>> allcounts[5, :] = allcounts[6, :] = allcounts[:, 0] = allcounts[:, 1] = 999

>>> h = aghast.Histogram(
...     [aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.above1,
...                                      loc_overflow=aghast.RealOverflow.above2))),
...      aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.below2,
...                                      loc_overflow=aghast.RealOverflow.below1)))],
...     aghast.UnweightedCounts(
...         aghast.InterpretedInlineBuffer.fromarray(allcounts)))

>>> h.loc[0.0:, None].counts[:]            # None projects (sums over) a dimension

array([2003, 2003, 2003])
```

Note: slicing, projecting, and rebinning are aspects of the same operation:

- `[slice-from:slice-to:rebin-by]` (like Python/Numpy)
- `[..., None, ...]` projection: sum over and remove dimension
- `loc[...]` in x-axis coordinates
- `iloc[...]` in integer indexes

Numpy advanced indexing: slices that leave gaps

```
>>> allcounts = numpy.ones((7, 7), int)
>>> allcounts[5, :] = allcounts[6, :] = allcounts[:, 0] = allcounts[:, 1] = 999

>>> h = aghast.Histogram(
...     [aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.above1,
...                                      loc_overflow=aghast.RealOverflow.above2))),
...      aghast.Axis(aghast.RegularBinning(5, aghast.RealInterval(-5, 5),
...                  aghast.RealOverflow(loc_underflow=aghast.RealOverflow.below2,
...                                      loc_overflow=aghast.RealOverflow.below1)))],
...     aghast.UnweightedCounts(
...         aghast.InterpretedInlineBuffer.fromarray(allcounts)))

>>> h.iloc[[True, False, True, False, True], None].axis[0].binning.dump()

IrregularBinning(
  intervals=[
    RealInterval(low=-5.0, high=-3.0),
    RealInterval(low=-1.0, high=1.0),
    RealInterval(low=3.0, high=5.0)
  ],
  overflow=RealOverflow(loc_underflow=BinLocation.above1, loc_overflow=BinLocation.above2
```

Summary

- Instead of writing $\frac{N(N-1)}{2}$ converters among N histogramming libraries, create one **ontology** and convert the N libraries into and out of it.
- Allows Python histogramming libraries to specialize: a user can access all features without one library implementing all features.
- Provides a "reduced output" format for a future **Query Service** without tying it to any particular histogramming library.
- Histograms are n-dimensional and **sliced/projected/rebinned** like Numpy.
- Choice of **Flatbuffers**:
  - Minimal wire protocol, can be selectively deserialized (like `TDirectory`).
  - Schema evolution: we can add classes as needed.
  - Multilingual: supports 12 languages (most development in C++).