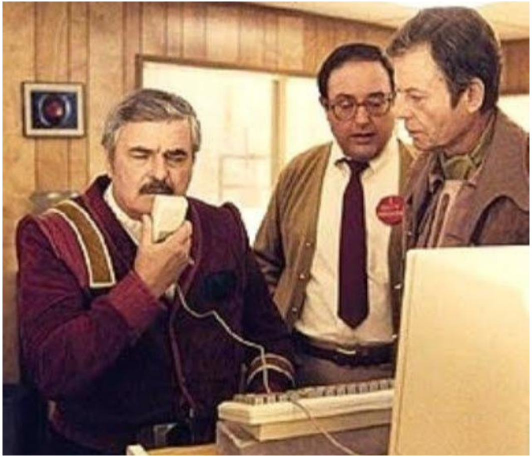
uproot update

Jim Pivarski

Princeton University – DIANA-HEP

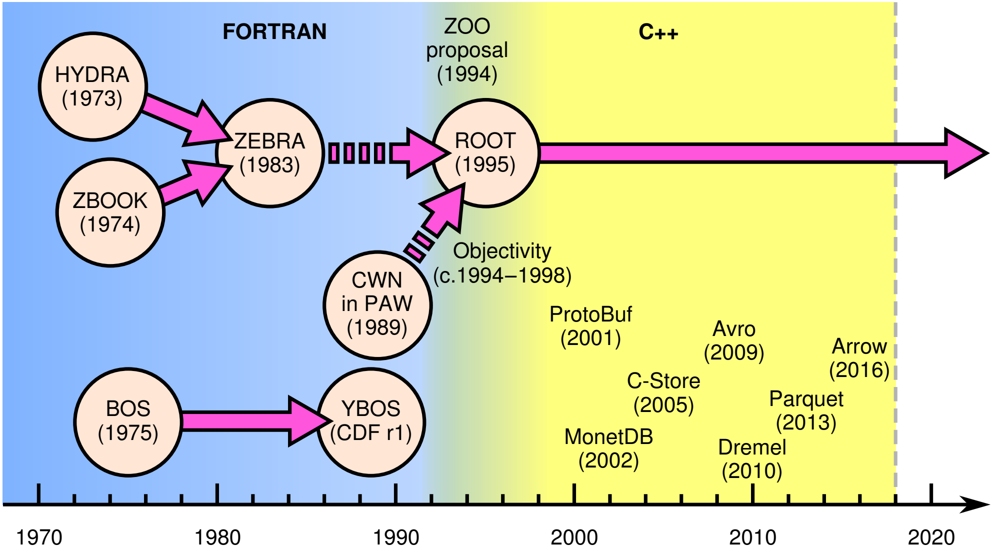
December 12, 2017

(Everybody check out the ZEBRA manual!)



"Hello, computer?"

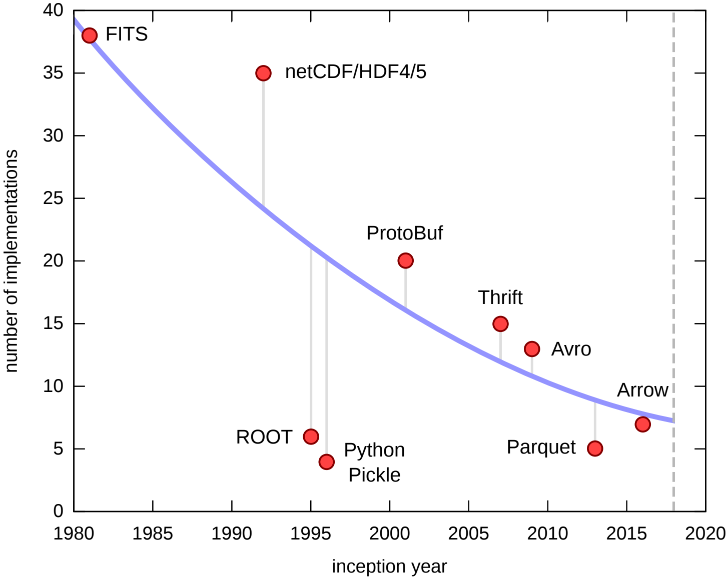
"Data bank" management systems



File formats *and specifications*

	inception	specification	implementations
FITS	1981	https://fits.gsfc.nasa.gov/standard30/fits_standard30aa.pdf	38
netCDF,HDF4/5	1992	https://support.hdfgroup.org/HDF5/doc/H5.format.html	35
ROOT	1995	some class headers like TFile and TKey; not enough info to read a file	6
Pickle	1996	*implementation* changes: 1→2 PEP-307, 3→4 PEP-3154; not a real spec	4
Protocol buffers	2001	https://developers.google.com/protocol-buffers/docs/encoding	20
Thrift	2007	**UNOFFICIAL:** https://erikvanoosten.github.io/thrift-missing-specification/	15
Avro	2009	http://avro.apache.org/docs/current/spec.html	13
Parquet	2013	http://parquet.apache.org/documentation/latest/	5
Arrow/Feather	2016	https://arrow.apache.org/docs/memory_layout.html	7

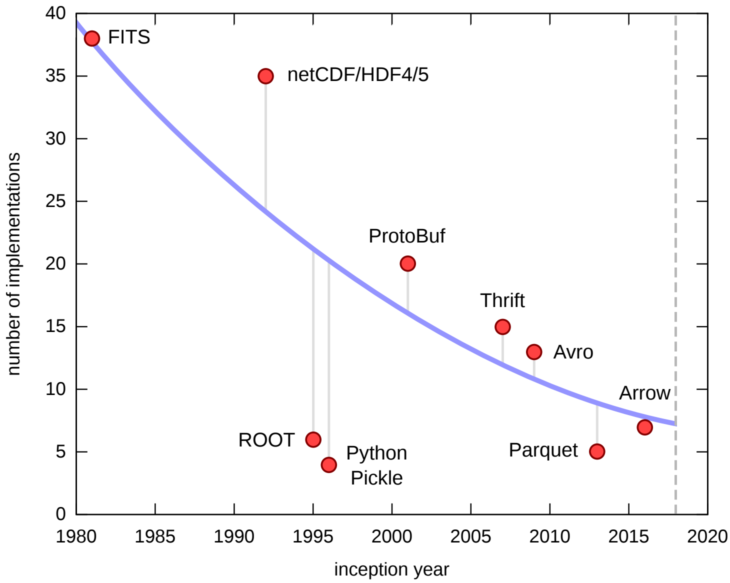
Same table, as a plot



General trend

File formats that are used for many years tend to accrete implementations, to access the data in different ways.

Same table, as a plot

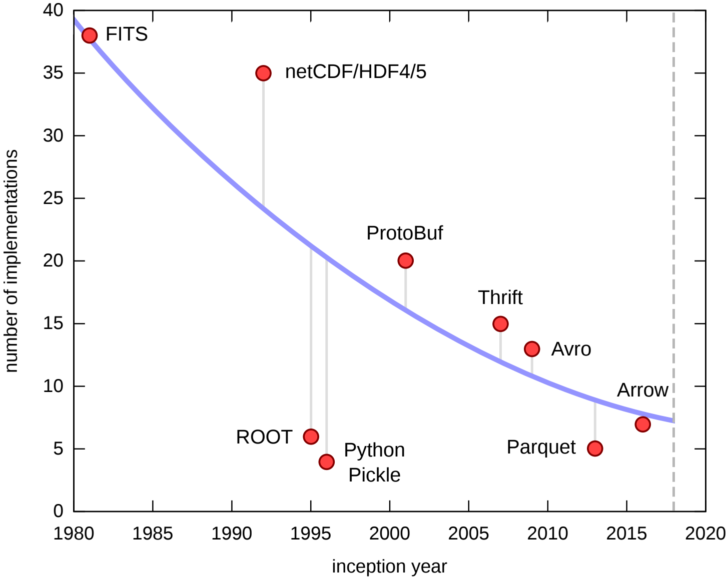


General trend

File formats that are used for many years tend to accrete implementations, to access the data in different ways.

Outlier: netCDF/HDF4/5 may be too broad to call one format.

Same table, as a plot



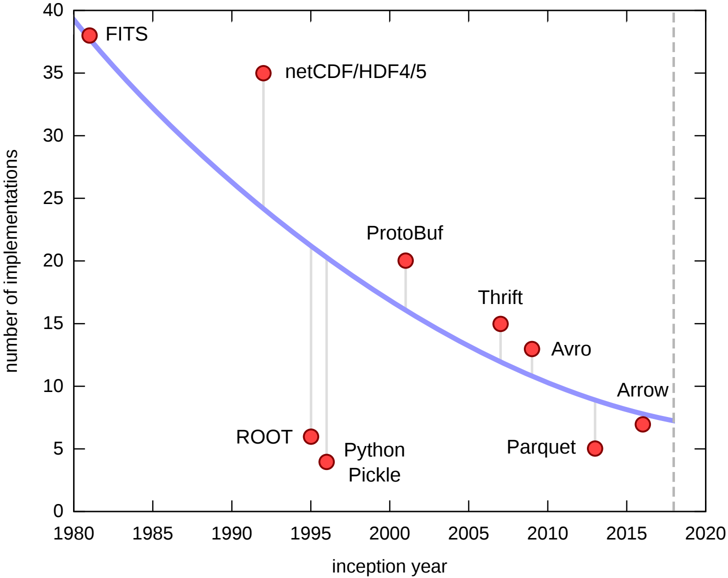
General trend

File formats that are used for many years tend to accrete implementations, to access the data in different ways.

Outlier: netCDF/HDF4/5 may be too broad to call one format.

Outlier: Python Pickle is only reimplemented when the whole language is reimplemented.

Same table, as a plot



General trend

File formats that are used for many years tend to accrete implementations, to access the data in different ways.

Outlier: netCDF/HDF4/5 may be too broad to call one format.

Outlier: Python Pickle is only reimplemented when the whole language is reimplemented.

Outlier: the ROOT format is not defined by a specification, making it hard to reimplement.

ROOT I/O implementations

ROOT	C++	ROOT itself.	The ROOT Team
JsRoot	Javascript	For interacting with ROOT in web browsers or standalone	Bertrand Bellenot, Sergey Linev (within the ROOT Team)
root4j/ spark-root	Java/Scala	For Spark and other Big Data projects that run on Java	Started by Tony Johnson in 2001, updated by Viktor Khristenko
inlib/exlib	C++	Intended as an alternative, embedded in GEANT-4	Guy Barrand
rootio	Go	go-hep ecosystem in Go	Sebastien Binet
uproot	Python	For quickly getting ROOT data into Numpy and Pandas for machine learning	Jim Pivarski (me)
	Rust?	(typesafe object ownership without a garbage collector)	

ROOT I/O implementations

ROOT	C++	ROOT itself.	The ROOT Team
JsRoot	Javascript	For interacting with ROOT in web browsers or standalone	Bertrand Bellenot, Sergey Linev (within the ROOT Team)
root4j/ spark-root	Java/Scala	For Spark and other Big Data projects that run on Java	Started by Tony Johnson in 2001, updated by Viktor Khristenko
inlib/exlib	C++	Intended as an alternative, embedded in GEANT-4	Guy Barrand
rootio	Go	go-hep ecosystem in Go	Sebastien Binet
uproot	Python	For quickly getting ROOT data into Numpy and Pandas for machine learning	Jim Pivarski (me)
	Rust?	(typesafe object ownership without a garbage collector)	

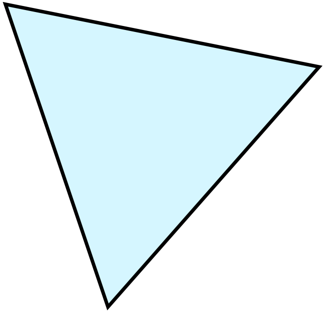
Motivations for uproot

Data Preservation

Combine Python's dynamic typing with ROOT's embedded streamers to access any data structures without the original header files/object libraries.

Physics Analysis

Complete Numpy integration for accessing machine learning libraries, convenience of Python backed by fast, compiled code.



My Research

A codebase that I can control and quickly modify to test new modes of data access, such as object stores, database-style indexing, query servers.

Motivations for uproot

Data Preservation

Combine Python's dynamic typing with ROOT's embedded streamers to access any data structures without the original header files/object libraries.

reduce version constraints on analysis

Physics Analysis

Complete Numpy integration for accessing machine learning libraries, convenience of Python backed by fast, compiled code.

share CMS public data with CS colleagues

user base in a position to try new features

My Research

A codebase that I can control and quickly modify to test new modes of data access, such as object stores, database-style indexing, query servers.

Let's try something dangerous

- The safest thing I could do is keep showing high-level slides, without getting into the code itself.

Let's try something dangerous

- The safest thing I could do is keep showing high-level slides, without getting into the code itself.
- It would be more dangerous to attempt a live demo. No matter how well you prepare, the Demo Gods will smite you.

Let's try something dangerous

- The safest thing I could do is keep showing high-level slides, without getting into the code itself.
- It would be more dangerous to attempt a live demo. No matter how well you prepare, the Demo Gods will smite you.
- Worse still, I'm going to ask you to download it and apply it to your own data while I show examples of what *should* work.

Let's try something dangerous

- The safest thing I could do is keep showing high-level slides, without getting into the code itself.
- It would be more dangerous to attempt a live demo. No matter how well you prepare, the Demo Gods will smite you.
- Worse still, I'm going to ask you to download it and apply it to your own data while I show examples of what *should* work.

You will find bugs. Not all of them will be real. Ask me or a neighbor to take a look at it; if the problem isn't obvious, submit a bug report with a small test file. Thanks!

Here it goes...

```
pip install uproot --user
```

Here it goes. . .

```
pip install uproot --user
```

https://github.com/scikit-hep/uproot

http://uproot.readthedocs.io

https://groups.google.com/forum/#!forum/uproot-users/join

Roadmap and scope

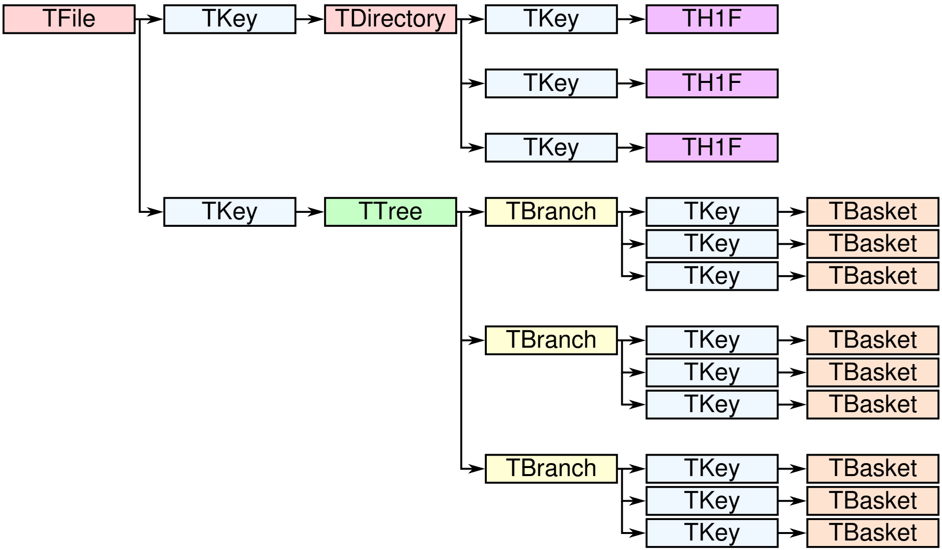
version 1.x *(last talk here)* minimalistic ROOT-to-Numpy implementation because this feature was delayed in ROOT 6.12.

version 2.x *(current)* a complete rewrite using ROOT streamers to recognize any data type. Anything can now be read from TDirectories but currently only numbers, strings, and `std::vector<number>` can be read from TTrees.

Classes that have been "split" (the default) usually satisfy the above.

version 3.x *(future)* will add the ability to write files. Will be limited to reading from one file and writing to another.

Terminology (may refer back to this slide)



Levels of trustworthiness

The feature in question is an essential part of the codebase; I designed around it and have included a formal test suite. Also, there's documentation (references, docstrings).

Mostly the same code paths as above and *informally* tested, but not yet integrated into the formal tests.

I wrote the feature for this talk and have only tested the examples shown here.

The basics: poking around


```
>>> import uproot
>>> f = uproot.open("~/TrackResonanceNtuple.root")
>>> f.keys()

['TrackResonanceNtuple;1']

>>> f.allkeys()

['TrackResonanceNtuple;1', 'TrackResonanceNtuple/twoTrack;2',
 'TrackResonanceNtuple/twoTrack;1',
 'TrackResonanceNtuple/twoMuon;1']

>>> f.classes()

[('TrackResonanceNtuple;1', <class 'uproot.rootio.ROOTDirectory'>)]

>>> f["TrackResonanceNtuple"].classes()

[('twoTrack;2', <class 'uproot.rootio.TTree'>),
 ('twoTrack;1', <class 'uproot.rootio.TTree'>),
 ('twoMuon;1', <class 'uproot.rootio.TTree'>)]
```

The basics: getting data from a TTree

```
>>> import uproot
>>> t = uproot.open("tests/samples/Zmumu.root")["events"]
>>> t.keys()

['Type', 'Run', 'Event', 'E1', 'px1', 'py1', 'pz1', 'pt1',
 'eta1', 'phi1', 'Q1', 'E2', 'px2', 'py2', 'pz2', 'pt2',
 'eta2', 'phi2', 'Q2', 'M']

>>> t["M"].array()

array([ 82.46269156,  83.62620401,  83.30846467, ...,  95.96547966,
        96.49594381,  96.65672765])

>>> t.arrays(["px1", "py1", "pz1"])

{'py1': array([ 17.433243, -16.5703623, -16.5703623, ..., 1.1994057,
 ...

>>> t.arrays()      # all of them!

 ...
```

The basics: doing meaningful calculations with them

```
>>> import numpy
>>> for px,py,pz in t.iterate(["px1","py1","pz1"], outputtype=tuple):
...     pt = numpy.sqrt(px**2 + py**2)
...     eta = numpy.arctanh(pz / numpy.sqrt(px**2 + py**2 + pz**2))
...     phi = numpy.arctan2(py, px)
...     print(pt)
...     print(eta)
...     print(phi)
...
[ 44.7322 38.8311  38.8311  ...,  32.3997  32.3997 32.5076  ]
[-1.21769 -1.05139 -1.05139 ..., -1.57044 -1.57044 -1.57078 ]
[ 2.74126 -0.44087 -0.44087 ...,  0.03702  0.03702  0.036964]
```

Also `uproot.iterate("~/files*.root", "events", ...)` for a collection of files.

Connector to an external package: Pandas

```
$ pip install pandas --user
```

```
>>> df = t.pandas.df()    # all the same arguments as t.arrays()
>>> df

               E1          E2      Event           M  Q1  Q2     Run  \
0       82.201866   60.621875   10507008   82.462692   1  -1  148031
1       62.344929   82.201866   10507008   83.626204  -1   1  148031
2       62.344929   81.582778   10507008   83.308465  -1   1  148031
3       60.621875   81.582778   10507008   82.149373  -1   1  148031
...
2302    1.199406 -26.398400  -74.532431 -153.847604   GT
2303    1.201350 -26.398400  -74.808372 -153.847604   GG

[2304 rows x 20 columns]
```

Then search the web to learn how to do exploratory analysis, make plots, apply machine learning algorithms, etc. (Or read the *Python for Data Analysis* book.)

Accessing data of non-uniform width (not scalar numbers)

```
>>> t = uproot.open("tests/samples/mc10events.root")["Events"]
>>> a = t.array("Muon.pt")      # such as std::vector<numbers>
>>> a                           # variable length for each event

jaggedarray([[ 28.07074928],
             [],
             [ 5.52336693  5.4780116 4.13222885],
             ...
             [],
             [ 6.85138178],
             []])
```

Accessing data of non-uniform width (not scalar numbers)


```
>>> t = uproot.open("tests/samples/mc10events.root")["Events"]
>>> a = t.array("Muon.pt")      # such as std::vector<numbers>
>>> a                           # variable length for each event

jaggedarray([[ 28.07074928],
             [],
             [ 5.52336693  5.4780116 4.13222885],
             ...
             [],
             [ 6.85138178],
             []])

>>> a.contents

array([ 28.07074928,   5.52336693,   5.47801161,   4.13222885, ...
         5.06344414,   6.85138178], dtype=float32)

>>> a.stops

array([ 1,  1,  4,  7,  7,  8, 13, 13, 14, 14])
```

Also, strings! (can store any variable-width data in jagged array)

```
>>> a = uproot.open("foriter2.root")["foriter2"]["data"]
>>> a

strings(['zero' 'one' 'two' ... 'twenty-nine' 'thirty'])
```

Also, strings! (can store any variable-width data in jagged array)


```
>>> a = uproot.open("foriter2.root")["foriter2"]["data"]
>>> a
strings(['zero' 'one' 'two' ... 'twenty-nine' 'thirty'])
>>> a.jaggedarray.contents
array([122, 101, 114, 111, 111, 110, 101, 116, 119, 111, 116, 104,
       114, 101, 101, 102, 111, 117, 114, 102, 105, 118, 101, 115,
       ...
       101, 105, 103, 104, 116, 116, 119, 101, 110, 116, 121,  45,
       110, 105, 110, 101, 116, 104, 105, 114, 116, 121], dtype=uint8)
>>> a.jaggedarray.stops
array([  4,   7,  10,  15,  19,  23,  26,  31,  36,  40,  43,  49,
       ...
       179, 191, 203, 214, 220])
```

As complex as STL vectors in a MiniAOD file

```
>>> t = uproot.open("~/cmssw-miniaod.root")["Events"]

>>> for basket in (t["GenEventInfoProduct_generator__HLT.obj.weights_"]
                  .iterate_baskets()):
...     print(basket)
...
[[  1.000000e+00   4.201630e-05 ...   4.242230e-05   3.990430e-05],
 [  1.000000e+00   4.201630e-05 ...   2.648060e-05   2.773620e-05],
 [  1.000000e+00   4.201630e-05 ...   4.329220e-05   4.058060e-05],
 ...
```

Though uproot was designed for analysis-ready TTrees, it will someday cover all data types by using the streamer info provided in each file.

Can already read arbitrary objects from TDirectory

```
>>> f = uproot.open("~/histograms.root")
>>> f.allclasses()

[('one;1', <class 'uproot.rootio.TH1F'>), ('two;1', <class 'upr t
 ('three;1', <class 'uproot.rootio.TH1F'>)]

>>> hist = f["one"]
>>> for n, v in hist.__dict__.items():    # class generated on the fly
...     if n.startswith("f"):             # with all the private fields
...         print n + "\t", v
...

fMarkerStyle    1
fMaximum        -1111.0
fEntries        10000.0
fLineColor      602
fContour        []
fYaxis  <TAxis 'yaxis' at 0x7e3a12cfee50>
fTsumwx2        10388.1526213
```

Histogram inspection and manipulation:


```
>>> hist.numbins, hist.low, hist.high
(10, -3.0, 3.0)
>>> hist.values
[68.0, 285.0, 755.0, 1580.0, 2296.0, 2286.0, 1570.0, 795.0, 289.0, 76.0
>>> hist[4] = 0
>>> hist.values
[68.0, 285.0, 755.0, 0, 2296.0, 2286.0, 1570.0, 795.0, 289.0, 76.0]
>>> hist.allvalues
[0.0, 68.0, 285.0, 755.0, 0, 2296.0, 2286.0, 1570.0, 795.0, 289.0, 76.0
>>> hist.underflows, hist.overflows
(0.0, 0.0)
```

You can always view histograms with a line printer

```
>>> hist.show(width=70)

                  0                                               2411
                  +--------------------------------------------------+
[-inf, -3)   0    |                                                  |
[-3, -2.4)   68   |*                                                 |
[-2.4, -1.8) 285  |******                                            |
[-1.8, -1.2) 755  |****************                                  |
[-1.2, -0.6) 0    |                                                  |
[-0.6, 0)    2296 |************************************************  |
[0, 0.6)     2286 |***********************************************   |
[0.6, 1.2)   1570 |*********************************                 |
[1.2, 1.8)   795  |****************                                  |
[1.8, 2.4)   289  |******                                            |
[2.4, 3)     76   |**                                                |
[3, inf]     0    |                                                  |
                  +--------------------------------------------------+
```

But you can also use Bokeh as your TCanvas (even remotely)

```
$ pip install bokeh --user
```

```
>>> canvas = uproot.BokehCanvas()    # canvas is a singleton
>>> canvas.show()                    # could set hosts="*", port=12345
```

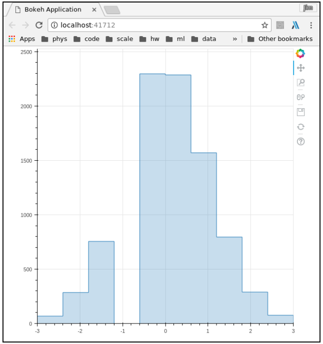
```
Created new window in existing browser session.
WARNING:tornado.access:404 GET /favicon.ico (::1) 1.09ms
```

```
>>> hist.bokeh.plot()
>>> uproot.BokehCanvas().url
```

```
'http://localhost:41712'
```

If Python is running on a remote machine, set `hosts="*"` or your IP address (for more security) and manually enter the URL in your web browser.

The (single) web browser will have a live view of anything you `hist.bokeh.plot()`.



Caching intermediate results

uproot adheres to the Python philosophy of avoiding implicit actions.

Caching must be explicit: every time you call `tree.arrays()`, it reads from the file *unless a cache is provided.*


```
>>> t = uproot.open("~/bigfile.root")["Events"]
>>> cache = {}
>>> t.array("Muon.pt", cache=cache)

jaggedarray([[ 28.07074928],
              ...,
              [ 33.39884186  30.11572647  14.1813221 ]])

>>> cache

{'/home/pivarski/bigfile.root;Events;Muon.pt;asjagged(asdtype(Bf4,Lf4,
(),()));0-47407': jaggedarray([[ 28.07074928],
              ...,
              [ 33.39884186  30.11572647  14.1813221 ]])}
```

Caching options

Any dict-like object can be a cache. (On the previous page, we used a dict.)

But dicts don't release objects when running low on memory. Therefore, uproot provides a suite of dict-like objects that *do* release the least-recently used (LRU) objects.

- `uproot.cache.MemoryCache`: a subclass of dict with LRU policy.
- `uproot.cache.ThreadSafeMemoryCache`: same with a lock for multithreading.
- `uproot.cache.DiskCache`: directory of files, uses POSIX operations like linking and file-locking for multi-process safety. Can be resumed after processes exit.

Caching options

Any dict-like object can be a cache. (On the previous page, we used a dict.)

But dicts don't release objects when running low on memory. Therefore, uproot provides a suite of dict-like objects that *do* release the least-recently used (LRU) objects.

- `uproot.cache.MemoryCache`: a subclass of dict with LRU policy.
- `uproot.cache.ThreadSafeMemoryCache`: same with a lock for multithreading.
- `uproot.cache.DiskCache`: directory of files, uses POSIX operations like linking and file-locking for multi-process safety. Can be resumed after processes exit.

They can be used in any of three arguments:

- *cache:* caches final, fully interpreted arrays.
- *basketcache:* caches decompressed TBasket data, to avoid cache-misses when slicing or interpreting the same branch different ways.
- *keycache:* tiny TKey data; probably put this in an ordinary dict without LRU.

Similarly, parallel-processing is not implicit

The `concurrent.futures` module is part of Python 3 and a package in Python 2.

```
>>> from concurrent.futures import ThreadPoolExecutor
>>> executor = ThreadPoolExecutor(16)
>>>
>>> t.arrays(["Muon.pt", "Muon.eta", "Muon.phi"], executor=executor)
```

In the above, as many as 16 threads will share the work of

- reading from disk (memory-mapped file can be multithreaded)
- decompressing TBaskets belonging to the same TBranch
- constructing arrays belonging to different TBranches.

Even though Python has a global interpreter lock (GIL), most of the numerical processing is performed in compiled code with the GIL released.

Non-blocking calls

```
>>> results = t.arrays(executor=executor, blocking=False)
>>> results
```

```
<function await at 0x747a5d2907d0>
```

Returns “await” function as soon as the work has been *submitted* to the executor.

To get the result (waiting as long as necessary), call the function:

```
>>> results()
```

```
{'CA8Puppi.nNeutrals': jaggedarray([[], [], [], ...,
                                    [], [19 68 14], [10]]),
 'AK4Puppi.hadronFlavor': jaggedarray([[5 0 5 0],
                                       [4 0 5 4 5 0],
                                       [5 0 4 0 0 5],
                                       ...,
```

Lazy arrays

```
>>> lazy = t.lazyarrays()
>>> lazy
```

```
{'CA8Puppi.nNeutrals':
     <uproot.tree._LazyArray object at 0x747a5d4f97d0>,
 'AK4Puppi.hadronFlavor':
     <uproot.tree._LazyArray object at 0x747a5d50ff10>,
 ...
```

Returns immediately and *does no work at all* until/unless you ask for items.

```
>>> lazy["Muon.pt"][:100]
```

```
jaggedarray([[ 28.07074928], ...
```

Hint: use with caching to avoid re-reading when asking for the same items twice. Nothing is implicit!

```
>>> read_only_once = t.lazyarrays(basketcache={})
```

Numba integration

Numba is a just-in-time (JIT) compiler for Python. Install Numba standalone or use CMSSW.

```
$ conda install numba    # conda, rather than pip, to get LLVM
```

Now any function preceded by `@numba.njit` gets natively compiled if Numba knows how.
Data structures produced by uproot are Numba-aware.

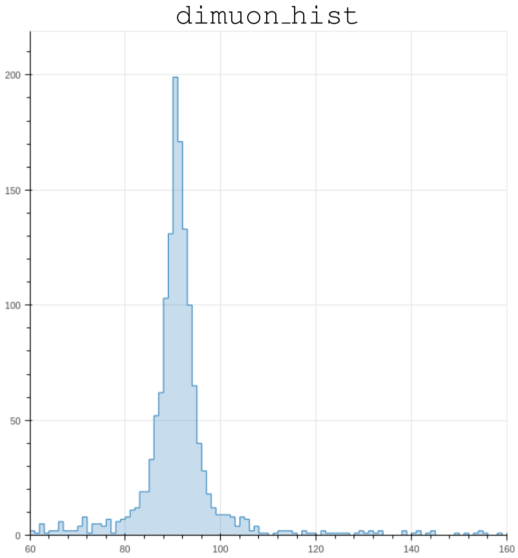
```
>>> import numba
>>> @numba.njit
... def fillhist(pthist, ptarray):
...     for event in ptarray:
...         for pt in event:
...             pthist.fill(pt)
...     return pthist                        # have to return it
...
>>> pthist = uproot.hist(100, 0, 50  )   # create empty TH1
>>> ptarray = t.array("Muon.pt")         # jagged array of arrays
>>> pthist = fillhist(pthist, ptarray)  # runs at the speed of C code
>>> pthist.show(width=70)
```

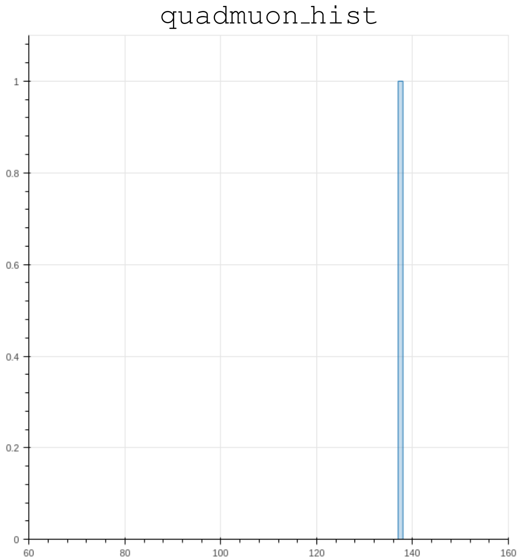
Complex, but realistic example

```python
import numba
from math import sqrt

@numba.njit
def fillhist(dimuon_hist, quadmuon_hist, NMuon, Muon_Px, Muon_Py, Muon_Pz, Muon_E):
    for event_i in range(len(NMuon)):
        totE  = 0.0
        totPx = 0.0
        totPy = 0.0
        totPz = 0.0
        for muon_i in range(NMuon[event_i]):
            for muon_j in range(muon_i + 1, NMuon[event_i]):
                E  = Muon_E[event_i][muon_i]  + Muon_E[event_i][muon_j]
                Px = Muon_Px[event_i][muon_i] + Muon_Px[event_i][muon_j]
                Py = Muon_Py[event_i][muon_i] + Muon_Py[event_i][muon_j]
                Pz = Muon_Pz[event_i][muon_i] + Muon_Pz[event_i][muon_j]
                dimuon_hist.fill(sqrt(E**2 - Px**2 - Py**2 - Pz**2))
            totE  += Muon_E[event_i][muon_i]
            totPx += Muon_Px[event_i][muon_i]
            totPy += Muon_Py[event_i][muon_i]
            totPz += Muon_Pz[event_i][muon_i]
            quadmuon_hist.fill(totE**2 - totPx**2 - totPy**2 - totPz**2)
    return dimuon_hist, quadmuon_hist
```

Previous page's plots





Functional programming

```
>>> t = uproot.open("tests/samples/Zmumu.root")["events"]
>>> from math import sqrt
>>> def mass(E1, px1, py1, pz1, E2, px2, py2, pz2):
...     return sqrt((E1 + E2)**2 - (px1 + px2)**2 - (py1 + py2)**2 - (pz1 + pz2)**2)
...
>>> t.hist(10, 70, 110, mass).show()
>>> t.filter("E1 > 10 and E2 > 10").hist(10, 70, 110, mass).show()

                     0                                                        926.1
                     +-------------------------------------------------------------+
[-inf, 70)  101      |*******                                                      |
[70, 74)    45       |***                                                          |
[74, 78)    30       |**                                                           |
[78, 82)    70       |*****                                                        |
[82, 86)    114      |********                                                     |
[86, 90)    527      |************************************                         |
[90, 94)    882      |************************************************************ |
[94, 98)    188      |*************                                                |
[98, 102)   51       |***                                                          |
[102, 106)  12       |*                                                            |
[106, 110)  7        |                                                             |
[110, inf]  13       |*                                                            |
                     +-------------------------------------------------------------+
```

Functional programming

Full suite of Spark-like methods for chaining calculations. Like everything else, they can be cached, executed in parallel, non-blocking, compiled by Numba, etc.

The following terminate a chain, causing it to be evaluated:

- `newarrays(exprs)` and `newarray(expr)`: calculate new arrays from old.
- `iterate_newarrays(exprs)`: do so iteratively over a large file.
- `reduceall(identity, increment)` and `reduce`: turn arrays into scalars.
- `hists(specs)` and `hist(numbins, low, high, dataexpr, weightexpr)`: special case of reduction for making one or many histograms.

The following can be used within a chain:

- `filter(expr)`: eliminate events.
- `define(**exprs)`: define quantities for use further down the chain.
- `intermediate(cache=None, **exprs)`: define intermediate arrays that will be computed exactly once in the chain. Cache not yet implemented.

Anything not required will not be computed, compiled, or even read from the file.

Functional programming

All of the expressions you provide are functions of one event, like

```python
def complicated(met_pt, jets_pt):
    total = met_pt
    for jet_pt in jets_pt:
        total += jet_pt
    return total
```

or maybe

```python
lambda met_pt, jets_pt: met_pt + sum(x for x in jets_pt)
```

or maybe

```python
"met_pt + sum(x for x in jets_pt)"
```

The upstream requirements, ultimately going back to the TTree branches, are taken from the names of function arguments. In the case of a function defined by a string, variable names in order of appearance are taken to be function arguments.

Although these are Python functions, they do get compiled (and inlined by LLVM).

Final remarks

uproot is designed to be:

- combinational: raw pieces that have to be put together to do a useful thing; it is not a guided path.
- consistent: most functions have long parameter lists, but they're the *same* parameters, applicable to every function that has those parameters.
- explicit: uproot does exactly what you ask it to do, so if you don't ask for caching or parallel processing, it won't happen (i.e. your memory and CPU usage won't grow unexpectedly).
- fast: any operation applied to every event is performed in compiled code; use Numba to make your user functions compiled.
- just I/O: all the non-I/O features (plotting, statistics, compilation, and perhaps fitting) are offloaded to PyData projects.