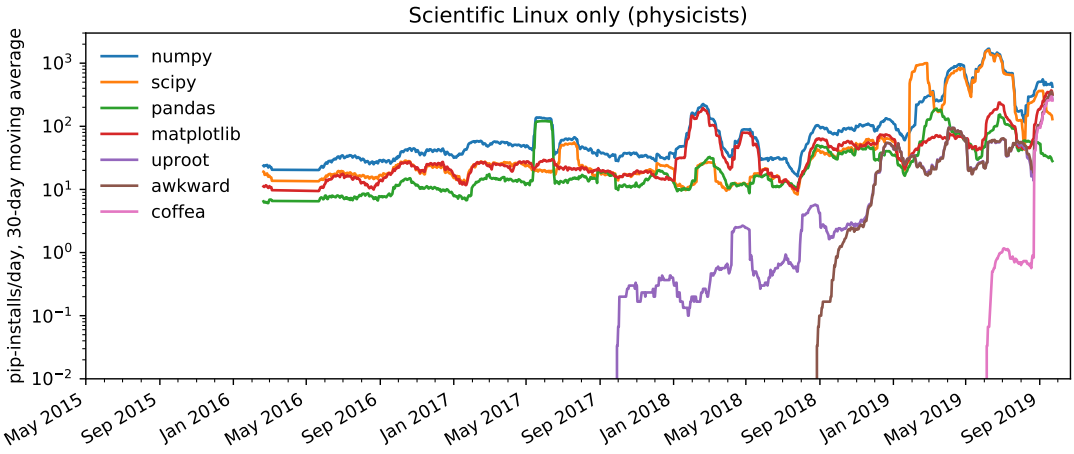
Awkward 1.0

Jim Pivarski

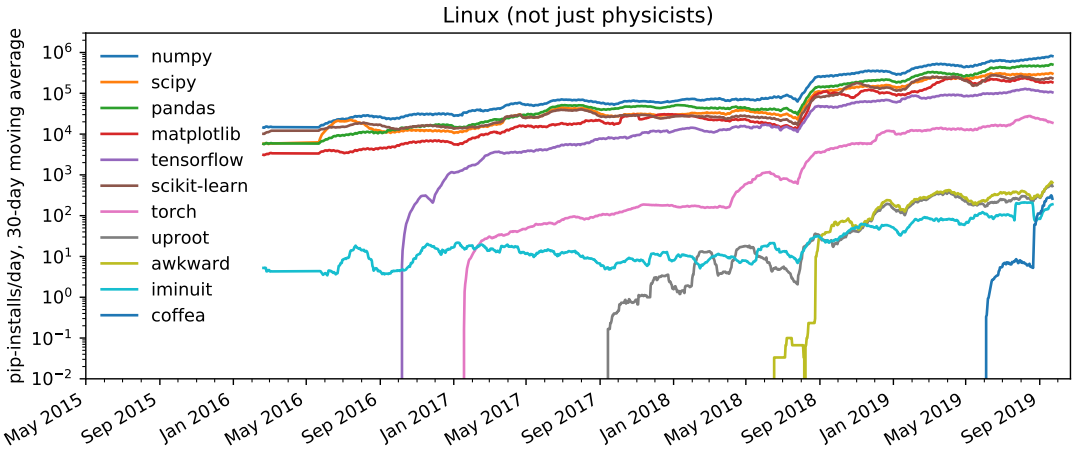
Princeton University – IRIS-HEP

October 17, 2019

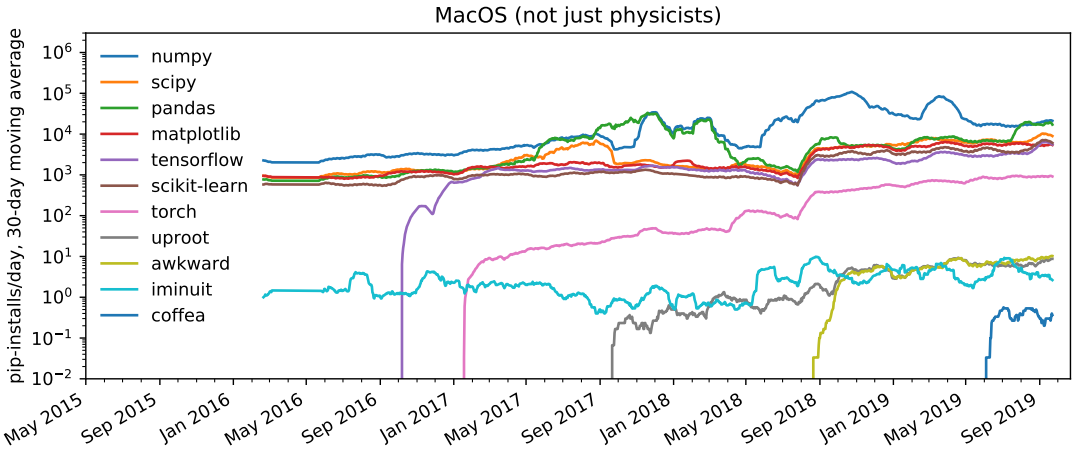
On Scientific Linux, uproot/awkward/coffea is mainstream



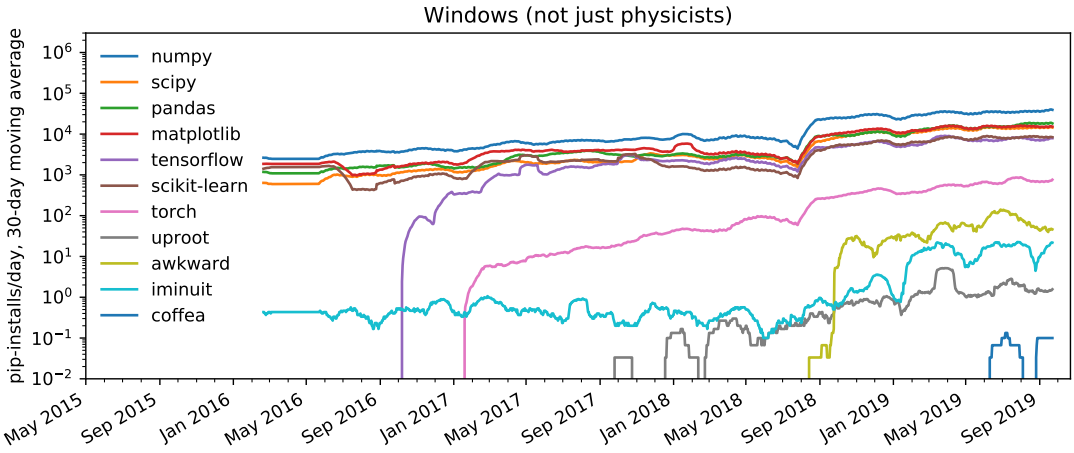
But not outside of particle physics, obviously



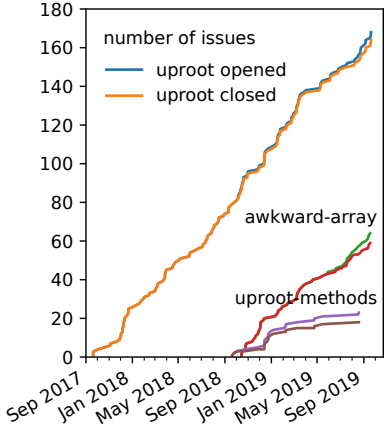
But not outside of particle physics, obviously

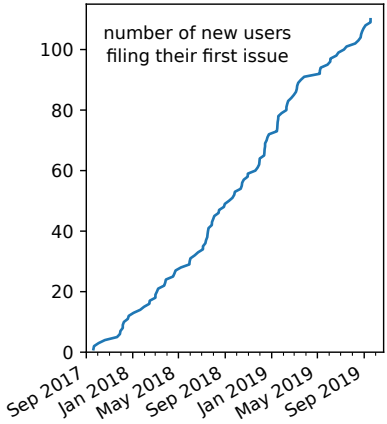


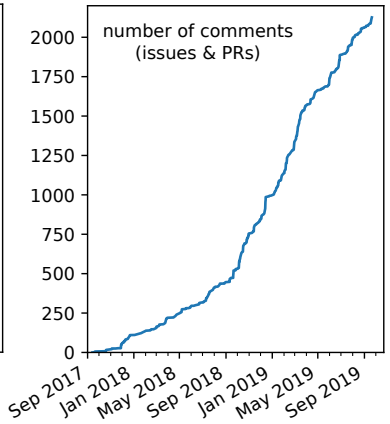
But not outside of particle physics, obviously



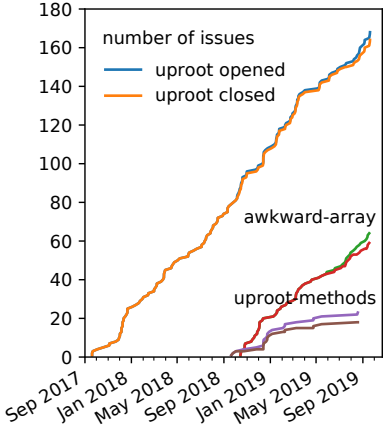
Uproot/Awkward maintainance is pretty much constant

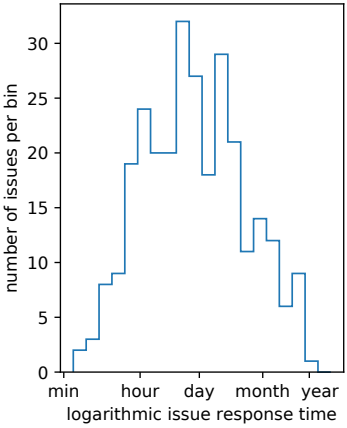


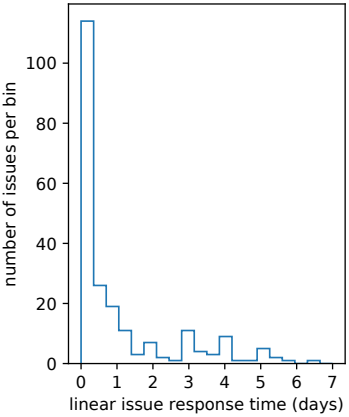




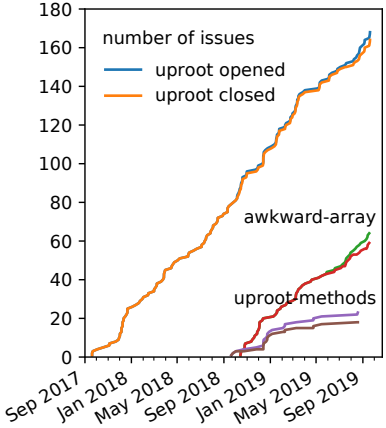
Uproot/Awkward maintainance is pretty much constant

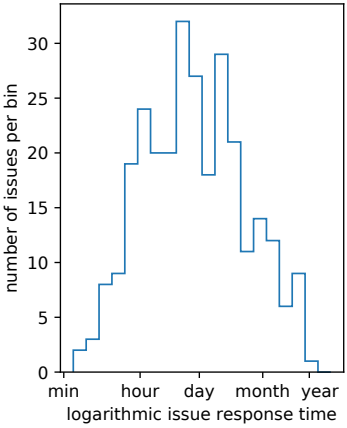


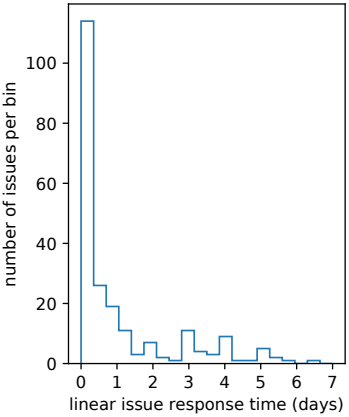




Uproot/Awkward maintainance is pretty much constant







The problem with GitHub issues is that once closed, they disappear.

Let's use StackOverflow (like most non-HEP software communities)

Questions

If you have a question about how to use uproot that is not answered in the document below, I recommend asking your question on StackOverflow with the `[uproot]` tag. (I get notified of questions with this tag.)

If you believe you have found a bug in uproot, post it on the GitHub issues tab.

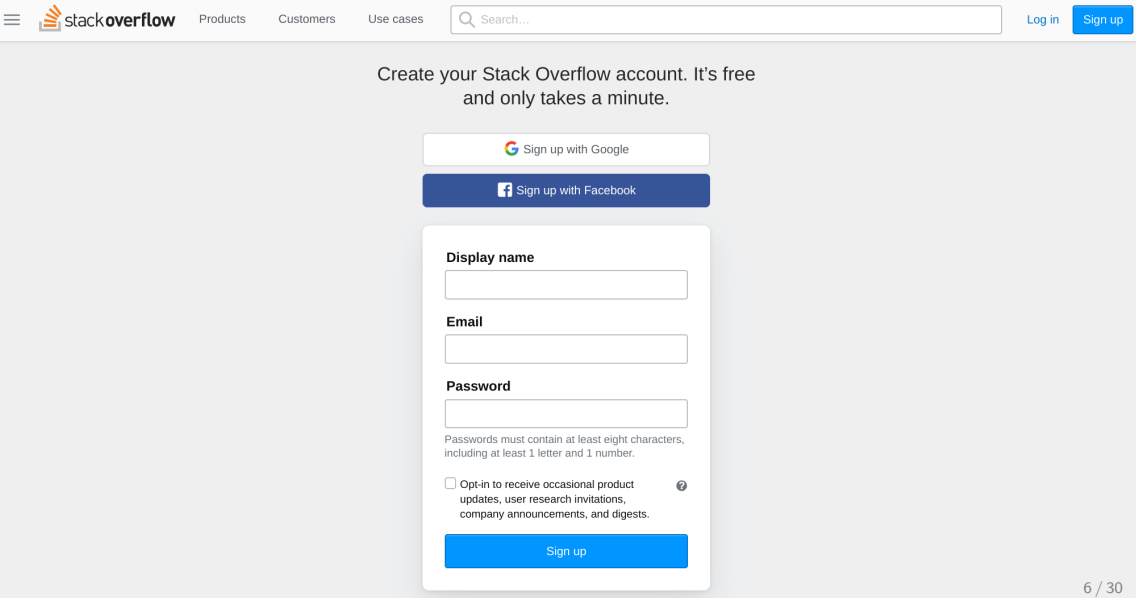
Tutorial

Run this tutorial on Binder.

Tutorial contents:

- Introduction
- What is uproot?
- Exploring a file
  - Compressed objects in ROOT files
  - Exploring a TTree
  - Some terminology
- Reading arrays from a TTree

No, seriously, do it now.



Future of Uproot and Awkward

Future of Uproot: maintenance

- TTree-writing was the last *major* feature planned.

Future of Uproot: maintenance

- TTree-writing was the last *major* feature planned.
- Bugs will be fixed.

Future of Uproot: maintenance

- TTree-writing was the last *major* feature planned.
- Bugs will be fixed.
- Uproot will keep ahead of changes in ROOT I/O.

  (Only one change in ROOT I/O in uproot's two-year existence: `TIOFeatures`.)

Future of Uproot: maintenance

- TTree-writing was the last *major* feature planned.
- Bugs will be fixed.
- Uproot will keep ahead of changes in ROOT I/O.

  (Only one change in ROOT I/O in uproot's two-year existence: `TIOFeatures`.)
- ROOT's future `RNtuple` can probably be handled with semi-independent code, as uproot-methods is now.

Future of Uproot: maintenance

- TTree-writing was the last *major* feature planned.
- Bugs will be fixed.
- Uproot will keep ahead of changes in ROOT I/O.

  (Only one change in ROOT I/O in uproot's two-year existence: `TIOFeatures`.)
- ROOT's future `RNtuple` can probably be handled with semi-independent code, as uproot-methods is now.
- "Uproot 4.0" will be a transition to Awkward 1.0.

Future of Uproot: maintenance

- TTree-writing was the last *major* feature planned.
- Bugs will be fixed.
- Uproot will keep ahead of changes in ROOT I/O.

  (Only one change in ROOT I/O in uproot's two-year existence: `TIOFeatures`.)
- ROOT's future `RNtuple` can probably be handled with semi-independent code, as uproot-methods is now.
- "Uproot 4.0" will be a transition to Awkward 1.0.

(Apart from TTree-writing, uproot has been in maintenance mode for a year already.)

Future of Awkward: consolidation

- Awkward has been tested "in the wild" for a year now.

Future of Awkward: consolidation

- Awkward has been tested "in the wild" for a year now.
- Pure Numpy implementation does some complex (clever!) things to perform jagged operations: no `for` loops allowed.

Future of Awkward: consolidation

- Awkward has been tested "in the wild" for a year now.
- Pure Numpy implementation does some complex (clever!) things to perform jagged operations: no `for` loops allowed.
  - There are limits to cleverness: many edge cases not handled.

Future of Awkward: consolidation

- Awkward has been tested "in the wild" for a year now.
- Pure Numpy implementation does some complex (clever!) things to perform jagged operations: no `for` loops allowed.
  - There are limits to cleverness: many edge cases not handled.
  - Most frequent bugs are due to Numpy usage (e.g. `numpy.max([])`).

Future of Awkward: consolidation

- Awkward has been tested "in the wild" for a year now.
- Pure Numpy implementation does some complex (clever!) things to perform jagged operations: no **`for`** loops allowed.
  - There are limits to cleverness: many edge cases not handled.
  - Most frequent bugs are due to Numpy usage (e.g. `numpy.max([])`).
  - Desire to use awkward-arrays in Numba, on GPUs, and in C++ library interfaces leads to duplication; hard to synchronize implementations.

Future of Awkward: consolidation

- Awkward has been tested "in the wild" for a year now.
- Pure Numpy implementation does some complex (clever!) things to perform jagged operations: no `for` loops allowed.
  - There are limits to cleverness: many edge cases not handled.
  - Most frequent bugs are due to Numpy usage (e.g. `numpy.max([])`).
  - Desire to use awkward-arrays in Numba, on GPUs, and in C++ library interfaces leads to duplication; hard to synchronize implementations.
- Feedback from users revealed some interface mistakes.

Future of Awkward: consolidation

- Awkward has been tested "in the wild" for a year now.
- Pure Numpy implementation does some complex (clever!) things to perform jagged operations: no **`for`** loops allowed.
  - There are limits to cleverness: many edge cases not handled.
  - Most frequent bugs are due to Numpy usage (e.g. `numpy.max([])`).
  - Desire to use awkward-arrays in Numba, on GPUs, and in C++ library interfaces leads to duplication; hard to synchronize implementations.
- Feedback from users revealed some interface mistakes.
  - `a.cross(b)` versus `awkward.cross(a, b)`

Future of Awkward: consolidation

- Awkward has been tested "in the wild" for a year now.
- Pure Numpy implementation does some complex (clever!) things to perform jagged operations: no **`for`** loops allowed.
  - There are limits to cleverness: many edge cases not handled.
  - Most frequent bugs are due to Numpy usage (e.g. `numpy.max([])`).
  - Desire to use awkward-arrays in Numba, on GPUs, and in C++ library interfaces leads to duplication; hard to synchronize implementations.
- Feedback from users revealed some interface mistakes.
  - `a.cross(b)` versus `awkward.cross(a, b)`
  - User-visible `JaggedArray` versus `ChunkedArray(JaggedArray)`

Awkward 1.0

Awkward 1.0 is a *rewrite*, improving structure and interface

scikit-hep / **awkward-1.0**

Unwatch 3 | Star 2 | Fork 0

Code | Issues 0 | Pull requests 1 | Actions | Projects 0 | Wiki | Security | Insights | Settings

Development of awkward 1.0, to replace scikit-hep/awkward-array in 2020. Edit

Manage topics

42 commits | **2** branches | **12** releases | **1** contributor | BSD-3-Clause

C++ 56.0% | **Python** 42.9% | **Other** 1.1%

Branch: **master** | New pull request | Create new file | Upload files | Find File | Clone or download

jpivarski Update VERSION_INFO — Latest commit c052ece 2 hours ago

.ci	Try to fix manylinux1 deployment again.	6 days ago
awkward1	Access ListArray::getitem in Numba. (#12)	3 hours ago
docs	Deep __getitem__ in C++. (#8)	11 days ago
include/awkward	Access ListArray::getitem in Numba. (#12)	3 hours ago
pybind11 @ e43e1cc	Include pybind11 as a submodule.	2 months ago
src	Access ListArray::getitem in Numba. (#12)	3 hours ago
studies	Deep __getitem__ in C++. (#8)	11 days ago
tests	Access ListArray::getitem in Numba. (#12)	3 hours ago

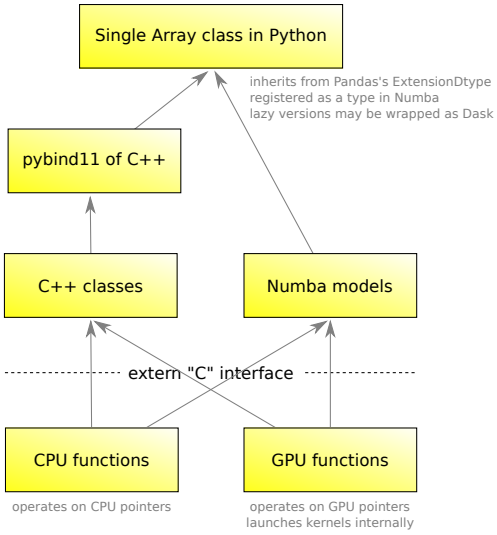
Layered architecture

Layer 1: Python user interface: a single `awkward.Array` class.

Layer 2: Structure classes, "layout" (e.g. `ListArray`/`RecordArray`).

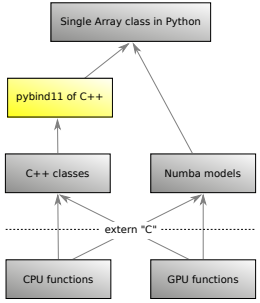
Layer 3: Memory management, array allocation and ownership; reference counting.

Layer 4: Implementations, where we write **`for`** loops. The only layer that needs to be optimized for speed.



Layer 2: pybind11 of C++

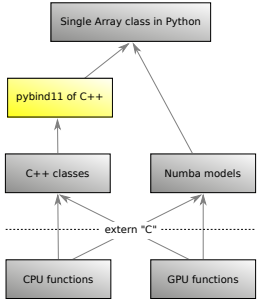
```
import numpy
import awkward1

content = awkward1.layout.NumpyArray(numpy.arange(10)*1.1)
listA   = awkward1.layout.ListOffsetArray32(
              awkward1.layout.Index32(numpy.array([0, 3, 3, 5, 6, 10])),
              content)
listB   = awkward1.layout.ListOffsetArray32(
              awkward1.layout.Index32(numpy.array([0, 3, 4, 4, 5])),
              listA)
```



Layer 2: pybind11 of C++

```
import numpy
import awkward1

content = awkward1.layout.NumpyArray(numpy.arange(10)*1.1)
listA   = awkward1.layout.ListOffsetArray32(
              awkward1.layout.Index32(numpy.array([0, 3, 3, 5, 6, 10])),
              content)
listB   = awkward1.layout.ListOffsetArray32(
              awkward1.layout.Index32(numpy.array([0, 3, 4, 4, 5])),
              listA)

print(awkward1.tolist(listA))
```

```
[[0.0, 1.1, 2.2], [], [3.3, 4.4], [5.5], [6.6, 7.7, 8.8, 9.9]]
```



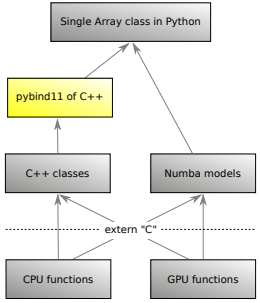
Layer 2: pybind11 of C++

```
import numpy
import awkward1

content = awkward1.layout.NumpyArray(numpy.arange(10)*1.1)
listA   = awkward1.layout.ListOffsetArray32(
              awkward1.layout.Index32(numpy.array([0, 3, 3, 5, 6, 10])),
              content)
listB   = awkward1.layout.ListOffsetArray32(
              awkward1.layout.Index32(numpy.array([0, 3, 4, 4, 5])),
              listA)

print(awkward1.tolist(listA))
```

```
[[0.0, 1.1, 2.2], [], [3.3, 4.4], [5.5], [6.6, 7.7, 8.8, 9.9]]
```

```
print(awkward1.tolist(listB))
```

```
[[[0.0, 1.1, 2.2], [], [3.3, 4.4]], [[5.5]], [], [[6.6, 7.7, 8.8, 9.9]]]
```



Layer 2: pybind11 of C++

```
import numpy
import awkward1

content = awkward1.layout.NumpyArray(numpy.arange(10)*1.1)
listA   = awkward1.layout.ListOffsetArray32(
              awkward1.layout.Index32(numpy.array([0, 3, 3, 5, 6, 10])),
              content)
listB   = awkward1.layout.ListOffsetArray32(
              awkward1.layout.Index32(numpy.array([0, 3, 4, 4, 5])),
              listA)

print(awkward1.tolist(listA))
```

```
[[0.0, 1.1, 2.2], [], [3.3, 4.4], [5.5], [6.6, 7.7, 8.8, 9.9]]
```

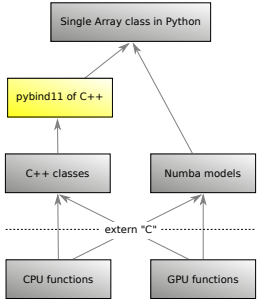
```
print(awkward1.tolist(listB))
```

```
[[[0.0, 1.1, 2.2], [], [3.3, 4.4]], [[5.5]], [], [[6.6, 7.7, 8.8, 9.9]]]
```

```
print(awkward1.tolist(listB[:, ::-1, ::2]))
```

```
[[[3.3], [], [0.0, 2.2]], [[5.5]], [], [[6.6, 8.8]]]
```

(old awkward-array can't do this)



Layer 2: pybind11 of C++

```
import numpy
import awkward1

content = awkward1.layout.NumpyArray(numpy.arange(10)*1.1)
listA   = awkward1.layout.ListOffsetArray32(
              awkward1.layout.Index32(numpy.array([0, 3, 3, 5, 6, 10])),
              content)
listB   = awkward1.layout.ListOffsetArray32(
              awkward1.layout.Index32(numpy.array([0, 3, 4, 4, 5])),
              listA)

print(awkward1.tolist(listA))
```

```
[[0.0, 1.1, 2.2], [], [3.3, 4.4], [5.5], [6.6, 7.7, 8.8, 9.9]]
```

```
print(awkward1.tolist(listB))
```

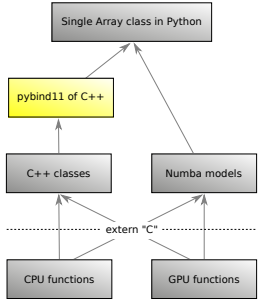
```
[[[0.0, 1.1, 2.2], [], [3.3, 4.4]], [[5.5]], [], [[6.6, 7.7, 8.8, 9.9]]]
```

```
print(awkward1.tolist(listB[:, ::-1, ::2]))
```

```
[[[3.3], [], [0.0, 2.2]], [[5.5]], [], [[6.6, 8.8]]]     (old awkward-array can't do this)
```

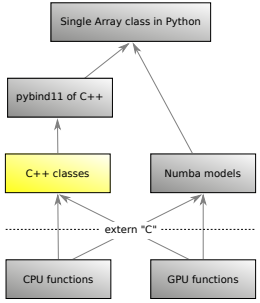
```
print(awkward1.tolist(listB[[0, 0, -1, -1], [0, -1, 0, -1], 1:-1]))
```

```
[[1.1], [], [7.7, 8.8], [7.7, 8.8]]                      (mixing fancy and basic indexing)
```



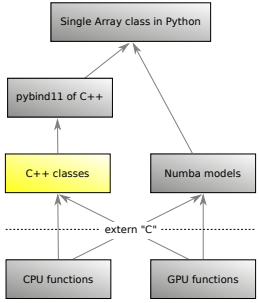
Layer 3: C++ classes

```
Index32 offsets(6);
offsets.ptr().get()[0] = 0;     offsets.ptr().get()[3] = 5;
offsets.ptr().get()[1] = 3;     offsets.ptr().get()[4] = 6;
offsets.ptr().get()[2] = 3;     offsets.ptr().get()[5] = 10;

auto raw = new RawArrayOf<double>(Identity::none(), 10);
for (int i = 0;  i < 10;  i++) {
  *raw->borrow(i) = 1.1*i;
}
std::shared_ptr<Content> content(raw);
std::shared_ptr<Content> list(new ListOffsetArray32(Identity::none(),
                                                    offsets, content));
```



Layer 3: C++ classes

```
Index32 offsets(6);
offsets.ptr().get()[0] = 0;     offsets.ptr().get()[3] = 5;
offsets.ptr().get()[1] = 3;     offsets.ptr().get()[4] = 6;
offsets.ptr().get()[2] = 3;     offsets.ptr().get()[5] = 10;

auto raw = new RawArrayOf<double>(Identity::none(), 10);
for (int i = 0;  i < 10;  i++) {
  *raw->borrow(i) = 1.1*i;
}
std::shared_ptr<Content> content(raw);
std::shared_ptr<Content> list(new ListOffsetArray32(Identity::none(),
                                                    offsets, content));
```




```
tostring(list);
```

```
"[[0, 1.1, 2.2], [], [3.3, 4.4], [5.5], [6.6, 7.7, 8.8, 9.9]]"
```

```
tostring(list.get()->getitem_range(1, -1));
```

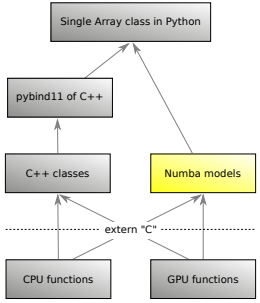
```
"[[], [3.3, 4.4], [5.5]]"
```

```
tostring(list.get()->getitem(slice(new SliceRange(2, Slice::none(), Slice::none()),
                                   new SliceRange(Slice::none(), Slice::none(), -1))));
```

```
"[[4.4, 3.3], [5.5], [9.9, 8.8, 7.7, 6.6]]"
```

Layer 3: Numba models

```
import numba

@numba.jit(nopython=True)
def iterate(array):
    out = 0.0
    for subarray in array:                  # for loops in a Numba-
        for subsubarray in subarray:        # compiled function are
            for item in subsubarray:        # just as fast as C or C++
                out += item
    return out

print(iterate(listB))
49.5
```



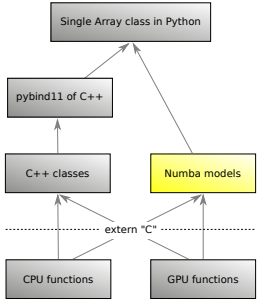
Layer 3: Numba models

```
import numba

@numba.jit(nopython=True)
def iterate(array):
    out = 0.0
    for subarray in array:                  # for loops in a Numba-
        for subsubarray in subarray:        # compiled function are
            for item in subsubarray:        # just as fast as C or C++
                out += item
    return out

print(iterate(listB))
```

```
49.5
```

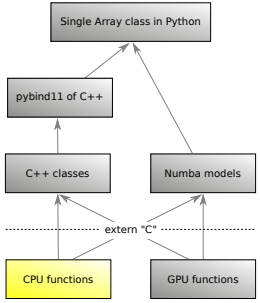
```
@numba.jit(nopython=True)
def slices(array):                          # same slicing works in the compiled environment
    return (array[:, ::-1, ::2],
            array[[0, 0, -1, -1], [0, -1, 0, -1], 1:-1])

one, two = slices(listB)
print(awkward1.tolist(one), awkward1.tolist(two))
```

```
[[[3.3], [], [0.0, 2.2]], [[5.5]], [], [[6.6, 8.8]]]     (same results as before)
[[1.1], [], [7.7, 8.8], [7.7, 8.8]]
```



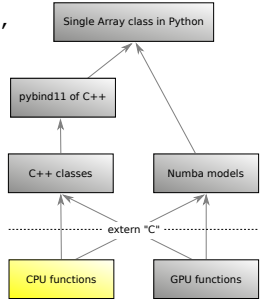
Layer 4: CPU functions

```
{
  for (int64_t i = 0;  i < lenstarts;  i++) {
    int64_t length = fromstops[stopsoffset + i] -
                     fromstarts[startsoffset + i];
    int64_t regular_at = at;
    if (regular_at < 0) {
      regular_at += length;
    }
    if (!(0 <= regular_at  &&  regular_at < length)) {
      return failure("index out of range", i, at);
    }
    tocarry[i] = fromstarts[startsoffset + i] + regular_at;
  }
  return success();
}
```



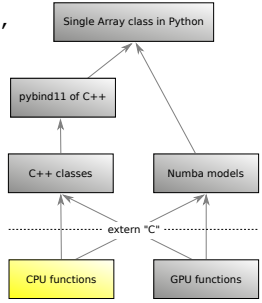
Layer 4: CPU functions

```
template <typename C, typename T>
Error awkward_listarray_getitem_next_at(T* tocarry, const C* fromstarts,
          const C* fromstops, int64_t lenstarts, int64_t startsoffset,
          int64_t stopsoffset, int64_t at)
{
  for (int64_t i = 0;  i < lenstarts;  i++) {
    int64_t length = fromstops[stopsoffset + i] -
                     fromstarts[startsoffset + i];
    int64_t regular_at = at;
    if (regular_at < 0) {
      regular_at += length;
    }
    if (!(0 <= regular_at  &&  regular_at < length)) {
      return failure("index out of range", i, at);
    }
    tocarry[i] = fromstarts[startsoffset + i] + regular_at;
  }
  return success();
}
```



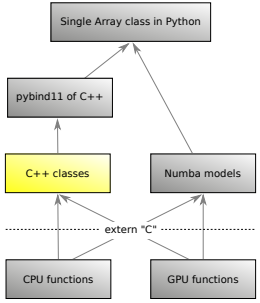
Layer 4: CPU functions

```
template <typename C, typename T>
Error awkward_listarray_getitem_next_at(T* tocarry, const C* fromstarts,
          const C* fromstops, int64_t lenstarts, int64_t startsoffset,
          int64_t stopsoffset, int64_t at)
{
  for (int64_t i = 0;  i < lenstarts;  i++) {
    int64_t length = fromstops[stopsoffset + i] -
                     fromstarts[startsoffset + i];
    int64_t regular_at = at;
    if (regular_at < 0) {
      regular_at += length;
    }
    if (!(0 <= regular_at  &&  regular_at < length)) {
      return failure("index out of range", i, at);
    }
    tocarry[i] = fromstarts[startsoffset + i] + regular_at;
  }
  return success();
}
extern "C" {
  Error awkward_listarray32_getitem_next_at_64(int64_t* tocarry, const int32_t* fromstarts,
            const int32_t* fromstops, int64_t lenstarts, int64_t startsoffset,
            int64_t stopsoffset, int64_t at);
```



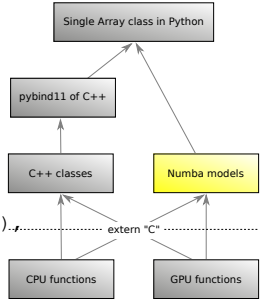
Layer 3: C++ classes

```
if (head.get() == nullptr) {
  return shallow_copy();
}

else if (SliceAt* at = dynamic_cast<SliceAt*>(head.get())) {
  std::shared_ptr<SliceItem> nexthead = tail.head();
  Slice nexttail = tail.tail();
  Index64 nextcarry(lenstarts);
  Error err = awkward_listarray32_getitem_next_at_64(
    nextcarry.ptr().get(),
    starts_.ptr().get(),
    stops_.ptr().get(),
    lenstarts,
    starts_.offset(),
    stops_.offset(),
    at->at());
  util::handle_error(err, classname(), id_.get());
  std::shared_ptr<Content> nextcontent = content_.get()->carry(nextcarry);
  return nextcontent.get()->getitem_next(nexthead, nexttail, advanced);
}

else if (SliceRange* range = dynamic_cast<SliceRange*>(head.get())) {
  ...
```



Layer 3: Numba models

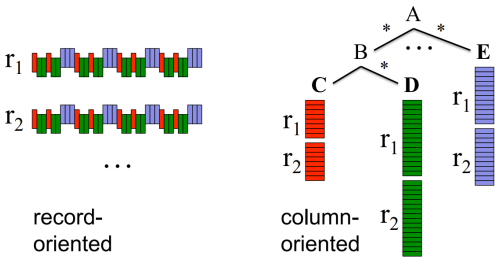
```
if isinstance(headtpe, numba.types.Integer):
  if arraytype.bitwidth == 64:
    kernel = cpu.kernels.awkward_listarray64_getitem_next_at_64
  elif arraytype.bitwidth == 32:
    kernel = cpu.kernels.awkward_listarray32_getitem_next_at_64

  nextcarry = util.newindex64(context, builder, numba.int64, lenstarts)
  util.call(context, builder, kernel,
    (util.arrayptr(context, builder, util.index64type, nextcarry),
     util.arrayptr(context, builder, arraytype.startstype, proxyin.starts),
     util.arrayptr(context, builder, arraytype.stopstype, proxyin.stops),
     lenstarts,
     context.get_constant(numba.int64, 0),
     context.get_constant(numba.int64, 0),
     util.cast(context, builder, headtpe, numba.int64, headval)),
    "in {}, indexing error".format(arraytype.shortname))
  nextcontenttype = arraytype.contenttype.carry()
  nextcontentval = arraytype.contenttype.lower_carry(context, builder, arraytype.contenttype,
                                                     util.index64type, proxyin.content, nextcarry)
  return nextcontenttype.lower_getitem_next(context, builder, nextcontenttype, tailtype,
                                            nextcontentval, tailval, advanced)
elif isinstance(headtpe, numba.types.SliceType):
  ...
```



Still following the array-at-a-time approach

Slow Python has been replaced by slow C++ (dynamic dispatch, runtime type-checks).

But only $\mathcal{O}$(depth of type) operations are performed in C++; $\mathcal{O}$(number of events) operations are performed in single-pass cpu-functions.



Deliverables

Compilable by CMake (for pure C++) or `python setup.py install`.

cpu-kernels.so suite of Layer 4 functions with an **`extern "C"`** interface, which can be accessed by any language (notably C++ and Numba).

libawkward.so library of Layer 3 classes that can be used in any C++ project.

awkward1 Python library: Layer 1 (user interface), Layer 2 (extension module), and Layer 3 (Numba extensions, if Numba is installed).

`https://pypi.org/project/awkward1/#files` hosts 29 binary wheels and 1 source package; most users will `pip install` without compiling.

Mathematical aspects

Functional programming for arrays

Arrays are functions:

$$\texttt{array} : [0, n) \to \texttt{dtype}$$

such that `array[i]` for integer `i` is a function call.

Functional programming for arrays

Arrays are functions:

$$\texttt{array} : [0, n) \to \texttt{dtype}$$

such that `array[i]` for integer `i` is a function call.

Indexing by an integer array is functional composition:

$$\texttt{ints} : [0, m) \to [0, n) \qquad \Rightarrow \qquad \texttt{array[ints]} : [0, m) \to \texttt{dtype}$$

Functional programming for arrays

Arrays are functions:

$$\texttt{array} : [0, n) \to \texttt{dtype}$$

such that `array[i]` for integer `i` is a function call.

Indexing by an integer array is functional composition:

$$\texttt{ints} : [0, m) \to [0, n) \qquad \Rightarrow \qquad \texttt{array[ints]} : [0, m) \to \texttt{dtype}$$

So if `f` and `g` are $\mathbb{Z}^{\geq 0} \to \mathbb{Z}^{\geq 0}$ functions and we sample them as `F` and `G`,

```
F   = numpy.array([f(i)    for i in range(...)])
G   = numpy.array([g(i)    for i in range(...)])
GoF = numpy.array([g(f(i)) for i in range(...)])
```

then $\boxed{\texttt{G[F]} = \texttt{GoF}}$.

Functional programming for arrays

Arrays are functions:

$$\texttt{array} : [0, n) \to \texttt{dtype}$$

such that `array[i]` for integer `i` is a function call.

Indexing by an integer array is functional composition:

$$\texttt{ints} : [0, m) \to [0, n) \qquad \Rightarrow \qquad \texttt{array[ints]} : [0, m) \to \texttt{dtype}$$

So if `f` and `g` are $\mathbb{Z}^{\geq 0} \to \mathbb{Z}^{\geq 0}$ functions and we sample them as `F` and `G`,

```
F   = numpy.array([f(i)    for i in range(...)])
G   = numpy.array([g(i)    for i in range(...)])
GoF = numpy.array([g(f(i)) for i in range(...)])
```

then $\boxed{\texttt{G[F]} = \texttt{GoF}}$.

Functional composition is associative: if `H` is any array, $\boxed{\texttt{H[G][F]} = \texttt{H[G[F]]}}$.

Associativity of integer-array indexing is a very useful feature

scikit-hep / awkward-1.0

Unwatch 3 | Star 2 | Fork 0

Code | Issues 0 | Pull requests 0 | Actions | Projects 0 | Wiki | Security | Insights | Settings

Branch: master | awkward-1.0 / docs / theory / arrays-are-functions.pdf | Find file | Copy path

jpivarski Deep __getitem__ in C++, (v8) ecf83ca 20 days ago

1 contributor

101 KB | Download | History

Array manipulations as functional programming

Jim Pivarski

September 19, 2019

Introduction

The central features of an array library like Numpy or Awkward Array simplify if we think of arrays as functions and these features as function composition. A one-dimensional array of `dtype` d (e.g. `int32` or `float64`) can be thought of as a function from integer indexes to members of d. Thus,

`array[i]`

becomes

$$\texttt{array} : \mathbb{Z} \to d$$

because given an integer $\texttt{i} \in \mathbb{Z}$, it returns a value in d. In Python, this function is the implementation of the array's `__getitem__` method.

Specified this way, this is a partial function[1]—for some integers, it raises an exception rather than returning a value in d. (Integers greater than or equal to the array's length or less than its negated length, if the array implements Python's negative indexing, are outside the bounds of the array and do not return a value.) It can be made into a total function by restricting the domain to $[0, n)$ where n is the length of the array:

$$\texttt{array} : [0, n) \to d.$$

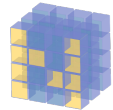
We can choose $[0, n)$ as the domain and work with total functions or $\mathbb{Z}$ as the domain and

https://github.com/scikit-hep/awkward-1.0/blob/master/docs/theory/arrays-are-functions.pdf

Used throughout `getitem_next` to "carry" information from one level of recursion to the next, in analogy with carrying digits in longhand addition.

Pandas-style indexing

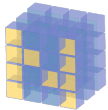
Indexing distinguishes Numpy from Pandas and xarray



vs.

Awkward 1.0 operations will optionally pass around an `Identity`, an extra array that attaches permanent coordinates to each number, list, and record in the data.

Indexing distinguishes Numpy from Pandas and xarray



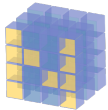
vs.

Awkward 1.0 operations will optionally pass around an `Identity`, an extra array that attaches permanent coordinates to each number, list, and record in the data.

Good for error messages. . .

```
>>> dataset.setid()    # generate Identities
>>> primary_jet_for_muon = dataset[:, "muons", :, "jets", 0]
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: in ListArray32 at id[10374, "muons", 1, "jets"] attempting
            to get 0, index out of range
```

Indexing distinguishes Numpy from Pandas and xarray



vs.

Awkward 1.0 operations will optionally pass around an `Identity`, an extra array that attaches permanent coordinates to each number, list, and record in the data.

Good for error messages. . .

```
>>> dataset.setid()    # generate Identities
>>> primary_jet_for_muon = dataset[:, "muons", :, "jets", 0]
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: in ListArray32 at id[10374, "muons", 1, "jets"] attempting
            to get 0, index out of range
```

. . . but it was motivated by investigations into set-based languages

https://github.com/jpivarski/PartiQL

```
# For events with at least three leptons (electrons or muons) and a same-flavor
# opposite-sign lepton pair, find the same-flavor opposite-sign lepton pair with a
# mass closest to 91.2 GeV; make a histogram of the pT of the leading other lepton.

leptons = electrons union muons

cut count(leptons) >= 3 named "three_leptons" {
    Z = electrons as (lep1, lep2) union muons as (lep1, lep2)
            where lep1.charge != lep2.charge
            min by abs(mass(lep1, lep2) - 91.2)

    third = leptons except [Z.lep1, Z.lep2] max by pt
    hist third.pt by regular(100, 0, 250) named "third_pt"
}
```

An `Identity` (surrogate-key index) is needed to define set operations like **`join`**, **`cross`**, **`union`**, and **`except`** such that particles are never duplicated.

Multi-paradigm arrays

With a dataset in Awkward form (from TTrees, RNtuples, Arrow...), we want to

Multi-paradigm arrays

With a dataset in Awkward form (from TTrees, RNtuples, Arrow...), we want to

- perform Numpy-like slicing, reduction, and vectorized operations,

Multi-paradigm arrays

With a dataset in Awkward form (from TTrees, RNtuples, Arrow...), we want to

- perform Numpy-like slicing, reduction, and vectorized operations,
- enter a compiled Numba function for imperative code in Python,

Multi-paradigm arrays

With a dataset in Awkward form (from TTrees, RNtuples, Arrow...), we want to

- perform Numpy-like slicing, reduction, and vectorized operations,
- enter a compiled Numba function for imperative code in Python,
- pass to/from a C++ library (e.g. jagged array of Lorentz vectors to FastJet),

Multi-paradigm arrays

With a dataset in Awkward form (from TTrees, RNtuples, Arrow...), we want to

- perform Numpy-like slicing, reduction, and vectorized operations,
- enter a compiled Numba function for imperative code in Python,
- pass to/from a C++ library (e.g. jagged array of Lorentz vectors to FastJet),
- compute combinatorics with a HEP-specific domain specific language,

Multi-paradigm arrays

With a dataset in Awkward form (from TTrees, RNtuples, Arrow...), we want to

- perform Numpy-like slicing, reduction, and vectorized operations,
- enter a compiled Numba function for imperative code in Python,
- pass to/from a C++ library (e.g. jagged array of Lorentz vectors to FastJet),
- compute combinatorics with a HEP-specific domain specific language,

interchangeably.

Multi-paradigm arrays

With a dataset in Awkward form (from TTrees, RNtuples, Arrow...), we want to

- perform Numpy-like slicing, reduction, and vectorized operations,
- enter a compiled Numba function for imperative code in Python,
- pass to/from a C++ library (e.g. jagged array of Lorentz vectors to FastJet),
- compute combinatorics with a HEP-specific domain specific language,

interchangeably.

Awkward 1.0 is intended as a solid foundation for that future.

When can I try it?

Nowish: it is in a testable state (for Coffea and thrill-seekers).

Will be minimally usable for physics analysis in "early 2020."

Start an `import awkward` → `import awkward0`
`import awkward1` → `import awkward` transition by spring.

Roadmap

The rough estimate for development time to a minimally usable library for physics was six months, starting in late August (i.e. finishing in late February). **Progress is currently on track.**

Approximate order of implementation

Completed items are ☑check-marked. See closed PRs for more details.

- [x] Cross-platform, cross-Python version build and deploy process. Regularly deploying 30 wheels after closing each PR.
- [x] Basic `NumpyArray`, `ListArray`, and `ListOffsetArray` with `__getitem__` for int/slice and `__iter__` in C++/pybind11 to establish structure and ensure proper reference counting.
- [x] Introduce `Identity` as a Pandas-style index to pass through `__getitem__`.
- [x] Reproduce all of the above as Numba extensions (make `NumpyArray`, `ListArray`, and `ListOffsetArray` usable in Numba-compiled functions).
- [x] Error messages with location-of-failure information if the array has an `Identity` (except in Numba).
- [x] Fully implement `__getitem__` for int/slice/intarray/boolarray/tuple (placeholders for newaxis/ellipsis), with perfect agreement with Numpy basic/advanced indexing, to all levels of depth.
- [ ] Appendable arrays (a distinct phase from readable arrays, when the type is still in flux) to implement `awkward.fromiter` in C++.
- [ ] JSON → Awkward via header-only simdjson and `awkward.fromiter`.
- [ ] Explicit broadcasting functions for jagged and non-jagged arrays and scalars.
- [ ] Extend `__getitem__` to take jagged arrays of integers and booleans (same behavior as old).
- [ ] Full suite of array types:
  - [x] `RawArray`: flat, 1-dimensional array type for pure C++ (header-only).
  - [x] `NumpyArray`: rectilinear, N-dimensional array type without Python/pybind11 dependencies, but intended for Numpy.
  - [x] `ListArray`: the new `JaggedArray`, based on `starts` and `stops` (i.e. fully general).
  - [x] `ListOffsetArray`: the `JaggedArray` case with no unreachable data between reachable data (gaps).
  - [ ] `RecordArray`: the new `Table` *without* lazy-slicing.

Did you get a StackOverflow account?