Future of User Analysis

Jim Pivarski

Princeton University – IRIS-HEP

October 1, 2020

~~Future of~~ Trends in User Analysis

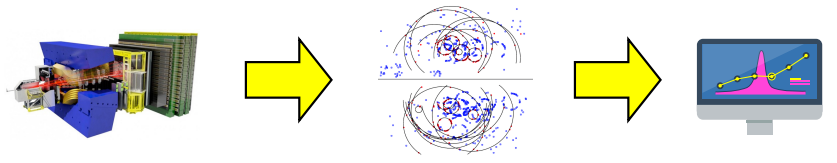
Jim Pivarski

Princeton University – IRIS-HEP

October 1, 2020

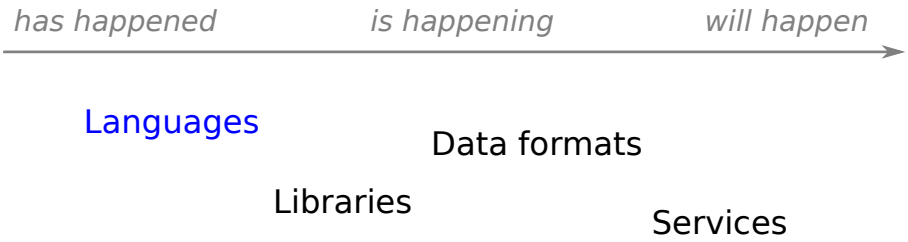
Define: "user analysis"

I'm aware that LHCb is moving important physics decisions earlier in the pipeline, but I think it's still meaningful to talk about "user analysis" as the last step.



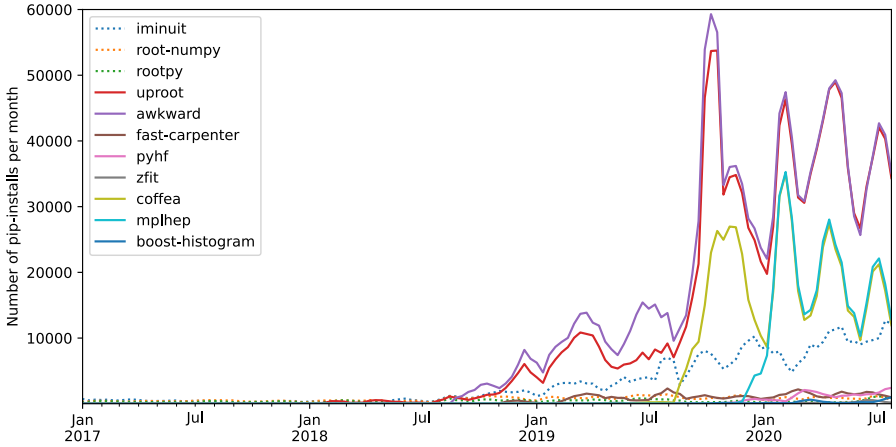
Reconstruction: what the collaboration does centrally

Analysis: small group producing a result

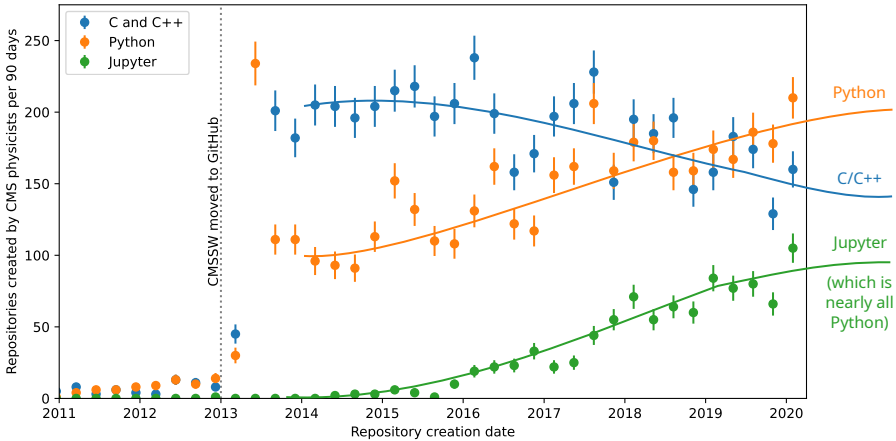

has happened
is happening
will happen
Languages
Data formats
Libraries
Services

Plot #1: pip-install statistics in the past three years

iminuit, root-numpy, and rootpy predate the rise, and therefore set the scale.

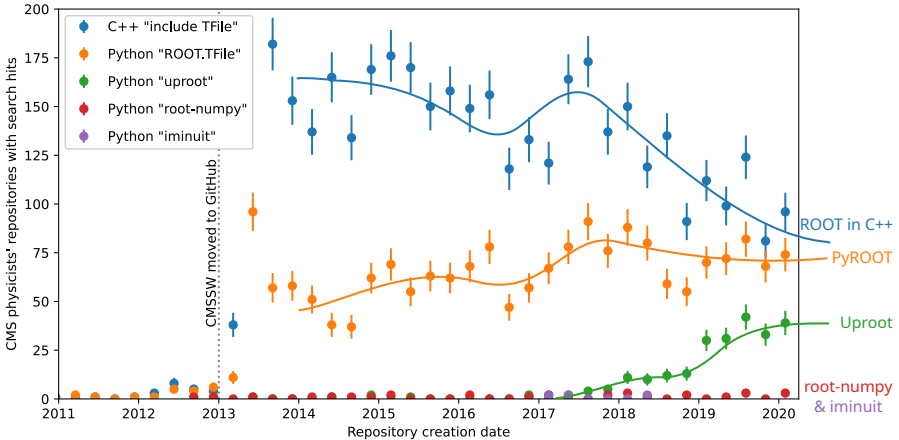


Plot #2: Language of non-fork GitHub repos by users who forked CMSSW



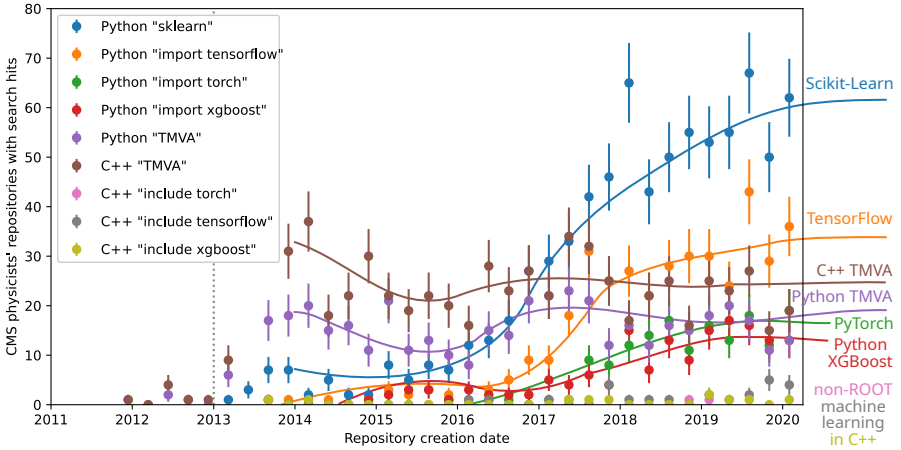
Plot #3: Search for substrings in those repos: what are those users typing?

We assume "tfile" correlates to ROOT; "root" by itself (case insensitive) is too generic.



Plot #3: What about machine learning?

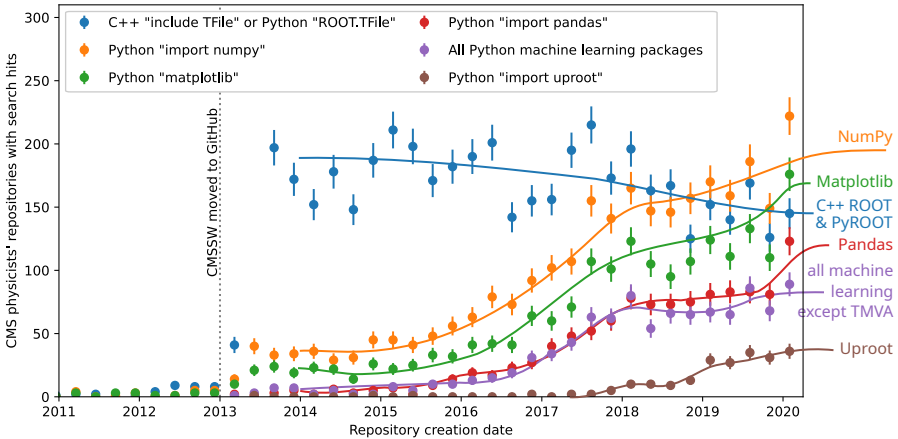
Surprise: Scikit-Learn! **Not a surprise:** TMVA is the only significant C++ library.



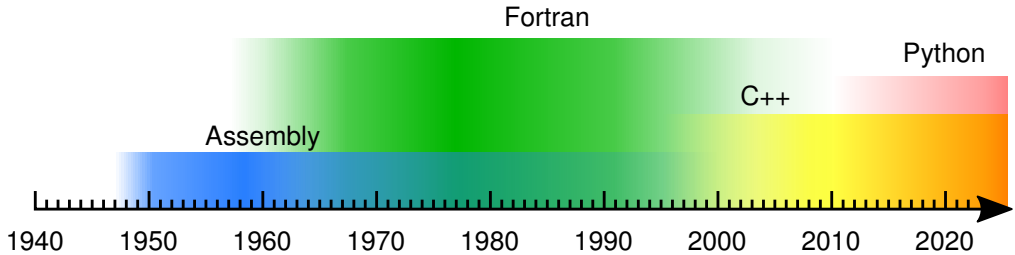
Python: the revolution has already happened

Plot #4: Was Python adoption driven by Uproot or machine learning (ML)?

Twice as much basic analysis (NumPy/Matplotlib) than ML; trend predates Uproot.



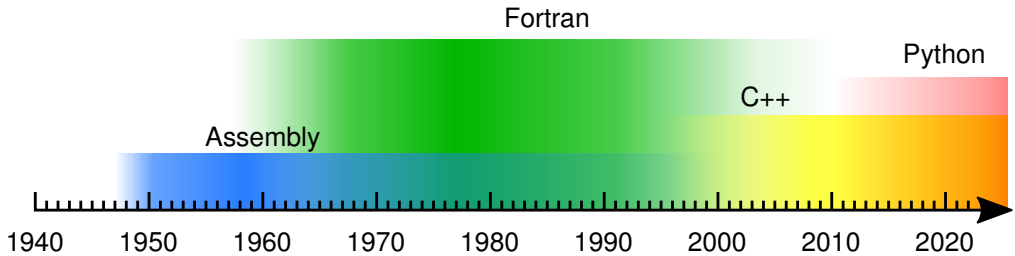
Our community doesn't change or add languages often



C++ first appeared in 1985: 10–15 years before we adopted it

Python first appeared in 1990: 20–25 years before we adopted it

Our community doesn't change or add languages often

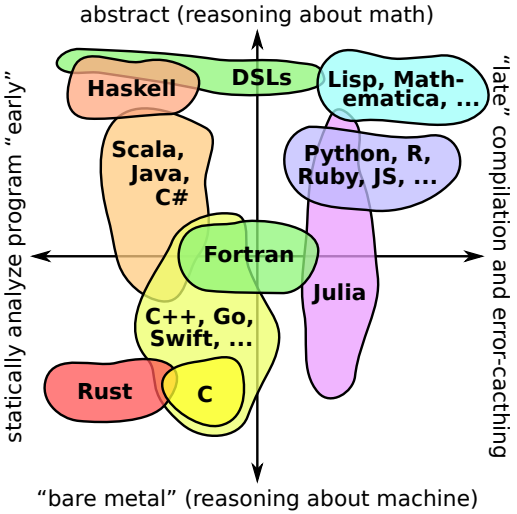


C++ first appeared in 1985: 10–15 years before we adopted it

Python first appeared in 1990: 20–25 years before we adopted it

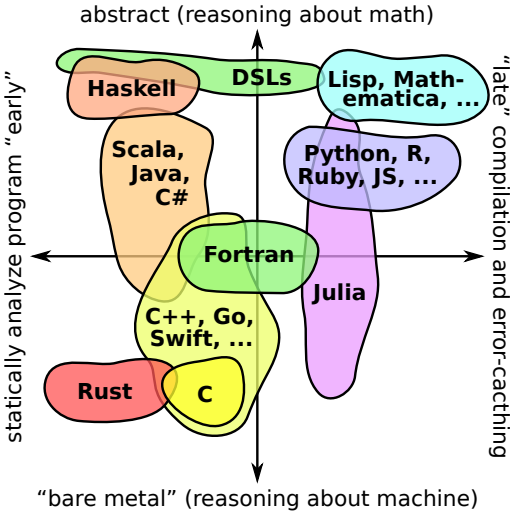
Julia first appeared in 2012: maybe we'll be using it by 2030

An aside on future languages



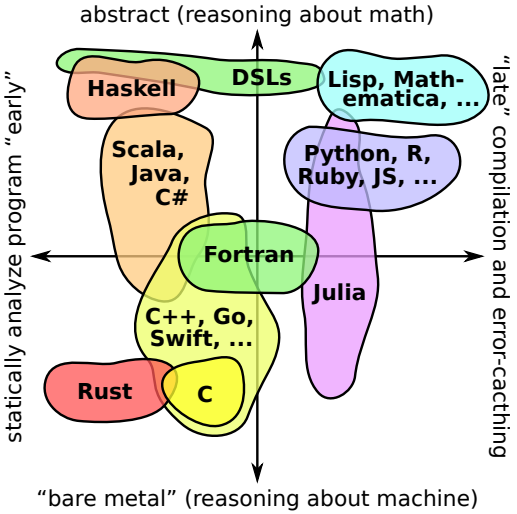
- Python, R, Ruby, Javascript, Lua, etc. are all relatively abstract, late error-catching glue languages.

An aside on future languages



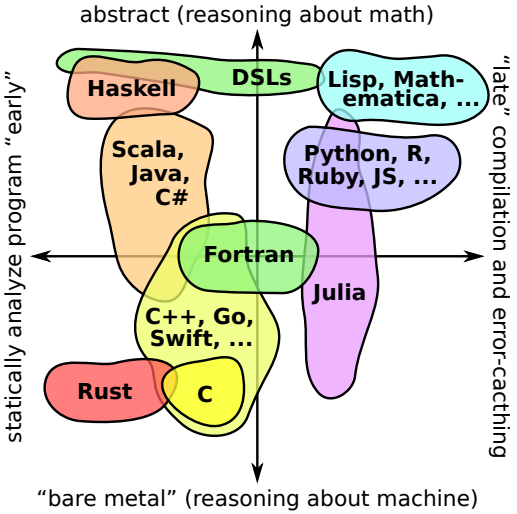
- Python, R, Ruby, Javascript, Lua, etc. are all relatively abstract, late error-catching glue languages. Python isn't special: it's just where the best libraries self-gravitated.

An aside on future languages



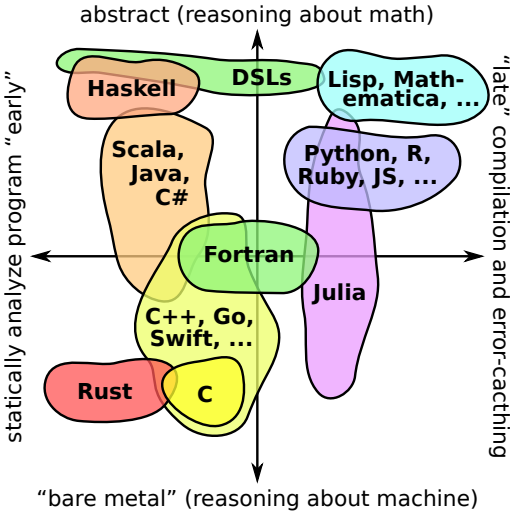
- Python, R, Ruby, Javascript, Lua, etc. are all relatively abstract, late error-catching glue languages. Python isn't special: it's just where the best libraries self-gravitated.
- Julia is different: everything is "just-in-time" (JIT) compiled, so it is flexible like Python, but fast, too.

An aside on future languages



- Python, R, Ruby, Javascript, Lua, etc. are all relatively abstract, late error-catching glue languages. Python isn't special: it's just where the best libraries self-gravitated.
- Julia is different: everything is "just-in-time" (JIT) compiled, so it is flexible like Python, but fast, too.
- Rust is a unique extreme. Triggers?

An aside on future languages



- Python, R, Ruby, Javascript, Lua, etc. are all relatively abstract, late error-catching glue languages. Python isn't special: it's just where the best libraries self-gravitated.
- Julia is different: everything is "just-in-time" (JIT) compiled, so it is flexible like Python, but fast, too.
- Rust is a unique extreme. Triggers?
- DSLs (domain specific languages) have physics concepts baked in with opportunities for strict error checking and flexible backends.

LHADA/CutLang —> ADL: syntax

Event selection regions

```
# preselection region
region preselection
  select size(AK4jets) >= 3
  select size(AK8jets) >= 1
  select MR > 800
  select Rsq > 0.08
```

```
# control region for tt+jets
region ttjetsCR
  select preselection
  select size(leptonsVeto) == 0
  select size(Wjets) >= 1
  select dphimegajets < 2.8
  select MT > 100
  select size(bjetsLoose) >= 1
  bin MR [] 800 1000 and Rsq [] 0.08 0.1
  bin MR [] 800 1000 and Rsq [] 0.1 0.2
  bin ...
```

F.A.S.T. F.A.S.T analysis tools based on YAML: Syntax

```yaml
BasicVars:
  variables:
      - Muon_Pt: "sqrt(Muon_Px ** 2 + Muon_Py ** 2)"
      - IsoMuon_Idx: (Muon_Iso / Muon_Pt) < 0.10
      # This next variable will create a single
      # number for each event, using a set of inputs
      # whose length varies for each event
      - NIsoMuon:
          formula: IsoMuon_Idx
          reduce: count_nonzero
      - HasTwoMuons: NIsoMuon >= 2
      # Capture first muon's Pt, padded
      # with NaNs if NMuon < 1
      - Muon_lead_Pt: {reduce: 0, formula: Muon_Pt}
      # Capture second muon's Pt, padded
      # with NaNs if NMuon < 2
      - Muon_sublead_Pt: {reduce: 1, formula: Muon_Pt}
```

```yaml
DiMu_controlRegion:
    weights: {nominal: weight}
    selection:
        All:
          - {reduce: 0, formula: Muon_pt > 30}
          - leadJet_pt > 100
          - All:
            - DiMuon_mass > 60
            - DiMuon_mass < 120
          - Any:
            - nCleanedJet == 1
            - DiJet_mass < 500
            - DiJet_deta < 2
```

```yaml
DiMuonMass:
  dataset_col: true
  binning:
      - in: DiMuon_Mass
        out: dimu_mass
        bins: {low: 60, high: 120, nbins: 60}
  weights: {weighted: EventWeight}
```

NAIL Natural Analysis Implementation Language: syntax

```
flow.Define("Muon_p4","@p4v(Muon)")
flow.Define("Electron_p4","@p4v(Electron)")
flow.Define("Electron_pid","Electron_pt*0+11")
flow.Define("Muon_pid","Muon_pt*0+13")
flow.MergeCollections("Lepton",["Muon","Electron"])
flow.Define("Lepton_index","Range(nLepton)")
flow.Distinct("LPair","Lepton")
flow.Selection("twoLeptons","nLepton>=2")
flow.Define("isOSSF","LPair0_charge != LPair1_charge && LPair0_pid == LPair1_pid",requires=["twoLeptons"])
flow.Selection("hasOSSF","Sum(isOSSF) > 0")
flow.TakePair("Z","Lepton","LPair","Argmax(-abs(MemberMap((LPair0_p4+LPair1_p4),M() )*isOSSF-91.2))",requires=["hasOSSF"])
flow.Selection("threeLeptons","nLepton>=3",requires=["twoLeptons","hasOSSF"])
flow.SubCollection("ResidualLeptons","Lepton",sel="Lepton_index != LPair0[Z_indices] && Lepton_index != LPair1[Z_indices]",
flow.ObjectAt("ResidualLepton","ResidualLeptons","Argmax(ResidualLeptons_pt)")

histosPerSelection={
"threeLeptons" : ["METplusLepton_Mt"]
}

flow.binningRules = [
(".*Mt.*","100,0,1000")
]
```

hep_tables and dataframe_expressions: syntax

Associating electrons with corresponding GEN particles:

```
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]
```

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...

```
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!!!

Some DSLs... (see Sezen Sekmen @ LHCP 2020)

PartiQL (demo) → AwkwardQL (real implementation)

```
# "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"
}
```

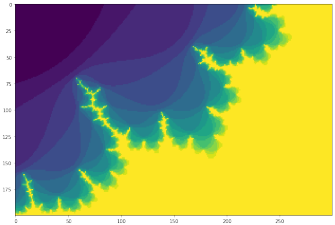
Languages

Data formats

Libraries

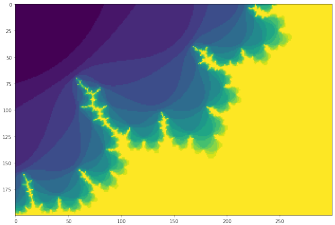
Services

Language choice does not determine performance!




```
import numpy

def run(height, width, maxiterations=20):
    y, x = numpy.ogrid[-1:0:height*1j, -1.5:0:width*1j]
    c = x + y*1j
    fractal = numpy.full(c.shape, maxiterations,
                                  dtype=numpy.int32)
    for h in range(height):
        for w in range(width):                  # for each pixel (h, w)...
            z = c[h, w]
            for i in range(maxiterations):      # iterate at most 20 times
                z = z**2 + c[h, w]              # applying z → z^2 + c
                if abs(z) > 2:                  # if it diverges (|z| > 2)
                    fractal[h, w] = i           # color the plane with the iteration number
                    break                       # we're done, no need to keep iterating

    return fractal
```

Language choice does not determine performance!



```
import numpy, numba
@numba.jit
def run(height, width, maxiterations=20):
    y, x = numpy.ogrid[-1:0:height*1j, -1.5:0:width*1j]
    c = x + y*1j
    fractal = numpy.full(c.shape, maxiterations,
                                  dtype=numpy.int32)
    for h in range(height):
        for w in range(width):                    # for each pixel (h, w)...
            z = c[h, w]
            for i in range(maxiterations):        # iterate at most 20 times
                z = z**2 + c[h, w]                # applying z → z^2 + c
                if abs(z) > 2:                    # if it diverges (|z| > 2)
                    fractal[h, w] = i             # color the plane with the iteration number
                    break                         # we're done, no need to keep iterating

    return fractal
```

Now **50× faster**, about as fast as C code (−O3).

Numba statically analyzes Python and compiles it with LLVM

Here's the catch:

GOOD

```
>>> @nb.njit
... def monte_carlo_pi(nsamples):
...     total = 0
...     for i in range(nsamples):
...         x = random.random()
...         y = random.random()
...         if x**2 + y**2 < 1:
...             total += 1
...     return 4.0 * total / nsamples
...
>>> monte_carlo_pi(int(1e9))
3.14156838
```

BAD

```
>>> @nb.njit
... def monte_carlo_pi(nsamples):
...     total = "hello"
...     for i in range(nsamples):
...         if total == "hello":
...             total = 0
...         x = random.random()
...         y = random.random()
...         if x**2 + y**2 < 1:
...             total += 1
...     return 4.0 * total / nsamples
...
>>> monte_carlo_pi(int(1e9))

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>

... (big traceback) ...

Cannot unify Literal[str](hello) and
    Literal[int](0) for 'total.5',
    defined at <stdin> (4)
```

Here's the catch: the right is legal Python, but dynamically typed

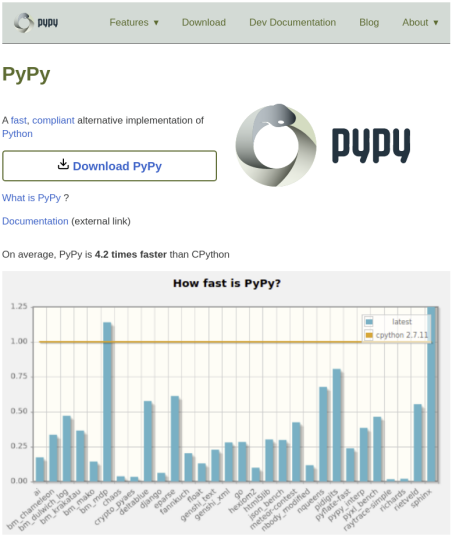
GOOD

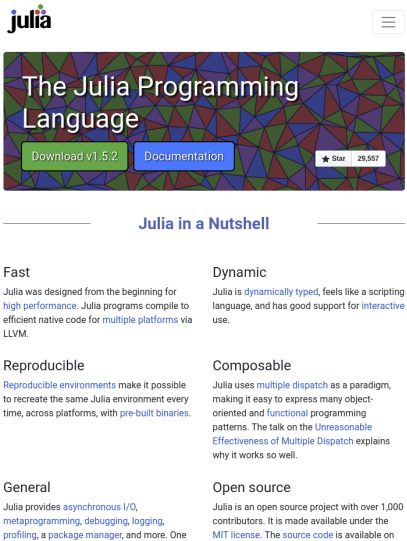
```
>>> @nb.njit
... def monte_carlo_pi(nsamples):
...     total = 0
...     for i in range(nsamples):
...         x = random.random()
...         y = random.random()
...         if x**2 + y**2 < 1:
...             total += 1
...     return 4.0 * total / nsamples
...
>>> monte_carlo_pi(int(1e9))
3.14156838
```

BAD

```
>>> @nb.njit
... def monte_carlo_pi(nsamples):
...     total = "hello"
...     for i in range(nsamples):
...         if total == "hello":
...             total = 0
...         x = random.random()
...         y = random.random()
...         if x**2 + y**2 < 1:
...             total += 1
...     return 4.0 * total / nsamples
...
>>> monte_carlo_pi(int(1e9))

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>

... (big traceback) ...

Cannot unify Literal[str](hello) and
    Literal[int](0) for 'total.5',
    defined at <stdin> (4)
```

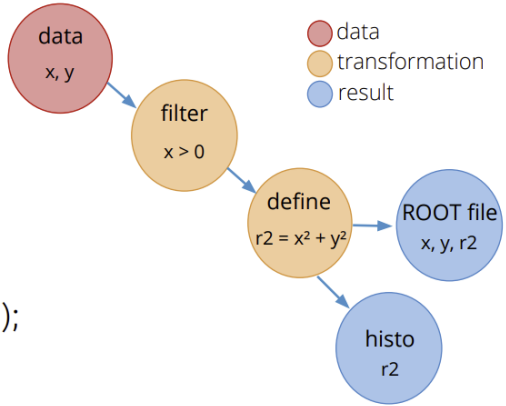
PyPy JIT-compiles *all* of Python; Julia defines less dynamism





Defining less is good: ROOT's RDataFrame takes away for loops

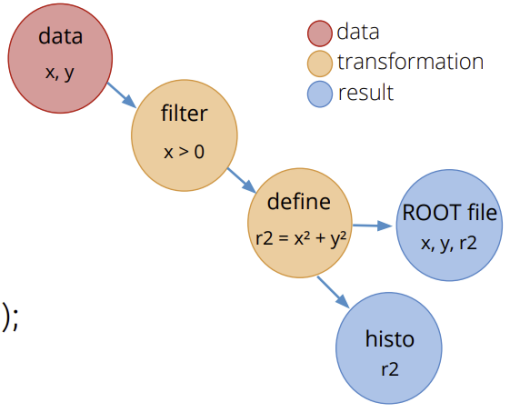
```
ROOT::RDataFrame df(dataset);
auto df2 = df.Filter("x > 0")
             .Define("r2", "x*x + y*y");
auto rHist = df2.Histo1D("r2");
df2.Snapshot("newtree", "newfile.root");
```



(see Enrico Guiraud @ CHEP 2018)

Defining less is good: ROOT's RDataFrame takes away for loops

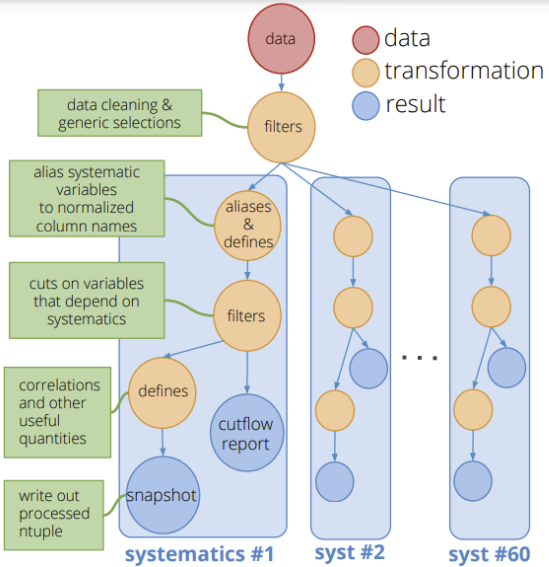
```
ROOT::RDataFrame df(dataset);
auto df2 = df.Filter("x > 0")
             .Define("r2", "x*x + y*y");
auto rHist = df2.Histo1D("r2");
df2.Snapshot("newtree", "newfile.root");
```



Without an explicit **for** loop, tasks without a dependency arrow can be parallelized or distributed across a cluster.

(see Enrico Guiraud @ CHEP 2018)

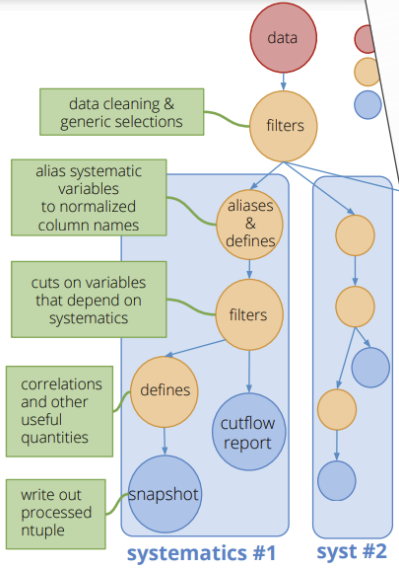
Case study: ATLAS SUSY ntuple → ntuple

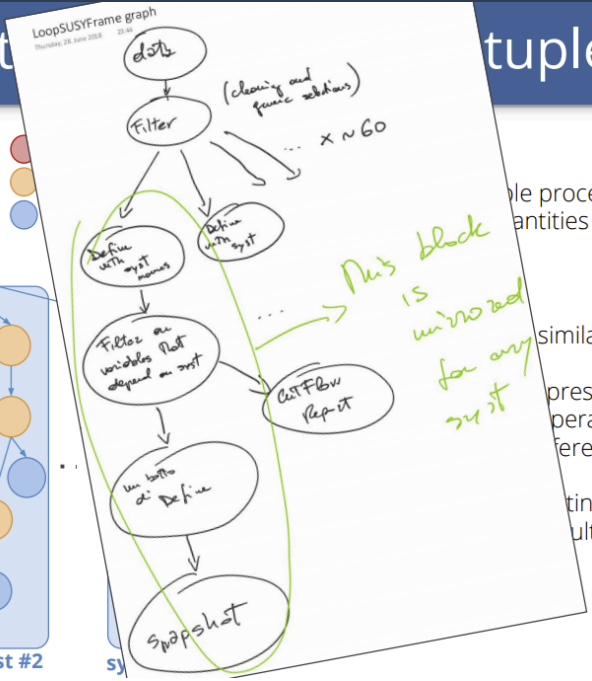


Local ntuple → ntuple processing, MC data is processed to add quantities relevant for publication

- ➔ program's **main** reads similarly to this graph
- ➔ the large blue boxes represent one single function that applies the same operations to an RDF variable and is re-used for all different systematics
- ➔ cuts, calculations and writing of the 60 output trees all happen in the same multi-thread event loop

Case st… tuple → ntuple





…le processing, MC data is
…antities relevant for publication

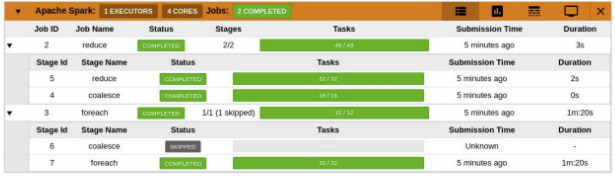
…similarly to this graph

…present one single function
…perations to an RDF variable
…ferent systematics

…ting of the 60 output trees
…ulti-thread event loop

Distributed Monitoring

- **Bridge the gap** between interactive computing and distributed data processing
- Automatically appears when a Spark job is submitted from a cell
- Progress bars, task timeline, resource utilisation



RDataFrame also JIT-compiles functions (with Cling)

C++

```
d.Filter([](double t) { return t > 0.; }, {"theta"})
 .Snapshot<vector<float>>("t","f.root",{"pt_x"});
```

C++ with cling's just-in-time compilation

```
d.Filter("theta > 0").Snapshot("t","f.root","pt_x");
```

PyROOT, automatically generated Python bindings

```
d.Filter("theta > 0").Snapshot("t","f.root","pt_x")
```

Exactly one level of abstraction: events

Events are distributed, but event code is C++ on `ROOT::VecOps::RVec<Particle>`.

```
getPt_code = """
using namespace ROOT::VecOps;
RVec<double> getPt(const RVec<FourVector> &tracks) {
   auto pt = [](const FourVector &v) { return v.pt(); };
   return Map(tracks, pt);
}
"""
ROOT.gInterpreter.Declare(getPt_code)

getPtWeights_code = """
using namespace ROOT::VecOps;
RVec<double> getPtWeights(const RVec<FourVector> &tracks) {
   auto ptWeight = [](const FourVector &v) { return 1. / v.Pt(); };
   return Map(tracks, ptWeight);
};
"""
ROOT.gInterpreter.Declare(getPtWeights_code)

augmented_d = (d.Define("tracks_n", "(int)tracks.size()")
                .Filter("tracks_n > 2")
                .Define("tracks_pts", "getPt(tracks)")
                .Define("tracks_pts_weights", "getPtWeights(tracks)"))
```

Implicit loops at all levels → analysis can be completely in Python

Awkward Array

```
>>> import awkward1 as ak
>>> import numpy as np
>>>
>>> @ak.mixin_class(ak.behavior)
... class Lorentz:
...     @property
...     def pt(self):
...         return np.sqrt(self.px**2 + self.py**2)
...
>>> array = ak.Array([{"px": 1, "py": 1, "pz": 1, "E": 1},
...                   {"px": 2, "py": 2, "pz": 2, "E": 2},
...                   {"px": 3, "py": 3, "pz": 3, "E": 3},
...                   {"px": 4, "py": 4, "pz": 4, "E": 4},
...                   {"px": 5, "py": 5, "pz": 5, "E": 5}],
...                  with_name="Lorentz")
...
```

Implicit loops at all levels → analysis can be completely in Python

Awkward Array

```
>>> import awkward1 as ak
>>> import numpy as np
>>>
>>> @ak.mixin_class(ak.behavior)
... class Lorentz:
...     @property
...     def pt(self):
...         return np.sqrt(self.px**2 + self.py**2)
...
>>> array = ak.Array([{"px": 1, "py": 1, "pz": 1, "E": 1},
...                   {"px": 2, "py": 2, "pz": 2, "E": 2},
...                   {"px": 3, "py": 3, "pz": 3, "E": 3},
...                   {"px": 4, "py": 4, "pz": 4, "E": 4},
...                   {"px": 5, "py": 5, "pz": 5, "E": 5}],
...                  with_name="Lorentz")
...
>>> array[-2]
<LorentzRecord {px: 4, py: 4, pz: 4, E: 4} type='Lorentz["px": int64, "py": int6...'>
```

Implicit loops at all levels → analysis can be completely in Python

Awkward Array

```
>>> import awkward1 as ak
>>> import numpy as np
>>>
>>> @ak.mixin_class(ak.behavior)
... class Lorentz:
...     @property
...     def pt(self):
...         return np.sqrt(self.px**2 + self.py**2)
...
>>> array = ak.Array([{"px": 1, "py": 1, "pz": 1, "E": 1},
...                   {"px": 2, "py": 2, "pz": 2, "E": 2},
...                   {"px": 3, "py": 3, "pz": 3, "E": 3},
...                   {"px": 4, "py": 4, "pz": 4, "E": 4},
...                   {"px": 5, "py": 5, "pz": 5, "E": 5}],
...                  with_name="Lorentz")
...
>>> array[-2]
<LorentzRecord {px: 4, py: 4, pz: 4, E: 4} type='Lorentz["px": int64, "py": int6...'>
>>> array * 10
<Array [{px: 10, py: 10, pz: 10, ... E: 50}] type='5 * {"px": int64, "py": int64...'>
```

Implicit loops at all levels → analysis can be completely in Python

Awkward Array

```
>>> import awkward1 as ak
>>> import numpy as np
>>>
>>> @ak.mixin_class(ak.behavior)
... class Lorentz:
...     @property
...     def pt(self):
...         return np.sqrt(self.px**2 + self.py**2)
...
>>> array = ak.Array([{"px": 1, "py": 1, "pz": 1, "E": 1},
...                   {"px": 2, "py": 2, "pz": 2, "E": 2},
...                   {"px": 3, "py": 3, "pz": 3, "E": 3},
...                   {"px": 4, "py": 4, "pz": 4, "E": 4},
...                   {"px": 5, "py": 5, "pz": 5, "E": 5}],
...                  with_name="Lorentz")
...
>>> array[-2]
<LorentzRecord {px: 4, py: 4, pz: 4, E: 4} type='Lorentz["px": int64, "py": int6...'>
>>> array * 10
<Array [{px: 10, py: 10, pz: 10, ... E: 50}] type='5 * {"px": int64, "py": int64...'>
>>> array.pt
<Array [1.41, 2.83, 4.24, 5.66, 7.07] type='5 * float64'>
```

Implicit loops at all levels → analysis can be completely in Python

Awkward Array

```
>>> import awkward1 as ak
>>> import numpy as np
>>>
>>> @ak.mixin_class(ak.behavior)
... class Lorentz:
...     @property
...     def pt(self):
...         return np.sqrt(self.px**2 + self.py**2)
...
>>> array = ak.Array([[{"px": 1, "py": 1, "pz": 1, "E": 1},
...                    {"px": 2, "py": 2, "pz": 2, "E": 2},
...                    {"px": 3, "py": 3, "pz": 3, "E": 3}],
...                   [],
...                   [{"px": 4, "py": 4, "pz": 4, "E": 4},
...                    {"px": 5, "py": 5, "pz": 5, "E": 5}]],
...                  with_name="Lorentz")
...
>>> array[0, -1]
<LorentzRecord {px: 3, py: 3, pz: 3, E: 3} type='Lorentz["px": int64, "py": int6...'>
>>> array * 10
<Array [[{px: 10, py: 10, ... E: 50}]] type='3 * var * {"px": int64, "py": int64...'>
>>> array.pt
<Array [[1.41, 2.83, 4.24], ... [5.66, 7.07]] type='3 * var * float64'>
```

Implicit loops at all levels → analysis can be completely in Python

Awkward Array

```
>>> import awkward1 as ak
>>> import numpy as np
>>>
>>> @ak.mixin_class(ak.behavior)
... class Lorentz:
...     @property
...     def pt(self):
...         return np.sqrt(self.px**2 + self.py**2)
...
>>> array = ak.Array([[[{"px": 1, "py": 1, "pz": 1, "E": 1},
...                     {"px": 2, "py": 2, "pz": 2, "E": 2}],
...                    [{"px": 3, "py": 3, "pz": 3, "E": 3}]],
...                   [[]],
...                   [[{"px": 4, "py": 4, "pz": 4, "E": 4}],
...                    [],
...                    [{"px": 5, "py": 5, "pz": 5, "E": 5}]]],
...                  with_name="Lorentz")
>>> array[-1, 2, 0]
<LorentzRecord {px: 5, py: 5, pz: 5, E: 5} type='Lorentz["px": int64, "py": int6...'>
>>> array * 10
<Array [[[{px: 10, py: 10, ... E: 50}]]] type='3 * var * var * {"px": int64, "py...'>
>>> array.pt
<Array [[[1.41, 2.83], [4.24, ... [], [7.07]]] type='3 * var * var * float64'>
```

Awkward Array is like NumPy, but for data structures

```
array = ak.Array([
    [{"x": 1.1, "y": [1]}, {"x": 2.2, "y": [1, 2]}, {"x": 3.3, "y": [1, 2, 3]}],
    [],
    [{"x": 4.4, "y": [1, 2, 3, 4]}, {"x": 5.5, "y": [1, 2, 3, 4, 5]}]
])
```

NumPy-like expression

```
output = np.square(array["y", ..., 1:])
```

```
[
    [[], [4], [4, 9]],
    [],
    [[4, 9, 16], [4, 9, 16, 25]]
]
```

4.6 seconds to run (2 GB footprint)

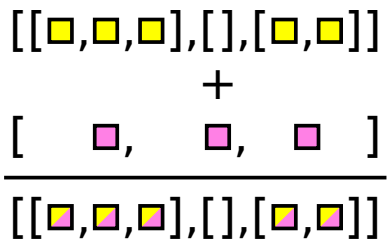
equivalent Python

```
output = []
for sublist in python_objects:
    tmp1 = []
    for record in sublist:
        tmp2 = []
        for number in record["y"][1:]:
            tmp2.append(np.square(number))
        tmp1.append(tmp2)
    output.append(tmp1)
```

138 seconds to run (22 GB footprint)

(single-threaded on a 2.2 GHz processor with a dataset 10 million times larger than the one shown)

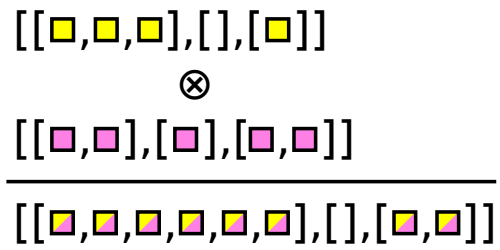
Like NumPy, but structured: broadcasting



```
>>> jagged = ak.Array([[1, 2, 3],  [], [4, 5]])
>>> flat   = ak.Array([      100, 200,    300])
>>> jagged + flat
<Array [[101, 102, 103], [], [304, 305]] type='3 * var * int64'>
```

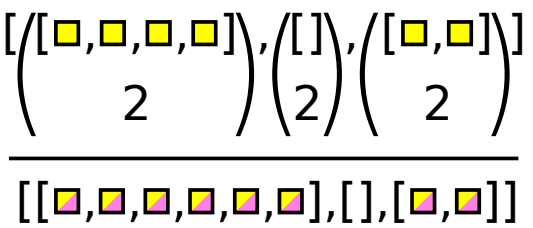
Like NumPy, but structured: combinatorics

Cartesian product



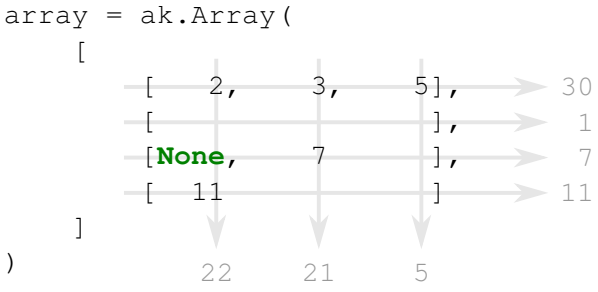
```
>>> x = [[1, 2, 3], [], [4]]
>>> y = [["a", "b"], ["c"], ["d", "e"]]
>>> ak.cartesian([x, y], axis=1)
[[(1, "a"), (1, "b"),
  (2, "a"), (2, "b"),
  (3, "a"), (3, "b")],
 [],
 [(4, "d"), (4, "e")]]
```

Combinations without replacement



```
>>> x = [[1, 2, 3, 4], [], [5, 6]]
>>> ak.combinations(x, 2, axis=1)
[[(1, 2), (1, 3), (1, 4),
  (2, 3), (2, 4), (3, 4)],
 [],
 [(5, 6)]]
```

Like NumPy, but structured: reduction



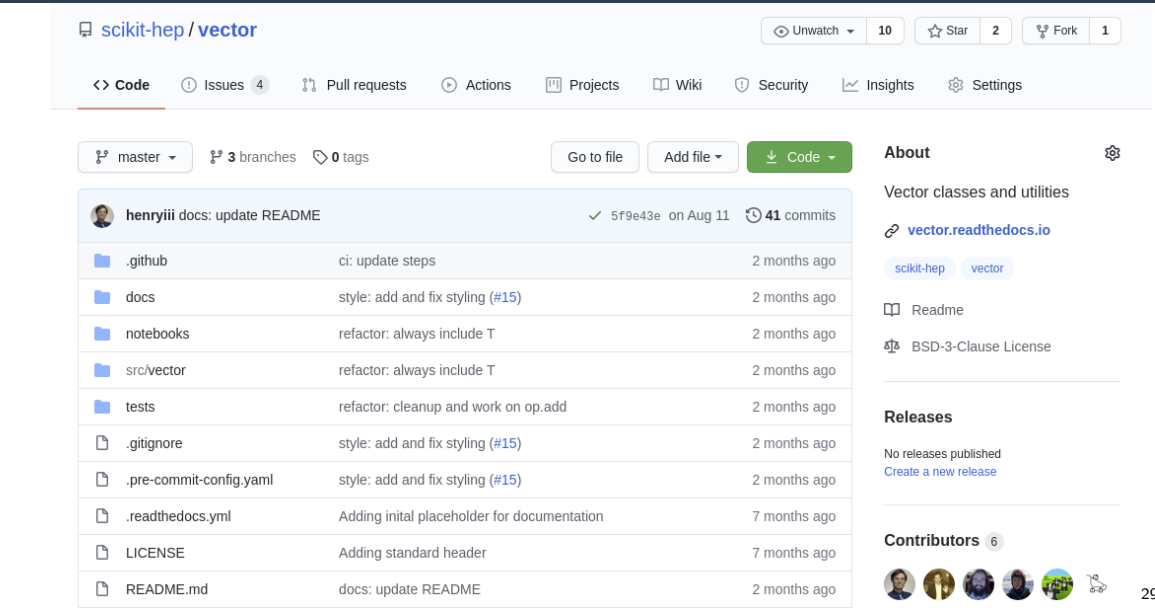

```
>>> ak.prod(array, axis=0)
<Array [22, 21, 5] type='3 * int64'>
>>> ak.prod(array, axis=1)
<Array [30, 1, 7, 11] type='4 * int64'>
```

Awkward Arrays can be used with Numba

You can mix fast, explicit **for** loops with NumPy-like slicing, all in Python.

```
>>> @nb.jit
... def make_cut(electrons, jets, builder):
...     for event_electrons, event_jets in zip(electrons, jets):
...         builder.begin_list()
...         for electron in event_electrons:
...             keep = False
...             for jet in event_jets:
...                 if abs(electron.x - jet.x) < 0.45:    # calculate delta-R here
...                     keep = True
...                     break
...             builder.append(keep)
...         builder.end_list()
...     return builder
...
>>> cut = make_cut(electrons, jets, ak.ArrayBuilder()).snapshot()
>>> cut
<Array [[True, False, True], ... [True, False]] type='3 * var * bool'>
>>> electrons[cut].tolist()
[[{'x': 1, 'y': 1, 'z': 1, 't': 1}, {'x': 3, 'y': 3, 'z': 3, 't': 3}],
 [],
 [{'x': 4, 'y': 4, 'z': 4, 't': 4}]]
```

vector: an Awkward 2D, 3D, Lorentz vector library in development



scikit-hep / vector

Unwatch 10 | Star 2 | Fork 1

Code | Issues 4 | Pull requests | Actions | Projects | Wiki | Security | Insights | Settings

master | 3 branches | 0 tags | Go to file | Add file | Code

henryiii docs: update README — 5f9e43e on Aug 11 — 41 commits

.github	ci: update steps	2 months ago
docs	style: add and fix styling (#15)	2 months ago
notebooks	refactor: always include T	2 months ago
src/vector	refactor: always include T	2 months ago
tests	refactor: cleanup and work on op.add	2 months ago
.gitignore	style: add and fix styling (#15)	2 months ago
.pre-commit-config.yaml	style: add and fix styling (#15)	2 months ago
.readthedocs.yml	Adding inital placeholder for documentation	7 months ago
LICENSE	Adding standard header	7 months ago
README.md	docs: update README	2 months ago

About

Vector classes and utilities

vector.readthedocs.io

scikit-hep | vector

Readme

BSD-3-Clause License

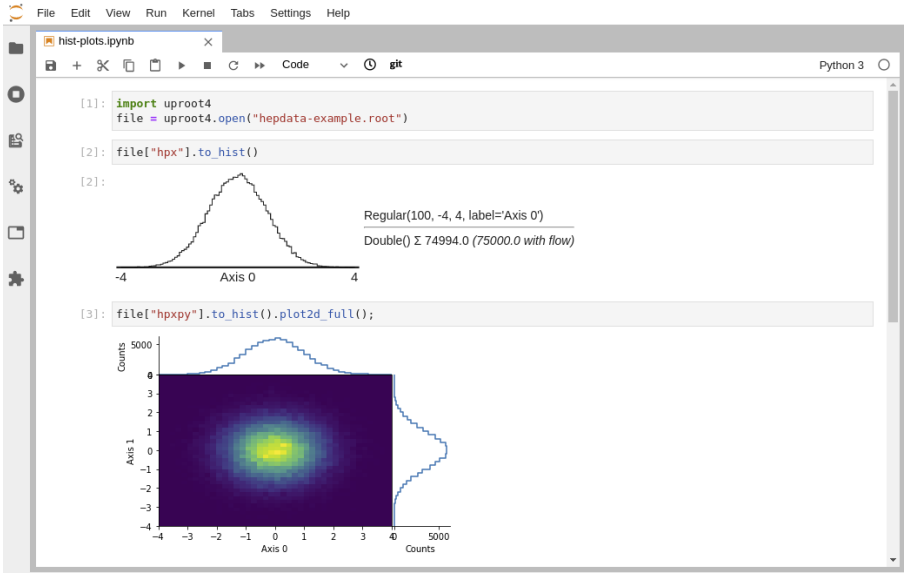
Releases

No releases published

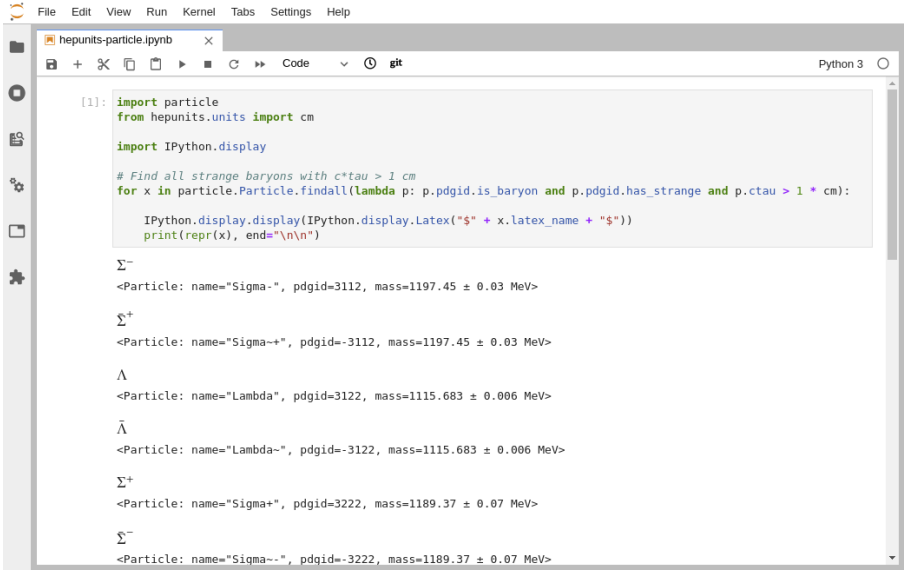
Create a new release

Contributors 6

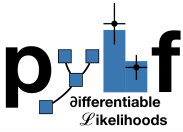
First release of hist, a Pythonic histogram library, on Tuesday

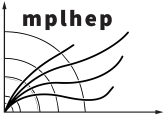


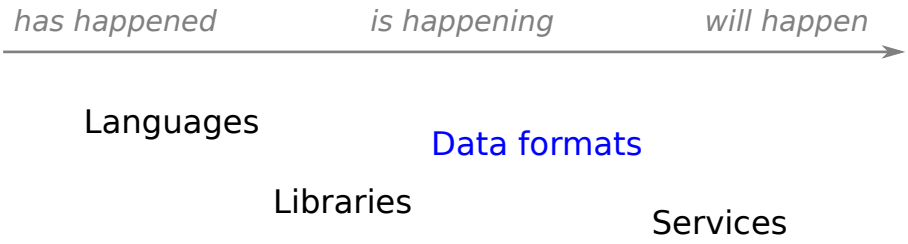
hepunits: common units in HEP and particle: PDG data



Scikit-HEP: a growing ecosystem of interoperating tools



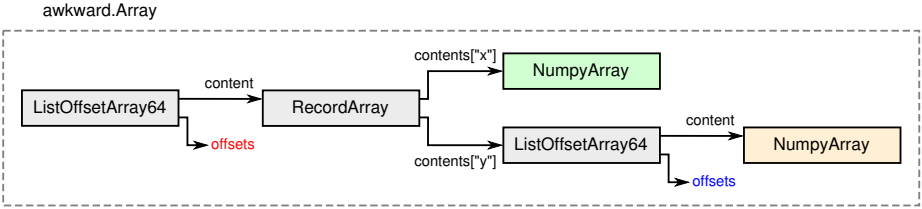



has happened
is happening
will happen
Languages
Data formats
Libraries
Services

A columnar form can be iterated or vectorized

```
array = ak.Array([
    [{"x": 1,  "y": [11]},
     {"x": 4,  "y": [12, 22]},
     {"x": 9,  "y": [13, 23, 33]}],
    [],
    [{"x": 16, "y": [14, 24, 34, 44]}]
])
```

```
outer offsets:  0,                      3,  3, 5
content for x:  1,  4,      9,             16
offsets for y:  0,  1,      3,              6,            10
content for y: 11, 12, 22, 13, 23, 33,     14, 24, 34, 44
```



ROOT's TTree is columnar to one level (only intended for iteration)

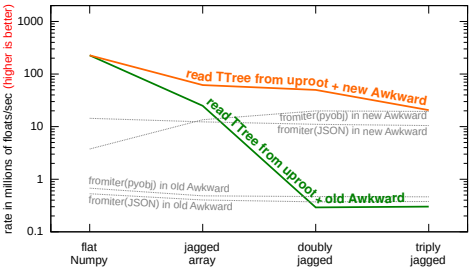
```
tree = TTree([
    [{"x": 1,  "y": [11]},
     {"x": 4,  "y": [12, 22]},
     {"x": 9,  "y": [13, 23, 33]}],
    [],
    [{"x": 16, "y": [14, 24, 34, 44]}]
])
```

```
0                                                    15 15
x 1 y(1) 11 x 4 y(2) 12 22 x 9 y(3) 13 23 33      x 16 y(4) 14 24 34 44
```

ROOT's TTree is columnar to one level (only intended for iteration)

```
tree = TTree([
    [{"x": 1,  "y": [11]},
     {"x": 4,  "y": [12, 22]},
     {"x": 9,  "y": [13, 23, 33]}],
    [],
    [{"x": 16, "y": [14, 24, 34, 44]}]
])
```

```
0                                                    15 15
x 1 y(1) 11 x 4 y(2) 12 22 x 9 y(3) 13 23 33      x 16 y(4) 14 24 34 44
```

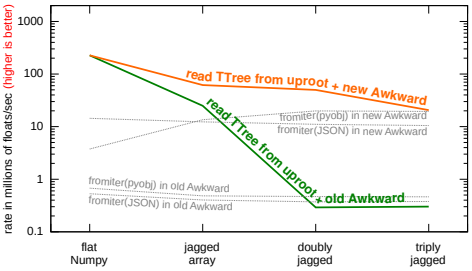


This is why Uproot reads **`float`** and `vector<`**`float`**`>` data much faster than `vector<vector<`**`float`**`>>` or higher.

ROOT's TTree is columnar to one level (only intended for iteration)

```
tree = TTree([
    [{"x": 1,  "y": [11]},
     {"x": 4,  "y": [12, 22]},
     {"x": 9,  "y": [13, 23, 33]}],
    [],
    [{"x": 16, "y": [14, 24, 34, 44]}]
])
```

```
0                                                    15 15
x 1 y(1) 11 x 4 y(2) 12 22 x 9 y(3) 13 23 33      x 16 y(4) 14 24 34 44
```



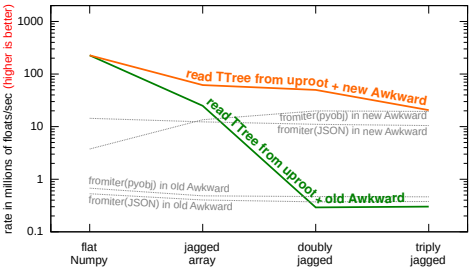
This is why Uproot reads **`float`** and `vector<`**`float`**`>` data much faster than `vector<vector<`**`float`**`>>` or higher.

TTree interleaves data of more than one level of nesting, which can't be unraveled by NumPy.

ROOT's TTree is columnar to one level (only intended for iteration)

```
tree = TTree([
    [{"x": 1,  "y": [11]},
     {"x": 4,  "y": [12, 22]},
     {"x": 9,  "y": [13, 23, 33]}],
    [],
    [{"x": 16, "y": [14, 24, 34, 44]}]
])
```

```
0                                                    15 15
x 1 y(1) 11 x 4 y(2) 12 22 x 9 y(3) 13 23 33      x 16 y(4) 14 24 34 44
```

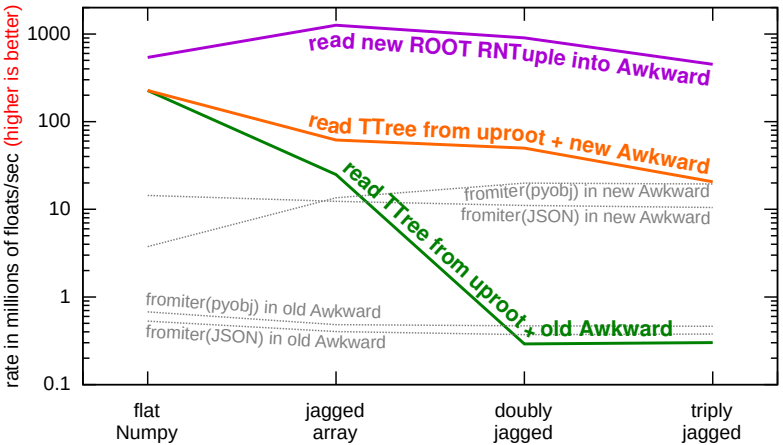


This is why Uproot reads **`float`** and `vector<`**`float`**`>` data much faster than `vector<vector<`**`float`**`>>` or higher.

TTree interleaves data of more than one level of nesting, which can't be unraveled by NumPy.

(But some performance can be recovered by moving the **`for`** loops from Python into C++.)

RNTuple: the ROOT 7 *replacement* for TTree!



RNTuple is a redesign of TTree from scratch, using columnar data structures at all levels of depth.

Not having to deal with interleaved data at all is better than moving loops from Python to C++.

RNTuple will also have a separable C++ interpretation

The first two layers are compatible/convertible to Awkward Array.

Event iteration
Reading and writing in event loops and through `RDataFrame`
`RNTupleDataSource`, `RNTupleView`, `RNTupleReader/Writer`

Logical layer / C++ objects
Mapping of C++ types onto columns
e.g. `std::vector<float>` ↦ index column and a value column
`RField`, `RNTupleModel`, `REntry`

Primitives layer / simple types
"Columns" containing elements of fundamental types (`float`, `int`, ...)
grouped into (compressed) pages and clusters
`RColumn`, `RColumnElement`, `RPage`

Storage layer / byte ranges
`RPageStorage`, `RCluster`, `RNTupleDescriptor`

Approximate translation between TTree and RNTuple classes:

TTree	≈	RNTupleReader
		RNTupleWriter
TTreeReader	≈	RNTupleView
TBranch	≈	RField
TBasket	≈	RPage
TTreeCache	≈	RClusterPool

- Jakob Blomer and his students are developing RNtuple in ROOT.
- Oksana Shadura is developing the Python interface.

Columnar data formats in industry: on disk, in memory, on GPUs

Parquet

Apache Software Foundation / Apache Parquet

Apache Parquet is a columnar storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language.

Parquet Videos (more presentations)

Tweets by @ApacheParquet

Apache Parquet Retweeted

Julien Le Dem @J...

PSA: If you use the page-level statistics in @ApacheParquet please chime in on JIRA: issues.apache.org/jira/browse/PA…

Nov 6, 2018

Apache Parquet Retweeted

Overview FAQ Blog Get Arrow Documentation Community ASF Links

APACHE ARROW

A cross-language development platform for in-memory analytics

Star 6,241 Follow @ApacheArrow 7,504 followers

What is Arrow?

Format

Apache Arrow defines a language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware like CPUs and GPUs. The Arrow memory format also supports zero-copy reads for lightning-fast data access without serialization overhead.

Learn more about the design or read the specification.

Libraries

Arrow's libraries implement the format and provide building blocks for a range of use cases, including high performance analytics. Many popular projects use Arrow to ship columnar data efficiently or as the basis for analytic engines.

Libraries are available for C, C++, C#, Go, Java, JavaScript, MATLAB, Python, R, Ruby, and

RAPIDS

RAPIDS

Open GPU Data Science

GET STARTED

GPU DATA SCIENCE

ⓘ ACCELERATED DATA SCIENCE

The RAPIDS suite of open source software libraries gives you the freedom to execute end-to-end data science and analytics pipelines entirely on GPUs.

Learn about RAPIDS »

SCALE OUT ON GPUS

Seamlessly scale from GPU workstations to multi-GPU servers and multi-node clusters with Dask.

Learn about Dask »

PYTHON INTEGRATION

Accelerate your Python data science toolchain with minimal code changes and no new tools to learn.

Learn about our libraries »

TOP MODEL ACCURACY

Increase machine learning model accuracy by iterating on models faster and deploying them more frequently.

Learn about RAPIDS for model optimization »

Columnar data formats in industry: on disk, in memory, on GPUs

Parquet

Apache Software Foundation / Apache Parquet

Apache Parquet is a columnar storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language.

Parquet Videos (more presentations)

Tweets by @ApacheParquet

Apache Parquet Retweeted

Julien Le Dem
@J...
PSA: If you use the page-
@ApacheParquet please chime in on JIRA:
issues.apache.org/jira/browse/PA…

Nov 6, 2018

Apache Parquet Retweeted

Overview FAQ Blog Get Arrow Documentation Community ASF Links

APACHE ARROW

A cross-language development platform for in-memory analytics

Star 6,241 Follow @ApacheArrow 7,504 followers

What is Arrow?

Format

Apache Arrow defines a language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware like CPUs and GPUs. The Arrow memory format also supports zero-copy reads for lightning-fast data access without serialization overhead.

Learn more about the design or read the specification.

Libraries

Arrow's libraries implement the format and provide building blocks for a range of use cases, including high performance analytics. Many popular projects use Arrow to ship columnar data efficiently or as the basis for analytic engines.

Libraries are available for C, C++, C#, Go, Java, JavaScript, MATLAB, Python, R, Ruby, and

RAPIDS

RAPIDS

Open GPU Data Science

GET STARTED

GPU DATA SCIENCE

ⓘ ACCELERATED DATA SCIENCE

The RAPIDS suite of open source software libraries gives you the freedom to execute end-to-end data science and analytics pipelines entirely on GPUs.

Learn about RAPIDS »

SCALE OUT ON GPUS

Seamlessly scale from GPU workstations to multi-GPU servers and multi-node clusters with Dask.

Learn about Dask »

PYTHON INTEGRATION

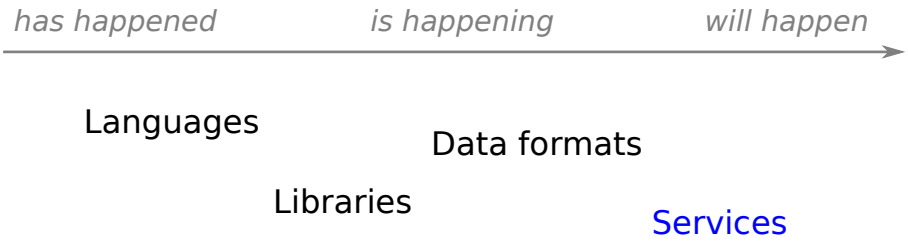
Accelerate your Python data science toolchain with minimal code changes and no new tools to learn.

TOP MODEL ACCURACY

Increase machine learning model accuracy by iterating on models faster and deploying them more frequently.

Learn about RAPIDS for model optimization »

Since RNTuple is fully columnar, it will be easier to convert.


has happened
is happening
will happen
Languages
Data formats
Libraries
Services

Query services (mostly future)

As datasets get too big to ntuplize, we'll have to move the pre-aggregation steps of analysis to a centralized service.

Query services (mostly future)

As datasets get too big to ntuplize, we'll have to move the pre-aggregation steps of analysis to a centralized service.

Need to reproduce the low-latency experience of a local ntuple on a shared system.

- It will be harder because it will be more in demand at peak times.
- It will be easier because popular data can be in shared caches.

Query services (mostly future)

As datasets get too big to ntuplize, we'll have to move the pre-aggregation steps of analysis to a centralized service.

Need to reproduce the low-latency experience of a local ntuple on a shared system.

- It will be harder because it will be more in demand at peak times.
- It will be easier because popular data can be in shared caches.

Strong bias toward reusing existing technologies: Kubernetes, Docker, Kafka, Spark, Dask, and perhaps Ray.

Query services (mostly future)

As datasets get too big to ntuplize, we'll have to move the pre-aggregation steps of analysis to a centralized service.

Need to reproduce the low-latency experience of a local ntuple on a shared system.

- It will be harder because it will be more in demand at peak times.
- It will be easier because popular data can be in shared caches.

Strong bias toward reusing existing technologies: Kubernetes, Docker, Kafka, Spark, Dask, and perhaps Ray.

Query services are now at the demo stage!

- ServiceX: https://youtu.be/NEgqGRdt360
- CoffeaCasa: https://youtu.be/CDIFd1gDbSc

Conclusions

- The transition to more-Python-than-C++ has already happened.
- DSLs are also being explored, but they're at a much earlier stage.
- Just-in-time compilation is important: Numba, PyPy, Julia, and RDataFrame.
- So is "removing the `for` loop."
- Awkward Array has implicit loops at all levels for NumPy-like manipulation, and also provides an onramp into Numba for conventional code.
- The Scikit-HEP ecosystem is booming!
- Columnar internal structures are essential: ROOT is developing RNTuple.
- Columnar data formats are popular in industry, too. We must interoperate!
- Query services are at the demo stage.