Discussion of Python/ROOT "interoperability"

David Lange

June 13, 2016

Thanks to

Noel Dawe, Wim Lavrijsen, Pere Mato, Danilo Piparo, Daniela Remenska, Tai Sakuma, Jeff Templon, Chris Tunnell

All mistakes, omissions and misrepresentations are mine… (including any and all simplistic python constructs)

Interoperability

- Wikipedia: “With respect to software, the term **interoperability** is used to describe the capability of different programs to exchange data via a common set of exchange formats, to read and write the same file formats and to use the same protocols”

- In the context of Python and ROOT, HEP analysts primarily think about this in one direction:
  - How can we use python to efficiently access root data, root graphics (etc).
  - But we should think in both directions

Goals

- Understand capabilities/limitations of existing Python←→ROOT interfaces
- Think about if/how the DIANA project should contribute
  - Starting assumption: Developing yet another independent interface tool is not the right way for us to go
- Today is largely a non-technical discussion of existing tools

What features could we wish to have?

- The ability to interact with ROOT data including complex types from python scripts/command line
  - To do so in a “python-ic” way
  - To be able to interface with other analysis tools and mathematical/statistical tools developed in the broader scientific community

- The ability to use ROOT functionality such as plotting and mathematical functions with python data (from ROOT or other data structures)

- The ability to have these functionality running at the same access speed as at the ROOT prompt

All of this at the same time is of course quite non-trivial…

Developments I looked at

1. PYROOT

2. ROOTPY, ROOT_NUMPY

3. Pandas based toolkit from Xenon experiment

PyROOT

- Project started in 2002
  - Part of ROOT since ROOT 4.00/04.
  - Python C-API based
    - Not based on numpy given missing features of numpy at that time (e.g., a proper buffer interface)
  - Dynamic interface: User defined complex types naturally via Cling and ROOT dictionaries

- Python3
  - Not yet supported by PyROOT
  - Python3.3, all tests pass thanks to recent work. Python3.4 garbage collection improvements need additional changes (PythonReleaseNotesPEP442)
    - (I briefly tried to verify this but haven't yet managed to build root with python3(.5) – so a work in progress)

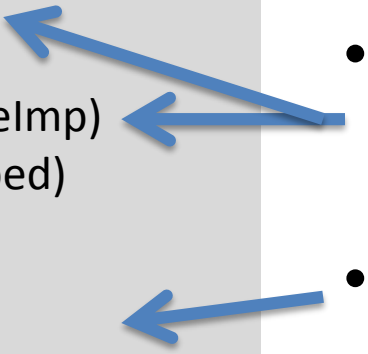
CMS example from long ago CHEP

FWLite Example

```
from PhysicsTools.PythonAnalysis import *
from ROOT import *

# prepare the FWLite autoloading mechanism
gSystem.Load("libFWCoreFWLite.so")
AutoLibraryLoader.enable()
events = EventTree("reco.root")
# book a histogram
histo = TH1F("photon_pt", "Pt of photons", 100, 0, 300)

# event loop
for event in events:
    photons = event.photons  # uses aliases
    print "# of photons in event %i: %i" % (event, len(photons))
    for photon in photons:
        if photon.eta() < 2:
            histo.Fill(photon.pt())
```

Toy example: Creating an ntuple (from random data)

```
import numpy, numpy.random
import ROOT

fields=['vx','vy','vz','pt','phi','eta','q']
types=[numpy.float32,numpy.float32,numpy.float32,
       numpy.float32,numpy.float32,numpy.float32,
       numpy.int32]
typesRoot=['F','F','F','F','F','F','I']

dtypeImp=[]
for i in range(len(fields)):
   dtypeImp.append( (fields[i],types[i]) )

mydata=numpy.recarray(100,dtype=dtypeImp)
# fill mydata with random numbers (skipped)

mydataT=[]
for i in range(len(fields)):
   mydataT.append(numpy.zeros(1,dtype=types[i]))
```



- Start with numpy array and fill ntuple
- Buffers for each branch

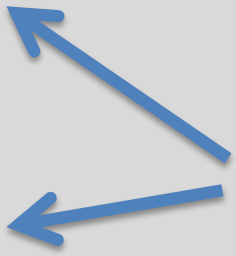
Toy example: Creating an ntuple (from random data)

```
f=ROOT.TFile("tree2.root","recreate")
t=ROOT.TTree("tree","tree")

for i in range(len(fields)):
   t.Branch(fields[i],mydataT[i],
            fields[i]+'/'+ typesRoot[i])

for i in range(mydata.shape[0]):
   for j in range(len(fields)):
      mydataT[j][0]=mydata[i][j]
   t.Fill()

f.Write()
f.Close()
```

- Initialize
- Declare each branch
- Double loop to fill the TTree

Toy example: Reading this ntuple into numpy array

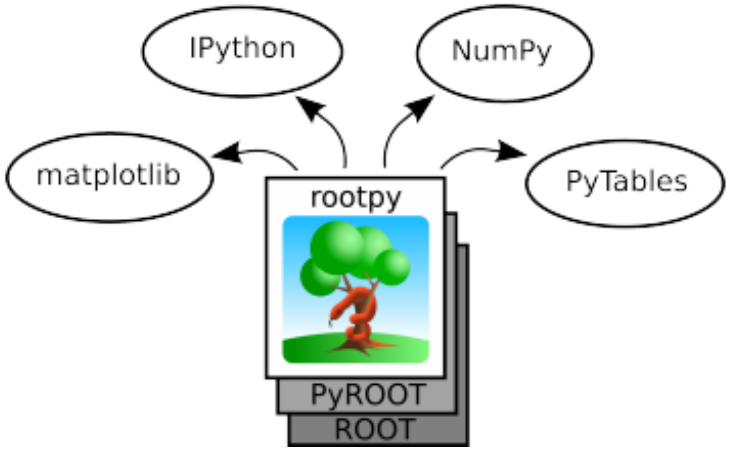
```
import numpy
import ROOT

f=ROOT.TFile("tree2.root")
t=f.Get("tree")

fields=['vx','vy','vz','pt','phi','eta','q']
types=[numpy.float32,numpy.float32,numpy.float32,numpy.float32,
numpy.float32,numpy.float32,numpy.int32]
typesRoot=['F','F','F','F','F','F','I']

dtypeImp=[]
for i in range(len(fields)):
   dtypeImp.append( (fields[i],types[i]) )

nEvts=t.GetEntries()
mydata=numpy.recarray(nEvts,dtype=dtypeImp)
for i,ev in enumerate(t):
   for j in range(len(fields)):
      mydata[i][j]=getattr(ev,fields[j])
print mydata[0:10]
```



Better to learn this by inspecting the branches of the saved tree. This is supported (I was lazy)

rootpy

- Developed as a layer on top of PyROOT (http://www.rootpy.org/) aiming to add "pythonic" bindings
- Not part of the ROOT distribution, installs as a python module
- Focus: Tools for easing the creation and manipulation of ntuples, histograms and cuts
- Expect V1.0 to be complete in ~1 year



Toy example via rootpy

```
import numpy, numpy.random
from rootpy.tree import Tree
from rootpy.io import root_open

fields=['vx','vy','vz','pt','phi','eta','q']
types=[numpy.float32,numpy.float32,numpy.float32,
numpy.float32,numpy.float32,numpy.float32, numpy.int32]

dtypeImp=[]
for i in range(len(fields)):
    dtypeImp.append( (fields[i],types[i]) )

froot=root_open("tree3.root","recreate")
tree=Tree("tree")
tree.create_branches(typesDict)

for i in range(mydata.shape[0]):
    for j,f in enumerate(fields):
        setattr(tree,f,mydata[i][j])
    tree.fill()
tree.write()
froot.close()
```

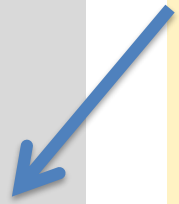
- Similar structure but different interface.
- Most notable change is the lack of an explicit buffer per branch

root_numpy

- Designed as an efficient interface between ROOT and numpy
  - C++ compiled backend that operates directly with ROOT C++ classes
  - Python3 is supported
  - Not part of the ROOT distribution, installs as a python module
  - Handles simple types, vectors of simple types, vectors of vectors of simple types, strings
    - Can work with custom types with loaded dictionaries within numpy limits (I want to explore this but have not done so yet)

Toy example via root_numpy

```
import numpy, numpy.random
import ROOT
Import root_numpy

fields=['vx','vy','vz','pt','phi','eta','q']
types=[numpy.float32,numpy.float32,numpy.float32,
numpy.float32,numpy.float32,numpy.float32,
numpy.int32]

dtypeImp=[]
for i in range(len(fields)):
   dtypeImp.append( (fields[i],types[i]) )

mydata=numpy.recarray(100,dtype=dtypeImp)
# fill mydata with random numbers (skipped)

root_numpy.array2root(mydata,'tree1.root','tree')
```



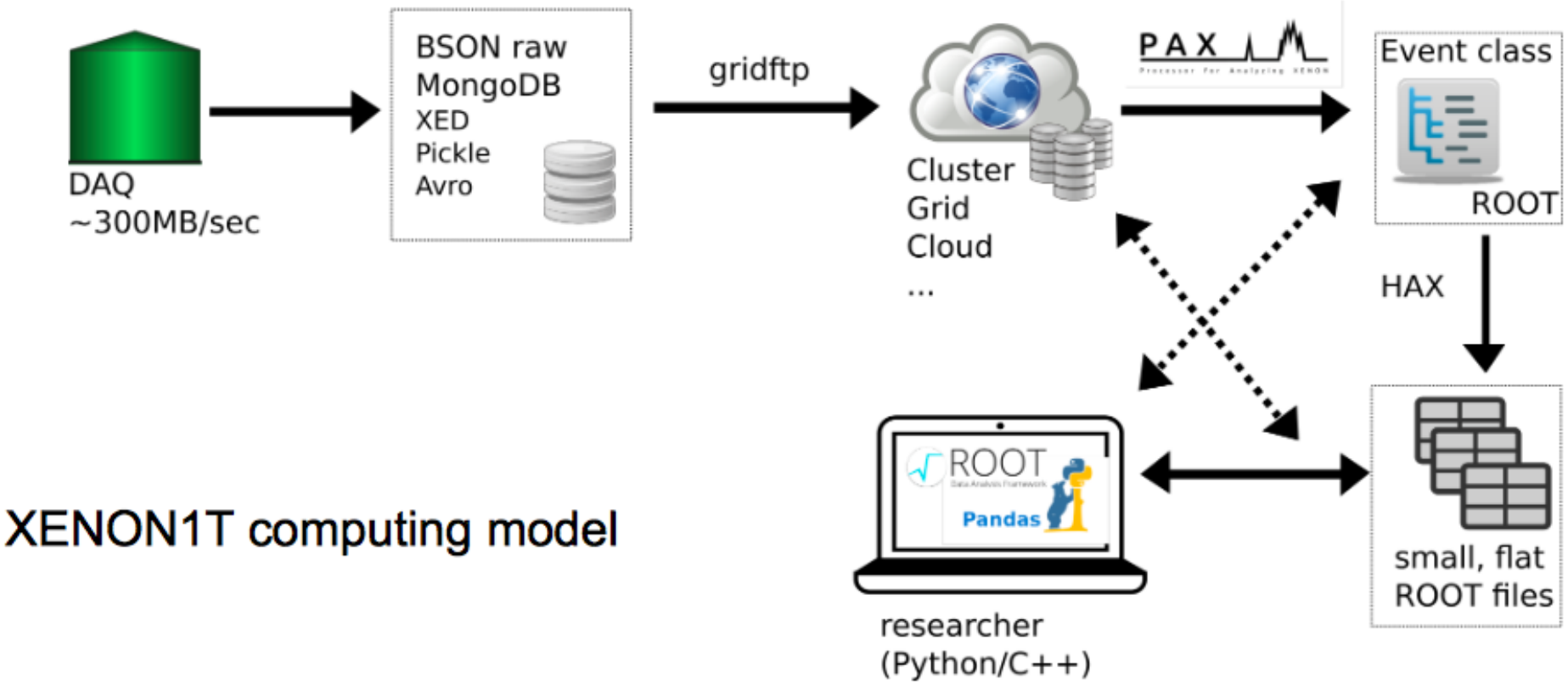
- The double loop becomes one call to c++
- Type conversion done by root_numpy package

Toy example via root_numpy

```
import numpy
import numpy.random
import root_numpy
import ROOT

rfile=ROOT.TFile('tree2.root')
tree=rfile.Get('tree')

array=root_numpy.tree2array(tree)

print array[0:10]
```

- Fast: ~15x faster reading time in my example using 10k "events"
- Of course this data conversion is what root_numpy is designed to perform effciently

ROOT + PANDAS + MongoDB based computing model at XENON1T experiment



XENON1T computing model

[HSF workshop presentation]

- Python(3) bridge between PANDAS data frames and ROOT

PANDAS and HAX

- root_pandas (GitHub) is a python package to convert to/from Pandas dataframes and ROOT data formats
  - Built around root_numpy
  - Limited to converting simple trees (no user-defined types)
- "HAX" (GitHub) written to flatten the XENON data formats for user data reduction and analysis in Pandas

- Example data model supported in HAX

```
class ReconstructedPosition(object):
    x = float('nan')
    y = float('nan')
    ....

class Peak(object):
    area = 0.0
    detector = 'ptc'
    ...
    rec_positions = list(ReconstructedPosition)

class Hit(object):
    channel = 0
    center = 0.0
    ...
class Event(object):
    event_number = 0
    dataset_name = 'Unknown'
    ...
    peaks = list(Peak)
    hits = np.array([], dtype=Hit.get_dtype())
```

Listing 2: The pax Event model

Possible future evolutions?

- Is a generalized python access to ROOT data to achieve numpy-like performance with complex types possible?
  - e.g., to efficiently read and operate on CMS miniAOD objects
- How closely to integrate python interface with ROOT distribution?
  - "Strong coupling" of interface and C++ for best performance
  - Other extreme: Python interface to ROOT data structures without ROOT installation. Easy for users outside of large experiments/labs (where ROOT is likely installed for them)
  - Implications of ROOT7?
    - Modernization means easier memory management (and big changes in PyROOT and rootpy(?)
- True Python3 support?
  - Some experiments are already using Python 3 (e.g., Belle2)

Possible future evolutions?

- Keep in mind other technologies, e.g. notebooks, spark, etc
- Moving beyond the GIL? Eventually the scientific programming community will need multi-core aware languages
  - PyPy (Wim's talk on benefits at the 2013 ROOT users workshop)
  - Julia : LLVM is behind Julia machinery. (Joosep's talk at the 2015 ROOT users workshop)