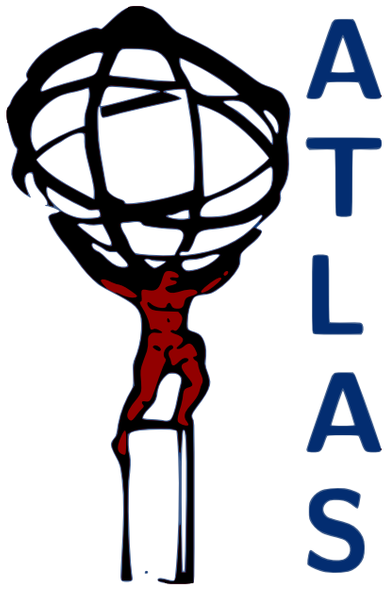
pyframe

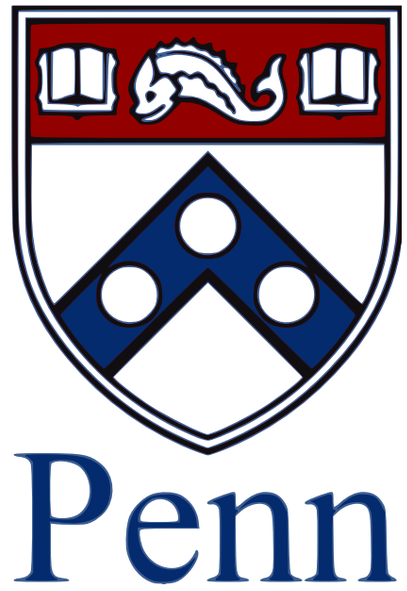
A light-weight Python framework for analyzing ROOT ntuples in ATLAS



Ryan Reece

University of Pennsylvania

ryan.reece@cern.ch



Physics Analysis Tools Meeting, CERN

September 21, 2011

What is this framework?

pyframe is a light-weight python framework for analyzing ROOT ntuples in ATLAS. It allows you to read virtually any kind of flat ntuples in python, quickly, and in a minimal framework that can scale in complexity to include several algorithms and tools. It's design goals are:

1. No EDM
2. Flat data
3. But treat objects like objects
4. Blackboard pattern for sharing data among algorithms
5. Easy addition of user data
6. Be fast, enough

1. No EDM

- Eliminate the need of a user to maintain classes describing the entire content of the data.
- Variables are accessed dynamically, on-demand.
- Of course the execution will halt if you try to access a variable that doesn't exist, but you aren't responsible for knowing more about the data than you ask of it.
- If you don't read a variable, it can't hurt you.
- No `MakeClass`. No `ReaderMaker`.

2. Flat data

- Data formats in active experiments are subject to change. pyframe makes no assumptions about the kind of data you want to analyze except:

  1. Your trees are flat, meaning that they are not filled with rich, user-defined types. The branches are basic types, arrays, or std::vectors.

  2. Groups of variables that represent the attributes of a common class of objects, should have a common prefix in their branch names, for example:

```
int el_n;
std::vector<float> el_eta;
std::vector<float> el_phi;
std::vector<float> el_pt;
```

- pyframe was designed with the use case of reading ATLAS D3PDs in mind, but there is no reason you can't read any flat ntuple with common branch prefixes.

3. But treat objects like objects

- Flat data has the huge advantages that it is easy to inspect and there is no schema evolution.
- But our problems in high energy physics really are ***object oriented***.
- pyframe benefits from assuming flat data (D3PDs), but allows you to group related variables into objects that behave as if they were a class; objects that can be collected, filtered, and sorted.
- This goal is realized with the `VarProxy` class.

VarProxy

- A `VarProxy` internally holds a reference to the tree being read, a string prefix, and an integer index.
- It has its `__getattribute__` method overridden such that when one calls `p.pt` it returns `tree.el_pt[1]`
- You can build a list of `VarProxies` for all the electrons in your tree with the `build_var_proxies` function:

```
electrons = pyframe.core.build_var_proxies(chain, chain.el_n,
                    prefix='el_')
```

  Building lists of `VarProxies` with a common prefix and saving the collection in the `store` dictionary is made more user friendly with the ListBuilder algorithm shown later.

- Now you can pt-sort the electrons

```
electrons.sort(lambda x, y: cmp(x.pt, y.pt), reverse=True)
```

- and treat each instance as if it were an instance of a class with members pt, eta, etc.

```
el = electrons[0]
el.pt
```

4. Blackboard pattern for data sharing among algorithms

- A pyframe job executes a list of algorithms in order, for each event in the data.
- Like the Gaudi framework's StoreGate, pyframe allows algorithms to share data with each other by storing any event-level derived data in the `store` dictionary, which can later be retrieved by any subsequent algorithms.
- There is also a `hists` dictionary, with the only difference that it is not cleared event-by-event, but `store` is cleared.

5. Easy addition of user data

- In the course of data analysis, one should be able to calculate new object-level derived quantities and associate them to the object.
- Python's ability to dynamically set any object's attributes makes attaching new variables to objects trivial:

```
p.my_new_var = 11.0
```

6. Be fast, enough

- Being written in python, pyframe's design favors the programmer's time and sanity, over the CPU. But of course your analysis needs to be quick enough to sample the data effectively.
- pyframe has been designed with event processing rates (per 2-3 GHz core) of 100 Hz being considered sufficienct and 1 kHz being preferred. My analysis currently runs at 400 Hz.
- Processing lots of data should be done in parallel. pyframe has been designed for easy parallelization using Python's standard module `multiprocessing`, or by submitting jobs to a batch system.

Selectors

- pyframe has modules for each object type: egamma, muon, tau, jet, met.

```
electron_selector = \
    pyframe.egamma.ElectronSelector(
        allowed_authors = [1, 3],
        min_pt = 15.0*GeV,
        allowed_etas = [(-2.47, -1.52),
            (-1.37, 1.37),(1.52, 2.47)],
        flags = ['tightWithTrack'] )
```

- Each selector inherits from a base selector that uses Python's built-in `filter`. Each selector just has to implement a `select` function that returns True/ False.

Configuring an EventLoop

- One adds algorithms to an EventLoop:

```
loop = pyframe.core.EventLoop('myloop')
loop += pyframe.grl.GRLFilter(config['grl'])
loop += pyframe.algs.ListBuilder(
                prefixes = ['vxp_'],
                keys = ['all_vertices'],
                )
loop += \
   pyframe.selectors.SelectorAlg('VertexSelectorAlg',
                selector = vxp_primary_selector,
                key_in='all_vertices',
                key_out='primary_vertices',
                )
...
loop.run(chain, 0, config['max_events'])
```

Turning Off Branches with TreeProxy

- O(10) speed improvements can be achieved by simply turning off branches that are not read in the analysis.
- The `TreeProxy` class makes this easy by logging every variable that is read, and optionally dumping the list to a file.
- When running an `EventLoop`, one can configure to dump the variable log, and/or select which branches to turn on by:

```
loop.run(chain, 0, config['max_events'],
    branches_on_file = config.get('branches_on_file'),
    do_var_log = config.get('do_var_log'),
    )
```

Profile your analysis as you develop

- A pyframe `EventLoop` automatically monitors the execution time for each algorithm.
- A report of the the time used is dumped at the end of the job:

```
ALGORITHM TIME SUMMARY
  # ALGORITHM                          TIME [s]  RATE [kHz]
  0 MCEventWeight                             0        31.3
  1 GRLFilter                                 0        30.9
  2 ListBuilder                               0        10.5
  3 PrimaryVertexSelectorAlg                  8        3.67
...
```

Parallelization

- On a multicore machine, one can make use of Python's multiprocessing module just by supplying a command line argument specifying how many cores to use:

  ```
  ./job.py -p 4
  ```

  This gives functionality similar to PROOF-lite, with pure python.

- On a cluster, I've been running Condor jobs easily. You just need Python and ROOT on all the worker nodes.

- `pyutils/condor.py` is a module I wrote for dividing up the input files and generating condor submission scripts.

Installation

- Requirements:
  - Python 2.6
  - ROOT
  - SVN
- Instructions and documentation:
  - https://twiki.cern.ch/twiki/bin/view/Sandbox/PyFrame
  - checkout pyframe and pyutils
  - add to your PYTHONPATH
  - only thing to compile is external packages using RootCore
  - has a helloworld that runs out of the box

pyframe summary

- Pure python framework for reading flat ntuples
- VarProxy allows you to treat related variables like objects, without maintaining classes.
- Parallelization with multiprocessing and condor.
- Uses RootCore to use external tools and ATLAS recommendations.
- Get the full advantage of Python
  - clean syntax
  - duck typing
  - string formatting
  - standard modules
  - garbage collection (no memory leaks)

My workflow

- `x.D3PD.root`
  → `tree_trimmer.py` → `x.skim.root`
  → `pyframe` → `x.hist.root`
  → `metaroot` → `x.canv.root`
  → `root2html.py` → `www`
- `tree_trimmer.py` is a general python script for skimming events and dropping branches.
- `pyframe` is used to make an event loop for filling histograms.
- `metaroot` is a python package for formatting histograms, producing a file of `TCanvases`.
- `root2html.py` is a general script that generates an html document of figures with captions with detailed statistics from any ROOT file of TCanvases. Some example output:
  - ditau analysis:
    https://reece.web.cern.ch/reece/share/ditau.tau-mu.2011-09-05/
  - tau performance studies:
    https://reece.web.cern.ch/reece/share/plot_vars/
  - TRT performance studies:
    https://reece.web.cern.ch/reece/share/trt_eff.data_182787.mc_J2/

More info

- `tree_trimmer.py`
  https://svnweb.cern.ch/trac/penn/browser/reece/rel/tree_trimmer/trunk
- `pyframe`
  https://twiki.cern.ch/twiki/bin/view/Sandbox/PyFrame
- `metaroot`
  https://svnweb.cern.ch/trac/penn/browser/reece/rel/metaroot/trunk
- `root2html.py`
  https://svnweb.cern.ch/trac/penn/browser/reece/rel/root2html/trunk