Python and ATLAS offline

Scott Snyder
On behalf of the ATLAS Collaboration

Brookhaven National Laboratory, Upton, NY, USA

July 08, 2018
PyHEP 2018

Python use by ATLAS offline

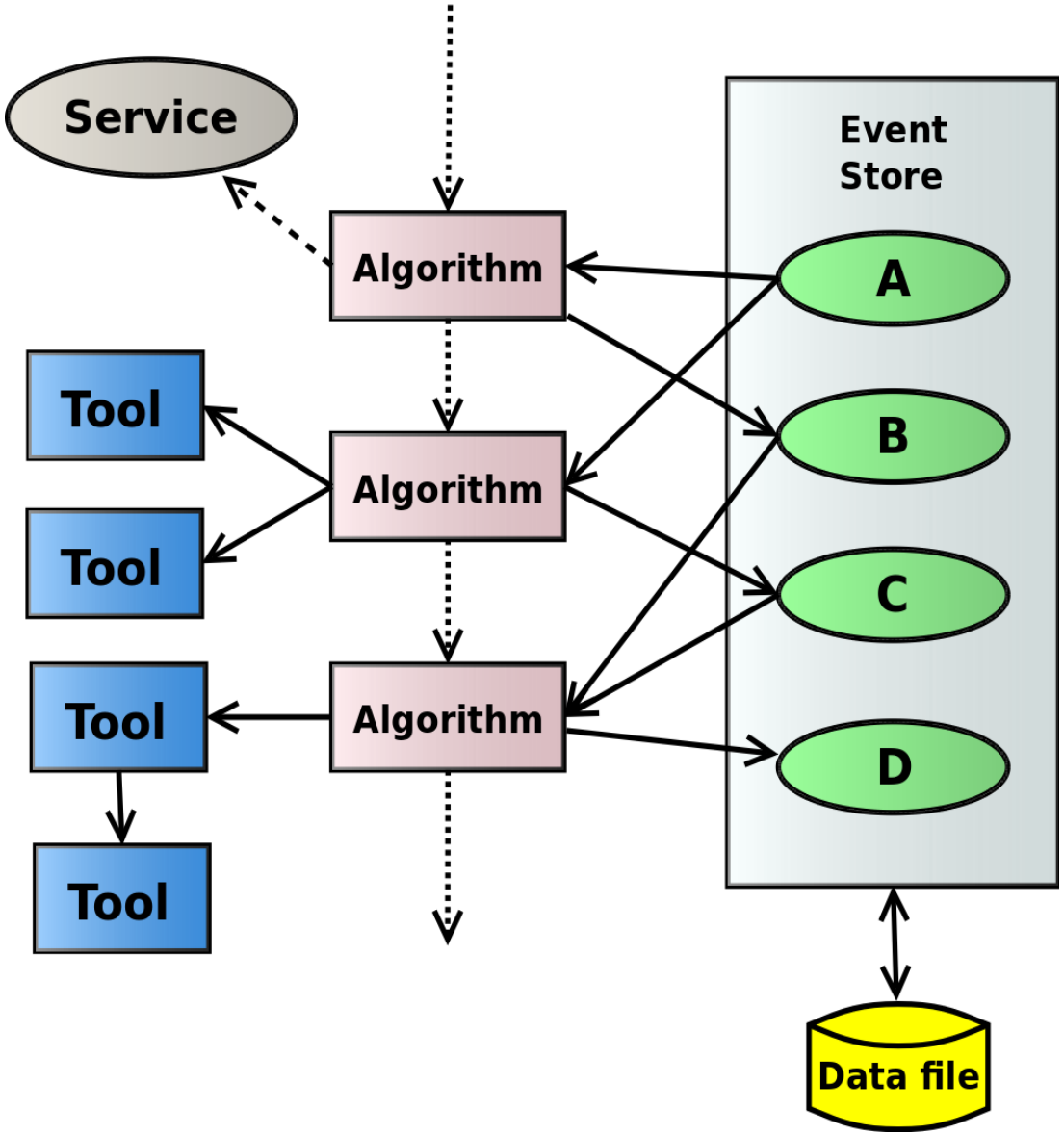
(Excluding here online and analysis use cases.)

- ATLAS offline (Athena) uses Python extensively.
- ATLAS repository contains $\sim$ 1M lines of python code ($\sim$ 20% of the total).
- Major uses:
  - Job configuration.
  - Direct use of reconstruction objects from Python.
  - Building/testing/general scripting.
- Python 2.7 used throughout.
  - No clear plans for Python 3 (but see later).
- Interaction with C++ code almost entirely through PyROOT.

Athena and Gaudi

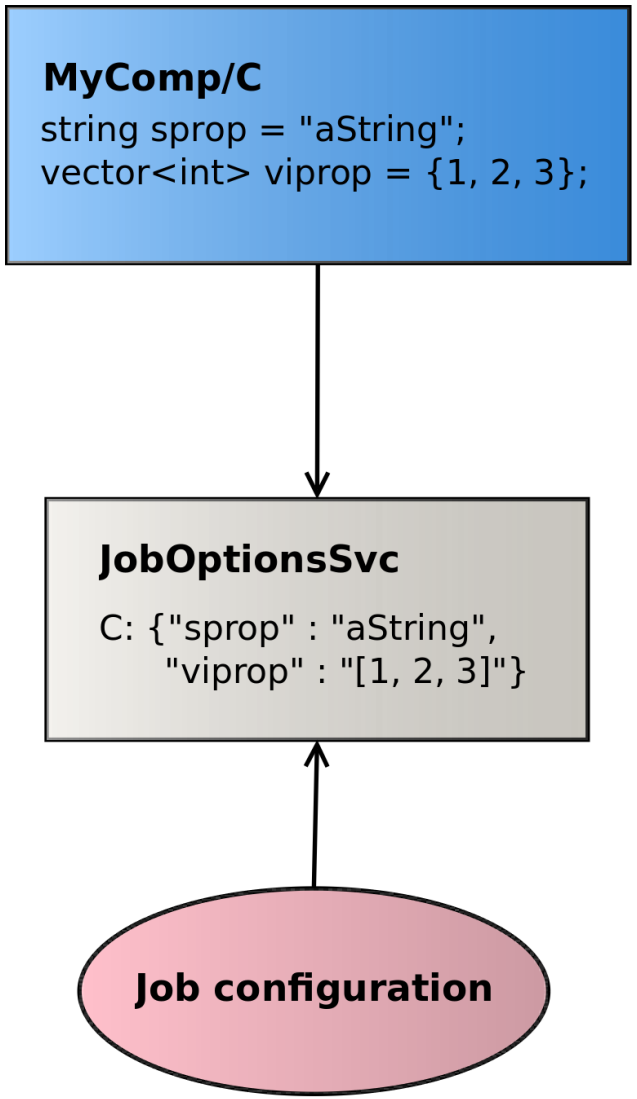
Athena uses Gaudi, shared with LHCb and other experiments.

- Dynamically loadable *components*.
  - Including algorithms, tools, services.
- Ideally used via abstract interfaces.
- Algorithms and tools can own other tools.
- Services are singletons.
- Identified with a type (C++ type) and name (instance).
- Each component has a list of *properties*.



Component properties

- Each component has a list of named properties.
  - Of various C++ types.
  - Corresponding to data members in the component C++ class.
- Also can have properties representing references to other components: ToolHandle and ServiceHandle.
- When a component initializes, it queries JobOptionsSvc for its properties.
- JobOptionsSvc holds property settings for each component, as strings.
- JobOptionsSvc populated by job configuration.



Python component configuration interface: Configurable

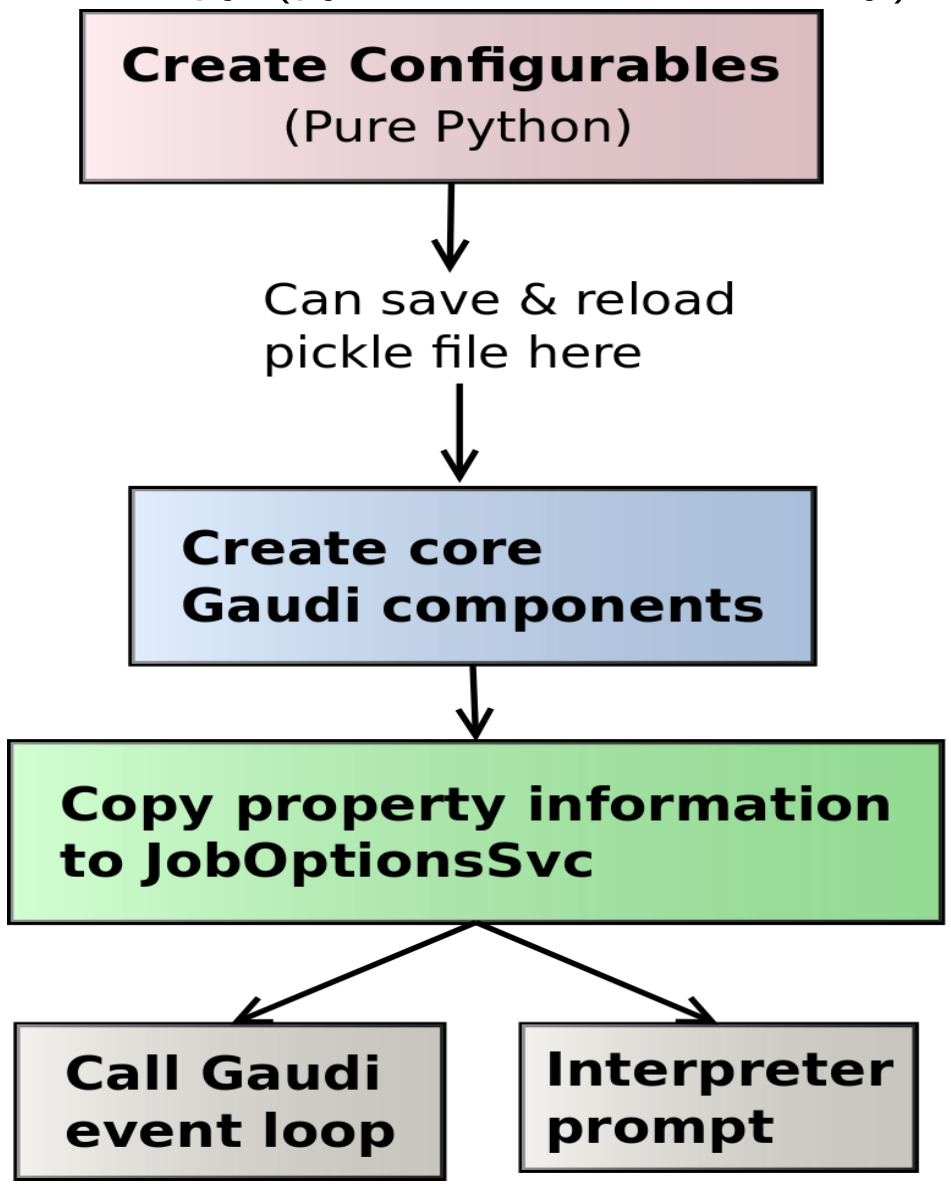
Each component has a Python "Configurable" class to collect information during job configuration:

```
from MyComponents.MyComponentsConf import MyComponent
# Create an instance of MyComponent named mycomp and
# set some properties.
mycomp = MyComponent ('mycomp',
                      sprop = "some string",
                      viprop = [3, 2, 1])
# Can change/set properties later.
mycomp.iprop = 10
mycomp.sprop = "some other string"
```

Configurables

- Configurables are pure Python classes.
  - Can be pickled, etc.
- Generated automatically during build from the shared libraries containing the components.
- Singleton behavior: `MyComponent("foo")` always gives the *same* object.
- Hold property settings, and can later transport them to JobOptionsSvc.

athena.py (python is Athena binary):



Component references

Configurables can reference other Configurables to build up a complete job configuration.

```
from MyComponents.MyComponentsConf import *

# Create a Service; register with ServiceMgr.
myserv = MyService ('myserv', parm=1)
svcMgr += myserv

# An algorithm using this service plus a tool.
myalg = MyAlgorithm ('myalg',
                     service = myserv,
                     tool = MyTool ('mytool',
                                    arg='something'))

# Schedule this algorithm to run in sequence.
topSequence += myalg
```

Other configuration helpers

- `include`: Textual inclusion of another Python fragment.

```
include('Calorimeter/CaloRec.py')
```

- Job configuration flags:

```
from CaloRec.CaloRecFlags import jobproperties as jp
jp.CaloRecFlags.doTileCorrection = True
if jp.CaloRecFlags.doTileMuID():
    ...
```

- Some properties set automatically from input file metadata:

```
tool = MyTool ('mytool',
               energy = jobproperties.Beam.energy)
```

Done? Maybe not.

- Doesn't scale — becomes unmanageable with thousands of components.
- Everything in the global namespace — easy to have collisions.
- Different pieces of the configuration can try to configure the same component in different ways.
- Many components have prerequisites that must be configured first.
  - Difficult even for experts to get it right.
- Several attempts made in the past to add more structure.
  - Put configuration into imported Python modules.
  - Various registries for looking up configuration information.
- Helped somewhat — but now different parts of the job are configured in different ways.

Run 3 job configuration

- Working on overhauling how job configuration works for run 3.
- Subject of a separate poster; only brief summary here.
- No global namespace; make dependencies explicit.
- Everything in imported modules; no include.
- Configuration of component also configures all its dependencies.
  - Configurations are self-contained.
- Modules can be run stand-alone and concatenated:
  - Need to remove duplicates if the some component is configured multiple times.
- Will probably be used for major parts of the Run 3 configuration.
  - May not have sufficient effort available to convert everything before Run 3, given other migrations also in progress.
- Trying to keep new code compatible with both Python 2 and 3.

Run 3 job configuration: simple example

```
def MyAlgoCfg(inputFlags):
    result=ComponentAccumulator()
    isMC=inputFlags.get("AthenaConfiguration.GlobalFlags.isMC"

    # Set up geometry.
    from Geometry.GeomConfig import GeomCfg
    result.addConfig (GeomCfg, inputFlags)

    form MyAlgoPackage.MyAlgoPackageConf import MyAlgo
    myAlgo = MyAlgo(isData = not isMC)
    result.addEventAlgo(myAlgo)
    return result
```

Access to event data and Athena components from Python

- Nearly all the Atlas event data classes are fully functional from Python.
- Can inspect event data interactively from the Python prompt.
- Many tools/services are callable from Python as well.
- Can write an algorithm entirely in Python:

```
import AthenaPoolCnvSvc.ReadAthenaPool
ServiceMgr.EventSelector.InputCollections = [ 'data.root' ]
from AthenaPython.PyAthenaComps import Alg, StatusCode
class MyAlg (Alg):
    def execute (self):
        for el in self.evtStore['Electrons']:
            print el.pt(), el.eta(), el.phi()
        return StatusCode.Success

from AthenaCommon.AppMgr import topSequence
topSequence += MyAlg ('myalg')
```

Python components in Athena

- Ability to access Athena components and event data has been very useful for testing and debugging.
- Numerous unit/regression tests are written in Python.
- Code to dump ATLAS events is written in Python.
- Used to have a few pure-Python algorithms in the standard reconstruction.
  - Eventually removed, not due to speed, but rather due to the memory required by the ROOT dictionaries.
- Some other workflows still use Python-based components (e.g., event display).
- Users also have Python analysis code that that processes ATLAS event data.

Building and scripting

- Python is the preferred ATLAS scripting language for tasks beyond what's appropriate for shell scripts.
  - Setting up Athena environment.
  - Managing the running of reconstruction jobs with complicated configurations.
  - Atlas-specific helpers for git.
  - Interacting with the conditions database.
  - Test drivers; test result validation.
  - Debugging; diagnostics; dumping.
  - Code generation.
  - Analysis of code performance and data quality.
- Much middleware / data handling code also heavily relies on Python.

Summary

- Python is heavily used in ATLAS offline code.
  - Second only to C++.
- Major use is for job configuration.
  - Currently being reworked to be more modular and maintainable.
- Access to Athena components and event data from Python is invaluable for debugging and testing.
- Python is also heavily used in building and testing Athena, and in many other roles.
- Currently no firm plans for moving to Python 3.
  - Would like to do it, but not a priority for Run 3.
  - Modernizing the way job configuration works should help prepare for this.
  - Being able to run both py2 and py3 with the same ROOT build would also be a help!