Usage of the Python programming language in the CMS experiment

R Wilkinson¹, B Hegner², and C D Jones³

¹California Institute of Technology, Pasadena, CA, USA
²CERN, Geneva, Switzerland
³Fermi National Accelerator Laboratory, Batavia, IL, USA

E-mail: rickw@caltech.edu

Abstract. Being a highly dynamic language and allowing reliable programming with quick turnarounds, Python is a widely used programming language in CMS. Most of the tools used in workflow management and the GRID interface tools are written in this language. Also most of the tools used in the context of release management: integration builds, release building and deploying, as well as performance measurements are in Python. With an interface to the CMS data formats, rapid prototyping of analyses and debugging is an additional use case. Finally in 2008 the CMS experiment switched to using Python as its configuration language. This paper will give an overview of the general usage of Python in the CMS experiment and discuss which features of the language make it well-suited for the existing use cases.

1. Introduction

Many software projects on the Compact Muon Solenoid experiment (CMS) have independently chosen to use Python [1], not as a result of a top-down decision from management. When asked why they use Python, some common reasons emerge: Python is seen to be easy to learn, without the steep learning curve of C++. Python syntax is seen as simpler and more comprehensible than either C++ or Perl, which makes it easier to understand code written by others. This simplicity leads to the option of writing prototype code in Python, and then, once objects and behaviors are determined, translating to C++ for performance. Finally, many standard tools are native to the language, and many useful external packages exist, such as cherrypy [2] for web programming, PyROOT [3] for physics analysis, and PyQt [4] for graphics.

2. Job Configuration

The CMS software framework uses a “software bus” model, where data is stored in the event which is passed to a series of modules. A single executable, cmsRun, is used, and the modules are loaded at runtime. A configuration file defines which modules are loaded, in which order they are run, and with which configurable parameters they are run. Note that this is not an interactive system. The entire configuration is defined once, at the beginning of the job, and cannot be changed during running. This design facilitates the tracking of event provenance [5], that is, the processing history of the event.

A software release contains over 6000 configuration files. Three quarters of these are fragments, meant to be shared, defining the parameters for a single module or sequence of modules. The rest are
full, executable configurations. Our standard full-chain release validation job defines over 700 modules, over 150 sequences of modules, and over 13,000 configurable parameters.

These configurations have a rich syntax. Many data types exist, including int32, uint32, int64, double, string, InputTag, EventID, and vectors of these types. These data types can be stored hierarchically in PSets or VPSets. Modules which operate on event data can be defined as EDProducers, EDFilters, or EDAnalyzers. Modules which handle time-dependent conditions can be defined as ESSources or ESProducers, and sets of conditions can be chosen using ESPrefer statements. Modules can be grouped into Sequences and execution Paths, which can be stopped by EDFilters. In addition, operators can define the relationships between elements in a path, such is inverting (“~”) or ignoring (“!”) a filter, or expressing whether or not modules depend on their previous element for data (“*” or “+”).

These configurations had been implemented using a custom language syntax, parsed by the GNU tools Flex and Bison [6]. This system was designed to be a simple declarative language, but it soon proved too inflexible. Users commonly wanted to copy and modify sets of parameters and modules, which meant that we had to provide that syntax for each of the many elements of the language. The production system especially needed easy access to modify input and output file names and random number seeds. We needed a full programming language, and since the production system was already written in Python, it was the natural choice.

To design the language syntax, we tried to mimic the original language as much as possible. The new Python configuration returns a single Python data structure, the Process, into which modules would be inserted. The Python configuration would be started from the C++ cmsRun program through a boost::python [7] interface, and the Python Process object would be converted into the same C++ data structure as the old language was, again, using boost::python. Examples of the old and new languages are shown in figure 1.

```python
process foo = {
    module jetAnalyzer = MyJetAnalyzer {}
    module jet50Filter = jetFilter from "Jets/Reco/data/JetFilter.cfi"
        replace jet50Filter.minPt = 50.
    module jet100Filter = jetFilter from "Jets/Reco/data/JetFilter.cfi"
        replace jet100Filter.minPt = 100.
    sequence jet50to100Filter = {jetFilter50 & !jetFilter100}
    path jet50to100 = {jet50to100Filter, jetAnalyzer}
}
```

```python
import FWCore.ParameterSet.Config as cms
process = cms.Process("foo")
process.jetAnalyzer = cms.EDAnalyzer("MyJetAnalyzer")
from Jets.Reco.JetFilter_cfi import jetFilter
process.jet50Filter = jetFilter.clone()
process.jet50Filter.minPt = 50.
process.jet100Filter = jetFilter.clone()
process.jet100Filter.minPt = 100.
process.jet50to100Filter = cms.Sequence(
    process.jet50Filter + ~process.jet100Filter
)
process.jet50to100 = cms.Path(
    process.jet50to100Filter * process.jetAnalyzer
)
```

Figure 1. Examples of the old (left) and new (right) configuration languages.

After the new Python data structures were defined, we created a tool to parse old configuration files into the new data structure, using the pyparsing [8] module. The new Python data structures could
dump themselves into both the new and the old formats, so the production system could immediately use this parsing tool to edit configurations.

Next came the task of translating the ~6000 configuration files in the system. We were given CVS permissions for the entire release, and we made scripts to automatically parse, translate, commit, and tag all packages that had configuration files. Some structures didn’t translate well automatically, so we edited the original configuration by hand, because we expected iterations of automatic translations. A period of one pre-release was defined where we would support both configuration languages, and we made scripts to detect when an old configuration file was changed without a change to the new one.

One disadvantage of this translation scheme was that neither comments nor order of configuration statements were preserved. To translate comments, we made a second-pass script which tried to match the line following the comment in the new file.

After we automatically translated all configuration files in CVS, we announced the location of the translation scripts, so users could translate their own configurations. We created a validation program, which dumped the C++ data structure that resulted from both old and new configurations, and checked them for differences.

After the transition, the maintenance burden was indeed reduced. In addition, the new system had powerful new features. The configurations became easier to debug, because users could inspect the configurations interactively, and could test their configuration syntax simply by compiling it. It became easier to build configurations, and to keep the configurations internally consistent by, for example, changing input and output file names simultaneously. We no longer need separate Perl or shell scripts to edit the configurations. Users can now use command-line arguments, and higher-level Python functions such as loops. Finally, a browsing GUI was created, which uses PyQt [9].

We also have configuration builder utilities, written in Python, to assemble common uses, such as simulation and reconstruction chains, and to add common sets options to them, such as those needed for fast simulation or cosmic ray reconstruction. This configuration builder is used for release validation, and to create standard workflows for production.

3. Analysis
CMS stores its data in ROOT files. There are two common methods to access this data, described in more detail in [10]. The most common is to use the standard framework, creating a C++ “EDAnalyzer” module to read the data, and perhaps create a new ROOT file, extracting selected data. Another way is to use “FWLite”, a utility which provides easy access to the objects in the event, and their C++ member functions, in either C++ or Python. FWLite auto-loads the C++ objects it needs, with class dictionaries provided by ROOT’s REFLEX utility.

FWLite can be combined with PyROOT to create a powerful and intuitive analysis tool. The code in figure 2 can be used to create analysis plots, produces histogram windows directly from a Python prompt, with a simplicity approaching pseudocode. To further simplify the use of Python interactively, the user could save the first four lines of setup code into a new script, and use that script to start an interactive Python session:

```python
> python -i openRecoFile.py
```
4. Production
The CMS Data Management and Workflow Management team uses Python extensively in data processing. Clusters of Python daemons perform tasks such as request management, allocation, job submission, tracking, bookkeeping, and error handling [11]. These daemons use a common Python framework for event-driven message passing and MySQL persistency.

In this architecture, the processing work is done by “ProdAgents” [12], independent daemons which act as front ends to diverse resources, ranging from farms to grid systems. These ProdAgents receive tasks from a “ProdMgr”, which manages requests, and does the final accounting.

In addition, many web applications use cherrypy servers, for such tasks as data moving [13], site monitoring [14], data monitoring [15], and database browsing [16]. These tools are being consolidated into a single framework [17].

5. Conclusion

CMS uses Python for a wide variety of applications, including scripting, job configuration, analysis, GUIs, web interfaces, message passing, and database interfaces. We anticipate that Python’s popularity on the experiment will continue to grow, as more shared utilities and frameworks in Python become available.

References
[5] Jones C D “File level provenance tracking in CMS”, these proceedings
Hinzmann A “Visualization of the CMS Python configuration system”, these proceedings


Wakefield S et al. “CMS production and processing system – design and experiences”, these proceedings

Spiga D 2007 Nuclear Physics B Proceedings Supplements p 141-144

Kuznetsov V “CMS FileMover: one click data”, these proceedings

Metson S “SiteDB: marshalling the people and resources available to CMS”, these proceedings

Tuura L “Authentication and authorization in CMS’ monitoring and computing web services”, these proceedings

Pierro A “CMS Conditions Database Web Application Service”, these proceedings

Wakefield S et al. “Job life cycle management libraries for CMS workflow management projects”, these proceedings