PRINCETON UNIVERSITY

Python for “core software” in CMS

David Lange

Princeton University

July 8, 2018

Python is a central component of the CMS software environment

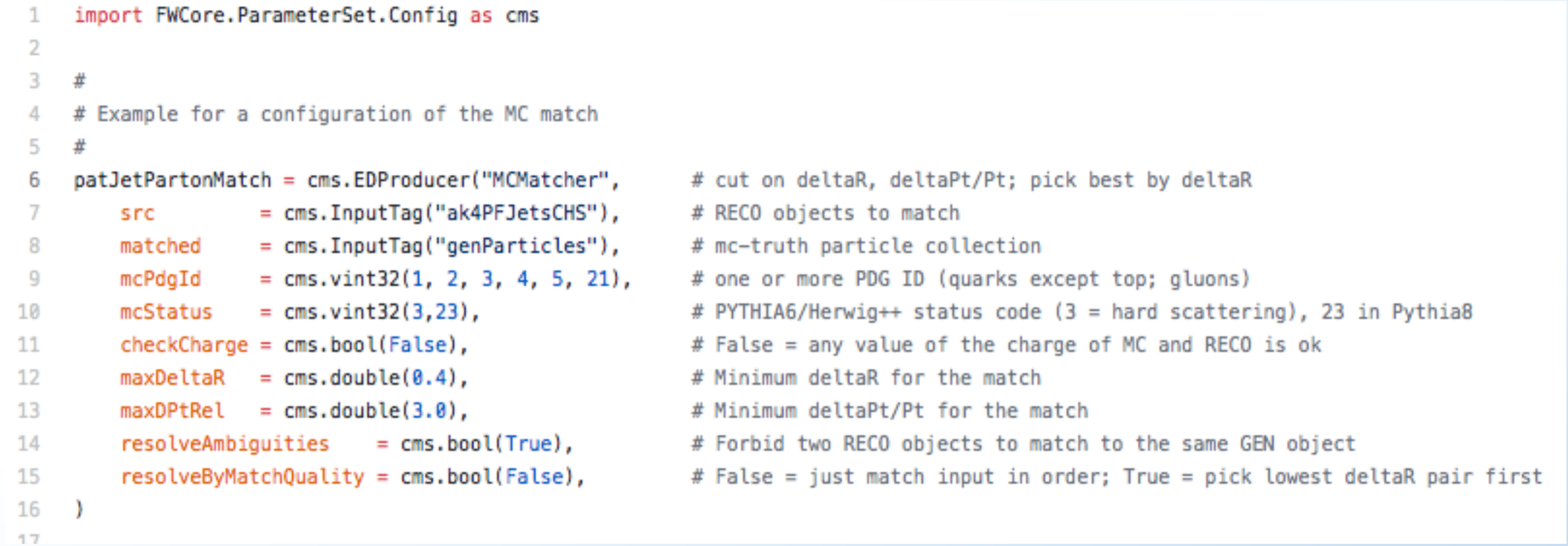
- CMSSW framework configuration
- Many (nearly all?) analyses are Python driven for at least some stages of the analysis if not all of them (relying on PyROOT etc)
- Basis of data science tools used across the collaboration
- Scripting language of choice
- Workflow management, monitoring, data management, etc are primarily Python

CMSSW Framework changed to Python as its job control / configuration language in ~2008

- Jobs are defined by configuration files (glued together from fragments associated with one or more C++ plugins) and are not interactive
  - In the CMSSW release, we are now at the scale of
    - 10K fragments
    - 4K configuration files maintained in the CMSSW release
    - Configuration files for production workflows are produced by a (Python) tool given their complexity (O(1000) fragments organized hierarchially)

- Use boost::python to translate Python configurations into C++ data structures (the Python state is discarded before event processing begins)

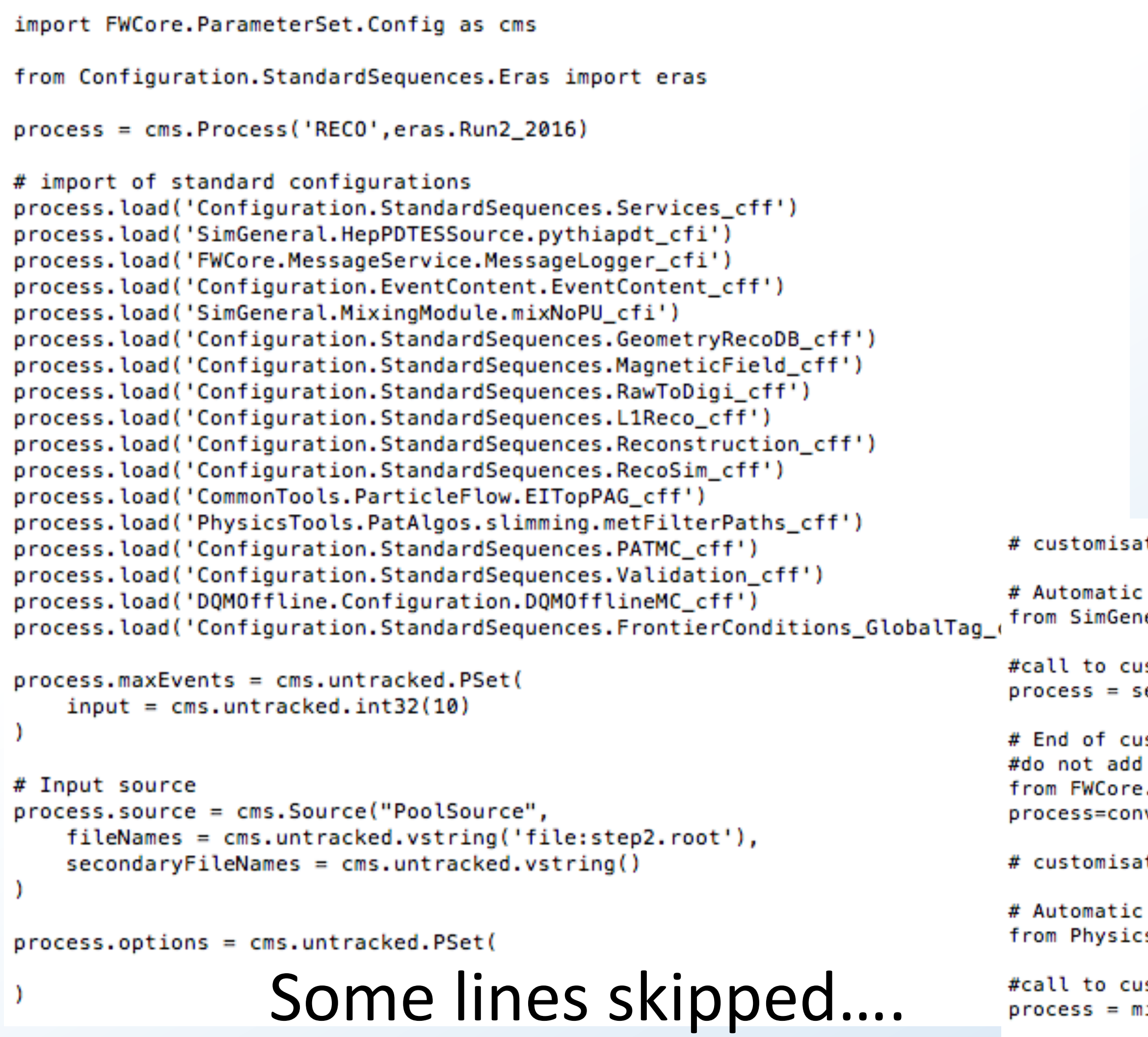
A simple example of a basic configuration snippet defining a module



```python
import FWCore.ParameterSet.Config as cms

#
# Example for a configuration of the MC match
#
patJetPartonMatch = cms.EDProducer("MCMatcher",      # cut on deltaR, deltaPt/Pt; pick best by deltaR
    src          = cms.InputTag("ak4PFJetsCHS"),      # RECO objects to match
    matched      = cms.InputTag("genParticles"),      # mc-truth particle collection
    mcPdgId      = cms.vint32(1, 2, 3, 4, 5, 21),     # one or more PDG ID (quarks except top; gluons)
    mcStatus     = cms.vint32(3,23),                  # PYTHIA6/Herwig++ status code (3 = hard scattering), 23 in Pythia8
    checkCharge = cms.bool(False),                    # False = any value of the charge of MC and RECO is ok
    maxDeltaR    = cms.double(0.4),                   # Minimum deltaR for the match
    maxDPtRel    = cms.double(3.0),                   # Minimum deltaPt/Pt for the match
    resolveAmbiguities    = cms.bool(True),           # Forbid two RECO objects to match to the same GEN object
    resolveByMatchQuality = cms.bool(False),          # False = just match input in order; True = pick lowest deltaR pair first
)
```

- Fragments are grouped hierarchically to build full applications

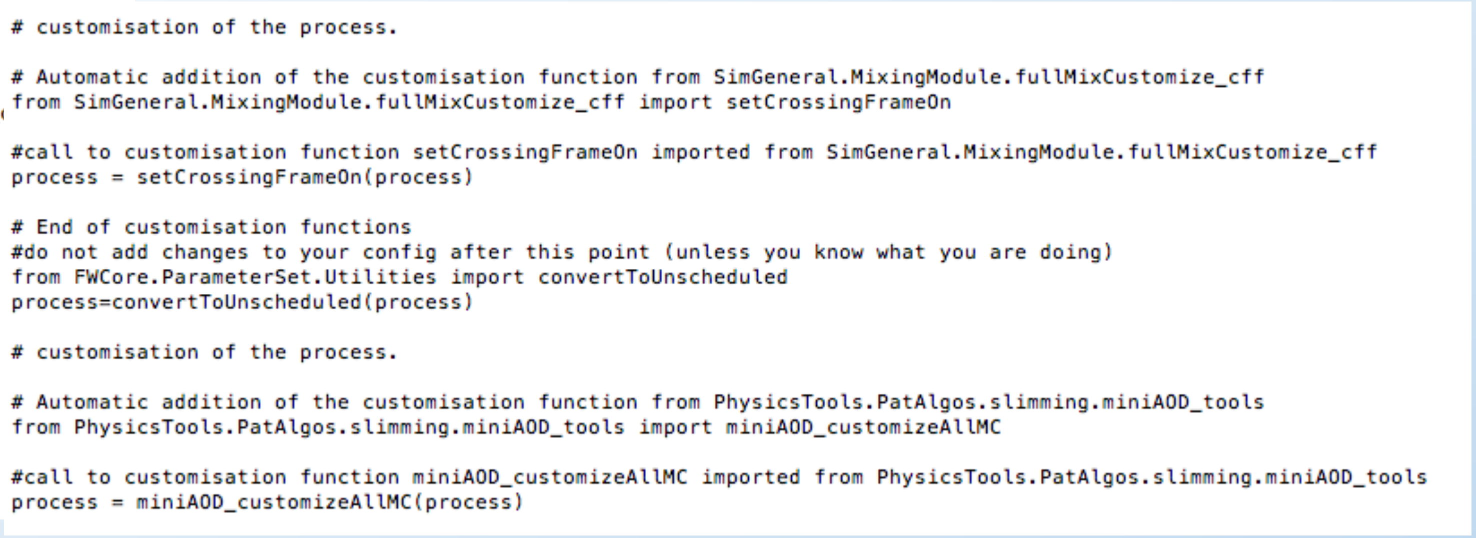
Example application



```
import FWCore.ParameterSet.Config as cms

from Configuration.StandardSequences.Eras import eras

process = cms.Process('RECO',eras.Run2_2016)

# import of standard configurations
process.load('Configuration.StandardSequences.Services_cff')
process.load('SimGeneral.HepPDTESSource.pythiapdt_cfi')
process.load('FWCore.MessageService.MessageLogger_cfi')
process.load('Configuration.EventContent.EventContent_cff')
process.load('SimGeneral.MixingModule.mixNoPU_cfi')
process.load('Configuration.StandardSequences.GeometryRecoDB_cff')
process.load('Configuration.StandardSequences.MagneticField_cff')
process.load('Configuration.StandardSequences.RawToDigi_cff')
process.load('Configuration.StandardSequences.L1Reco_cff')
process.load('Configuration.StandardSequences.Reconstruction_cff')
process.load('Configuration.StandardSequences.RecoSim_cff')
process.load('CommonTools.ParticleFlow.EITopPAG_cff')
process.load('PhysicsTools.PatAlgos.slimming.metFilterPaths_cff')
process.load('Configuration.StandardSequences.PATMC_cff')
process.load('Configuration.StandardSequences.Validation_cff')
process.load('DQMOffline.Configuration.DQMOfflineMC_cff')
process.load('Configuration.StandardSequences.FrontierConditions_GlobalTag_

process.maxEvents = cms.untracked.PSet(
    input = cms.untracked.int32(10)
)

# Input source
process.source = cms.Source("PoolSource",
    fileNames = cms.untracked.vstring('file:step2.root'),
    secondaryFileNames = cms.untracked.vstring()
)

process.options = cms.untracked.PSet(

)
```

Some lines skipped....

Applications can be consumed by ”cmsRun” executable to process events, inspected at Python prompt, or otherwise scripted against



```
# customisation of the process.

# Automatic addition of the customisation function from SimGeneral.MixingModule.fullMixCustomize_cff
from SimGeneral.MixingModule.fullMixCustomize_cff import setCrossingFrameOn

#call to customisation function setCrossingFrameOn imported from SimGeneral.MixingModule.fullMixCustomize_cff
process = setCrossingFrameOn(process)

# End of customisation functions
#do not add changes to your config after this point (unless you know what you are doing)
from FWCore.ParameterSet.Utilities import convertToUnscheduled
process=convertToUnscheduled(process)

# customisation of the process.

# Automatic addition of the customisation function from PhysicsTools.PatAlgos.slimming.miniAOD_tools
from PhysicsTools.PatAlgos.slimming.miniAOD_tools import miniAOD_customizeAllMC

#call to customisation function miniAOD_customizeAllMC imported from PhysicsTools.PatAlgos.slimming.miniAOD_tools
process = miniAOD_customizeAllMC(process)
```

Using Python language complexity

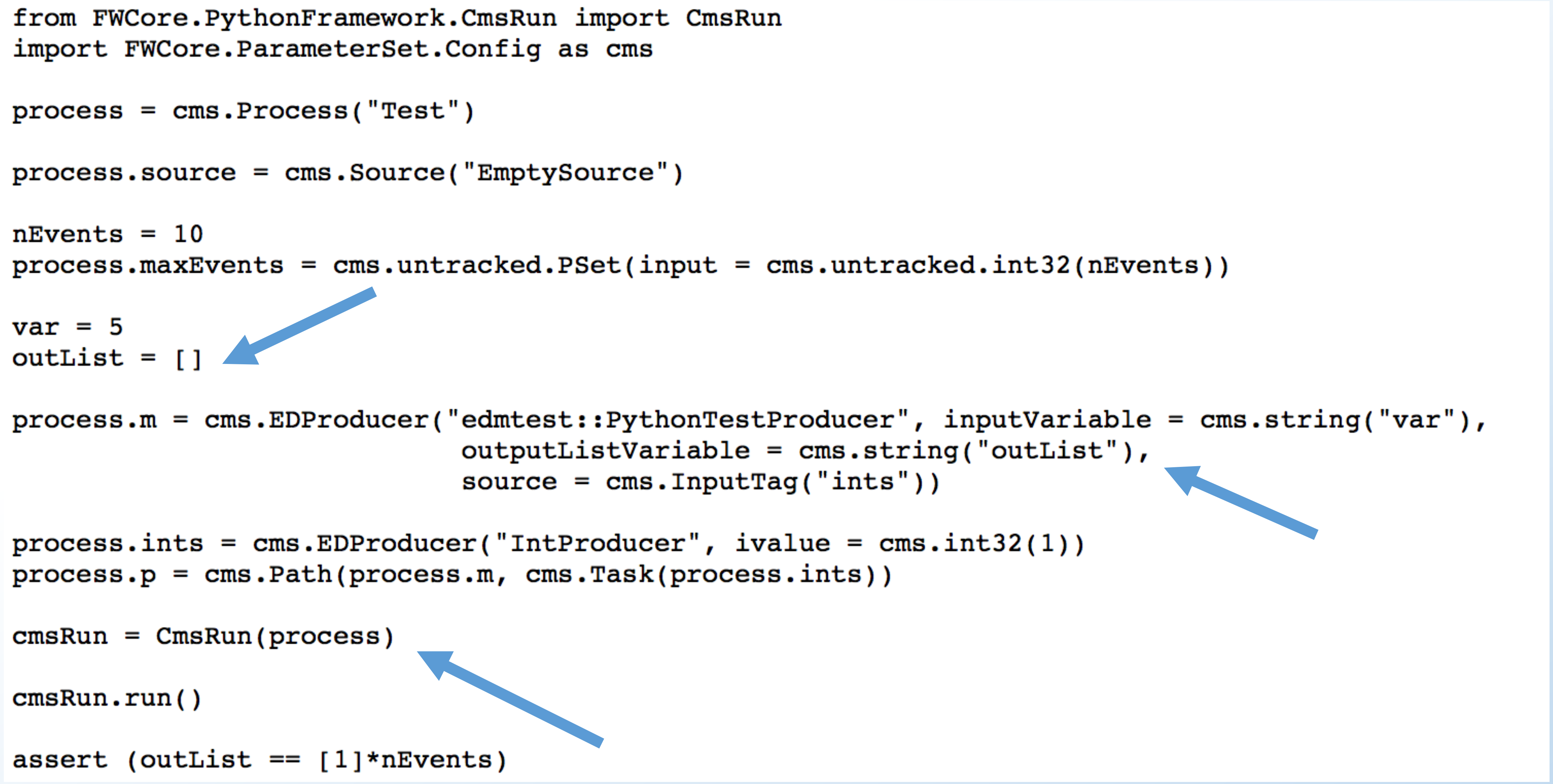
- We started from a relatively basic Python usage
  (was largely a straight migration from a custom language format)
  - Its generally straightforward to understand CMSSW job configurations without being an expert in Python.

- Developers now take advantage of Python functionality to write more maintainable configuration.
  - Complexity tends to increase closer to analysis codes / developers
  - Growth in code complexity is managed by policy of having all manipulations for a module instance in a single file (this is not always enforced, but we try…)

Some examples of how we take advantage of Python capabilities [or generically, a programming language for configuration..]

- Typical motivation is to avoid code duplication. Examples:
  - Analysis configurations are often quite complex to facilitate multiple use cases of the same code
  - Global manipulations of the configuration. Examples are allowing modules to configure themselves depending on application or input data, and process-level changes applied to entire configuration
  - Managing two configuration components that need to stay in sync (via information sharing in one form or another)
  - Defining a series of modules with similar configurations (eg, monitoring for pT bin)

We can also run applications from within Python and use the in-memory results of the C++ application



```
from FWCore.PythonFramework.CmsRun import CmsRun
import FWCore.ParameterSet.Config as cms

process = cms.Process("Test")

process.source = cms.Source("EmptySource")

nEvents = 10
process.maxEvents = cms.untracked.PSet(input = cms.untracked.int32(nEvents))

var = 5
outList = []

process.m = cms.EDProducer("edmtest::PythonTestProducer", inputVariable = cms.string("var"),
                           outputListVariable = cms.string("outList"),
                           source = cms.InputTag("ints"))

process.ints = cms.EDProducer("IntProducer", ivalue = cms.int32(1))
process.p = cms.Path(process.m, cms.Task(process.ints))

cmsRun = CmsRun(process)

cmsRun.run()

assert (outList == [1]*nEvents)
```

Maintaining Python for CMSSW

- As with other dependencies, CMSSW is distributed with its own Python.
  - Switched to Python 2.7 in 2012
  - Now in the process of changing to Python3

- Package management for Python tools is done almost exclusively via pip and PyPI repository for sources.
  - Almost 150 packages in part thanks to push to include broad suite of data science packages
  - Typically very simple build recipes, so easy to maintain.
  - Build and distribute python2 and python3 versions in parallel within a release build

Targeting change over to Python early in long-shutdown 2

- Working through most significant syntax changes
  - print
  - keys(), values() and their iter friends
  - Etc [The next presentation will explain much more about this process]
- Relying heavily on futurize toolkit, but clearly we have lots of hand work to be done

- Staged approach enabled by having python2 and python3 distributed within same release stack.

Looking forward: We are evaluating a change from Boost::python to PyBind11

- Pybind11 is essentially a header only reimplementation of Boost::Python with C++11, and without any dependency on boost.

- Facilitates simultaneous support for Python2 and Python3 in framework

- Pros/cons including performance will be evaluated before making a decision to change. We do not have an extensive interface, so this migration should not be difficult