Interfacing Python and the C++ Frameworks used by High Energy Physicists

Pere Mato/CERN, Wim Lavrijsen/LBNL

EuroPython Conference
CERN, 3-5 July 2006

Outline

- Motivation and Introduction
  - Scripting languages
  - Python - C++ interoperation
- Technology Overview
  - Different game-plans
  - Available products
- PyROOT
  - Overview and status
- Example use cases

The Case for Scripting

- Typically, scripting languages are
  - Simple, high-level, dynamically typed
  - Designed for "gluing" existing components
  - Interactive, interpreted
  - Missing steps in write/build/nap/run/debug
- Improved productivity
  - Reduced learning curve (learn by doing)
  - More effective re-use of components
  - Shorter development cycle

Scripting in High Energy Physics

- Scripting has been an essential component in the HEP analysis software for the last decades
  - PAW macros (kumac) in the FORTRAN era
  - C++ interpreter (CINT) in the C++ era
  - Python recently introduced
- Most of the statistical data analysis and final presentation is done with scripts
  - Interactive analysis and rapid prototyping
- Scripts are also used to "configure" complex C++ programs developed and used by the experiments
  - "Simulation" and "Reconstruction" programs

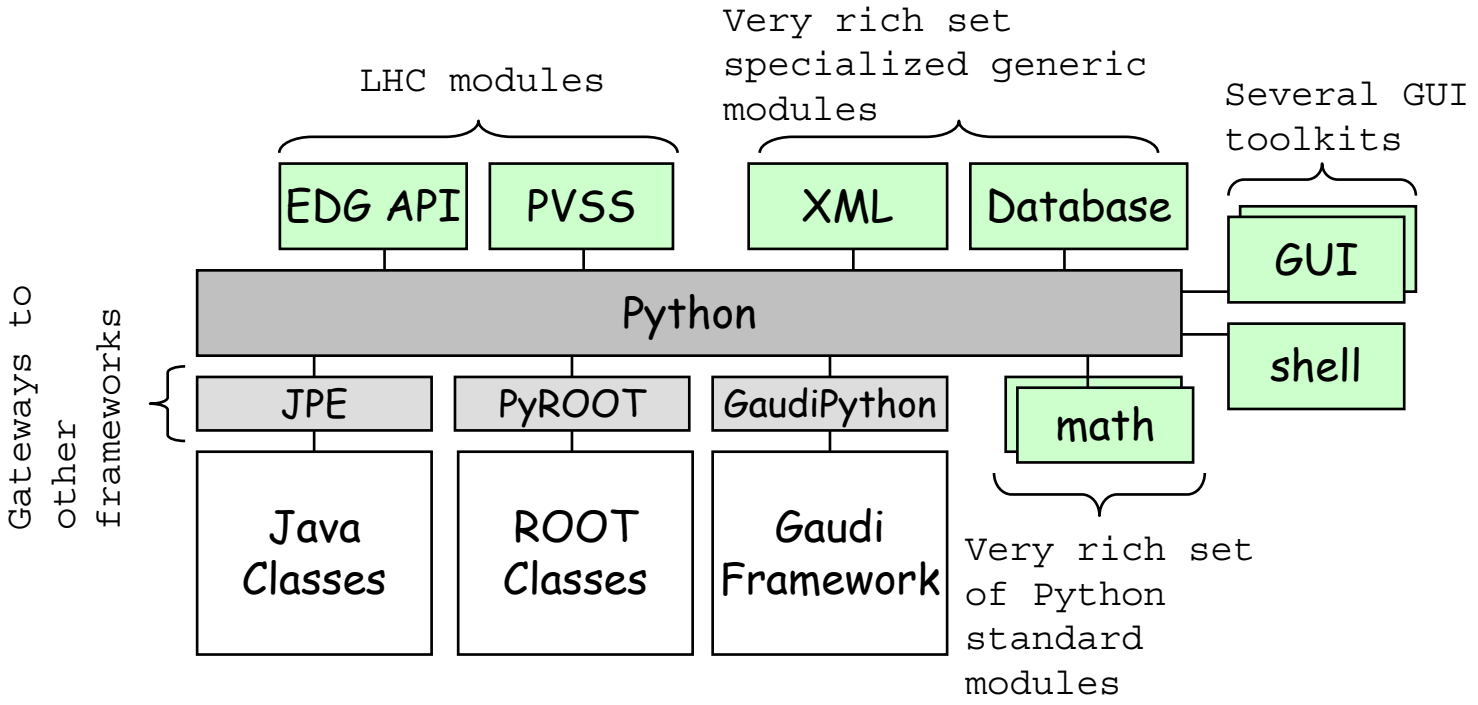
The Case for Python

- Simple, elegant, easy to learn
  - Based on ABC, a teaching language
  - Tutorials available online (www.python.org)
  - Many standard and 3rd party modules
  - 2nd most popular in use, most for bindings
- Used for scientific programming
  - Open source, freely available
  - Extensions for high performance and distributed parallel code (www.scipy.org)

Python <-> C++ Interoperation

- The bulk of code for the new HEP experiments is written in C++
  - Still some portions of FORTRAN with plans to migrate
  - Java and other languages almost non-existent
- Need Python *bindings* to C++ code
  - Hand-written (C-API) or generated
  - Requires taking care of:
    - » Object, parameter conversions
    - » Memory management
    - » C++ function overloading
    - » C++ templates
    - » Inheritance and function callbacks

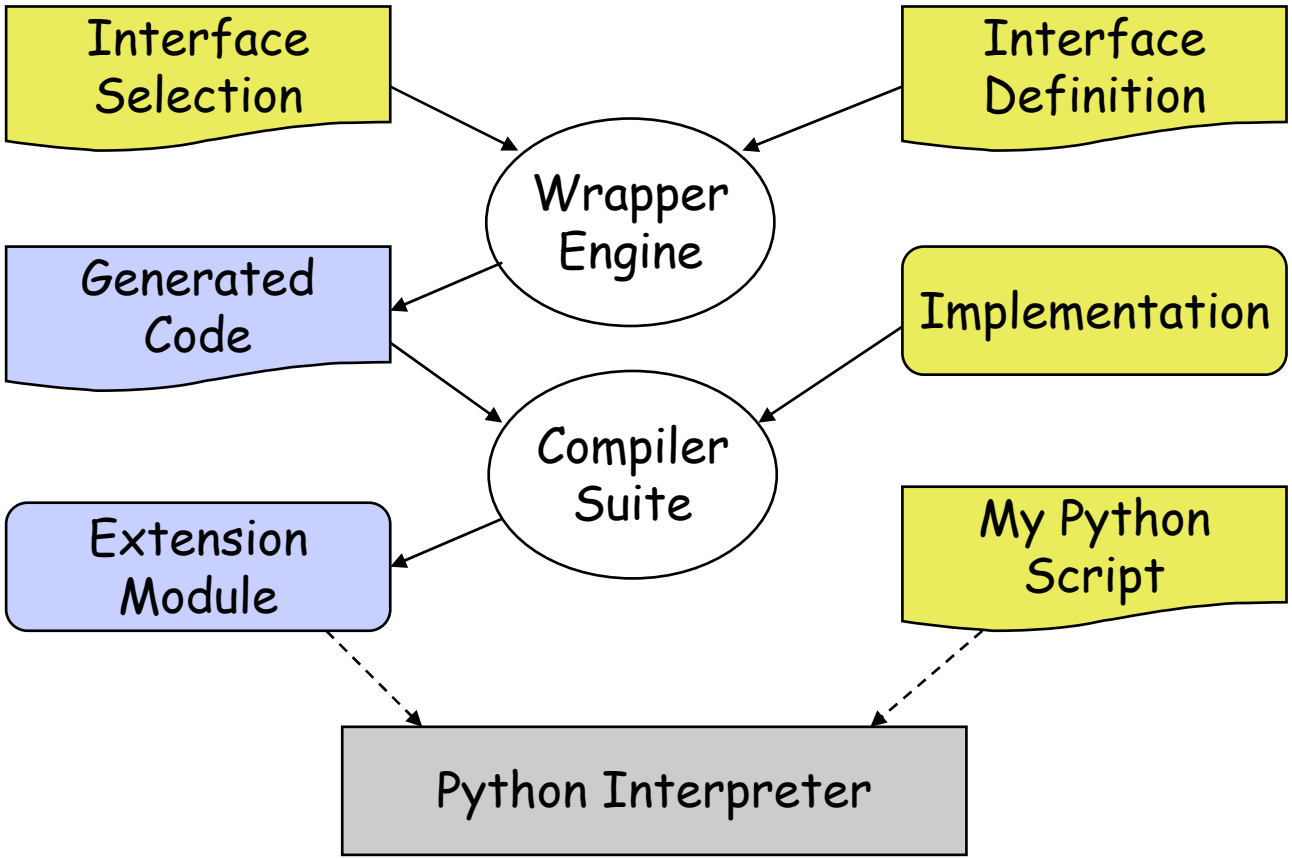
Python as "Glue"



Static Wrapping

- Wrappers created from interface file
  - Several steps required (can be automated)
  - Control creation with a selection file
  - E.g. SWIG, Boost.Python, SIP, etc.
- Wrappers are written out as code
  - Compiled into an extension module
    - » Missing classes replaced by stubs
  - Internal bookkeeping for class sharing
    - » Types are registered to allow conversions
  - Function returns follow bound signature
    - » But allow explicit (dynamic) casts by the user

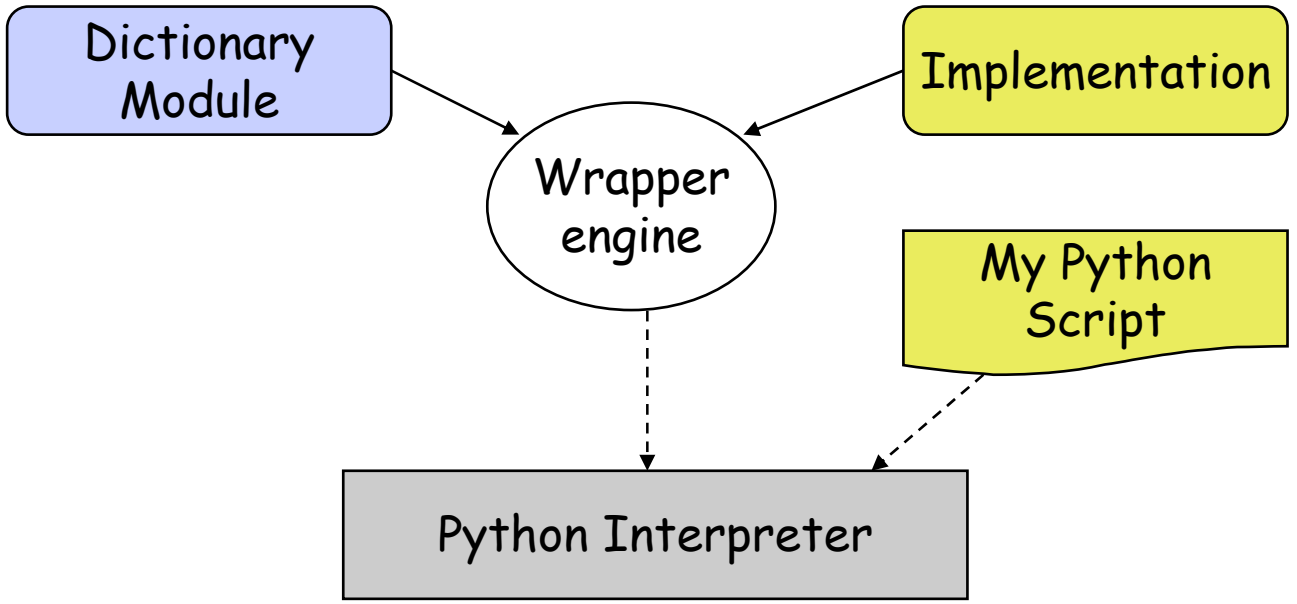
Static Wrapping



Dynamic Wrapping

- Wrappers created from reflection information
  - If dictionary available: minimal user effort
    - » If not, very similar effort as static wrapping
  - No/Little control over creation mechanism
- Python classes, built-up in memory
  - Loaded and created on-demand
    - » Missing classes can be automatically loaded
  - "Bookkeeping" is in Python classes itself
    - » Conversions are derived from reflection information
  - Function returns the concrete dynamic type

Dynamic Wrapping



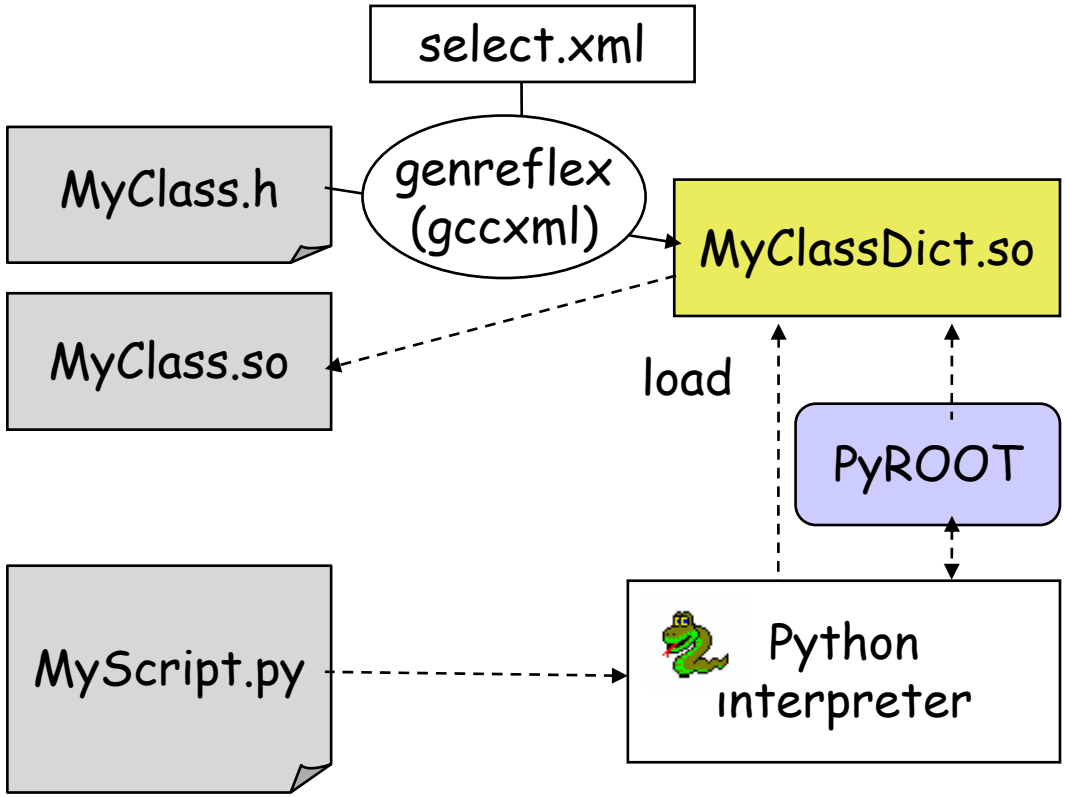
C++ Reflection

- Reflection is the ability of a language to introspect its own structures at *runtime* and interact with them in a generic way
  - C++ provides natively very limited functionality (RTTI)
- Reflection information is used in HEP for several essential functionalities:
  - Object persistency
  - Scripting and interactivity
  - Plug-in management
- "Dictionaries" are the dynamic loadable libraries containing the reflection information

Python bindings based on Dictionaries

- Generic Python bindings based on C++ reflection information (dictionaries) has proven to be a very practical way to get the job done
  - Create the dictionaries (automated by the provided tools)
  - Have direct access to all C++ functionality from Python
  - Get for free interesting "pythonizations" of C++ constructs
- Several incarnations of this approach
  - Using the CINT (C++ interpreter) dictionaries:
    - » PyROOT module in ROOT - Object-Oriented Analysis Framework (http://root.cern.ch)
  - Using new Reflex dictionaries:
    - » PyROOT module + Cintex (gateway between Reflex and CINT dictionaries)

PyROOT: Mode d'emploi



- From class definitions (.h files) a "dictionary" library is produced
  - Description of the class
  - "stub" functions to class methods
- Absolutely non-intrusive
- The PyROOT module does the adaptation between Python objects and C++ objects in a generic way
  - It works for any dictionary

PyROOT: Supported Features

- C++ types
  - Conversion between C++ and Python primitive types
  - Python classes loaded on demand. Templated classes supported.
- C++ namespaces
  - Mapped to python scopes. The "::" separator is replaced by the python "." separator. Global namespace: ROOT
- Class methods
  - Static and non static class methods are supported. Default arguments.
  - Method arguments are passed by value or by reference
  - The return values are converted into Python types and new Python classes are created if required. Dynamic type returned if possible.
  - Method overloading works by dispatching sequentially to the available methods with the same name until a match with the provided arguments (match cached for subsequent calls)

PyROOT: Supported Features

- Support for most of the C++ features
  - Public data members are accessible as Python properties
  - Enums, typedefs, global functions, global variables
  - Doc strings of signatures
  - C++ reference function arguments (for float, int/long, and objects)
  - Settable operator[](int), smart pointers, etc.
- Emulation of Python containers
  - Container C++ classes (std::vector, std::list, std::map like) are given the behavior of the Python collections to be used in iterations and slicing operations.
- Operator overloading
  - Standard C++ operators are mapped to the corresponding Python overloading operators
- Memory management
  - Several schemas supported. Fine control to the user.

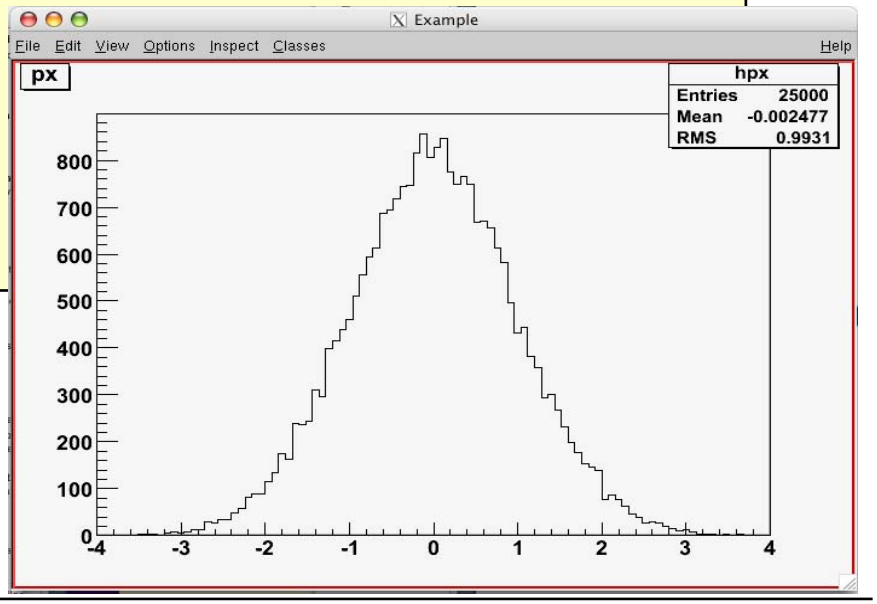
Mapping C++ to Python

C++ construct	Python equivalent	Example
Primitive types: bool, char, short, int, long, unsigned log, float, double, char*, std::string	Converted to: bool, int, int, int, long, long, float, float, string, string	
Namespace separation (::)	scope separation (.)	`Gaudi.Time`
Global namespace (::)	ROOT, gaudimodule.gbl	`gbl.Gaudi.Time`
Template class <…>	<…> replaced by (…)	`std.vector('double')`
NULL pointer	None	`func(None, None)`
Dynamic cast	No need. Always dynamic type	`tk = event('track')`
Complex object value, pointers and references	Always object references	
Iterators	Iteration protocol	`for o in vector: ...`

PyROOT: Using ROOT in Python

- Example on how to create an histogram and display its contents

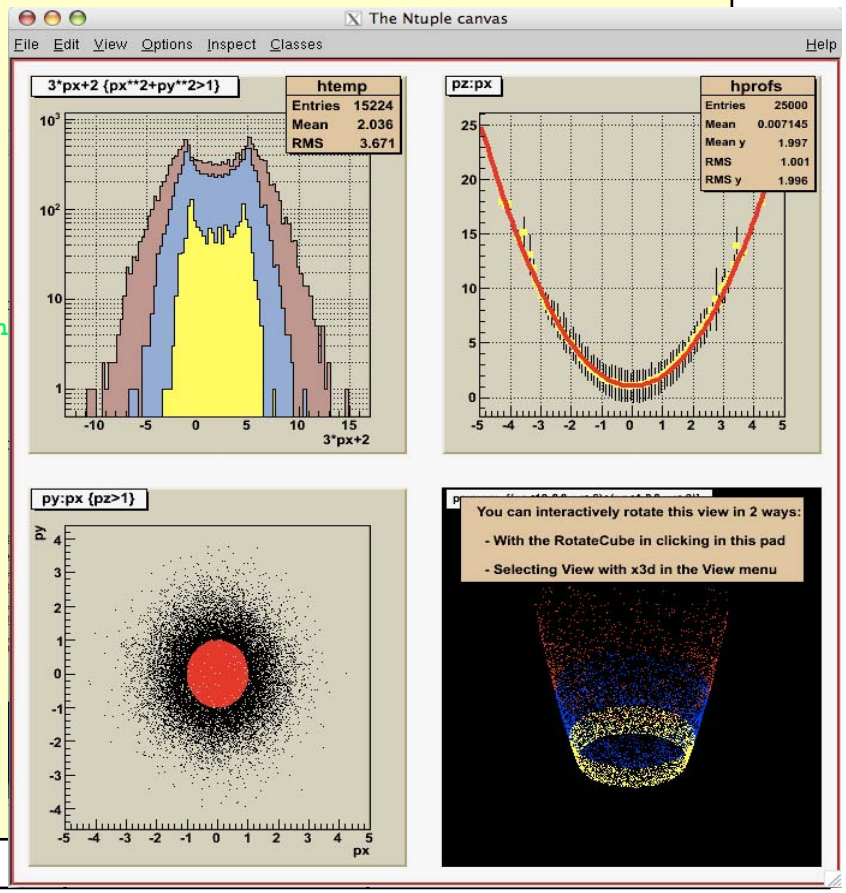
```
>>> from ROOT import gRandom, TCanvas, TH1F
>>> c1 = TCanvas('c1','Example',200,10,700,500)
>>> hpx = TH1F('hpx','px',100,-4,4)
>>> for i in xrange(25000):
...   px = gRandom.Gaus()
...   hpx.Fill(px)
...
>>> hpx.Draw()
```



PyROOT: ntuple1

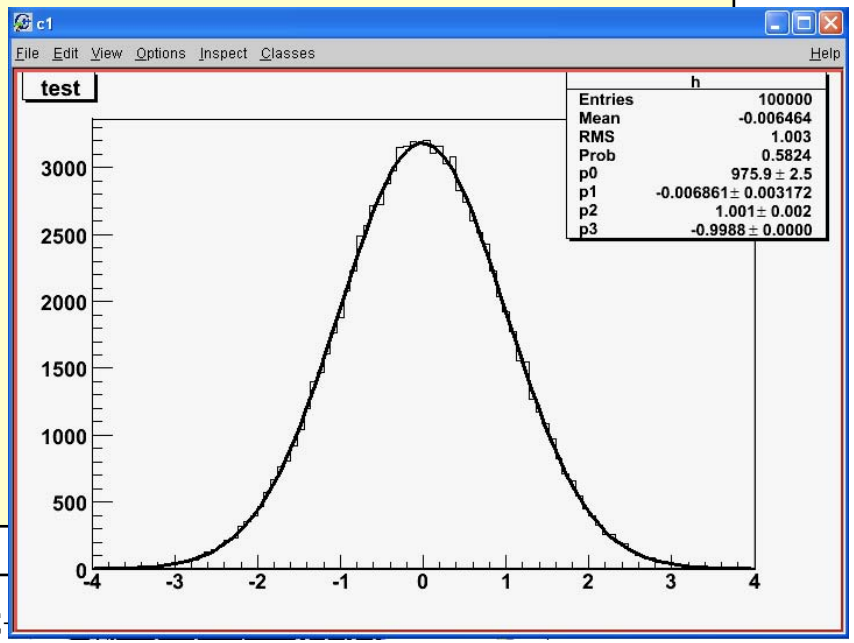
ntuple1.py

```
from ROOT import TCanvas, TPad, TFile, TPaveText
from ROOT import gBenchmark, gStyle, gROOT

c1 = TCanvas('c1','The Ntuple canvas',200,10,700,780)
f1 = TFile('hsimple.root')

# Inside this canvas, we create 4 pads
pad1 = TPad('pad1','This is pad1',0.02,0.52,0.48,0.98,21)
pad2 = TPad('pad2','This is pad2',0.52,0.52,0.98,0.98,21)
pad3 = TPad('pad3','This is pad3',0.02,0.02,0.48,0.48,21)
pad4 = TPad('pad4','This is pad4',0.52,0.02,0.98,0.48,1)
pad1.Draw()
pad2.Draw()
pad3.Draw()
pad4.Draw()
# Display a function of one ntuple column imposing a condition on
pad1.cd()
pad1.SetGrid()
pad1.SetLogy()
pad1.GetFrame().SetFillColor(15)
ntuple = gROOT.FindObject('ntuple')
ntuple.SetLineColor(1)
ntuple.SetFillStyle(1001)
ntuple.SetFillColor(45)
ntuple.Draw('3*px+2','px**2+py**2>1')
ntuple.SetFillColor(38)
ntuple.Draw('2*px+2','pz>2','same')
ntuple.SetFillColor(5)
ntuple.Draw('1.3*px+2','(px^2+py^2>4) && py>0','same')
...
```



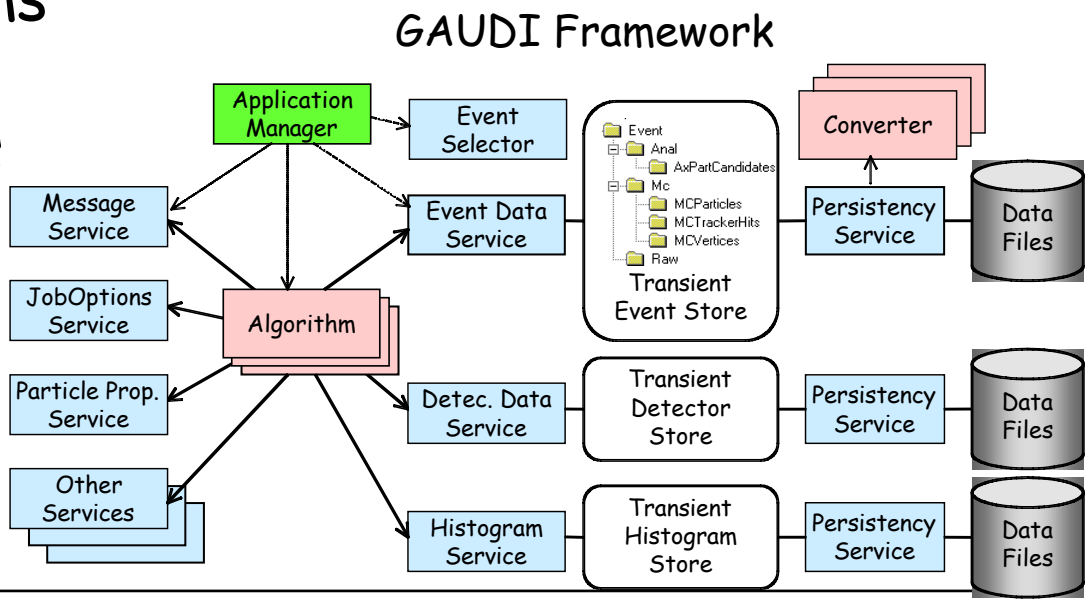
PyROOT: fitting

```
import ROOT, math
def pygaus( x, par ):
    ddx = 0.01
    if (par[2] != 0.0):
        arg1 = (x[0]-par[1])/par[2]
        scale1 = (ddx*0.39894228)/par[2]
        h1 = par[0]/(1+par[3])

        gauss = h1*scale1*math.exp(-0.5*arg1*arg1)
    else:
        gauss = 0.
    return gauss

if __name__ == '__main__':
   f = ROOT.TF1( 'pygaus', pygaus, -4, 4, 4 )
   f.SetParameters( 600, 0.43, 0.35, 600 )

   h = ROOT.TH1F( "h"," test", 100, -4, 4 )
   h.FillRandom( "gaus", 100000 )
   h.Fit( f )
   h.Draw()
```



Experiment Data Processing Frameworks

- HEP experiments have developed software frameworks for their event data processing applications (simulation, reconstruction, analysis, etc.)
  - E.g. the GAUDI framework used by LHCb and ATLAS
- These frameworks help the physicists to focus on developing their algorithms and provide a palette of common functionalities
  - Python scripting is one of the provided functionalities
  - Other examples are: I/O, GUI, 3D Graphics, RDBMS,etc.



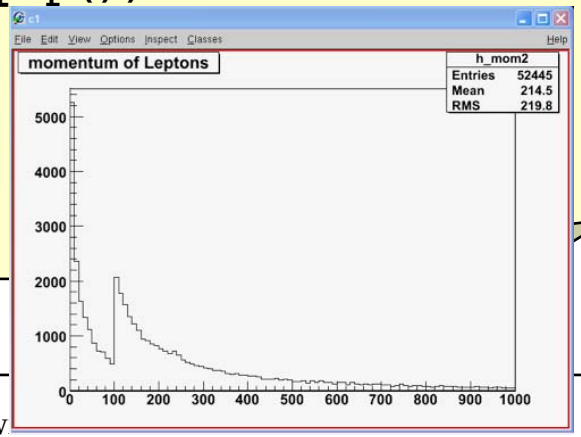
GaudiPython

- Enabling the interaction of GAUDI components from the Python interpreter
  - Configuration, interactivity, developing analysis, gluing with other tools, etc.
- Starting from Gaudi v18r0, GaudiPython has been re-implemented using PyROOT
  - Generated dictionaries for most common GAUDI "Interfaces" and "Base classes" (~80 classes)
  - Not need to generate dictionaries for all classes (in particular the implementations)
- Generation of dictionaries for user classes fully automated
  - "Event" classes (also needed for object persistency), "Detector Description" classes, "Analysis Tool" interfaces, etc.

GaudiPython: simple example

```
>>> import gaudimodule
>>> gaudi = gaudimodule.AppMgr( outputlevel = 3,
                                joboptions  = 'myjob.opts')
>>> evt = gaudi.evtsvc()
>>> gaudi.run(1)
>>> evt.dump()

>>> h1 = his.book('h1','momentum of Leptons', 100, 0., 1000.)
>>> class MyAlg(gaudimodule.PyAlgorithm):
>>>   def execute(self):
>>>     particles = evt['MC/Particles']
>>>     for p in particles :
>>>       if p.isLepton() : h1.Fill(p.p())

>>> gaudi.addAlgorithm(MyAlg())
>>> gaudi.run(100)
>>> h1.Draw()
```

```
>>> evt.dump()
/Event
/Event/Gen
/Event/MC
/Event/MC/Header
/Event/MC/TestParticles
/Event/MC/TestVertices
/Event/MC/Velo
/Event/MC/PuVeto
/Event/MC/TT
/Event/MC/IT
/Event/MC/OT
/Event/MC/Muon
/Event/MC/Spd
/Event/MC/Prs
/Event/MC/Ecal
/Event/MC/Hcal
/Event/MC/Rich
/Event/MC/DigiHeader
/Event/MC/TrackInfo
/Event/MC/Particles
/Event/MC/Vertices
/Event/DAQ
/Event/Prev
/Event/PrevPrev
/Event/Next
/Event/Link
/Event/pSim
/Event/pSim/MCParticles
/Event/pSim/MCVertices
>>>
```



High-level Analysis built on GaudiPython

- Experiments are developing Python modules (e.g. Bender in LHCb) to facilitate final physics analysis to non-computer literates
  - Example: nested loops over particles to reconstruct specific particle decay channels

```
from bendermodule import *
class Dstar(Algo):
  def analyse( self ) :
    self.select ( tag='K-', cuts=('K-' ==ID)&(PT>1*GeV) )
    self.select ( tag='pi+',cuts=('pi+'==ID)&(P >3*GeV) )
    dmass = ABSDM("D0") < 30 * MeV
    for D0 in self.loop ( formula='K- pi+' , pid='D0' ) :
      if ( VCHI2(D0) < 4 ) &  dmass( D0 ) : D0.save('D0')
      tup = self.nTuple ( title = "D*+ N-Tuple " )
      for Dst in self.loop ( formula='D0 pi+', pid='D*(2010)+' ) :
        dm = M(Dst)-M1(Dst)
        h1 = self.plot( title = "Delta mass for D*+",
                        value = dm , low=130 , high=170 )
        tup.column( name = 'M'   , value = M(Dst)  / GeV )
        tup.column( name = 'DM'  , value = dm      / GeV )
        tup.column( name = 'p'   , value = P (Dst) / GeV )
        tup.column( name = 'pt'  , value = PT(Dst) / GeV )
        tup.write ()
    return SUCCESS
```

$D^{*+} \rightarrow K^- \pi^+ \pi^+$ analysis ▪

Conclusions

- Physicists' frameworks and toolkits beefed up with Python
  - Adds advantages of scripting languages
  - Very easy connection to many existing libraries
  - Learn by doing fostered (docs are hardly read)
  - Very high-level physics analysis possible
- Developed tools to provide ease of use
  - PyROOT: generic, dynamic
  - Automatic binding for user classes
  - Provide binding for standard physics libraries non-intrusively