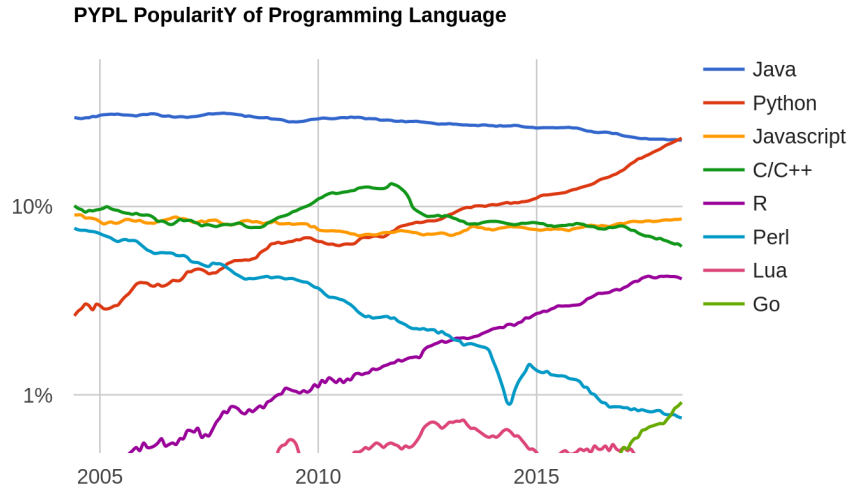
The Python Scientific Software Ecosystem: Past, Present and Future

Jim Pivarski

Princeton University – DIANA-HEP

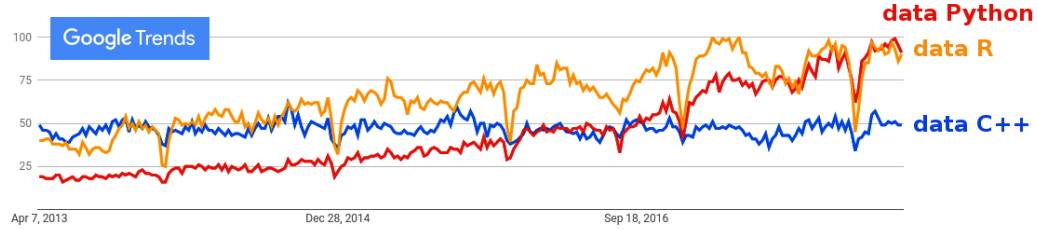
July 7, 2018

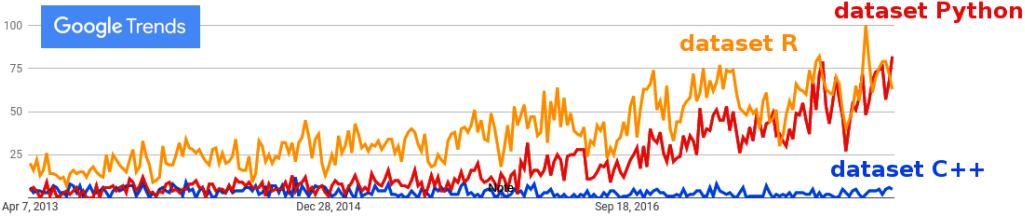
Why we're here



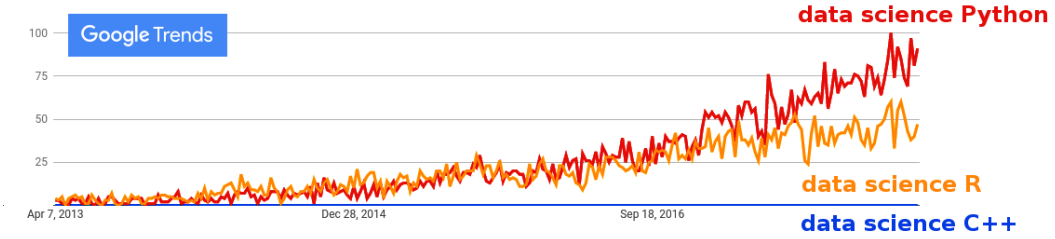
http://pypl.github.io/PYPL.html

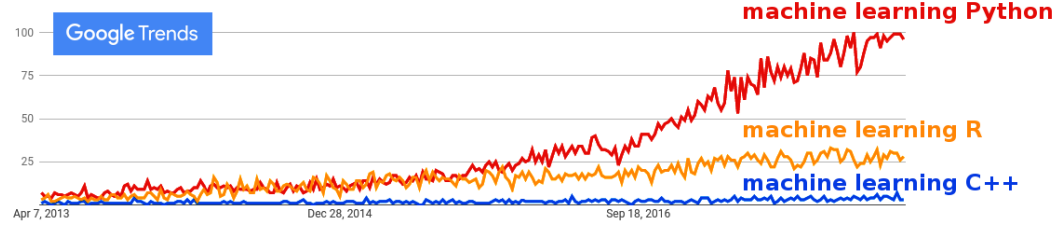
Why we're here





Why we're here

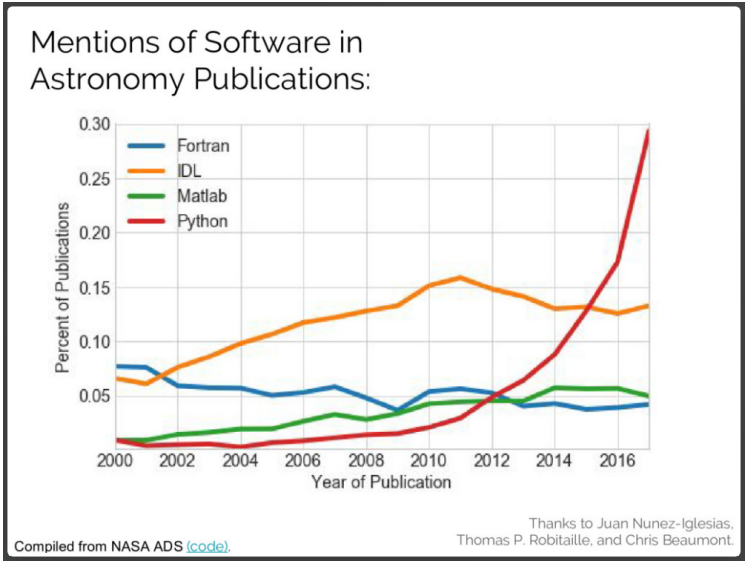




Why we're here

All of the machine learning libraries I could find either have a Python interface or are primarily/exclusively Python.

Why we're here



Python in HEP?

Python in HEP?

- Collaboration frameworks like Athena and CMSSW are configured or driven by Python.

Python in HEP?

- Collaboration frameworks like Athena and CMSSW are configured or driven by Python.
- Today, it is common for physicists to do their analyses in Python/PyROOT.
  (Half Python, half C++? Anyway, a lot more than in LHC Run I.)

Python in HEP?

- Collaboration frameworks like Athena and CMSSW are configured or driven by Python.
- Today, it is common for physicists to do their analyses in Python/PyROOT.
  (Half Python, half C++? Anyway, a lot more than in LHC Run I.)
- Python is the most natural bridge to machine learning and other statistical software written outside of HEP.

I was an early adopter (thesis workflow from 2006)

Earlier steps in Mathematica, Perl, GnuPlot, Emacs Lisp, PAW, and ROOT

SEAL-MINUIT

C++

```
#include "Minuit/MnMinos.h"
#include "Minuit/FunctionMinimum.h"
#include "Minuit/FCNBase.h"
#include "Minuit/MnFunctionCross.h"
#include "Minuit/MnCross.h"
#include "Minuit/MinosError.h"

std::pair<double,double> MnMinos::operator()(unsigned int par, unsigned
int maxcalls) const {
  MinosError mnerr = minos(par, maxcalls);
  return mnerr();
}

double MnMinos::lower(unsigned int par, unsigned int maxcalls) const {

  MnUserParameterState upar = theMinimum.userState();
  double err = theMinimum.userState().error(par);

  MnCross aopt = loval(par, maxcalls);

  double lower = aopt.isValid() ? -1.*err*(1.+ aopt.value()) :
(aopt.atLimit() ? upar.parameter(par).lowerLimit() : upar.value(par));

  return lower;
}
```

PyMinuit

C API

```
// g++ minuit.cpp -I/nfs/cleo3/Offline/rel/current/other_sources/pyth
include/python2.4/ -I/cdat/daf9/mccann/software/src/minuit/Minuit-1_5_2 /
cdat/daf9/mccann/software/src/minuit/Minuit-1_5_2/src/*.o -shared -o
_minuit.so
// g++ minuit.cpp -I/usr/include/python2.3 -I/root/src/Minuit-1_5_2/ /
root/src/Minuit-1_5_2/src/*.o -shared -o _minuit.so

#include <Python.h>
#include "Minuit/MnUserParameters.h"
#include "Minuit/MnMigrad.h"
#include "Minuit/MnMinimize.h"
...
PyObject* dominos(PyObject *self, PyObject *args)
{
   // parameter list:
   PyObject *p_fcn;                    // objective function p_fcn
   int npar;                           // number of parameters in p_fcn
   double up;                          // 1 for chi^2, 0.5 for loglike
   PyObject *p_min;                    // the minimum you previously found

   if (!PyArg_ParseTuple(args, "OidOiiOOiOO", &p_fcn, &npar, &up, &p_min,
&maxcalls, &strategy, &p_dolower, &p_doupper, &parnum, &p_grad,
&p_checkgrad)) {
      PyErr_SetString(PyExc_TypeError, "calling format must be: FCN(f),
npar(i), up(d), minimum(FunctionMinimum), maxcalls(i or 0), strategy(i),
dolower(b), doupper(b), parnum(i), gradient(f or None), checkgrad(b or
None)");
      return NULL;
   }
```

Fit function

FORTRAN

```
 DOUBLE PRECISION FUNCTION GBWKF(RM,GAM,WSPREAD,HC)
       IMPLICIT NONE
c  implicit real (A-H,O-Z), integer (I-N)

C   Observed shape of resonance peak, starting with Breit-Wigner
C   Normalized per unit W, MeV^-1
C   F_KF(x,s) convoluted with unit-area Breit-Wigner, and with beam
resolution
C    unit-area Gaussian
C   If GAM<1 keV, the Breit-Wigner is replaced with a delta function
C   BW(w) = (GAM/2*pi)/[(w-M)^2+GAM^2/4] (unit integral over w)
C   Gaussian(w) = exp{-.5*[(w-M)^2/WSPREAD^2]}/(2*pi*WSPREAD) (unit int...)

        INTEGER   MKBIN, MTBIN, JTBIN
        REAL*8    ROOT2, PI, RTWOPI, RTWOBPI, PMAX, TINY

 REAL*8    FKFT(MKBIN)               ! F_KF(x,s) (Kuraev-Fadin eq.28)
 REAL*8    HA(MKBIN)                  ! WA-RM of centroid WA bin
 REAL*8    DWA(MKBIN)                 ! bin width in WA-RM
 REAL*8    HT(0:MTBIN)
 REAL*8    DWT(0:MTBIN)               ! WT-RM bin center
 REAL*8    BWKF(-MTBIN:MTBIN)         ! result of BW-F_KF convolution
...
 DO 11 ITBIN=-MTBIN,MTBIN           ! sum over WT's
   H=HT(IABS(ITBIN))                ! WT-RM
   IF(ITBIN.LT.0)H=-H
   SIGS=(H-HC)/(ROOT2*WSPREAD)      ! sqrt of exponent
   IF((ITBIN.EQ.MTBIN).AND.(SIGS.LT.2.))GO TO 12 ! running out of bins?
   IF(SIGS.GT.3.)GO TO 13           ! Gaussian getting negligible?
   P=SIGS**2                        ! exponent
   IF(P.LT.PMAX)THEN
     IF(DWT(IABS(ITBIN)).GT..2*WSPREAD)GO TO 12  ! binning too coarse?
            SUM=SUM+EXP(-P)*BWKF(ITBIN)*DWT(IABS(ITBIN))
   ENDIF
 11 CONTINUE
 13 GBWKF=SUM/(RTWOPI*WSPREAD)         ! convolution
C         1/(sqrt(2*pi)*WSPREAD) normalizes Gaussian
 RETURN
 END
```

Fitting script

Python

```
# get_runs has been given a thorough look-over: it is correct (7 Oct
# 2005) (get_runs contains all corrections, from numbers of events to
# real, live cross-section.)

from minuit import *
execfile("/home/mccann/antithesis/utilities.py")
import gbwkf
import gbwkftau
...
def dofitgauss(h):
  def gauss(m, s, x): return exp(-(x-m)**2/2./s**2)/sqrt(2.*pi)/s
  def fitgauss(m,s):
    c = 0.
    for x in h.data:
      c += -log(gauss(m, s, x))
    return c
  m = Minuit(fitgauss, start=[0., 1.], up=0.5)
  m.migrad()
  m.minos([0,1])
  err0 = (m.minos_errors[0][1] - m.minos_errors[0][0])/2.
  err1 = (m.minos_errors[1][1] - m.minos_errors[1][0])/2.
  return m.values[0], err0, m.values[1], err1, lambda x:
0.1*extraarea*len(h.data)*gauss(m.values[0], m.values[1], x)
```

(pickle)

GNU plotutils

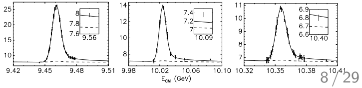
biggles **Numeric**

Plotting script

Python

```
from math import *
import biggles, Numeric, cPickle as pickle
import gbwkf
import gbwkftau

allthat = pickle.load(file("/home/mccann/antithesis/novemberdata.p"))
u1runs = allthat["u1runs"]
u2runs = allthat["u2runs"]
u3runs = allthat["u3runs"]
...
q = biggles.FramedPlot()
adddata(q, [None], u2data["high"], 0.)
addfunc(q, thefunc, 10080., 10090.)
addfunc(q, thefunc_bkgnd, 10080., 10090., linetype="dashed")
```



Which got me involved in open source (PyMinuit is now "iminuit")

What's new? | Help | Directory | Sign in

Google Code

pyminuit

Minuit numerical function minimization in Python

Project Home Downloads Wiki Issues Source

Search Projects | Search the Web

PyMinuit

Minuit numerical function minimization in Python

Minuit

Minuit has been the standard package for minimizing general N-dimensional functions in high-energy physics since its introduction in 1972. It features a robust set of algorithms for optimizing the search, correcting mistakes, and measuring non-linear error bounds. It is the minimization engine used behind-the-scenes in most high-energy physics curve fitting applications.

New: more robust installation instructions!

Python interface

PyMinuit is an extension module for Python that passes low-level Minuit functionality to Python functions. Interaction and data exploration is more user-friendly, in the sense that the user is protected from segmentation faults and index errors, parameters are referenced by their names, even in correlation matrices, and Python exceptions can be passed from the objective function during the minimization process. This extension module also makes it easier to calculate Minos errors and contour curves at an arbitrary number of sigmas from the minimum, and features a new N-dimensional scanning utility.

License: GNU General Public License v2

Labels: python, minuit, optimization, computation, HEP, math, fitting, physics

Featured Downloads: Show all

- Minuit-1_7_9.tar.gz
- pyminuit-1.0.2.tgz

Featured Wiki Pages: Show all

- FunctionReference
- GettingStartedGuide
- HowToInstall

Links:

- Official Minuit homepage at CERN
- Minuit documentation through the ages

Project owners: Join project

jpivarski

But to be honest, Python isn't my idea of an ideal language.

But to be honest, Python isn't my idea of an ideal language.

I wish the scientific and data analysis community had adopted something more functional and statically typed. In particular, I wish there wasn't a dichotomy between statements and expressions. And `True == 1` is evil.

But to be honest, Python isn't my idea of an ideal language.

I wish the scientific and data analysis community had adopted something more functional and statically typed. In particular, I wish there wasn't a dichotomy between statements and expressions. And `True == 1` is evil.

HOWEVER, a heavy dash of consensus is worth a smattering of language features!

Why Python?

Did Python just arrive at the right time?

Or is it a better language for the job?

Why Python?

Did Python just arrive at the right time?

- Ruby, Lua numerical stacks were not ready before Python already had a foothold.
- Python was one of the first glue languages of the Linux/open source era.

Or is it a better language for the job?

Why Python?

Did Python just arrive at the right time?

- Ruby, Lua numerical stacks were not ready before Python already had a foothold.
- Python was one of the first glue languages of the Linux/open source era.

Or is it a better language for the job?

- Perl $\rightarrow$ Python
- "Tcl War"
- R $\rightarrow$ Python

Why Python?

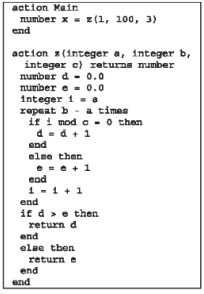
Did Python just arrive at the right time?

- Ruby, Lua numerical stacks were not ready before Python already had a foothold.
- Python was one of the first glue languages of the Linux/open source era.

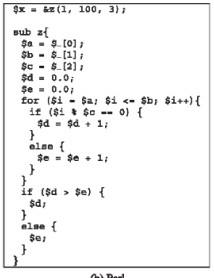
Or is it a better language for the job?

- Perl $\rightarrow$ Python
- "Tcl War"
- R $\rightarrow$ Python
- *An Empirical Investigation into Programming Language Syntax,* Andreas Stefik, Susanna Siebert.

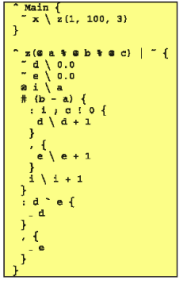
An Empirical Investigation into Programming Language Syntax



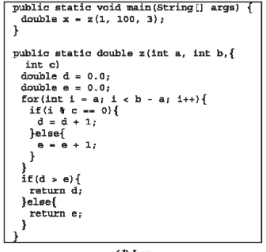
(a) Quorum



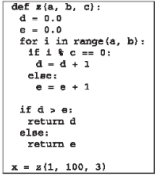
(b) Perl



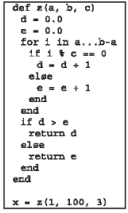
(c) Randomo



(d) Java



(e) Python



(f) Ruby

Abstract

Recent studies in the literature have shown that syntax remains a significant barrier to novice computer science students in the field. While this syntax barrier is known to exist, whether and how it varies across programming languages has not been carefully investigated. For this article, we conducted four empirical studies on programming language syntax as part of a larger analysis into the, so called, programming language wars. We first present two surveys conducted with students on the intuitiveness of syntax, which we used to garner formative clues on what words and symbols might be easy for novices to understand. We followed up with two studies on the accuracy rates of novices using a total of six programming languages: Ruby, Java, Perl, Python, Randomo, and Quorum. Randomo was designed by randomly choosing some keywords from the ASCII table (a metaphorical placebo). To our surprise, we found that languages using a more traditional C-style syntax (both Perl and Java) did not afford accuracy rates significantly higher than a language with randomly generated keywords, but that languages which deviate (Quorum, Python, and Ruby) did. These results, including the specifics of syntax that are particularly problematic for novices, may help teachers of introductory programming courses in choosing appropriate first languages and in helping students to overcome the challenges they face with syntax. (Less)

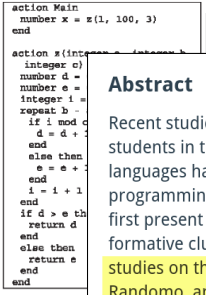
DOI: 10.1145/2534973

Table XXIII.

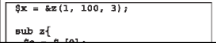
A table showing the cross-task average and standard deviation for each language and the Tukey HSD comparisons. Statistically significant differences are in bold.

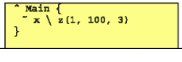
			Tukey HSD				
	Mean	**Std. Dev.**	**Python**	**Quorum**	**Perl**	**Java**	**Randomo**
Ruby	.558	.243	0.994	0.940	0.165	**0.044**	**0.003**
Python	.528	.246		0.999	0.412	0.141	**0.011**
Quorum	.508	.232			0.655	0.292	**0.033**
Perl	.429	.189				0.987	0.573
Java	.396	.194					0.922
Randomo	.344	.207					

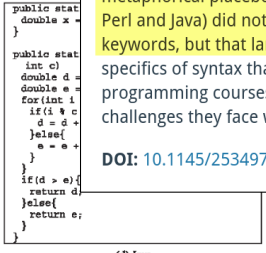
An Empirical Investigation into Programming Language Syntax



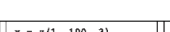
(a)







(d) Java



(e) Python

(f) Ruby

Abstract

Recent studies in the literature have shown that syntax remains a significant barrier to novice computer science students in the field. While this syntax barrier is known to exist, whether and how it varies across programming languages has not been carefully investigated. For this article, we conducted four empirical studies on programming language syntax as part of a larger analysis into the, so called, programming language wars. We first present two surveys conducted with students on the intuitiveness of syntax, which we used to garner formative clues on what words and symbols might be easy for novices to understand. We followed up with two studies on the accuracy rates of novices using a total of six programming languages: Ruby, Java, Perl, Python, Randomo, and Quorum. Randomo was designed by randomly choosing some keywords from the ASCII table (a metaphorical placebo). To our surprise, we found that languages using a more traditional C-style syntax (both Perl and Java) did not afford accuracy rates significantly higher than a language with randomly generated keywords, but that languages which deviate (Quorum, Python, and Ruby) did. These results, including the specifics of syntax that are particularly problematic for novices, may help teachers of introductory programming courses in choosing appropriate first languages and in helping students to overcome the challenges they face with syntax. (Less)

DOI: 10.1145/2534973

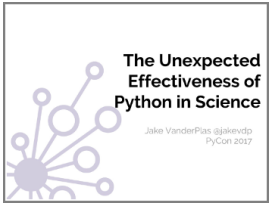
ch language and
bold.

						Java	Randomo
						0.044	**0.003**
						0.141	**0.011**
						0.292	**0.033**
						0.987	0.573
Java	.396	.194					0.922
Randomo	.344	.207					

My conclusion (debatable)

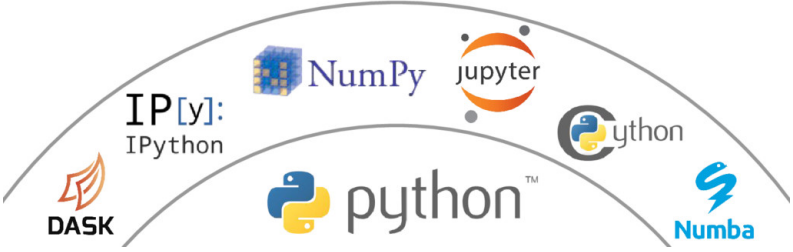
Python was good enough and first.

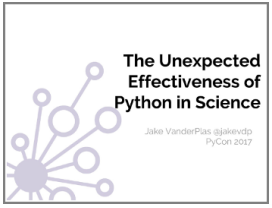
Python's Scientific Stack



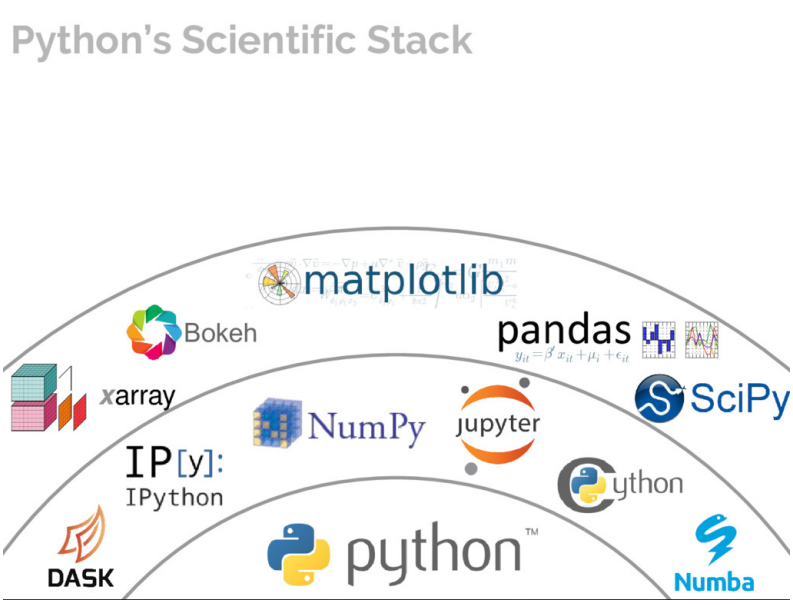
More significant: what has grown around Python

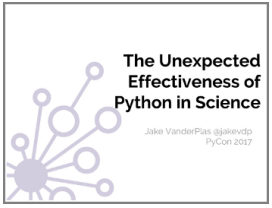
Python's Scientific Stack



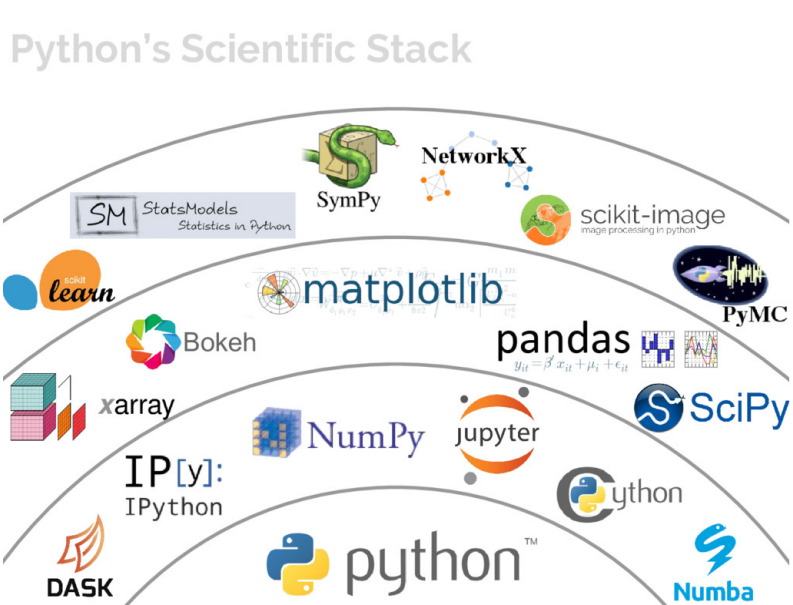


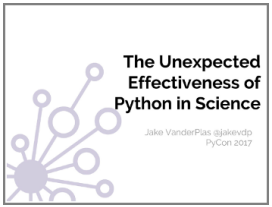
More significant: what has grown around Python



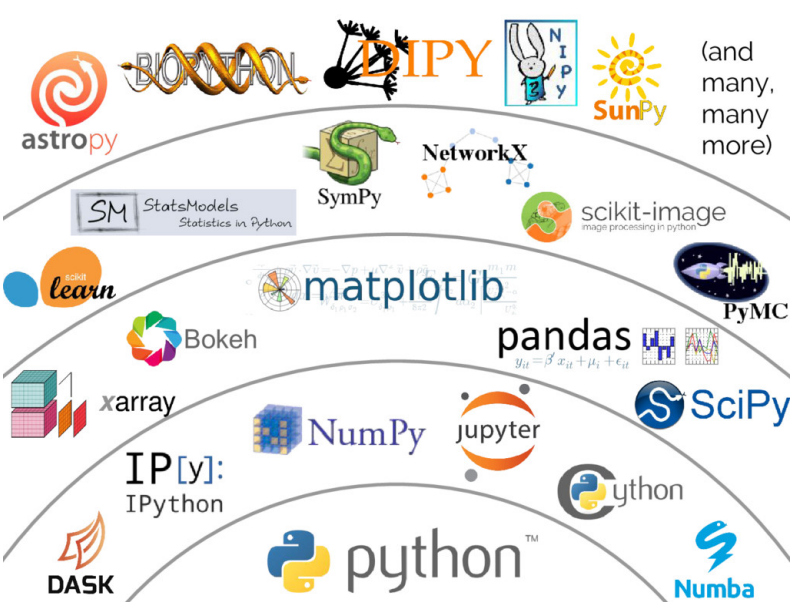


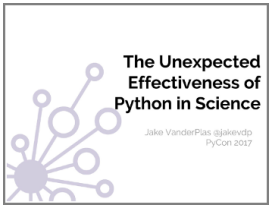
More significant: what has grown around Python





More significant: what has grown around Python





The key to ecosystem development was a common array library

1994 **Python** 1.0 released.

1995 First array package: **Numeric** (a.k.a. Numerical, Numerical Python, NumPy).

2001 Diverse scientific codebases merged into **SciPy**.

2003 **Matplotlib**

2003 Numeric was limited; **numarray** appeared as a competitor with more features (memory-mapped files, alignment, record arrays).

2005 Two packages were incompatible; could not integrate numarray-based code into SciPy. Travis Oliphant merged the codebases as **Numpy**.

2008 **Pandas**

2010 **Scikit-Learn**

2011 **AstroPy**

2012 **Anaconda**

2014 **Jupyter**

2015 **Keras**

The key to ecosystem development was a common array library

1994 **Python** 1.0 released.

1995 First array package: **Numeric** (a.k.a. Numerical, Numerical Python, NumPy).

2001 Diverse scientific codebases merged into **SciPy**.

2003 **Matplotlib**

2003 Numeric was limited; **numarray** appeared as a competitor with more features (memory-mapped files, alignment, record arrays).

2005 Two packages were incompatible; could not integrate numarray-based code into SciPy. Travis Oliphant merged the codebases as **Numpy**.

2008 **Pandas**

2010 **Scikit-Learn**

2011 **AstroPy**

2012 **Anaconda**

2014 **Jupyter**

2015 **Keras**

The scientific Python ecosystem could have failed before it started if the Numeric-numarray split hadn't been resolved!

Numpy is high-level, array-at-a-time math


```
>>> import numpy
>>> a = numpy.arange(12)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> a.shape = (3, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a.sum(axis=0)
array([12, 15, 18, 21])
>>> a.min(axis=1)
array([0, 4, 8])
>>> a**2
array([[  0,   1,   4,   9],
       [ 16,  25,  36,  49],
       [ 64,  81, 100, 121]])
>>> numpy.sqrt(a)
array([[0.        , 1.        , 1.41421356, 1.73205081],
       [2.        , 2.23606798, 2.44948974, 2.64575131],
       [2.82842712, 3.        , 3.16227766, 3.31662479]])
```

Numpy is also a low-level way to poke raw bytes


```
>>> import numpy
>>> hello = b"Hello, world!"

>>> # Python strings are immutable
>>> hello[4:8] = "????"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment

>>> # any buffer may be cast as an array
>>> a = numpy.frombuffer(hello, dtype=numpy.uint8)
>>> a
array([ 72, 101, 108, 108, 111,  44,  32, 119, 111, 114, 108, 100,  33],
      dtype=uint8)

>>> # and changed (with possibly disastrous consequences)
>>> a.flags.writeable = True
>>> a[4:8] = [69, 86, 73, 76]
>>> hello
b'HellEVILorld!'
```

Numpy is also a low-level way to poke raw bytes


```
>>> import ctypes

>>> # any *pointer* may be cast as an array
>>> ptr = ctypes.cast(id(hello), ctypes.POINTER(ctypes.c_uint8))
>>> ptr.__array_interface__ = {
...     "version": 3,
...     "typestr": numpy.ctypeslib._dtype(type(ptr.contents)).str,
...     "data": (ctypes.addressof(ptr.contents), False),
...     "shape": (100,)     # how many bytes do you want to access?
... }
>>> b = numpy.array(ptr, copy=False)
>>> b
array([  2,   0,   0,   0,   0,   0,   0,   0, 224, 136, 151,   0,   0,
         0,   0,   0,  13,   0,   0,   0,   0,   0,   0,   0,  47,  49,
       110, 120,  37, 235,  98,  94,  72, 101, 108, 108,  69,  86,  73,
        76, 111, 114, 108, 100,  33,   0,   0,   0,   1,   0,   0,   0,
         0,   0,   0,   0, 224, 136, 151,   0,   0,   0,   0,   0,  12,
         0,   0,   0,   0,   0,   0,   0, 255, 255, 255, 255, 255, 255,
       255, 255,   0,   1,  18,   1,   3,   1,  22,   1,  13,   1,   5,
         1,   0, 105,   0,   0,   1,   0,   0,   0], dtype=uint8)
>>> "".join(map(chr, b[32:45]))
'HellEVILorld!'
```

The Numpythonic mindset

Although you can write Python `for` loops over Numpy arrays, you don't reap the benefit unless you express your calculation in Numpy ufuncs (universal functions).

```
pz = numpy.empty(len(pt))
for i in range(len(pt)):
    pz[i] = pt[i]*numpy.sinh(eta[i])
```

$\mathcal{O}(N)$ Python bytecode instructions, type-checks, interpreter locks.

VS

```
pz = pt * numpy.sinh(eta)
```

$\mathcal{O}(1)$ Python bytecode instructions, type-checks, interpreter locks.

$\mathcal{O}(N)$ statically typed, probably vectorized native bytecode operations on contiguous memory.

The Numpythonic mindset

Although you can write Python `for` loops over Numpy arrays, you don't reap the benefit unless you express your calculation in Numpy ufuncs (universal functions).

```
pz = numpy.empty(len(pt))
for i in range(len(pt)):
    pz[i] = pt[i]*numpy.sinh(eta[i])
```

$\mathcal{O}(N)$ Python bytecode instructions, type-checks, interpreter locks.

VS

```
pz = pt * numpy.sinh(eta)
```

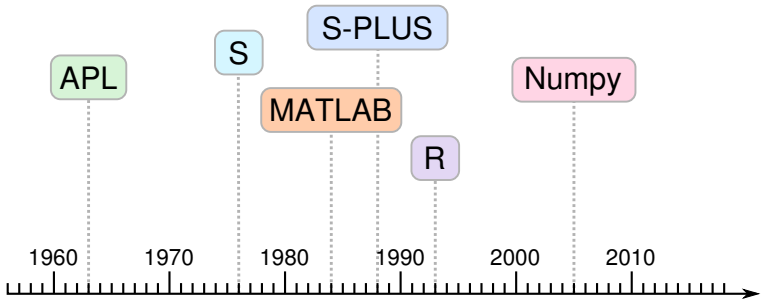
$\mathcal{O}(1)$ Python bytecode instructions, type-checks, interpreter locks.

$\mathcal{O}(N)$ statically typed, probably vectorized native bytecode operations on contiguous memory.

In other words, a Single (Python) Instruction on Multiple Data.

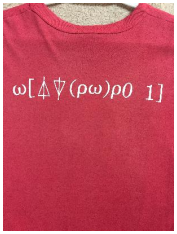
This is not new

APL, "A Programming Language" introduced the idea of single commands having sweeping effects across large arrays.



All members of the APL family are intended for interactive data analysis.
Numpy, however, is a library in a general-purpose language, not a language in itself.

APL pioneered conciseness;
discovered the mistake of being too concise.



Conway's Game of Life was one line of code:

```
life ← {↑ 1  ω ∨ .∧ 3  4 = +/,¯1  0  1 ∘ .Θ ¯1  0  1 ∘ .Φ ⊂ ω}
```

"Map" was implicit, "reduce" was a slash, functions were symbols. For example:

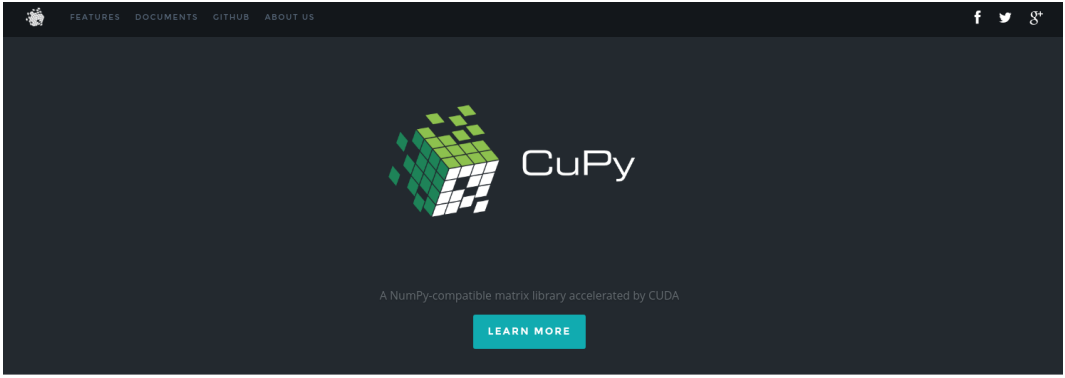
APL	Numpy
`m ← +/(3 + ⍳4)`	`m = (numpy.arange(4) + 3).sum()`

Numpythonic mindset: GPU and vectorization

As an array abstraction, Numpy presents a high-level way for users to think about vectorization.

Vectorization is key to using GPUs and modern CPUs efficiently.

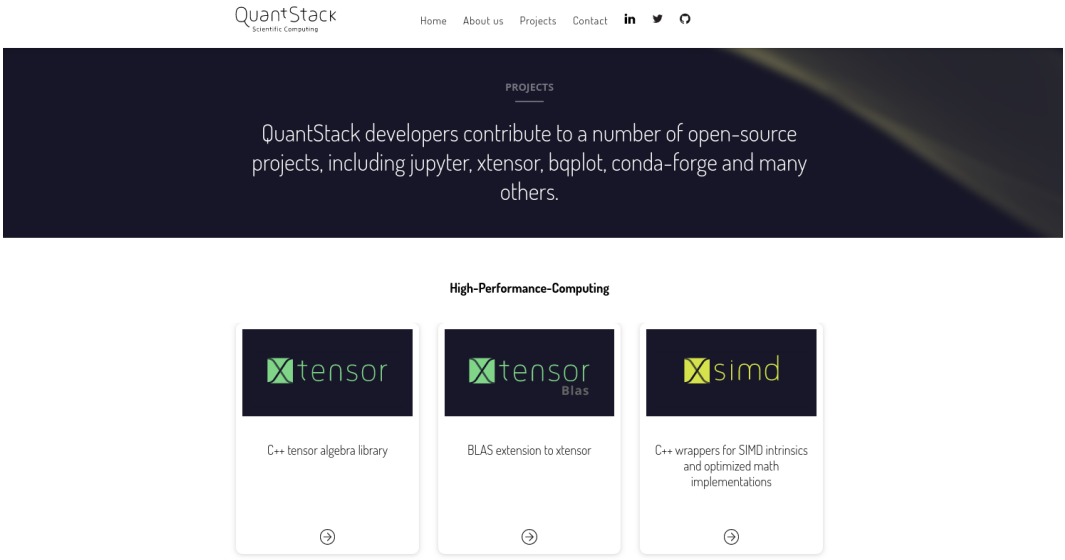
Numpythonic mindset: GPU and vectorization



HIGH PERFORMANCE WITH CUDA

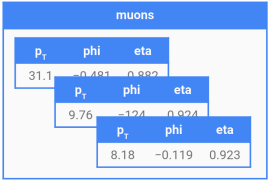
CuPy is an open-source matrix library accelerated with NVIDIA CUDA. It also uses CUDA-related libraries including cuBLAS, cuDNN, cuRand, cuSolver, cuSPARSE, cuFFT and NCCL to make full use of the GPU architecture.

Numpythonic mindset: GPU and vectorization



Can Numpy deal with HEP data?

N-dimensional arrays of values are great for image processing, signal processing, PDEs, but HEP data have nested structure with variable-length contents:

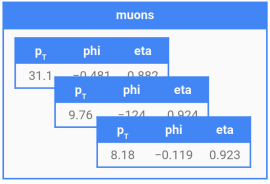


vs

mu1 p_T	mu1 phi	mu1 eta	mu2 p_T	mu2 phi	mu2 eta
31.1	−0.481	0.882	9.76	−0.124	0.924
5.27	1.246	−0.991	n/a	n/a	n/a
4.72	−0.207	0.953	n/a	n/a	n/a
8.59	−1.754	−0.264	8.714	0.185	0.629

Can Numpy deal with HEP data?

N-dimensional arrays of values are great for image processing, signal processing, PDEs, but HEP data have nested structure with variable-length contents:



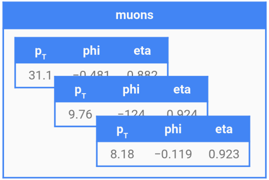
vs

mu1 p_T	mu1 phi	mu1 eta	mu2 p_T	mu2 phi	mu2 eta
31.1	−0.481	0.882	9.76	−0.124	0.924
5.27	1.246	−0.991	n/a	n/a	n/a
4.72	−0.207	0.953	n/a	n/a	n/a
8.59	−1.754	−0.264	8.714	0.185	0.629

```
[[Muon(31.1, -0.481, 0.882), Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
 [Muon(5.27, 1.246, -0.991)],
 [Muon(4.72, -0.207, 0.953)],
 [Muon(8.59, -1.754, -0.264), Muon(8.714, 0.185, 0.629)], ...
```

Can Numpy deal with HEP data?

N-dimensional arrays of values are great for image processing, signal processing, PDEs, but HEP data have nested structure with variable-length contents:



vs

mu1 p_T	mu1 phi	mu1 eta	mu2 p_T	mu2 phi	mu2 eta
31.1	−0.481	0.882	9.76	−0.124	0.924
5.27	1.246	−0.991	n/a	n/a	n/a
4.72	−0.207	0.953	n/a	n/a	n/a
8.59	−1.754	−0.264	8.714	0.185	0.629

```
[[Muon(31.1, -0.481, 0.882), Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
 [Muon(5.27, 1.246, -0.991)],
 [Muon(4.72, -0.207, 0.953)],
 [Muon(8.59, -1.754, -0.264), Muon(8.714, 0.185, 0.629)], ...
```

I've been working on ways to represent arbitrary physics objects as vectorizable arrays:

offsets	0,			3,	4,	5,		7
p_T	31.1,	9.76,	8.18,	5.27,	4.72,	8.59,	8.714	
phi	-0.481,	-0.123,	-0.119,	1.246,	-0.207,	-1.754,	0.185	
eta	0.882,	0.924,	0.923,	-0.991,	0.953,	-0.264,	0.629	

Extending Numpy's concept of "broadcasting"

In Numpy, arithmetic is applied element-wise; scalars are duplicated to fit:

```
>>> MET = numpy.array([(10.2, -0.480), (34.1, 1.251), (26.5, -0.22), (19.0, -1.75)],
...     dtype=[("E", float), ("phi", float)])
>>> MET["E"] * 1.1
array([11.22, 37.51, 29.15, 20.9 ])
```

Extending Numpy's concept of "broadcasting"

In Numpy, arithmetic is applied element-wise; scalars are duplicated to fit:

```
>>> MET = numpy.array([(10.2, -0.480), (34.1, 1.251), (26.5, -0.22), (19.0, -1.75)],
...     dtype=[("E", float), ("phi", float)])
>>> MET["E"] * 1.1
array([11.22, 37.51, 29.15, 20.9 ])
```

We could also apply operations element-wise if their nested structure is the same:

```
>>> muons = awkward.fromiter(
...      [[Muon(31.1, -0.481, 0.882), Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
...       [Muon(5.27, 1.246, -0.991)], [Muon(4.72, -0.207, 0.953)],
...       [Muon(8.59, -1.754, -0.264), Muon(8.714, 0.185, 0.629)]])
>>> muons["pt"]
<JaggedArray [[31.1   9.76  8.18] [5.27] [4.72] [8.59  8.714]] at 7d5022ab3f90>
>>> muons["pt"] * numpy.sinh(muons["eta"])
<JaggedArray [[31.12755703 10.35740718  8.66877254] [-6.12037182] [5.21063432]
              [-2.29419425  5.84974849]] at 7d50223d77d0>
```

Extending Numpy's concept of "broadcasting"

In Numpy, arithmetic is applied element-wise; scalars are duplicated to fit:

```
>>> MET = numpy.array([(10.2, -0.480), (34.1, 1.251), (26.5, -0.22), (19.0, -1.75)],
...     dtype=[("E", float), ("phi", float)])
>>> MET["E"] * 1.1
array([11.22, 37.51, 29.15, 20.9 ])
```

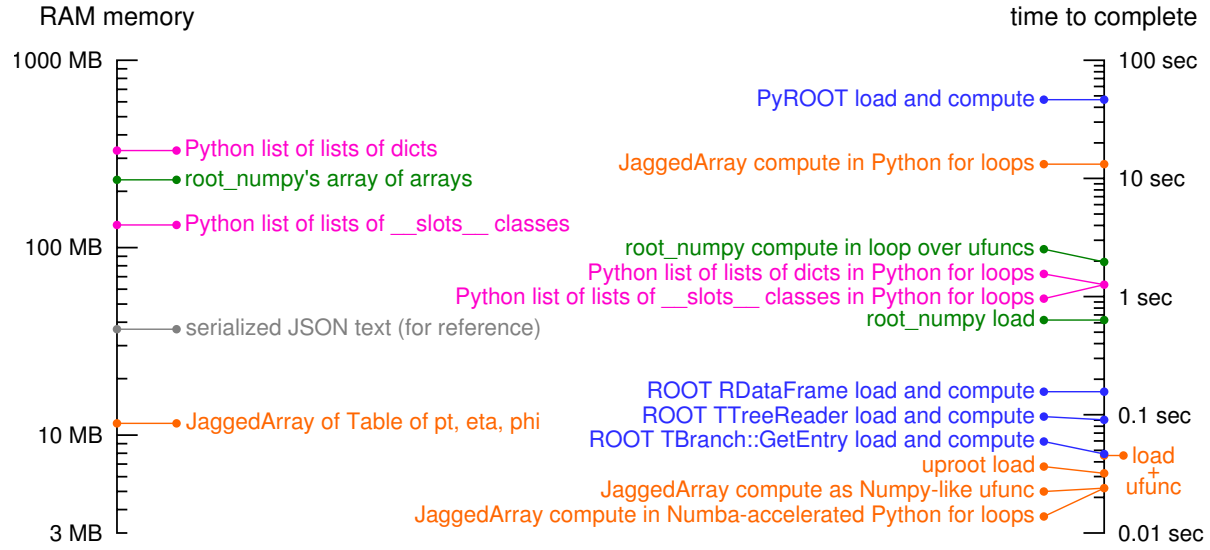
We could also apply operations element-wise if their nested structure is the same:

```
>>> muons = awkward.fromiter(
...     [[Muon(31.1, -0.481, 0.882), Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
...      [Muon(5.27, 1.246, -0.991)], [Muon(4.72, -0.207, 0.953)],
...      [Muon(8.59, -1.754, -0.264), Muon(8.714, 0.185, 0.629)]])
>>> muons["pt"]
<JaggedArray [[31.1   9.76  8.18] [5.27] [4.72] [8.59  8.714]] at 7d5022ab3f90>
>>> muons["pt"] * numpy.sinh(muons["eta"])
<JaggedArray [[31.12755703 10.35740718  8.66877254] [-6.12037182] [5.21063432]
              [-2.29419425  5.84974849]] at 7d50223d77d0>
```

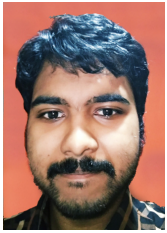
One-per-event scalars could also be broadcast down to multi-per-event jagged arrays:

```
>>> muons["phi"] - MET["phi"]
<JaggedArray [[-0.001  0.356  0.361] [-0.005] [0.013] [-0.004  1.935]] at 7d50223d7790>
```

Big performance gain, even without writing C code



How much more HEP analysis can be vectorized?



Jaydeep Nandi, a CERN/HSF Google Summer of Code student, is investigating vectorized algorithms to replace for-loop manipulations.

Explode to event-wise pairs (using only hardware-SIMD operations):

```
>>> pairs = muons.pairs()
>>> pairs
<JaggedArray [[<Pair 0> <Pair 1> <Pair 2>] [] [] [<Pair 3>]]>
>>> pairs[0][0].tolist()
{"_0": {"pt": 31.1, "phi": -0.481, "eta": 0.882, "pz": 31.128},
 "_1": {"pt": 9.76, "phi": -0.123, "eta": 0.924, "pz": 10.358}}
```

Now we can do such things as compute invariant masses without loops:

```
>>> pt1, eta1, phi1 = pairs["_0"]["pt"], pairs["_0"]["eta"], pairs["_0"]["phi"]
>>> pt2, eta2, phi2 = pairs["_1"]["pt"], pairs["_1"]["eta"], pairs["_1"]["phi"]
>>> mass = numpy.sqrt(2*pt1*pt2*(numpy.cosh(eta1 - eta2) - numpy.cos(phi1 - phi2)))
```

Also considering problems like “minimize per event” and “match gen/reco candidates” in hardware-SIMD operations, exposed in a Numpy-like interface.

Summary

- Python is a popular language, even in sciences where performance is critical.
- It has good features for readability and is easy to learn, particularly by scientists whose primary interest is not programming.
- It came early enough to build up a numerical ecosystem.
- The community was almost fractured by the Numeric-numarray split.
- Modern Numpy is a fixed API that can be swapped out for GPU and SIMD implementations.
- Numpy's existing API is flexible enough to represent complex data structures.
- May be a road toward vectorized HEP analysis, in the guise of interactive array primitives.