Fast and Efficient Python Programmming School: Setting the Scene

Jim Pivarski

Princeton University – IRIS-HEP

August 19, 2024

Welcome!

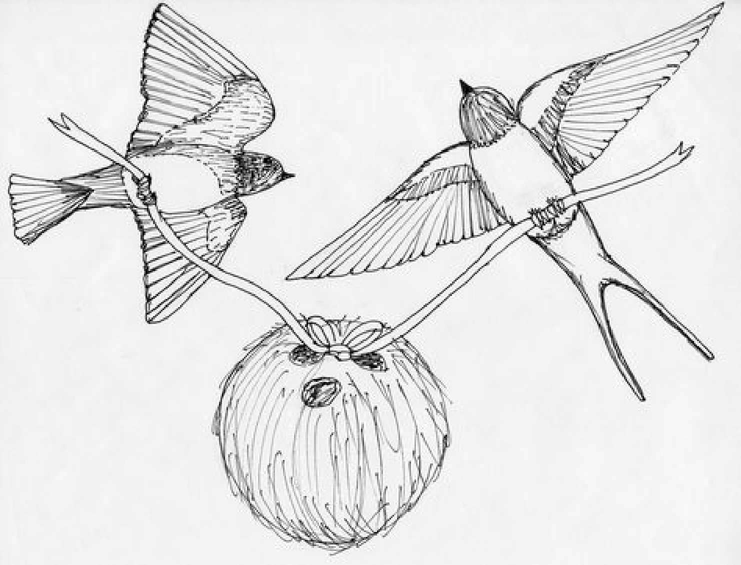
FAST & EFFICIENT PYTHON PROGRAMMING SCHOOL

Aachen

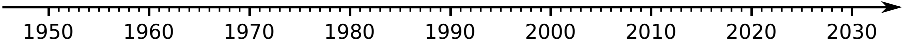
19. - 22. August 2024

Lectures, Tutorials, Computing Challenge

Setting the scene: Python and performance

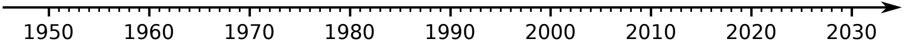


Big, oversimplified history



“just get it working” **“generalize it”** **“make it fast”**

Big, oversimplified history



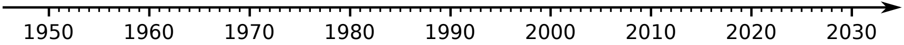
“just get it working”

“generalize it”

“make it fast”

- Early exploration: what can computers do for us?
- Specialized applications for business and science.
- Mostly assembly language.

Big, oversimplified history



"just get it working"

- Early exploration: what can computers do for us?
- Specialized applications for business and science.
- Mostly assembly language.

"generalize it"

- Distributing software for personalized computers and the web, need portability.
- High-level languages, particularly object-oriented.

"make it fast"

Big, oversimplified history

1950 1960 1970 1980 1990 2000 2010 2020 2030

“just get it working”

- Early exploration: what can computers do for us?
- Specialized applications for business and science.
- Mostly assembly language.

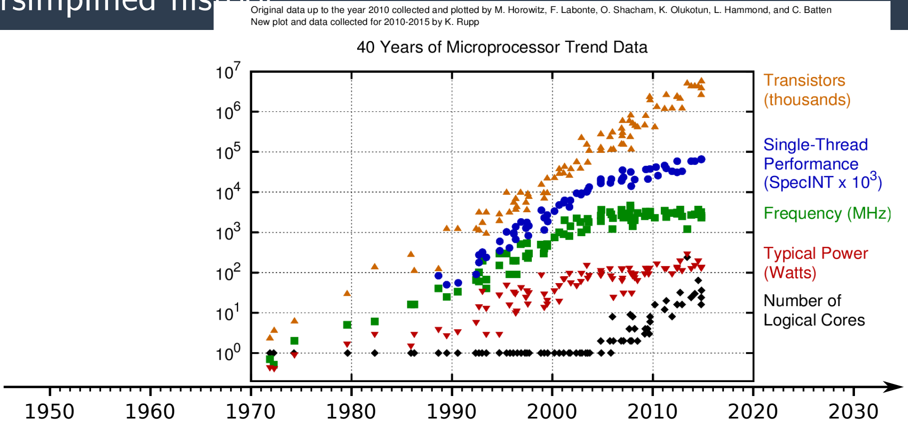
“generalize it”

- Distributing software for personalized computers and the web, need portability.
- High-level languages, particularly object-oriented.

“make it fast”

- Data analytics of web-sized datasets.
- Deep learning becoming effective in business and science.

Big, oversimplified history



"just get it working"

- Early exploration: what can computers do for us?
- Specialized applications for business and science.
- Mostly assembly language.

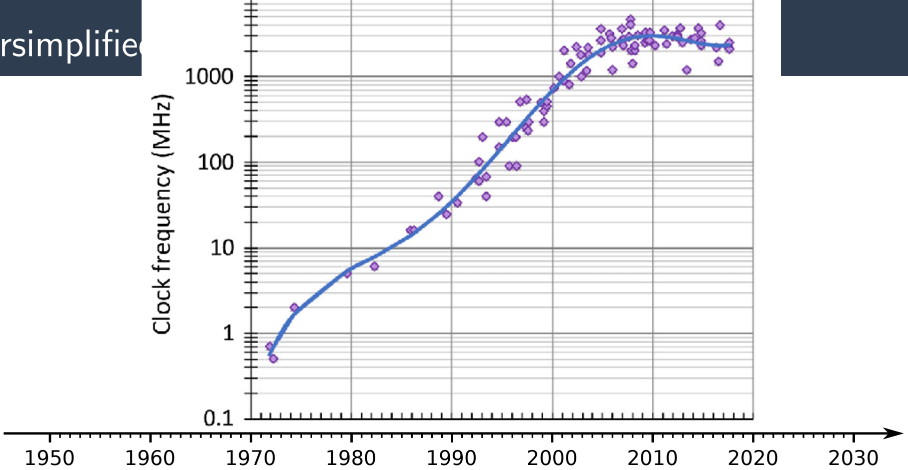
"generalize it"

- Distributing software for personalized computers and the web, need portability.
- High-level languages, particularly object-oriented.

"make it fast"

- Data analytics of web-sized datasets.
- Deep learning becoming effective in business and science.

Big, oversimplifie



"just get it working"

- Early exploration: what can computers do for us?
- Specialized applications for business and science.
- Mostly assembly language.

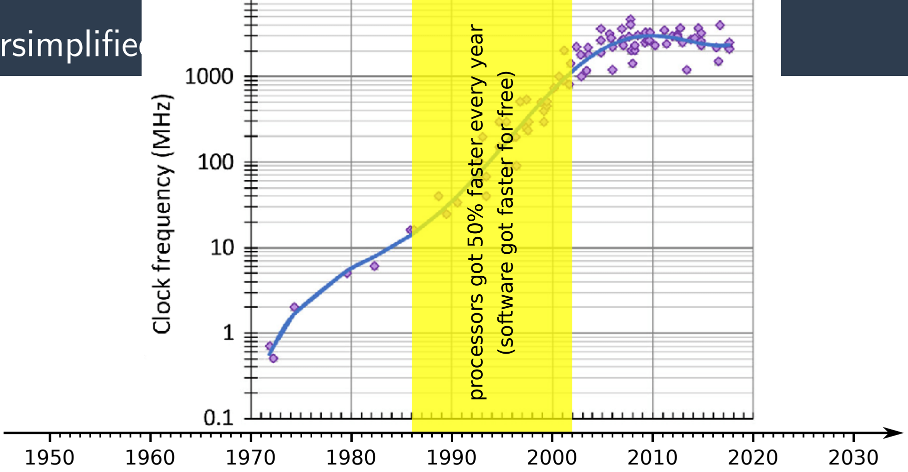
"generalize it"

- Distributing software for personalized computers and the web, need portability.
- High-level languages, particularly object-oriented.

"make it fast"

- Data analytics of web-sized datasets.
- Deep learning becoming effective in business and science.

Big, oversimplifie



"just get it working"

- Early exploration: what can computers do for us?
- Specialized applications for business and science.
- Mostly assembly language.

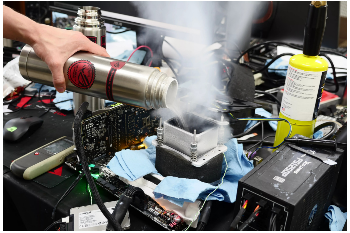
"generalize it"

- Distributing software for personalized computers and the web, need portability.
- High-level languages, particularly object-oriented.

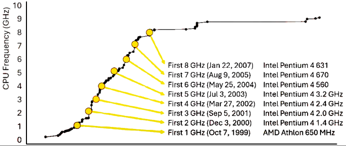
"make it fast"

- Data analytics of web-sized datasets.
- Deep learning becoming effective in business and science.

Big, oversimplified history



extreme overclocking records



1950 1960 1970 1980 1990 2000 2010 2020 2030

"just get it working"

- Early exploration: what can computers do for us?
- Specialized applications for business and science.
- Mostly assembly language.

"generalize it"

- Distributing software for personalized computers and the web, need portability.
- High-level languages, particularly object-oriented.

"make it fast"

- Data analytics of web-sized datasets.
- Deep learning becoming effective in business and science.

Example of the change in mindset...

Greg Owen's talk on Spark 2.0 (May 2016)

Greg Owen’s talk on Spark 2.0 (May 2016)

Volcano Iterator Model

Standard for 30 years: almost all databases do it

Each operator is an “iterator” that consumes records from its input operator

```
class Filter {
  def next(): Boolean = {
    var found = false
    while (!found && child.next()) {
      found = predicate(child.fetch())
    }
    return found
  }

  def fetch(): InternalRow = {
    child.fetch()
  }
  …
}
```

databricks

23

Greg Owen's talk on Spark 2.0 (May 2016)

What if we hire a college freshman to implement this query in Java in 10 mins?

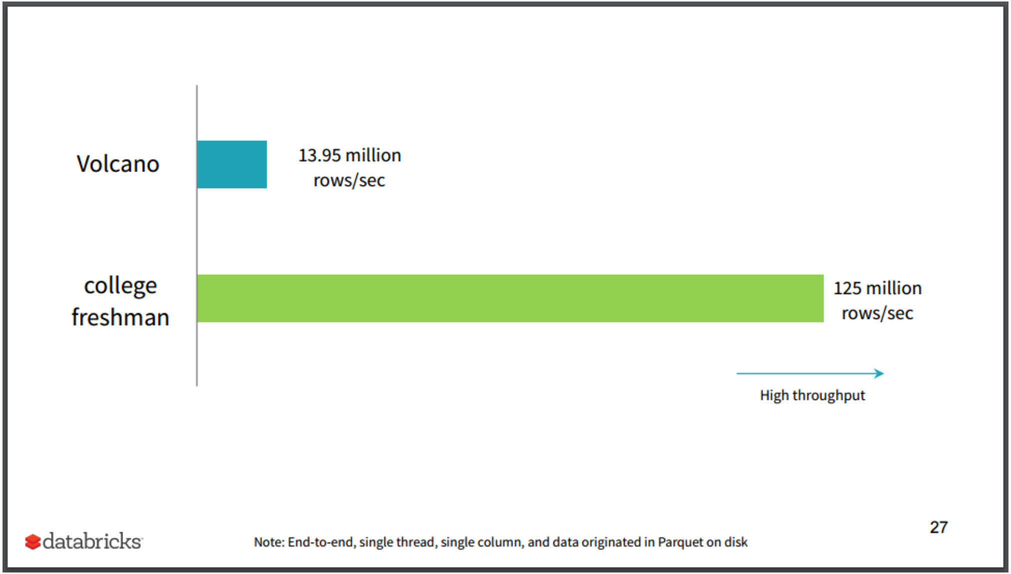
```
select count(*) from store_sales
where ss_item_sk = 1000
```

```
var count = 0
for (ss_item_sk in store_sales)
{
  if (ss_item_sk == 1000) {
    count += 1
  }
}
```

databricks

25

Greg Owen's talk on Spark 2.0 (May 2016)



Greg Owen's talk on Spark 2.0 (May 2016)

How does a student beat 30 years of research?

Volcano	Hand-written code
1. Many virtual function calls	1. No virtual function calls
2. Data in memory (or cache)	2. Data in CPU registers
3. No loop unrolling, SIMD, pipelining	3. Compiler loop unrolling, SIMD, pipelining

Take advantage of all the information that is known after query compilation

databricks

28

Tension between generalizability/portability and speed:

Tension between generalizability/portability and speed:

We want to write generic, abstract code so that it can be used on a variety of computers and so that it can be remixed in other applications.

Tension between generalizability/portability and speed:

We want to write generic, abstract code so that it can be used on a variety of computers and so that it can be remixed in other applications.

We want to write fast code so that we can analyze more data or perform more studies on it.

Tension between generalizability/portability and speed:

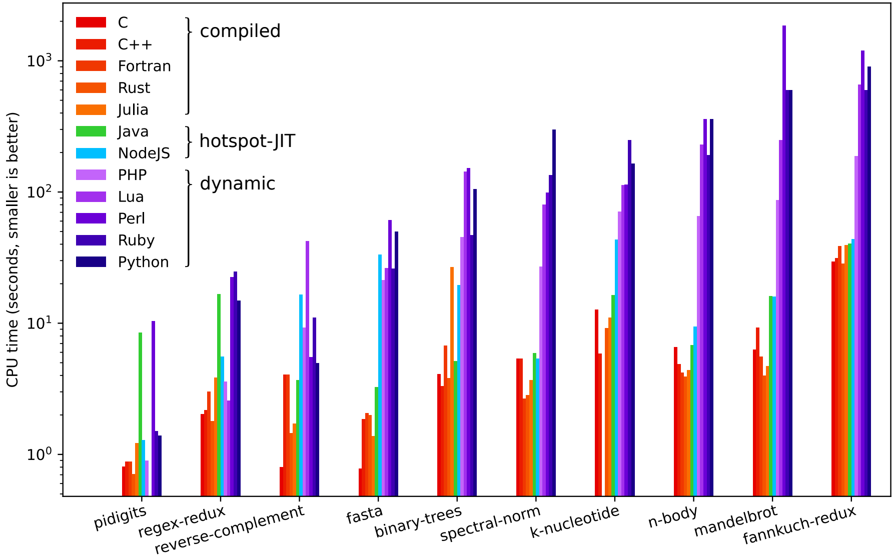
We want to write generic, abstract code so that it can be used on a variety of computers and so that it can be remixed in other applications.

We want to write fast code so that we can analyze more data or perform more studies on it.

Why can't we have it all?

There is a such thing as a “slow language.”

https://benchmarksgame-team.pages.debian.net/benchmarksgame (23.03)



Languages with dynamic features make the computer do more things at runtime; those things take time.

Caveat

Some language features are static, compile-time abstractions, which provide generalizability/portability *and* speed.

Caveat

Some language features are static, compile-time abstractions, which provide generalizability/portability ***and*** speed.

- Modern compilers are better at generating optimized machine code than most humans, taking full advantage of data locality, loop unrolling, SIMD vectorization, pipelining, etc.

Caveat

Some language features are static, compile-time abstractions, which provide generalizability/portability *and* speed.

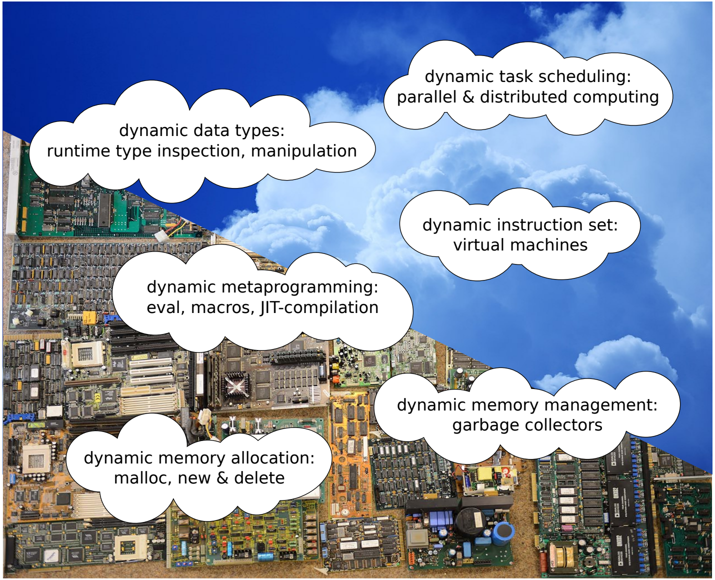
- Modern compilers are better at generating optimized machine code than most humans, taking full advantage of data locality, loop unrolling, SIMD vectorization, pipelining, etc.
- Rust's borrow checker eliminates all memory leaks and double-free segfaults before the code runs, albeit by pointing them out and making the developer fix them manually.

Caveat

Some language features are static, compile-time abstractions, which provide generalizability/portability *and* speed.

- Modern compilers are better at generating optimized machine code than most humans, taking full advantage of data locality, loop unrolling, SIMD vectorization, pipelining, etc.
- Rust's borrow checker eliminates all memory leaks and double-free segfaults before the code runs, albeit by pointing them out and making the developer fix them manually.
- Julia delays the compilation step, Just-In-Time or JIT-compilation, allowing developers to work with abstract code up to the point when it needs to run. (Many Python tools do this, too.)

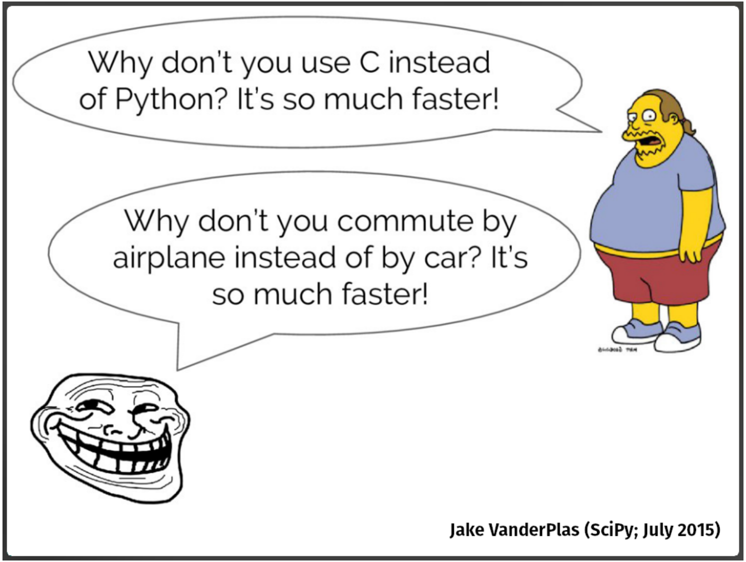
Dynamic language features



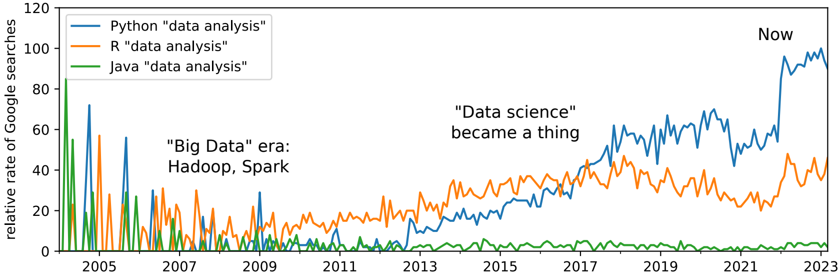
Dynamic language features

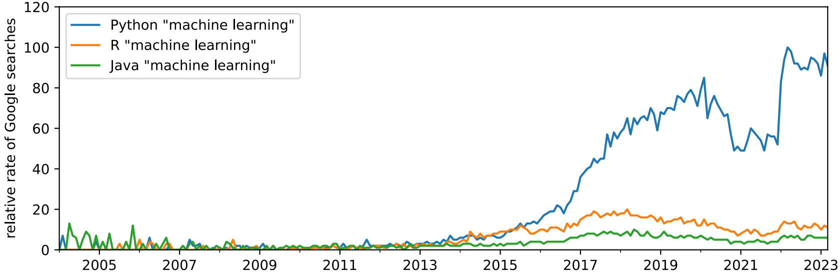
	alloc	reference count	GC	eval	VM	type reflect	scheduling
Fortran 77							
C	√						
C++	√	`shared_ptr<T>`				vtable only	std library
C++ with ROOT	√	`shared_ptr<T>`		√		√	√
Rust	√	`Rc<T>`				vtable only	√
Swift	√	√				vtable only	√
Julia	√		√	√		√	std macros
Go	√		√			vtable only	√
Java (JVM languages)	√		√		√	√	std library
Lua	√		√	√	√	√	
Python	√	√	√	√	√	√	√

Why use a language with these dynamic features, anyway?



Why use a language with these dynamic features, anyway?





Dynamic language features are for *developers!*

Dynamic language features are for *developers!*

- malloc, new & delete: make objects on the fly, arbitrary graph relationships

Dynamic language features are for *developers!*

- malloc, new & delete: make objects on the fly, arbitrary graph relationships
- garbage collectors: eliminate all memory leaks and double-free segfaults

Dynamic language features are for *developers!*

- malloc, new & delete: make objects on the fly, arbitrary graph relationships
- garbage collectors: eliminate all memory leaks and double-free segfaults
- eval, macros, JIT-compilation: deal with information that arrives "late"

Dynamic language features are for *developers!*

- malloc, new & delete: make objects on the fly, arbitrary graph relationships
- garbage collectors: eliminate all memory leaks and double-free segfaults
- eval, macros, JIT-compilation: deal with information that arrives “late”
- virtual machines: portability across hardware architectures

Dynamic language features are for *developers!*

- malloc, new & delete: make objects on the fly, arbitrary graph relationships
- garbage collectors: eliminate all memory leaks and double-free segfaults
- eval, macros, JIT-compilation: deal with information that arrives “late”
- virtual machines: portability across hardware architectures
- runtime type inspection, manipulation: make runtime choices based on types

Dynamic language features are for *developers!*

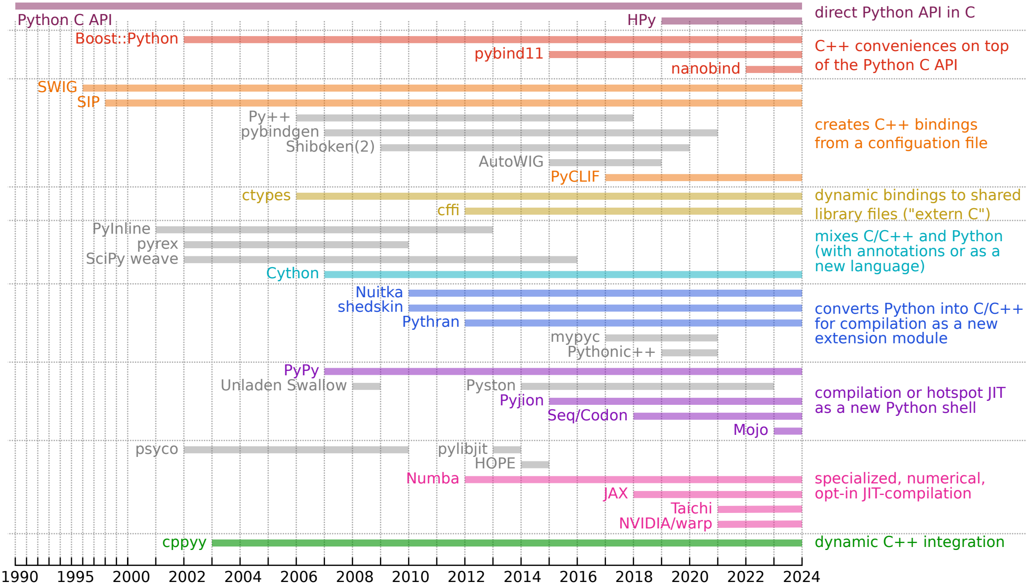
- malloc, new & delete: make objects on the fly, arbitrary graph relationships
- garbage collectors: eliminate all memory leaks and double-free segfaults
- eval, macros, JIT-compilation: deal with information that arrives "late"
- virtual machines: portability across hardware architectures
- runtime type inspection, manipulation: make runtime choices based on types
- parallel & distributed computing abstractions: let a scheduler worry about ordering tasks by data dependencies

In scientific programming and data analysis,
the developer ***is*** the user.

In scientific programming and data analysis,
the developer *is* the user.

The time it takes you to write the code
is part of the optimization.

Long history of attempts to use fast compiled code in Python



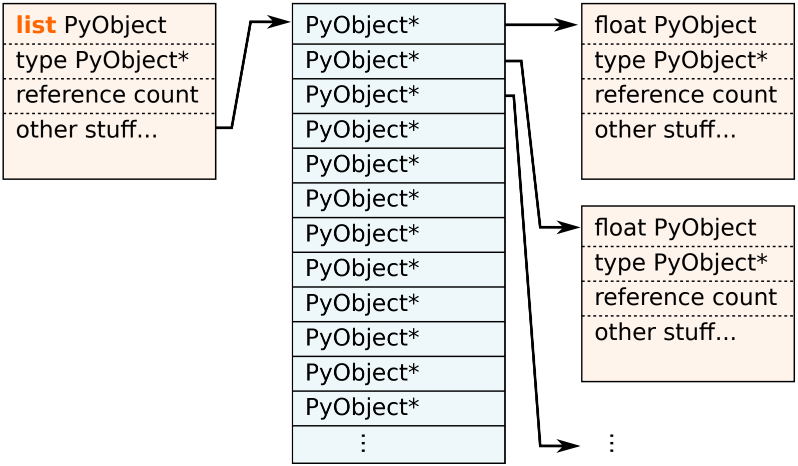
This week, you'll see how to use dynamic features when useful and how to avoid them when necessary.

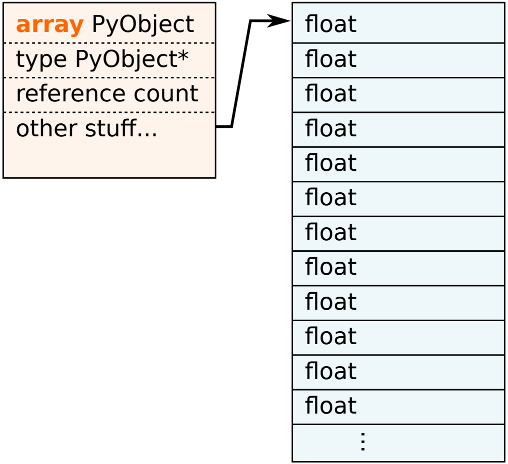
But first, let's see what an implementation looks like.

```
% c++ -std=c++11 -O3 baby-python.cpp -o baby-python
% ./baby-python
                     num = -123         add(x, x)   get(lst, i)   map(f, lst)
               oo    lst = [1, 2, 3]    mul(x, x)   len(lst)      reduce(f, lst)
. . . __/\_/\_/`'    f = def(x) single-expr   f = def(x, y) { ... ; last-expr }

>>
```

BACKUP


list PyObject
type PyObject*
reference count
other stuff...
PyObject*
PyObject*
PyObject*
PyObject*
PyObject*
PyObject*
PyObject*
PyObject*
PyObject*
PyObject*
PyObject*
PyObject*
float PyObject
type PyObject*
reference count
other stuff...
float PyObject
type PyObject*
reference count
other stuff...


array PyObject
type PyObject*
reference count
other stuff...
float
float
float
float
float
float
float
float
float
float
float
float
⋮

Python For Data Science *Cheat Sheet*

NumPy Basics

Learn Python for Data Science **Interactively** at www.DataCamp.com

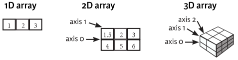
NumPy

The **NumPy** library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

Use the following import convention:

```
>>> import numpy as np
```

NumPy Arrays



Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]],
                 dtype = float)
```

Initial Placeholders

`>>> np.zeros((3,4))`	Create an array of zeros
`>>> np.ones((2,3,4),dtype=np.int16)`	Create an array of ones
`>>> d = np.arange(10,25,5)`	Create an array of evenly spaced values (step value)
`>>> np.linspace(0,2,9)`	Create an array of evenly spaced values (number of samples)
`>>> e = np.full((2,2),7)`	Create a constant array
`>>> f = np.eye(2)`	Create a 2X2 identity matrix
`>>> np.random.random((2,2))`	Create an array with random values
`>>> np.empty((3,2))`	Create an empty array

I/O

Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

Data Types

`>>> np.int64`	Signed 64-bit integer types
`>>> np.float32`	Standard double-precision floating point
`>>> np.complex`	Complex numbers represented by 128 floats
`>>> np.bool`	Boolean type storing `TRUE` and `FALSE` values
`>>> np.object`	Python object type
`>>> np.string_`	Fixed-length string type
`>>> np.unicode_`	Fixed-length unicode type

Inspecting Your Array

`>>> a.shape`	Array dimensions
`>>> len(a)`	Length of array
`>>> b.ndim`	Number of array dimensions
`>>> e.size`	Number of array elements
`>>> b.dtype`	Data type of array elements
`>>> b.dtype.name`	Name of data type
`>>> b.astype(int)`	Convert an array to a different type

Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

Array Mathematics

Arithmetic Operations

`>>> g = a - b`
`array([[-0.5, 0. , 0. ],`
`[-3. , -3. , -3. ]])`
	Subtraction

`>>> np.subtract(a,b)`	Subtraction

`>>> b + a`
`array([[ 2.5, 4. , 6. ],`
`[ 5. , 7. , 9. ]])`
	Addition

`>>> np.add(b,a)`	Addition

`>>> a / b`
`array([[ 0.66666667, 1. , 1. ],`
`[ 0.25 , 0.4 , 0.5 ]])`
	Division

`>>> np.divide(a,b)`	Division

`>>> a * b`
`array([[ 1.5, 4. , 9. ],`
`[ 4. , 10. , 18. ]])`
	Multiplication

`>>> np.multiply(a,b)`	Multiplication
`>>> np.exp(b)`	Exponentiation
`>>> np.sqrt(b)`	Square root
`>>> np.sin(a)`	Print sines of an array
`>>> np.cos(b)`	Element-wise cosine
`>>> np.log(a)`	Element-wise natural logarithm

`>>> e.dot(f)`
`array([[ 7., 7.],`
`[ 7., 7.]])`
	Dot product

Comparison

`>>> a == b`
`array([[False, True, True],`
`[False, False, False]], dtype=bool)`
	Element-wise comparison

`>>> a < 2` `array([True, False, False], dtype=bool)`	Element-wise comparison
`>>> np.array_equal(a, b)`	Array-wise comparison

Aggregate Functions

`>>> a.sum()`	Array-wise sum
`>>> a.min()`	Array-wise minimum value
`>>> b.max(axis=0)`	Maximum value of an array row
`>>> b.cumsum(axis=1)`	Cumulative sum of the elements
`>>> a.mean()`	Mean
`>>> b.median()`	Median
`>>> a.corrcoef()`	Correlation coefficient
`>>> np.std(b)`	Standard deviation

Copying Arrays

`>>> h = a.view()`	Create a view of the array with the same data
`>>> np.copy(a)`	Create a copy of the array
`>>> h = a.copy()`	Create a deep copy of the array

Sorting Arrays

`>>> a.sort()`	Sort an array
`>>> c.sort(axis=0)`	Sort the elements of an array's axis

Subsetting, Slicing, Indexing

Also see Lists

Subsetting

`>>> a[2]` `3`	Select the element at the 2nd index
`>>> b[1,2]` `6.0`	Select the element at row 1 column 2 (equivalent to `b[1][2]`)

Slicing

`>>> a[0:2]` `array([1, 2])`	Select items at index 0 and 1
`>>> b[0:2,1]` `array([ 2., 5.])`	Select items at rows 0 and 1 in column 1
`>>> b[:1]` `array([[1.5, 2., 3.]])`	Select all items at row 0 (equivalent to `b[0:1, :]`)

`>>> c[1,...]`
`array([[[ 3., 2., 1.],`
`[ 4., 5., 6.]]])`
	Same as `[1,:,:]`

`>>> a[ : :-1]` `array([3, 2, 1])`	Reversed array `a`

Boolean Indexing

`>>> a[a<2]` `array([1])`	Select elements from `a` less than 2

Fancy Indexing

`>>> b[[1, 0, 1, 0],[0, 1, 2, 0]]` `array([ 4. , 2. , 6. , 1.5])`	Select elements (1,0),(0,1),(1,2) and (0,0)

`>>> b[[1, 0, 1, 0]][:,[0,1,2,0]]`
`array([[ 4. ,5. , 6. , 4. ],`
`[ 1.5, 2. , 3. , 1.5],`
`[ 4. , 5. , 6. , 4. ],`
`[ 1.5, 2. , 3. , 1.5]])`
	Select a subset of the matrix's rows and columns

Array Manipulation

Transposing Array

`>>> i = np.transpose(b)`	Permute array dimensions
`>>> i.T`	Permute array dimensions

Changing Array Shape

`>>> b.ravel()`	Flatten the array
`>>> g.reshape(3,-2)`	Reshape, but don't change data

Adding/Removing Elements

`>>> h.resize((2,6))`	Return a new array with shape (2,6)
`>>> np.append(h,g)`	Append items to an array
`>>> np.insert(a, 1, 5)`	Insert items in an array
`>>> np.delete(a,[1])`	Delete items from an array

Combining Arrays

`>>> np.concatenate((a,d),axis=0)` `array([ 1, 2, 3, 10, 15, 20])`	Concatenate arrays

`>>> np.vstack((a,b))`
`array([[ 1. , 2. , 3. ],`
`[ 1.5, 2. , 3. ],`
`[ 4. , 5. , 6. ]])`
	Stack arrays vertically (row-wise)

`>>> np.r_[e,f]`	Stack arrays vertically (row-wise)

`>>> np.hstack((e,f))`
`array([[ 7., 7., 1., 0.],`
`[ 7., 7., 0., 1.]])`
	Stack arrays horizontally (column-wise)

`>>> np.column_stack((a,d))`
`array([[ 1, 10],`
`[ 2, 15],`
`[ 3, 20]])`
	Create stacked column-wise arrays

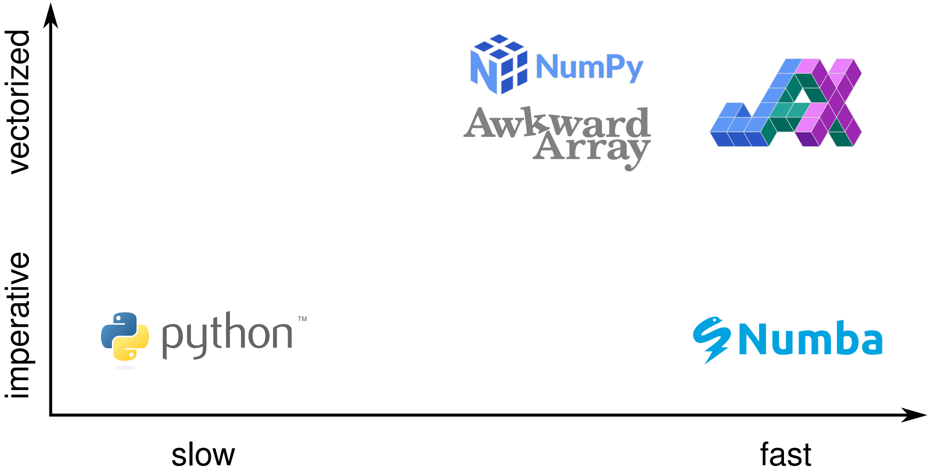
`>>> np.c_[a,d]`	Create stacked column-wise arrays

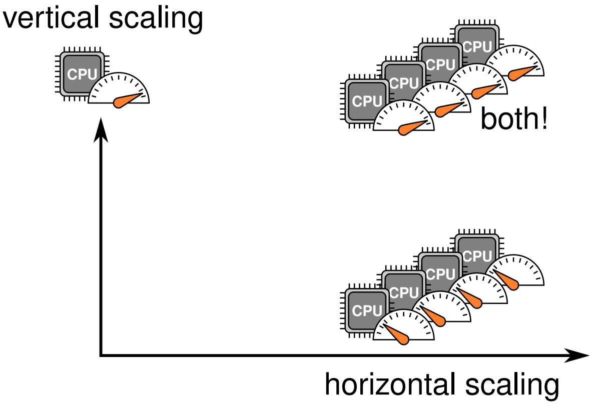
Splitting Arrays

`>>> np.hsplit(a,3)` `[array([1]),array([2]),array([3])]`	Split the array horizontally at the 3rd index

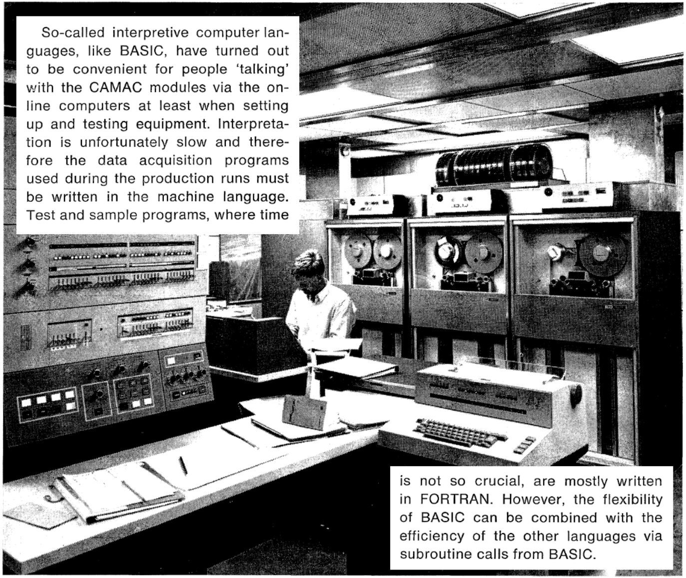
`>>> np.vsplit(c,2)`
`[array([[[ 1.5, 2. , 1. ],`
`[ 4. , 5. , 6. ]]]),`
`array([[[ 3., 2., 3.],`
`[ 4., 5., 6.]]])]`
	Split the array vertically at the 2nd index

DataCamp
Learn Python for Data Science **Interactively**


vectorized
imperative
NumPy
Awkward Array
python
Numba
slow
fast


vertical scaling
CPU
both!
horizontal scaling

CERN Courier, March 1972



So-called interpretive computer languages, like BASIC, have turned out to be convenient for people 'talking' with the CAMAC modules via the on-line computers at least when setting up and testing equipment. Interpretation is unfortunately slow and therefore the data acquisition programs used during the production runs must be written in the machine language. Test and sample programs, where time is not so crucial, are mostly written in FORTRAN. However, the flexibility of BASIC can be combined with the efficiency of the other languages via subroutine calls from BASIC.

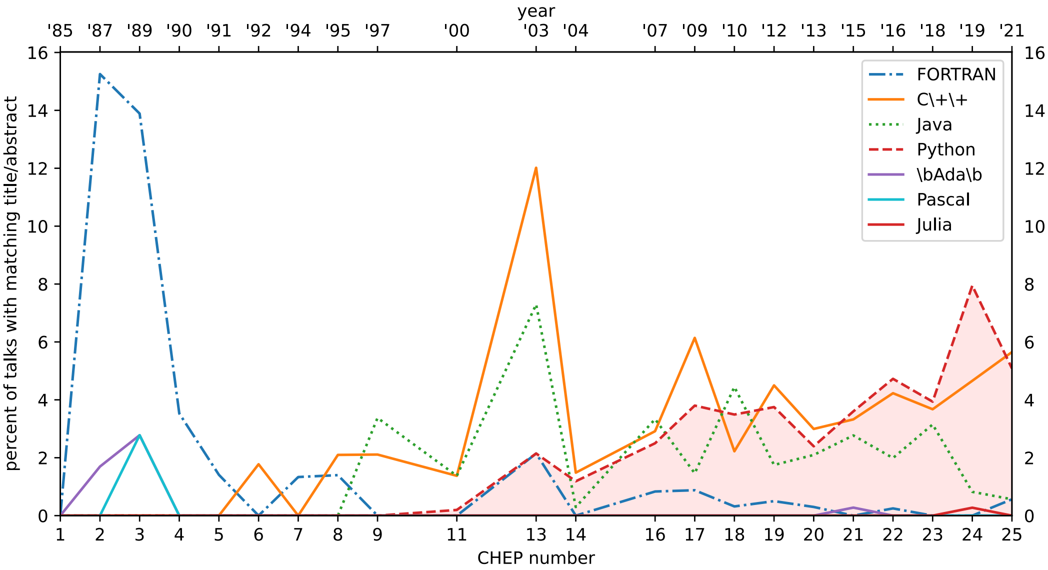
Lukas Taylor's summary talk at CHEP 2001

Emerging Standard ?
Python as "Software Glue"

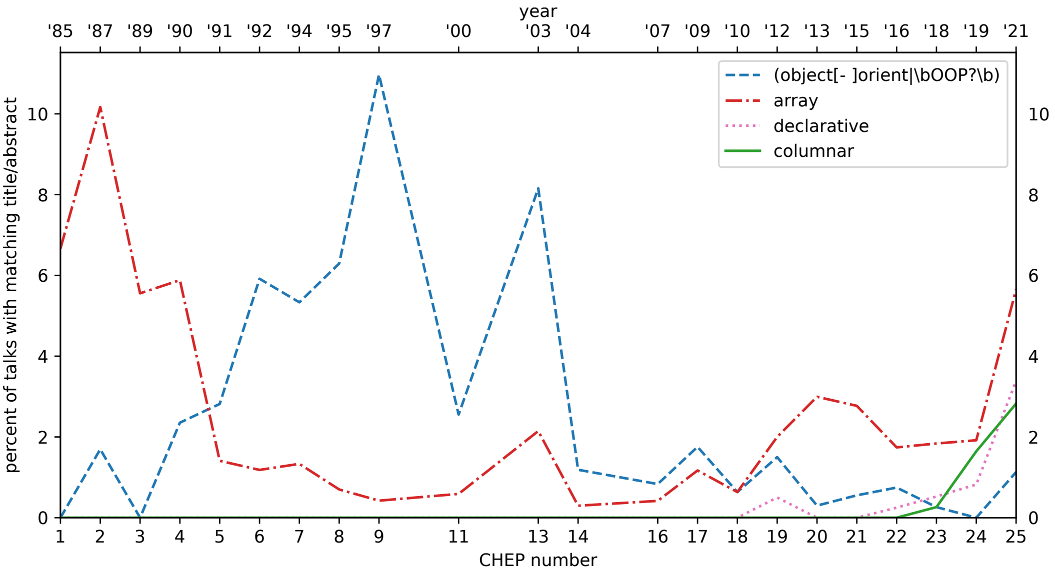
- **Clear trend towards Python**
  - Used by: ATLAS (Athena),CMS, D0, LHCb (Gaudi), SND,...
  - Used by: Lizard/Anaphe, HippoDraw, JAS (Jython)...
  - Architecturally, scripting is "just another service"
  - ROOT is the exception to the "Python rule"
    - CINT interpreter plays a central role
    - Developers and users seem happy
- **Python is popular with developers...**
  - Rapid prototyping; gluing together code
  - (Almost) auto-generation of wrappers (SWIG)
- **...but acceptance by users not yet proven**
  - Another language to learn, syntax,...

"Summary of Track 2: Data Analysis and Visualisa
Lucas Taylor, Northeastern U. CHEP 01, Beijing, 3-7 S

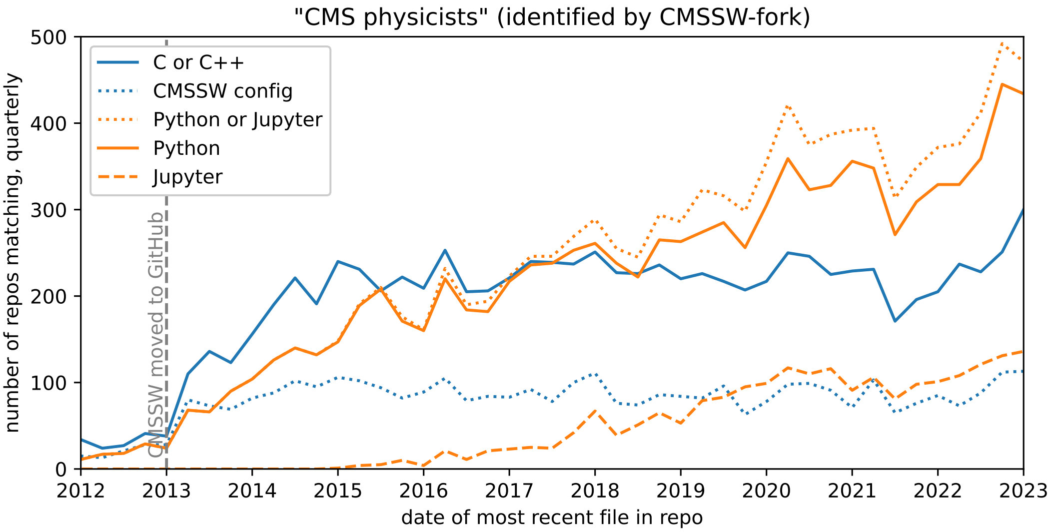
Mentions of programming languages in CHEP talks



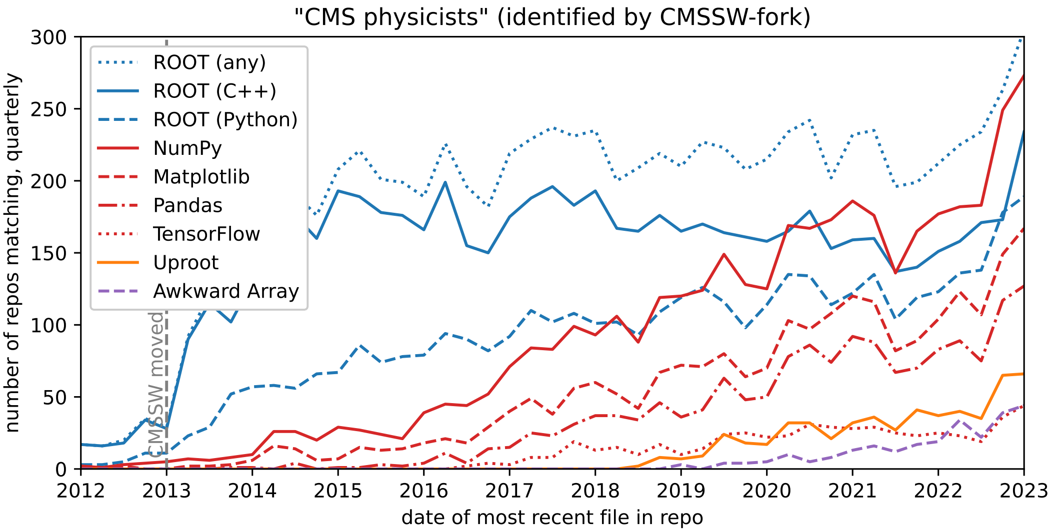
Mentions of programming paradigms in CHEP talks



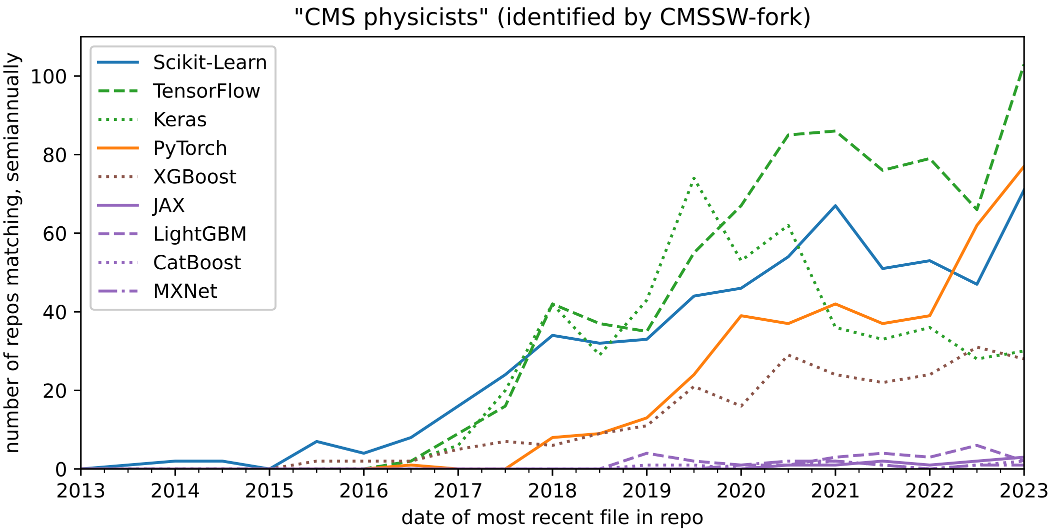
Adoption of Python for CMS analysis

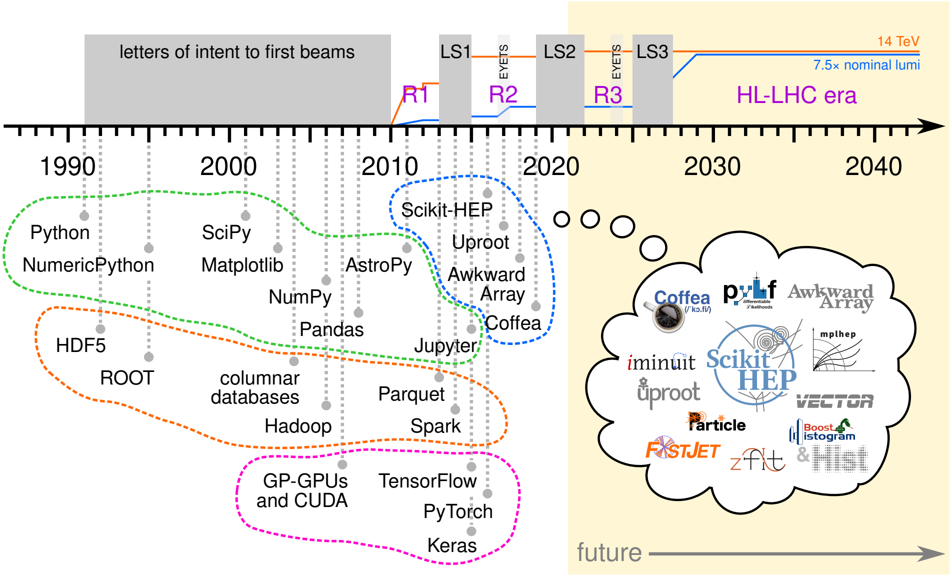


Adoption of Python for CMS analysis



Adoption of Python for CMS analysis




letters of intent to first beams
LS1
EYETS
LS2
EYETS
LS3
14 TeV
7.5× nominal lumi
R1
R2
R3
HL-LHC era
1990
2000
2010
2020
2030
2040
Python
NumericPython
SciPy
Matplotlib
NumPy
Pandas
AstroPy
Scikit-HEP
Uproot
Awkward Array
Coffea
Jupyter
HDF5
ROOT
columnar databases
Hadoop
Parquet
Spark
GP-GPUs and CUDA
TensorFlow
PyTorch
Keras
Coffea
pyhf
Awkward Array
iminuit
Scikit HEP
mplhep
uproot
VECTOR
Particle
Boost histogram
FASTJET
zfit
& Hist
future