History and Adoption of Programming Languages in NHEP

Jim Pivarski

Princeton University – IRIS-HEP

February 8, 2202
This talk is a historical overview of widely used programming languages in NHEP, focusing on the motivations for each language’s adoption.
This talk is a historical overview of widely used programming languages in NHEP, focusing on the motivations for each language’s adoption.

There have always been physicists at the edge, trying out new languages, but most physicists only use one or two. The field as a whole changes slowly.
This talk is a historical overview of widely used programming languages in NHEP, focusing on the motivations for each language’s adoption.

There have always been physicists at the edge, trying out new languages, but most physicists only use one or two. The field as a whole changes slowly.

**Thesis:** (1) change motivated more by “pain points” than incremental benefits,
This talk is a historical overview of widely used programming languages in NHEP, focusing on the motivations for each language’s adoption.

There have always been physicists at the edge, trying out new languages, but most physicists only use one or two. The field as a whole changes slowly.

**Thesis:** (1) change motivated more by “pain points” than incremental benefits, (2) though each of these transitions happened in a unique way.
Adoption of

Fortran: immediate; for syntax and portability; no infrastructure to replace

C++: long overdue; for data structures; replaced infrastructure in a burst

Python: slowly overtook its alternatives; for interactivity; different niche
Part 1: Fortran
NHEP was an early adopter of digital computers

One of the very first applications was Monte Carlo (neutron transport).
And so was data analysis

Luis Alvarez’s group at the Bevatron: $2\text{M} \text{ bubble chamber, } $0.2\text{M} \text{ IBM 650.}
The problem was the same then as it is now

Unlabeled photos come out of the detector.
The problem was the same then as it is now

Unlabeled photos come out of the detector.

Labeling them turns them into quantities to compute.
The problem was the same then as it is now

Unlabeled photos come out of the detector.

Labeling them turns them into quantities to compute.

THE MORE EVENTS THE BETTER!!!
The problem was the same then as it is now.
Identifying tracks was beyond the capabilities of software

So they invented special input devices to streamline data-entry.
Identifying tracks was beyond the capabilities of software

Madeleine (née Goldstein) Isenberg, UCLA class of ’65

“We scanners would review each frame of film, and per the brief instructions we had been given, looked for any ‘unusual activity.’

“The scanner had to use both hands, a joystick in each, and turn them clockwise or anti-clockwise, to align a double crosshair cursor at several sequential positions on a track.”
Identifying tracks was beyond the capabilities of software

Madeleine (née Goldstein) Isenberg, UCLA class of ’65

“A quick but firm tap on the foot-pedal punched the coordinate values onto an IBM card that had been fed into the keypunch machine.

“The precious stack of IBM cards were passed to the physicists, who would then process the data in the existing IBM processors, using software that would calculate the best fit for these coordinates, and thereby mathematically simulate the curvature of the track.”
At first, the software was written in assembly (example from 1958)
At first, the software was written in assembly (example from 1958)

The program is in fixed point, and values are given with eight decimals. This sets an upper bound on the magnitude of numbers, which can be only 99.9999999. An overflow occurs a little before this value, about 109, in the computation of \( v^2 \); however, a rescaling could prevent this. The interval of each advance in the numerical integration \( b \) is variable and has been used at 0.1 radians for most orbits. The error \( e \) between successive values of the variables as computed in the iterative integration procedure is also adjustable and has been used at 0.000000020 for most cases. The similar terms \( n_1 \) and \( n_2 \) are used in the starting program and are set at one-half the values for \( e \) and \( h \).

At the end of each integration step, the square of the total velocity is computed. The constancy of this term is used to judge the accumulation of errors in the integration of the three equations.

**COMPUTATIONS**

**Equations of Motion**

The three-component equations obtained for the Lorentz equation are:

\[
\begin{align*}
\frac{ds^2}{dt^2} &= \frac{(dp^2)}{d\theta^2} = \frac{c}{m} \frac{d\theta}{dt} \frac{d\theta}{dt} - s^2 \\
\frac{d^2}{dt^2} + \frac{1}{m} \frac{db}{dt} \frac{db}{dt} &= - \frac{\omega}{m} \left( \frac{\omega}{dt} B_x - \frac{\omega}{dt} B_y \right) \\
\end{align*}
\]

where we have assumed an azimuthally uniform field, i.e., \( B_y = 0 \), and note that \( B_x = B_x(r, \theta) \) and \( B_y = B_y(r, \theta) \).

The synchronous radius \( R \) corresponds to the beginning radius of the regenerator, and \( \rho \) is the orbit departure from this radius. The radius from the center of the cyclotron is \( r + R - \rho \). The magnetic field values are normalized by dividing them by the field on the median plane at the synchronous orbit \( B_x(R, 0) \).

Our equations then become:

\[
\begin{align*}
\frac{ds^2}{dt^2} &= \left( \frac{(dp^2)}{d\theta^2} \right) = \frac{-\omega B_x(R, 0)}{m} \left( \frac{(p + \rho)}{dt} \right) B_x(r, \theta) \\
\end{align*}
\]
At first, the software was written in assembly (example from 1958)
At first, the software was written in assembly (example from 1958)
At first, the software was written in assembly (example from 1958)
At first, the software was written in assembly (example from 1958)
But it was quickly replaced by Fortran (SUMX from 1964)
CERN Courier special issues on computing

September 1967

Fortran mentioned 3 times

March 1972

Fortran mentioned 18 times
Most of the programming was done in the basic instructions of the computer, but advantages in the use of Autocode (a so-called higher-level language in which the user writes his program in text and algebra-like equations which the computer translates into its own basic instructions) were becoming appreciated, amidst protests from the computer purists who believed that widespread use of such languages would lead to inefficient use of the computer. So it did, in a way, but it also allowed many more scientists to use the computer, which is perhaps a better criterion for efficiency. Essentially, all research laboratories to-day use Fortran, Algol, Autocode or similar language for scientific work.

The present central computing service operates 24 hours per day, seven days per week including most holidays. It is still predominately 'batch-oriented' with the main programming languages being FORTRAN and assembly language. A very large program and subroutine library is available on disk in re-locatable form, and a tape library of some 35 000 labelled tape reels is maintained close to the two computers.
NHEP adopted Fortran for data analysis in the first years after its release in 1956.
NHEP adopted Fortran for data analysis in the first years after its release in 1956.

“Algebra-like equations” (i.e. FORMula-TRANslat) was an obvious benefit.
NHEP adopted Fortran for data analysis in the first years after its release in 1956.

“Algebra-like equations” (i.e. Formula-TRANslation) was an obvious benefit.

Also portability: assembly programs only work on a single model of computer.
NHEP adopted Fortran for data analysis in the first years after its release in 1956.

“Algebra-like equations” (i.e. **FOR**mula-**TRAN**slation) was an obvious benefit.

Also portability: assembly programs only work on a single model of computer.

Why Fortran and not something else (e.g. ALGOL)?

It was IBM’s product, and most labs were buying IBM computers.
Part 2: C++
Fortran lacked an essential feature for NHEP

**Example:** High Energy Physics events are made up of vertices, every vertex has tracks associated to it. Also, to each event is associated a bank of information concerning electronic counter information to be used later. Assume the event to be a two-prong with an associated $V^\circ$.

The pictorial graph for this event information is then

Fortran lacked an essential feature for NHEP

Example: High Energy Physics events are made up of vertices, every vertex has tracks associated to it. Also, to each event is associated a bank of information concerning electronic counter information to be used later. Assume the event to be a two-prong with an associated $V^\circ$.

This wasn’t a part of Fortran until 1991. Physicists created *libraries* for tree-like data.

Fortran lacked an essential feature for NHEP


Years ago, the need for "pointer based" FORTRAN packages, such as HYDRA or BOS, later on ZBOOK [1], ZEBRA [2] or YBOS, became a necessity to efficiently manage the user heap space on a tight fixed size physical memory, in order to support large applications, such as code management systems or histogram packages. [1] Later on, these memory management systems were used not only to allocate dynamically space in a fixed size heap space, but also as a tool to organize and manage sensibly a complicate set of structures describing a detector, or an elementary particle collision. These packages are the essential building blocks for HEP data bases. They were designed to run mainly on single CPU systems and ignored entirely the existence of virtual memory available within FORTRAN through - system dependent! - system calls.
inside a program. Logical relations between banks are expressed by including the address of one bank in the link-table of another bank. For example, all tracks of a vertex-point are linked together by each track pointing to the next. Such a data-structure contains not only the numeric information but also logical information about the object it describes.

The program modularity is achieved by organizing the program into processors each having a well defined task. This task is entirely describable as a transformation applied to a data-structure in the dynamic store: some banks provide the input data to the processor and some contain the desired results. For a given application, a steering program is written to coordinate the operations of the processors needed. Any processor consists of at least one FORTRAN subroutine, its operation being invoked by transferring control to this subroutine. As a matter of internal organization, the processor may be divided up into the primary and several secondary subroutines. The programming of a processor has to observe certain conventions in order to be compatible with the HYDRA system. Precisely these conventions, which are the same through the whole program (indeed through all HYDRA programs) are responsible for the easy documentation and the good readability of the program.

The processors are supported by the HYDRA system. Its services are requested with CALL statements much like the services of the FORTRAN system which are part of the definition of the basic language. In this sense, the HYDRA system is an extension of the FORTRAN language to provide - primarily - dynamic memory management facilities. Some languages contain these facilities in their basic definition, but the HYDRA-

FORTRAN combination has two important advantages — the execution speed is that of a normal FORTRAN program, with very little overhead for the HYDRA system, and FORTRAN is a commonly accepted language. Because of the need for machine independence (so that the same programs can be used on a variety of computers) the processors for the new bubble chamber program, as well as the HYDRA system packages, have been written in ANSI FORTRAN which is the internationally accepted minimum requirement expected from anybody’s compiler.

The bubble chamber programs of the HYDRA form will come into operation in 1972. They should help to tear down the walls that have sometimes threatened, on the data handling side, to separate physicists from computer specialists, or bubble chamber groups from each other and from physicists using other techniques.

Beyond Fortran, data structures were a common language feature

<table>
<thead>
<tr>
<th>COBOL (1959)</th>
<th>Simula (1962)</th>
<th>PL/I (1964)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 Point.</td>
<td>Class Point (x, y);</td>
<td>define structure</td>
</tr>
<tr>
<td>05 x  pic 9(3).</td>
<td>Integer x, y;</td>
<td>1 point,</td>
</tr>
<tr>
<td>05 y  pic 9(3).</td>
<td>! define attributes...</td>
<td>2 x integer,</td>
</tr>
<tr>
<td></td>
<td>! define methods...</td>
<td>2 y integer;</td>
</tr>
<tr>
<td></td>
<td>End of Point;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MODE POINT = STRUCT(</td>
<td>type Point = record</td>
<td>typedef struct Point {</td>
</tr>
<tr>
<td>INT x,</td>
<td>x, y: integer;</td>
<td>int x;</td>
</tr>
<tr>
<td>INT y</td>
<td>end;</td>
<td>int y;</td>
</tr>
<tr>
<td>)</td>
<td></td>
<td>} Point;</td>
</tr>
</tbody>
</table>
Beyond Fortran, data structures were a common language feature


The C programming language is much better than
FORTRAN for both data structures and configuration con-
trol. Shown in Figure 4 are some segments of C code that
one might use in dealing with a track entity. Note the ex-
pressive power of the language in that access to variables is
by full name. Also, the C language deals directly with the
dynamic memory allocation of such structures since the
memory allocation functions are part of its standard library.
Finally, there’s nothing lost in using a symbolic debugger
because structures are part of the language, thus known to
the existing debugger.

Clearly, C is much better at handling data structures
then FORTRAN plus some additional package. Although
many large collaborations have discussed abandoning FOR-
TRAN, none has done so (yet). One reason they stayed with
FORTRAN is reluctance to learn a new language, which is
ironic since in each case they had to learn a big system to
complement FORTRAN.

```c
struct track {
    float px;
    float py;
    float pz;
};
...
...
struct track **mctrack;
...
...
px = mctrack[it]->px;
```

*Figure 4. Segments of C code.*
Beyond Fortran, data structures were a common language feature


The C programming language is much better than FORTRAN for both data structures and configuration control. Shown in Figure 4 are some segments of C code that one might use in dealing with a track entity. Note the expressive power of the language in that access to variables is by full name. Also, the C language deals directly with the dynamic memory allocation of such structures since the memory allocation functions are part of its standard library. Finally, there’s nothing lost in using a symbolic debugger because structures are part of the language, thus known to the existing debugger.

Clearly, C is much better at handling data structures than FORTRAN plus some additional package. Although many large collaborations have discussed abandoning FORTRAN, none has done so (yet). One reason they stayed with FORTRAN is reluctance to learn a new language, which is ironic since in each case they had to learn a big system to complement FORTRAN.

```c
struct track {
    float px;
    float py;
    float pz;
};
...
... struct track **mctrack;
...
... px = mctrack[it]->px;
```

*Figure 4. Segments of C code.*

In the first 15 years of CHEP (1985–2000), similar suggestions were made for ALGOL, PL/I, Pascal, Ada, Eiffel, Objective C, Java, and of course C++. 
Fortran-90 (1991)

type Point
    integer :: x
    integer :: y
end type Point
Fortran did finally add data structures


One major stumbling block in the move to FORTRAN 90 was the question of Input/Output. With ZEBRA, we had a simple way to describe data structures (banks) built out of basic types (typically integers and floats). Because FORTRAN 90 supported derived data types, it was theoretically possible to implement the most complex data structures that we used to model with ZEBRA. In particular ZEBRA was able to write and read these data structures from machine independent files.

Using FORTRAN 90, it appeared pretty hard to make a general implementation equivalent to ZEBRA without parsing the data type description in the FORTRAN 90 modules. In fact, we encountered the same problem later with C++, but we naively ignored at that time how much work it was to implement a data dictionary or reflection system describing at run time the type of objects. Mike Metcalf was aware of the problem and we reported this to a special session of the FORTRAN committee at CERN in 1992. As most members in the committee had no experience with this problem and thought that this was a database problem and not a language problem, the enhancements that we were expecting in the language did not happen.
So actually, NHEP needs:

1. rich data structures in the programming language,
2. serialization of those structures to/from disk,
3. read compatibility for old data versions (schema evolution),
4. mapping between persistent data and language’s structures (which can be implemented with type-introspection).
So actually, NHEP needs:

1. rich data structures in the programming language,
2. serialization of those structures to/from disk,
3. read compatibility for old data versions (schema evolution),
4. mapping between persistent data and language’s structures (which can be implemented with type-introspection).

HYDRA/ZEBRA/etc. approached these as a single problem.
So actually, NHEP needs:

1. rich data structures in the programming language,
2. serialization of those structures to/from disk,
3. read compatibility for old data versions (schema evolution),
4. mapping between persistent data and language’s structures (which can be implemented with type-introspection).

HYDRA/ZEBRA/etc. approached these as a single problem.

In Java, (1–2) is a language problem, (3–4) is for databases.
So actually, NHEP needs:

1. rich data structures in the programming language,
2. serialization of those structures to/from disk,
3. read compatibility for old data versions (schema evolution),
4. mapping between persistent data and language’s structures (which can be implemented with type-introspection).

HYDRA/ZEBRA/etc. approached these as a single problem.

In Java, (1–2) is a language problem, (3–4) is for databases.

Object databases focus on (4)…
So it wasn’t just a matter of adding another language to the mix.

The whole infrastructure, including I/O, had to change.

You only want to do that once!
Programming languages in CHEP title/abstract regex matches

- FORTRAN
- C\+
- Java
- Ada
- Pascal

Year:
- '85
- '87
- '89
- '90
- '91
- '92
- '94
- '95
- '97
- '98
- '00
- '01
- '03
- '04
- '06
- '07
- '09
- '10
- '12
- '13
- '15
- '16
- '18
- '19
- '21

Percent of talks with matching title/abstract

CHEP number:
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
Data infrastructure in CHEP title/abstract regex matches

![Graph showing percentage of talks with matching title/abstract over years 1985 to 2021, with markers for various regex matches including \bROOT\b, \b(Objectivity|ORM|OODB(|MS))\b, \b(Hippoplotamus|HippoDraw)\b, \b(java Analysis Studio|JAS|Jas4pp|FreeHEP)\b, and \bLHC\++\+.
Total number of talks (denominator): dips are small CHEPs
ROOT I/O is custom, but was more advanced than alternatives.
Some advances had to be rediscovered outside NHEP

Dremel: Interactive Analysis of Web-Scale Datasets (Google, 2010)

storage and reduce CPU cost due to cheaper compression. Column stores have been adopted for analyzing relational data [1] but to the best of our knowledge have not been extended to nested data models. The columnar storage format that we present is supported by many data processing tools at Google, including MR, Sawzall [20], and FlumeJava [7].

In this paper we make the following contributions:

- We describe a novel columnar storage format for nested data. We present algorithms for dissecting nested records into columns and reassembling them (Section 4).

Columnar, nested data was a ROOT feature 15 years earlier.
NHEP adoption of C++ (or similar) was held back by the fact that we had unique infrastructure.
NHEP adoption of C++ (or similar) was held back by the fact that we had unique infrastructure.

It’s important to have data structures in the language, but also on disk.
NHEP adoption of C++ (or similar) was held back by the fact that we had unique infrastructure.

It's important to have data structures in the language, but also on disk.

Our solutions were both *more advanced* and *less modular* than others.
NHEP adoption of C++ (or similar) was held back by the fact that we had unique infrastructure.

It’s important to have data structures in the language, but also on disk.

Our solutions were both *more advanced* and *less modular* than others.

Many options considered in late 1990’s; C++ with ROOT I/O became dominant.
Part 3: Python
So-called interpretive computer languages, like BASIC, have turned out to be convenient for people 'talking' with the CAMAC modules via the online 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.
NHEP has a history of custom solutions for interactivity: SPEAKEASY, Minuit, PAW, KUIP, CINT, Cling...
NHEP has a history of custom solutions for interactivity: SPEAKEASY, Minuit, PAW, KUIP, CINT, Cling…

But the number of industry solutions is also vast.

Interactive mode languages  [edit]

Interactive mode languages act as a kind of shell: expressions or statements can be entered one at a time, and the result of their evaluation is seen immediately. The interactive mode is also termed a *read–eval–print loop* (REPL).

- APL
- BASIC (some dialects)
- Clojure
- Common Lisp
- Dart (with Observatory or Dartium’s developer tools)
- ECMAScript
  - ActionScript
  - ECMAScript for XML
  - JavaScript
  - JScript
  - Source
- Erlang
- Elixir (with iex)
- F#
- Fri
- GAUSS
- Groovy
- Haskell (with the GHCi or Hugs interpreter)
- IDL
- J
- Java (since version 9)
- Julia
- Lua
- MUMPS (an ANSI standard general purpose language)
- Maple
- Mathematica (Wolfram language)
- MATLAB
- ML
- OCaml
- Perl
- PHP
- Pike
- PostScript
- Prolog
- Python
- PROSE
- R
- REBOL
- Rexx
- Ruby (with IRB)
- Scala
- Scheme
- Smalltalk (anywhere in a Smalltalk environment)
- S-Lang (with the S-Lang shell, slish)
- SPEAKEASY
- swift
- Tcl (with the Tcl shell, tclsh)
- Unix shell
- Windows PowerShell (.NET-based CLI)
- Visual FoxPro
In recent years, the industry has consolidated on Python. Python is currently leading every “most popular programming language” index.
Including data analytics and especially machine learning

Correlated words in Google searches (Google Trends).

Machine Learning era started when Python was already popular
Python use has also been rising—steadily—in NHEP.
Since before the return of machine learning
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,…

Note: PyROOT introduced in 2004 (v4.00/04).
Early applications: gluing compiled modules together

Stephan Lammel, *Computing models of major HEP experiments: DØ and CDF*, 1997

DØ has made the decision to move all large software projects to C++. Their framework approach has a set of modules that execute sequentially, each having a specific task. The glue that holds the individual software packages together will be an interpreted script system. The main task of this framework is to “guide” data between the various modules/packages ... prototype framework based on the Python scripting language has been developed and is ready for use.


Most of the code would be written in C or C++, but the integration would be done through Python. This enables the uninitiated to make simple modifications to the analysis which were perhaps not thought of by the authors; all the neophyte needs to know is how the interfaces work. On the other hand, it will force the code authors to make the analysis subsystems independent of each other (one of the big problems with the current code), and will encourage rigorous testing of subunits.
Trends in code written by CMS users (in GitHub)

Number of repos created by all CMS users

- C and C++
- Python
- Jupyter Notebook
- Java
- Fortran
- Julia

Repositories created by CMS physicists per 90 days


Repository creation date

CMSSW moved to GitHub
Trends in code written by CMS users (in GitHub)

Number of repos created per CMS user

- C and C++
- Python
- Jupyter Notebook
- Java
- Fortran
- Julia
Trends in code written by CMS users (in GitHub)

Number of repos with code matching search strings

- Python "numpy"
- Python "matplotlib"
- Python "pandas"
- Python "sklearn"
- Python "tensorflow"
- Python "torch"
- Python "xgboost"
Trends in code written by CMS users (in GitHub)

Number of repos with code matching search strings

- "TFile" (C++)
- "TFile" (Python)
- "import ROOT"
- "uproot"
- "numpy"
- "matplotlib"
- "pandas"

Repository creation date
Survey responses from PyHEP 2020 attendees ($N = 406$)

**Which do you use regularly (> 10% of your work)?**

- C or C++
- Python
- Matlab
- Javascript or other browser-based (e.g. TypeScript, CoffeeScript)
- Verilog, VHDL, or other hardware description language
- R
- Java or other JVM-based (e.g. Kotlin, Scala, Clojure)
- Perl
- PHP
- C#
- Julia
- Go
- Swift
- Rust
- Ruby
- Haskell
- Raw assembly or machine code
- Other, not listed above

**How often do you use Python relative to C or C++?**

- More C++
- Half-and-half
- Always C++
- Neither
- Always Python
- More Python

**What are your main uses of Python?**

- Physics analysis (other than machine learning)
- Physics analysis with machine learning
- Scripting routine tasks (e.g. job submission)
- Interacting with collaboration software frameworks (e.g. configuration)
- Non-physics data analysis
- Computing infrastructure (e.g. GRID middleware, analyzing log files)
- Developing general libraries for others to use
- Demos for outreach and education
- Developing specialized applications (e.g. dashboards)
- Other uses, not listed above
NHEP adoption of Python started long before machine learning and columnar analysis trends.
NHEP adoption of Python started long before machine learning and columnar analysis trends.

Interactive/fluid programming has always been a need, and has traditionally been met by a variety of alternatives.
NHEP adoption of Python started long before machine learning and columnar analysis trends.

Interactive/fluid programming has always been a need, and has traditionally been met by a variety of alternatives.

What’s new is the consolidation on one language, Python, and an increase in how much analysis logic can be “driven” from the interactive language.
Concluding conclusions
Adoption of

**Fortran**: immediate; for syntax and portability; no infrastructure to replace

**C++**: long overdue; for data structures; replaced infrastructure in a burst

**Python**: slowly overtook its alternatives; for interactivity; different niche

**Julia?** slowly mix in among the Python and C++ until it’s all that’s left?
Backup
Python ecosystem has had as much time to evolve as the LHC
Widespread familiarity with data science tools (PyHEP survey)

"Do you use these software packages?"

- NumPy
- Matplotlib
- SciPy
- Jupyter
- Pandas
- SciKit-Learn
- IPython
- TensorFlow
- Keras
- SciKit-Optimize
- hSpy (HDFS)
- Seaborn
- PyTorch
- SciKit-Image
- Numba
- xarray
- ROOT in C++
- ROOT through PyROOT
- Uproot
- root-numpy
- root-pandas
- rootpy
- lumi
- Awkward Array
- particle
- Coffea
- pyhf

Color codes:
- No selection
- Don't know what it is
- Never
- Through dependencies only
- Regularly
- All the time

Number of PyHEP 2020 survey respondents:

- 0
- 50
- 100
- 150
- 200
- 250
- 300
- 350
More CHEP history

- (object[- ]orient\bOOP?!b)
- array
- declarative
- columnar

Year: '85 '87 '89 '90 '91 '92 '94 '95 '97 '98 '00 '01 '03 '04 '06 '07 '09 '10 '12 '13 '15 '16 '18 '19 '21

Percent of talks with matching title/abstract

CHEP number: 0 0 2 2 4 4 6 6 8 8 10 10

Year: '85 '87 '89 '90 '91 '92 '94 '95 '97 '98 '00 '01 '03 '04 '06 '07 '09 '10 '12 '13 '15 '16 '18 '19 '21

Percent of talks with matching title/abstract

CHEP number: 0 0 2 2 4 4 6 6 8 8 10 10
More CHEP history

- **CHEP number:**
  - 0
  - 2
  - 4
  - 6
  - 8

- **Percent of talks with matching title/abstract:**
  - '85
  - '87
  - '89
  - '90
  - '91
  - '92
  - '94
  - '95
  - '97
  - '98
  - '00
  - '01
  - '03
  - '04
  - '06
  - '07
  - '09
  - '10
  - '12
  - '13
  - '15
  - '16
  - '18
  - '19
  - '21

- **Year:**
  - '85
  - '87
  - '89
  - '90
  - '91
  - '92
  - '94
  - '95
  - '97
  - '98
  - '00
  - '01
  - '03
  - '04
  - '06
  - '07
  - '09
  - '10
  - '12
  - '13
  - '15
  - '16
  - '18
  - '19
  - '21
More CHEP history

- CPUs?
- GPUs? | CUDA
- FPGAs? | VHDL | Verilog
- TPUs?
- RISCs?

- Percent of talks with matching title/abstract.

- CHEP number:
  - 0 0
  - 2 2
  - 4 4
  - 6 6
  - 8 8
  - 10 10
  - 12 12
  - 14
  - 16

- Year:
  - '85
  - '87
  - '89
  - '90
  - '91
  - '92
  - '94
  - '95
  - '97
  - '98
  - '00
  - '01
  - '03
  - '04
  - '06
  - '07
  - '09
  - '10
  - '12
  - '13
  - '15
  - '16
  - '18
  - '19
  - '21

- CHEP number vs. percent of talks with matching title/abstract.