Friedrich-Alexander-Universität Erlangen-Nürnberg

ERLANGEN CENTRE FOR ASTROPARTICLE PHYSICS

THE MISSING **PYTHON INTRO**DUCTION **FOR SCIENTISTS**

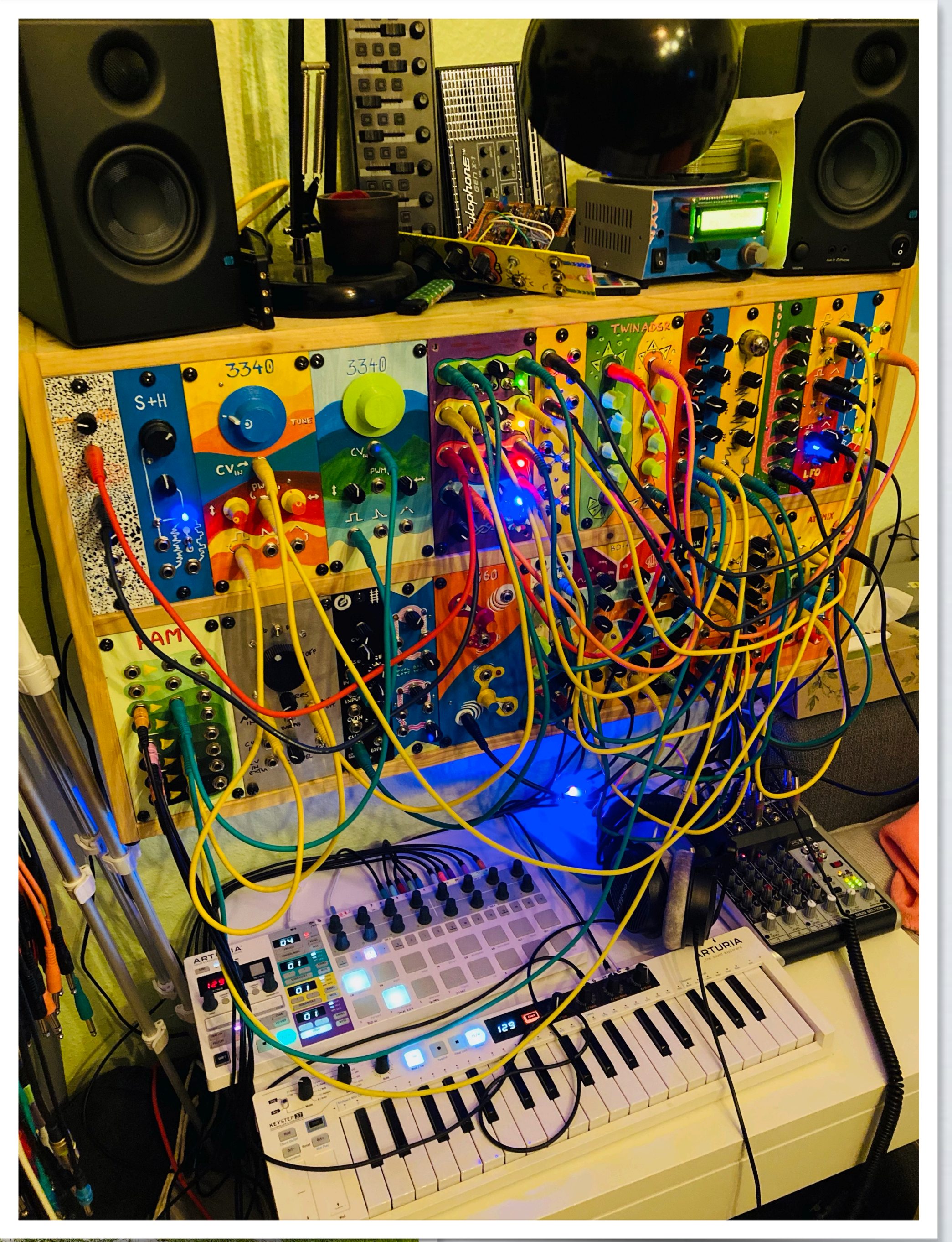
TAMAS GAL – ERLANGEN CENTRE FOR ASTROPARTICLE PHYSICS (ECAP)

WHO AM I?

- **Tamás Gál**, born 1985 in Debrecen (Hungary)
- **Astroparticle physicist** at the Erlangen Centre for Astroparticle Physics (**ECAP**) working on the **KM3NeT neutrino detector** experiment and **open science/data**
- **Sysadmin** (DevOps) at **ECAP** (including the ECAP and KM3NeT IT services)
- **Programming** background:
  - Coding enthusiast since ~**1993**
  - First real application written in **Amiga Basic** (toilet manager, tons of **GOTOs** ;)
  - Mostly **Julia, Python**, Rust, JavaScript and C/C++ for **work**
  - **Haskell** for **fun**
  - Earlier also Obj-C, Java, Perl, PHP, Delphi, MATLAB, whatsoever…
- Started with **Python** around **1998** (to replace Perl/Shell)
- Editor: **Vim for ~25 years** but switched to **Emacs (EVIL mode)** in 2020
- Other: ADV motorbikes, climbing, electronics, modular synths, DIY…

@tamasgal

My desk in 2003



My DIY modular synth

My GS

DISCLAIMER

The following presentation contains oversimplifications and blatant omissions due to time constraints. The viewer may also experience well-timed product placements.

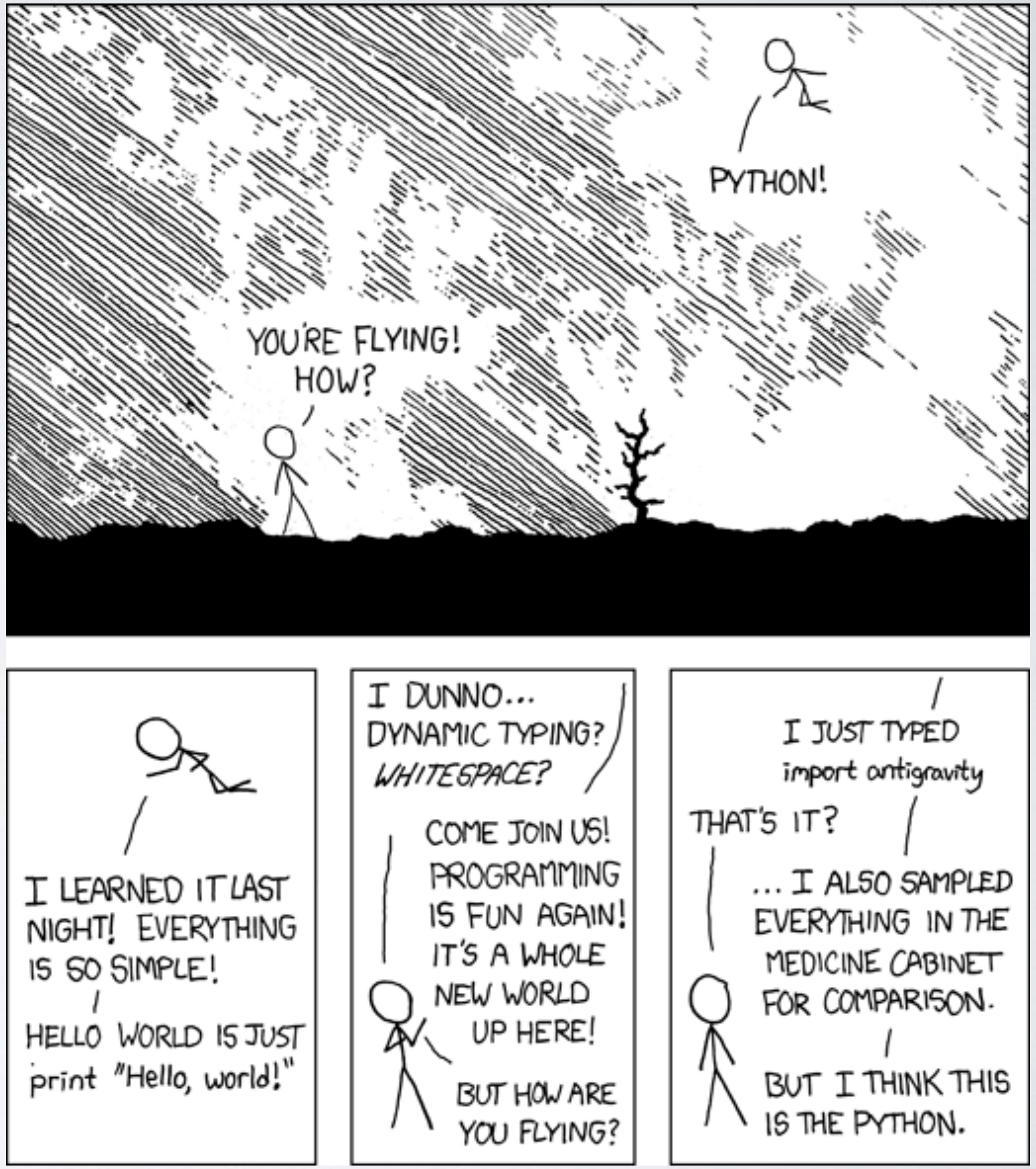
PYTHON

THE PYTHON PROGRAMMING LANGUAGE

- **Interpreted** high-level **general-purpose** programming language
- **Object-oriented**, procedural (imperative), functional, structured, reflective
- **Dynamically-typed** and **garbage-collected**
- "**batteries included**"
- Tries to **avoid premature optimisation**: move time-critical functions to extension modules written in "faster" languages (like C or Fortran) when necessary

THE ZEN OF PYTHON

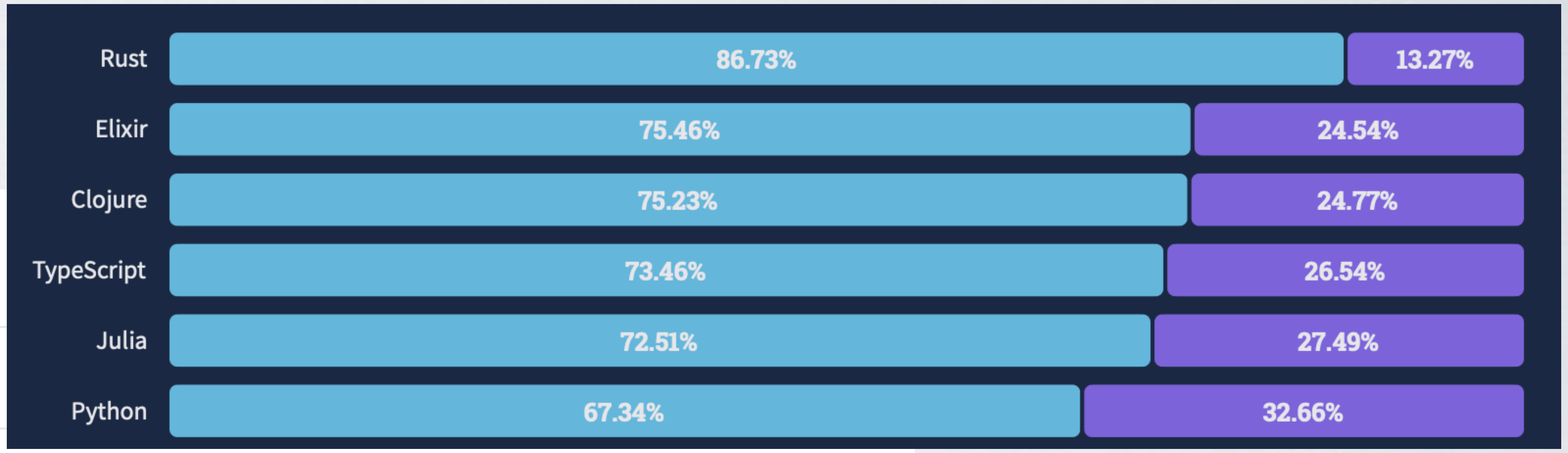
```
>>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those
```

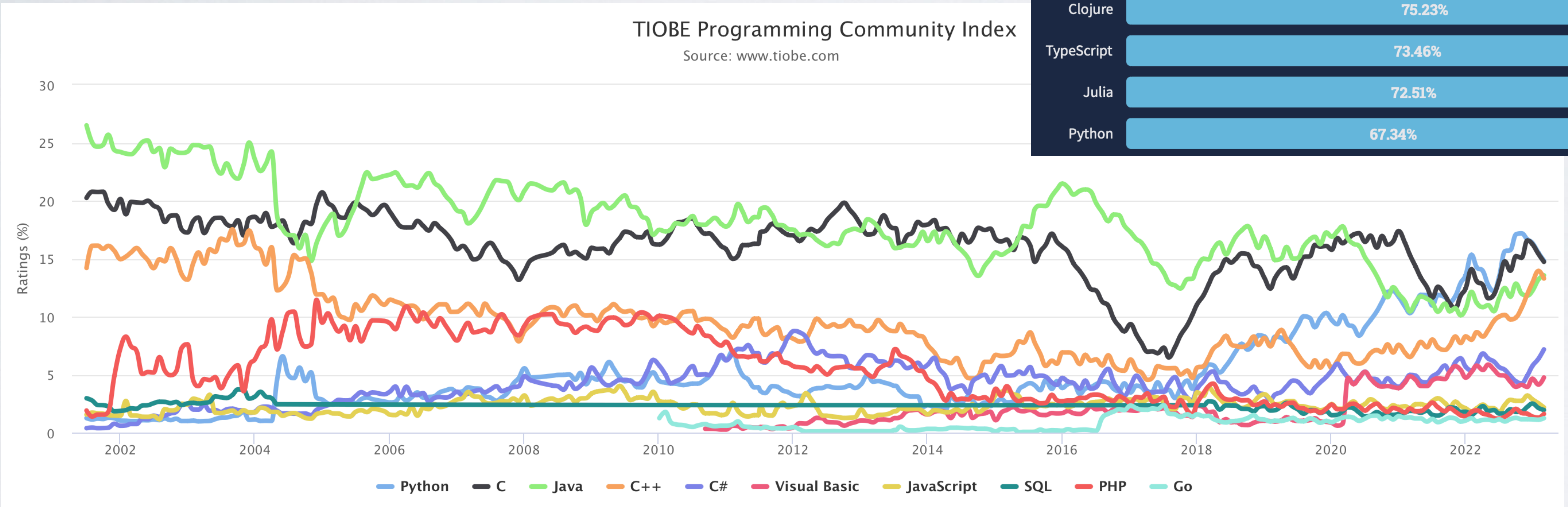


xkcd.com/535

POPULAR LANGUAGES

Most loved languages https://survey.stackoverflow.co/2022





`Python` is the most popular language (according to TIOBE)!

… and has beaten **Java** and **C++**! Btw. **Julia** is #33 ;)

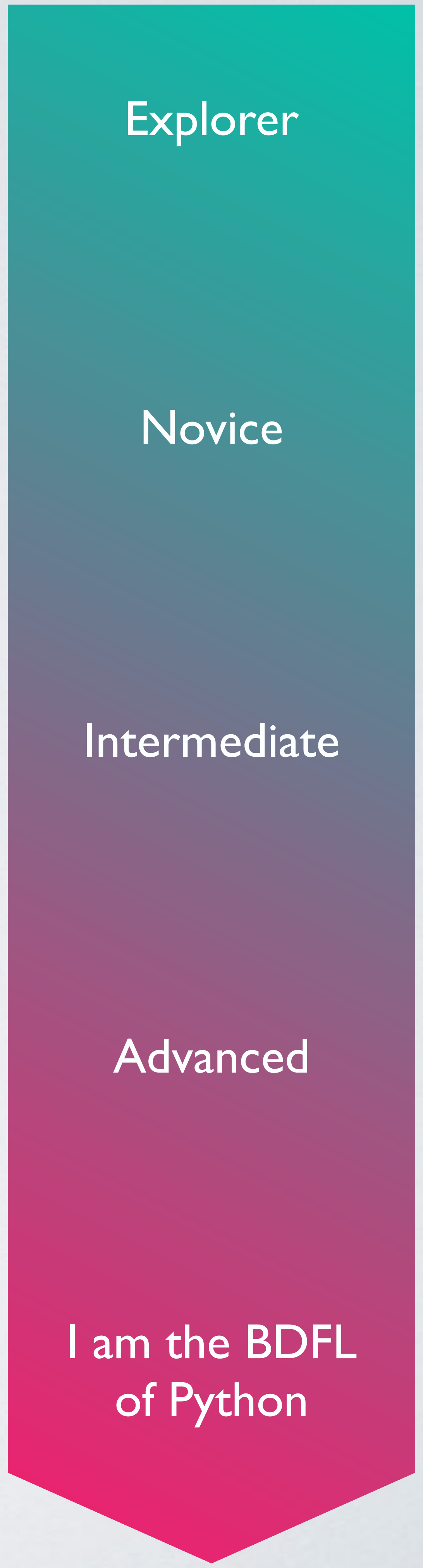
YOUR JOURNEY THROUGH PYTHON?

(JUST A VERY ROUGH GUESS, NOT A MEAN GAME)

Raise your hand and keep it up until you answer a question with "no" (means you're out).

- Have you ever launched the Python interpreter?
- Wrote for/while-loops or if/else statements?
- ...your own functions?
- ...classes?
- ...list/dict/set comprehensions?
- Do you know what a generator is?
- Have you ever implemented a decorator?
- ...a metaclass?
- ...a C-extension?
- Do you know and can you explain the output of the following line for Python?

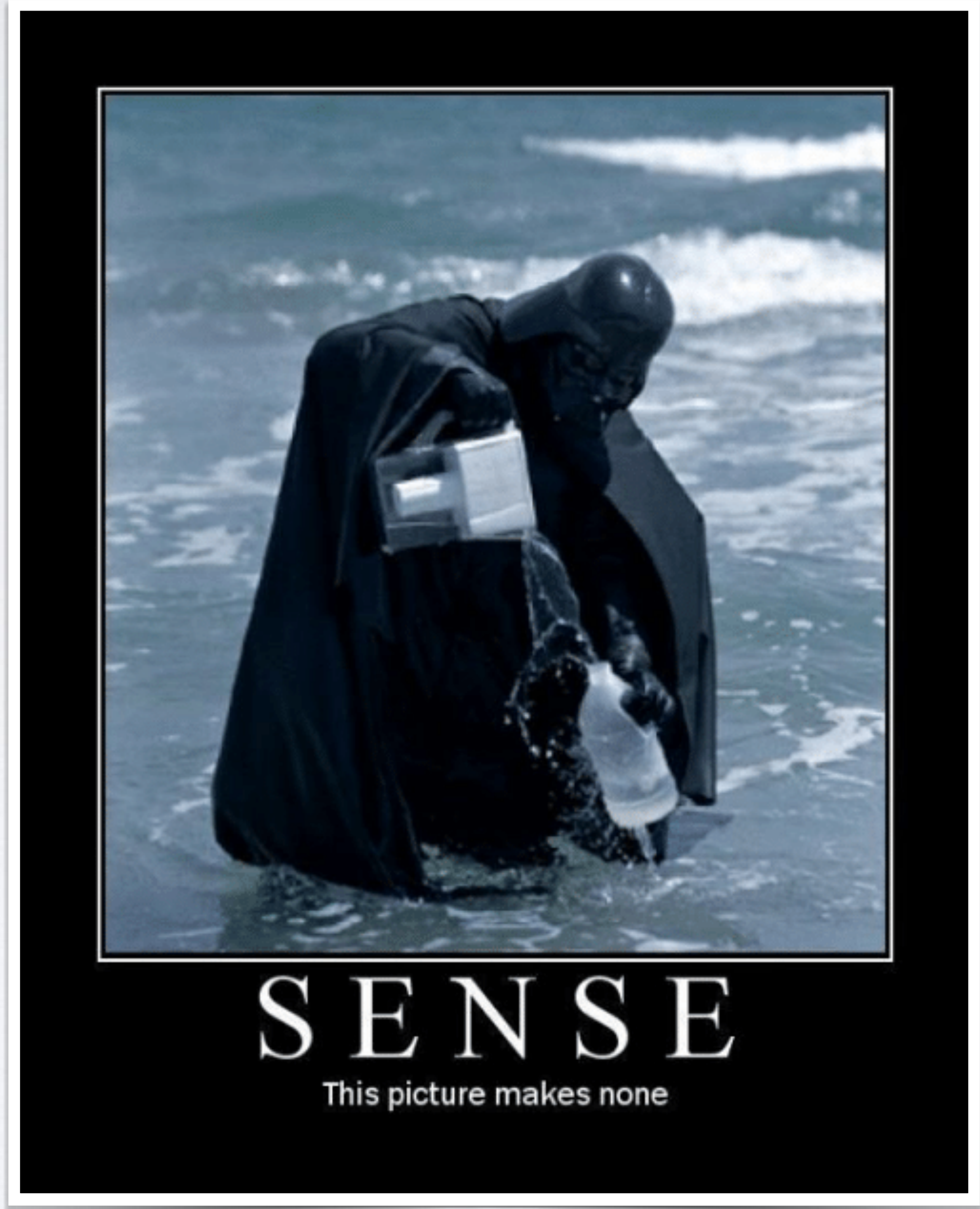
```
print(5 is 7 - 2, 300 is 302 - 2)
```



ANSWER TO

`print(5 is 7 - 2, 300 is 302 - 2)`

Python 2.7: True, False
Python 3.6: True, False
Python 3.7: True, True
Python 3.8: True, True,
and warnings ...
Python 3.9: True, True,
and warnings ...
Python 3.10: True, True,
and warnings ...



EXPLANATION OF

`print(5 is 7 - 2, 300 is 302 - 2)`

```
PyObject* PyLong_FromLong(long v)
Return value: New reference.
Return a new PyLongObject object from v,
or NULL on failure.
```

cpython/Include/internal/**pycore_interp.h**

```
/* Small integers are preallocated in this array so that they
   can be shared.
   The integers that are preallocated are those in the range
   -_PY_NSMALLNEGINTS (inclusive) to _PY_NSMALLPOSINTS (not inclusive).
*/
PyLongObject* small_ints[_PY_NSMALLNEGINTS + _PY_NSMALLPOSINTS];
```

The current implementation keeps an array of integer objects for all **integers between -5 and 256**, when you create an int in that range you actually just get back a reference to the existing object.

"is" is an operator which checks if **two objects are identical**: "x is y" is true iff x and y are pointing to the same object.

In Python 3.7+ the constant folding is moved from the peephole optimiser to the new AST optimiser, which effectively avoids the extra allocation. (https://github.com/python/cpython/commit/7ea143ae795a9fd57eaccf490d316bdc13ee9065)

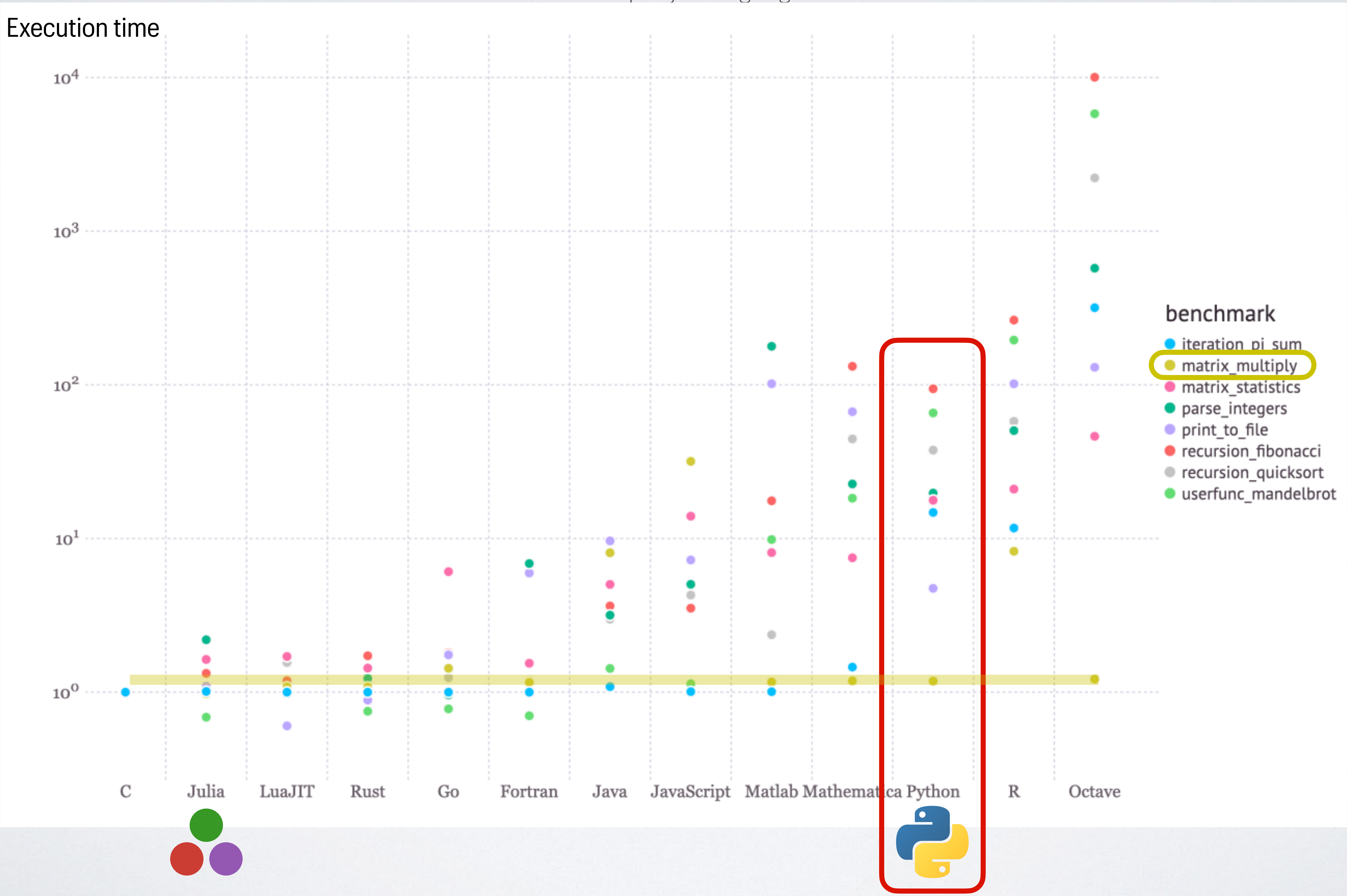
WHY IS PYTHON SO POPULAR (FOR SCIENCE)?

- **Ease of use** – scientists don't know/want how to program
- **Readable** code – source code is more often read than written
- **Interactive** workflow
- Lots of **scientific libraries** (and machine learning is everywhere)
- **Batteries included**: tons of (built-in) useful supplementary functionalities
- General purpose language so that scientists can **focus on a single language to rule them all**...
- ...**can they?**

julia

PERFORMANCE OF LANGUAGES

Microbenchmarks from https://julialang.org/benchmarks/



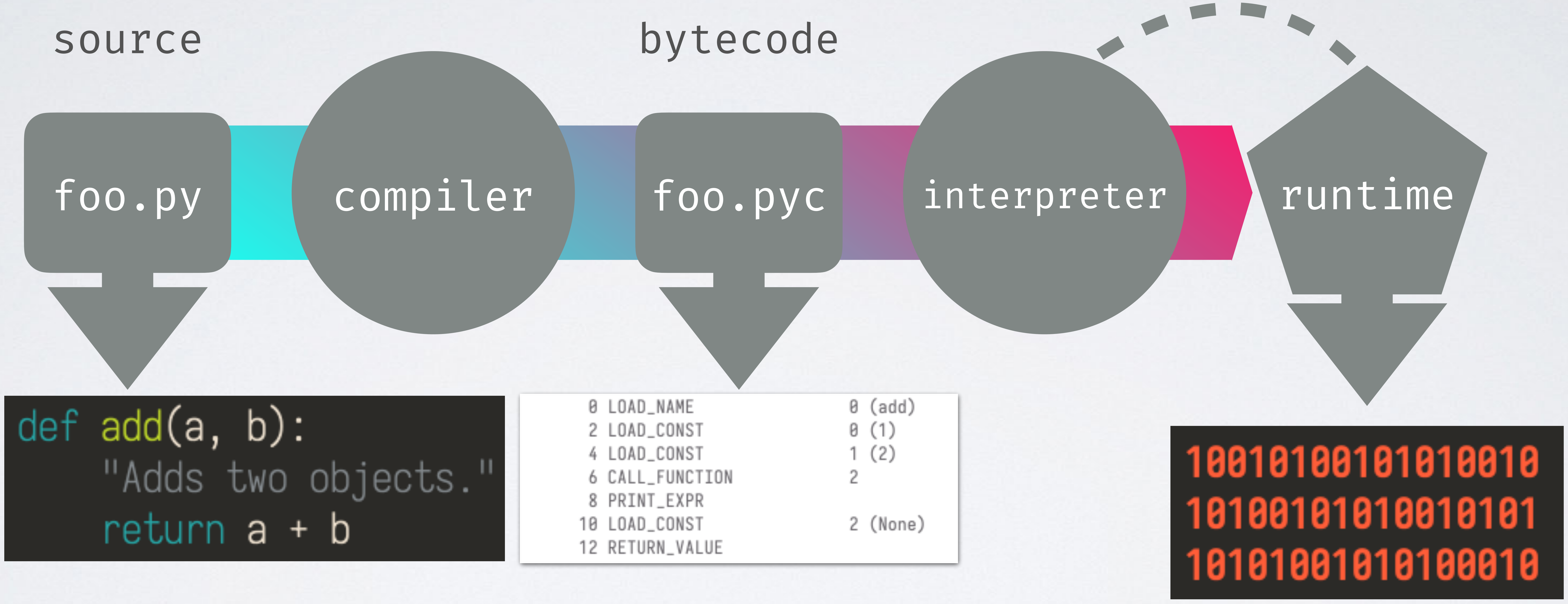
PYTHON INTERNALS

TO UNDERSTAND THE PERFORMANCE ISSUES

CPYTHON

- **Python** (in contrast to other languages like C, Julia) has **no formal specification**
- The **Python Language Reference** is written in **English**: https://docs.python.org/3.9/reference
- **CPython** is the **reference implementation** which contains implementation details which are not part of the language (e.g. GC with reference counting)
- Python is defined **partly** by the **Python Language Reference** and its main implementation **CPython**
- There are several other implementations: **PyPy**, **Jython** (stuck at Python 2.7), **IronPython** (2.7 and 3.4), etc.
- From now on, with "**Python**" we refer to **CPython**

FROM SOURCE TO RUNTIME



Recommended reading (**Victor Skvortsov - Python behind the scenes**): https://tenthousandmeters.com/blog/python-behind-the-scenes-1-how-the-cpython-vm-works/

PYTHON BYTECODE

print(5 is 7 - 2, 300 is 302 - 2)

Python 3.6

True, False

```
import dis; dis.dis(compile('print(5 is 7 - 2, 300 is 302 - 2)', '', 'single'))
  1           0 LOAD_NAME                0 (print)
              2 LOAD_CONST               0 (5)
              4 LOAD_CONST               6 (5)
              6 COMPARE_OP               8 (is)
              8 LOAD_CONST               3 (300)
             10 LOAD_CONST               7 (300)
             12 COMPARE_OP               8 (is)
             14 CALL_FUNCTION            2
             16 PRINT_EXPR
             18 LOAD_CONST               5 (None)
             20 RETURN_VALUE
```

co_consts[i]

Python 3.7

True, True

```
import dis; dis.dis(compile('print(5 is 7 - 2, 300 is 302 - 2)', '', 'single'))
  1           0 LOAD_NAME                0 (print)
              2 LOAD_CONST               0 (5)
              4 LOAD_CONST               0 (5)
              6 COMPARE_OP               8 (is)
              8 LOAD_CONST               1 (300)
             10 LOAD_CONST               1 (300)
             12 COMPARE_OP               8 (is)
             14 CALL_FUNCTION            2
             16 PRINT_EXPR
             18 LOAD_CONST               2 (None)
             20 RETURN_VALUE
```

co_consts[i]

THE TYPE OF A `PyObject`

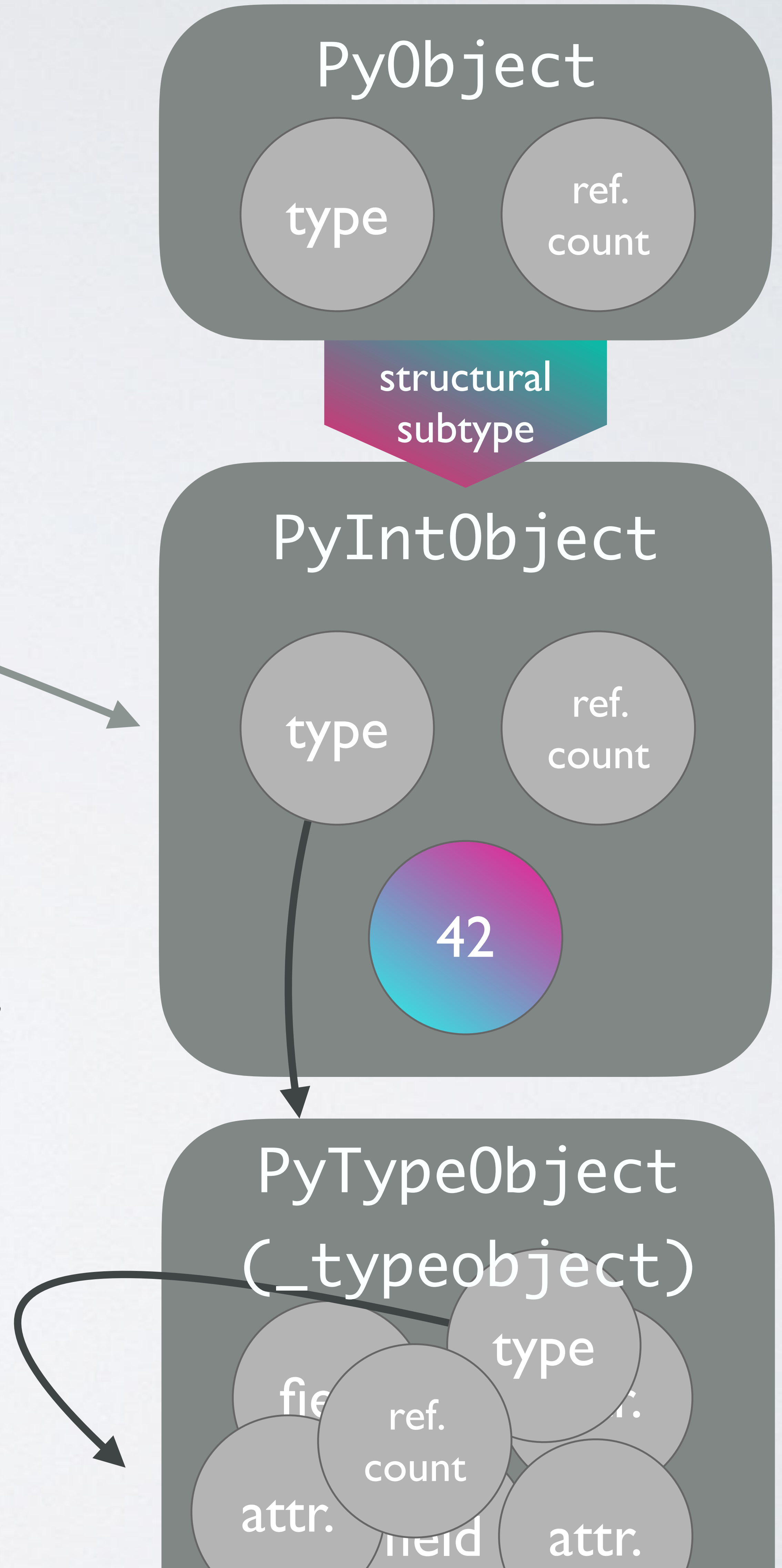
“An object has a ‘type’ that determines what it represents and what kind of data it contains.
An object’s type is fixed when it is created. Types themselves are represented as objects. The type itself has a type pointer pointing to the object representing the type ‘type’, which contains a pointer to itself!”

— object.h

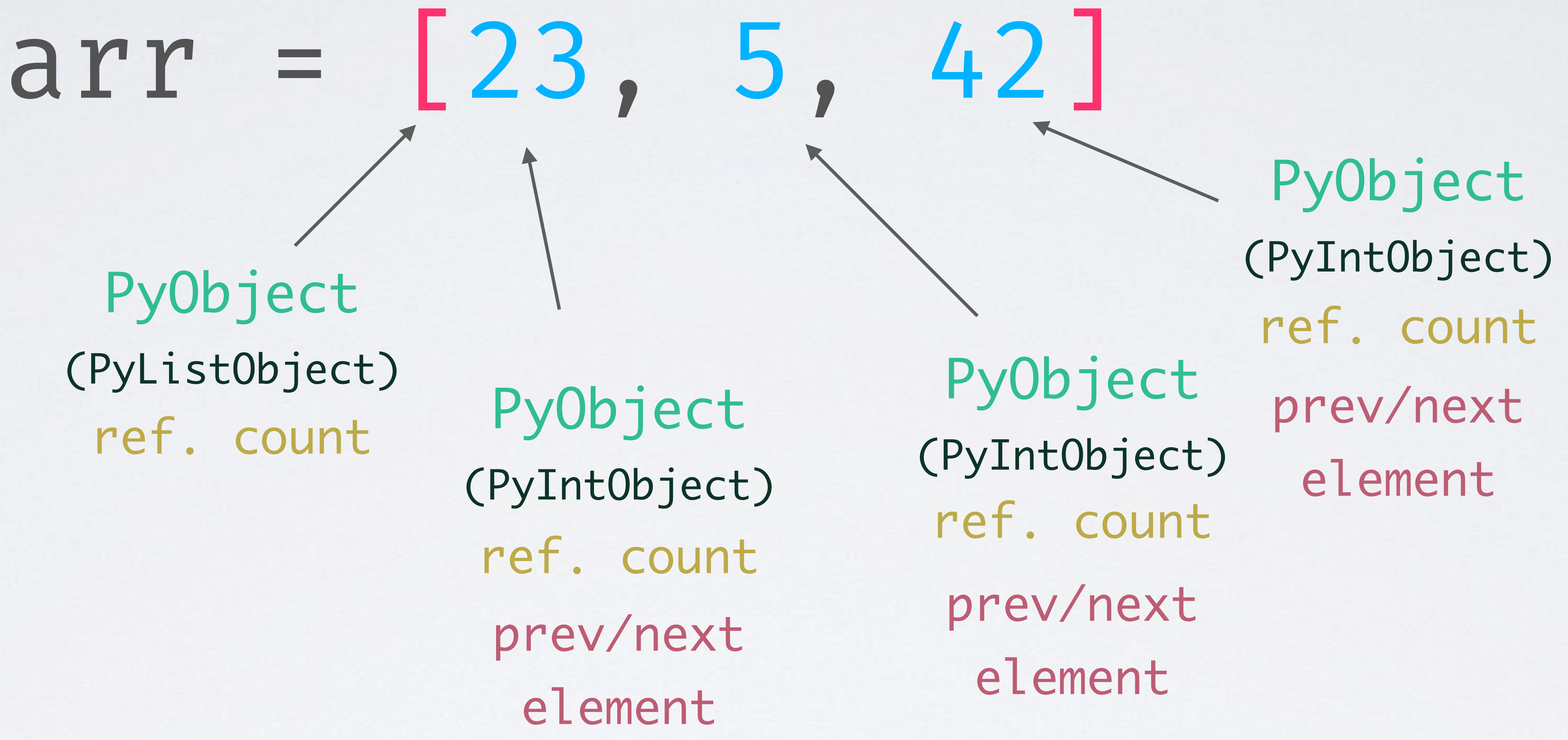
DATA IN PYTHON

- Every piece of data is a PyObject

```
>>> dir(42)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__',
'__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length',
'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real',
'to_bytes']
```



DATA IN PYTHON



- **Lower limit** for the **size** in **bytes** (on a 64bit system), oversimplified:
  8+8+8 (list) + 3*(8+8+8+8+8+8) (integer) = **168**
- **"Technically"** it's only **24 bytes** of information if we see it as an array of integers

YOUR BEST FRIEND AND WORST ENEMY:

GIL - Global Interpreter Lock

- The **GIL** prevents parallel execution of (Python) bytecode
- It is a **very simple solution to memory safety** (Python uses reference counting, which can cause trouble with race conditions and deadlocks)
- Even though Python has **real threads**, they **never execute (byte)code at the same time**
- **Context switching** between threads creates overhead (the user cannot control thread-priority)
- Threads perform pretty badly on **CPU bound** tasks
- They do a great job speeding up **I/O heavy** tasks

THREADS AND CPU BOUND TASKS

single thread:

```
>>> N = 100_000_000

>>> def count(n):
...     while n != 0:
...         n -= 1
...

>>> %time count(N)
CPU times: user 2.97 s, sys: 6.28 ms, total: 2.98 s
Wall time: 2.98 s
```

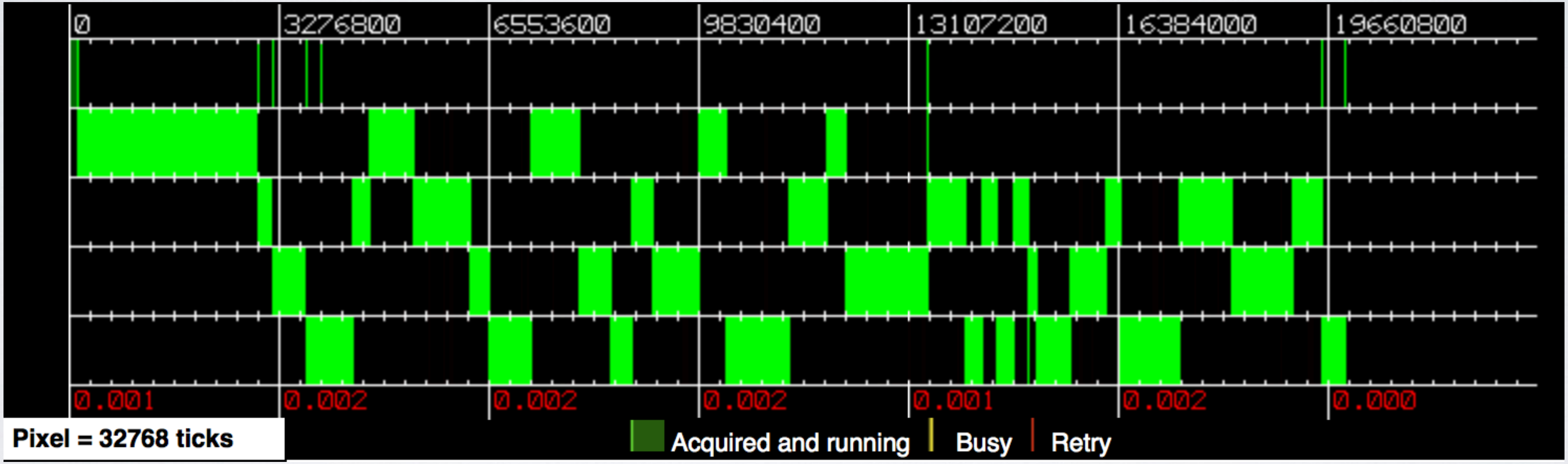
two threads:

```
>>> from threading import Thread

>>> def count_threaded(n):
...     t1 = Thread(target=count, args=(N/2,))
...     t2 = Thread(target=count, args=(N/2,))
...     t1.start()
...     t2.start()
...     t1.join()
...     t2.join()
...

>>> %time count_threaded(N)
CPU times: user 3.18 s, sys: 15.3 ms, total: 3.19 s
Wall time: 3.22 s
```

This is probably not really what you expected…

THREADS FIGHTING FOR THE GIL

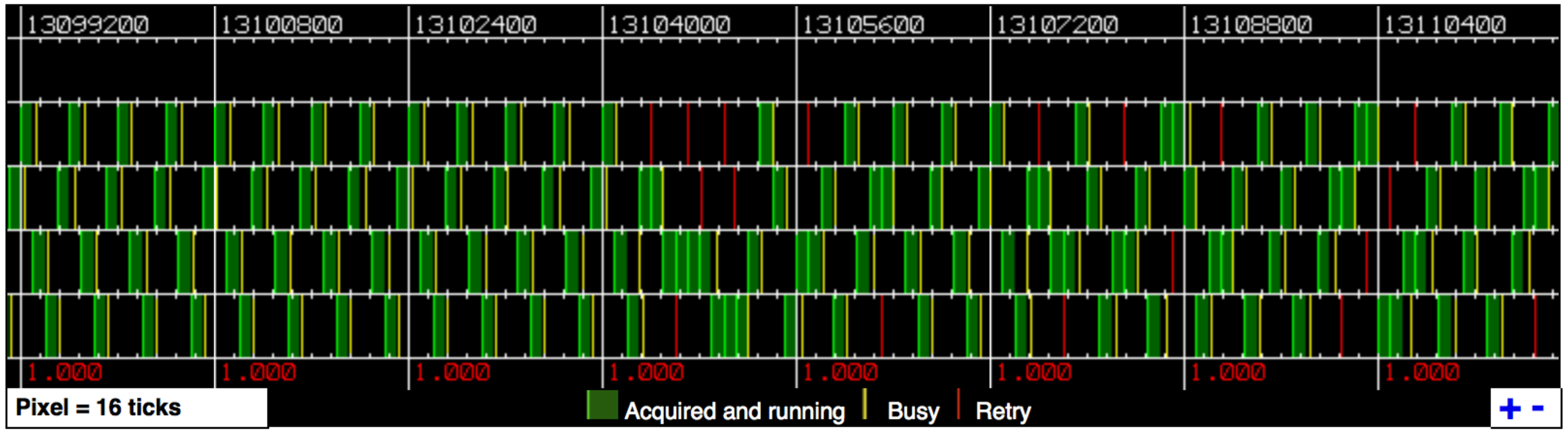
OS X: 4 threads on 1 CPU (Python 2.6)



By David M Beazley @PyCON'2010: http://dabeaz.com/GIL/gilvis

THREADS FIGHTING FOR THE GIL

OS X: 4 threads on 4 CPUs (Python 2.6)



By David M Beazley @PyCON'2010: http://dabeaz.com/GIL/gilvis

OK, HUGE OVERHEAD FOR EVERY SINGLE OBJECT, NO REAL PARALLEL EXECUTION OF CODE...

HOW COULD PYTHON EVER COMPETE WITH ALL THOSE SUPER FAST C/C++/FORTRAN SOFTWARE?

C-EXTENSIONS AND INTERFACES TO "FAST" LANGUAGES LIKE C/C++/FORTRAN!

THOSE CAN RELEASE THE GIL AND DO THE HEAVY STUFF IN THE BACKGROUND.

A DUMB SPEED COMPARISON

Calculating the mean of 1000000 randomly generated numbers.

pure Python

```
>>> def mean(numbers):
...     return sum(numbers) / len(numbers)
...

>>> numbers = list(range(1_000_000))

>>> %timeit mean(numbers)
2.98 ms ± 21.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

NumPy (~16x faster)

```
>>> import numpy as np

>>> numbers = np.random.random(1_000_000)

>>> %timeit np.mean(numbers)
190 µs ± 1.35 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Numba (~3x faster)

```
>>> @nb.njit
... def numba_mean(numbers):
...     s = 0
...     N = len(numbers)
...     for i in range(N):
...         s += numbers[i]
...     return s/N
...

>>> numbers = np.random.random(1_000_000)

>>> %timeit numba_mean(numbers)
1.03 ms ± 35.2 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Julia (~16x faster)

```
julia> numbers = rand(1_000_000);

julia> using BenchmarkTools

julia> @benchmark mean($numbers)
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  182.292 µs … 230.167 µs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     183.166 µs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   183.927 µs ±   2.457 µs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
  182 µs        Histogram: log(frequency) by time        195 µs <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

CRAZY LLVM COMPILER OPTIMISATIONS

Summing up consecutive numbers from 0 to N=100,000,000

pure Python

```
>>> def simple_sum(N):
...     s = 0
...     for i in range(1, N+1):
...         s += i
...     return s
...

>>> %timeit simple_sum(N)
3.02 s ± 56.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

NumPy (~75x faster)

```
>>> np_numbers = np.arange(1, N+1, dtype=np.int64)

>>> %timeit np.sum(np_numbers)
38 ms ± 27.8 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Numba (~34000000x faster)

```
>>> @nb.njit
... def simple_sum(N):
...     s = 0
...     for i in range(1, N+1):
...         s += i
...     return s
...

>>> %timeit simple_sum(N)
85.6 ns ± 0.602 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

⇒ N*(N+1)/2 = 5000000050000000

Julia (~150000000x faster)

```
julia> function simple_sum(N)
           s = 0
           for i ∈ 1:N
               s += i
           end
           s
       end
simple_sum (generic function with 1 method)

julia> @benchmark simple_sum(N)
BenchmarkTools.Trial: 10000 samples with 997 evaluations.
 Range (min … max):  19.015 ns … 489.468 ns  ┊ GC (min … max): 0.00% …
 Time  (median):     19.267 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   20.046 ns ±   6.731 ns  ┊ GC (mean ± σ):  0.46% ±
```

```
 19 ns          Histogram: log(frequency) by time        23.9 ns <

 Memory estimate: 16 bytes, allocs estimate: 1.
```

```
julia> @code_native debuginfo=:none simple_sum(1_000_000_000)
        .section        __TEXT,__text,regular,pure_instructions
        .build_version macos, 13, 0
        .globl  _julia_simple_sum_1025          ; -- Begin function
        .p2align        2
_julia_simple_sum_1025:                 ; @julia_simple_sum_1025
        .cfi_startproc
; %bb.0:                                ; %top
        cmp     x0, #1                          ; =1
        b.lt    LBB0_2
; %bb.1:                                ; %L18.preheader
        bic     x8, x0, x0, asr #63
        sub     x9, x8, #1                      ; =1
        sub     x10, x8, #2                     ; =2
        mul     x11, x9, x10
        umulh   x9, x9, x10
        extr    x9, x9, x11, #1
        add     x8, x9, x8, lsl #1
        sub     x0, x8, #1                      ; =1
        ret
LBB0_2:
        mov     x0, #0
        ret
        .cfi_endproc
```

"WHY WE CAN'T HAVE NICE THINGS"

AKA VECTORISE FOR YOUR LIFE!

CLASSES TO STRUCTURE CODE

- **Classes** and cleverly designed **class hierarchies** make it **easier** to **understand code, structure and architecture**
- It's **tempting** to take "**object-oriented**" seriously and create a **detailed model** of your application
- The **more fine-grained** it gets, the bigger the **impact on performance**
- **Imagine** a **neutrino detector** which consists of PMTs to measure photons, each **triggered signal represented by** an instance of the **"Hit"-class**:

```
>>> class Hit:
...     def __init__(self, pmt_id, time, tot, triggermask):
...         self.pmt_id = pmt_id
...         self.time = time
...         self.tot = tot
...         self.triggermask = triggermask
...
```

CLASSES TO STRUCTURE CODE

- **Now imagine** that a KM3NeT detector (the experiment I work on) produces **several thousands of hits per event**, with a trigger rate of more than **200Hz**
- It's certainly a **bad idea** to have an **array of thousands of hits** with instances of this class due to the **large overhead**

```
>>> class Hit:
...     def __init__(self, pmt_id, time, tot, triggerm
...         self.pmt_id = pmt_id
...         self.time = time
...         self.tot = tot
...         self.triggermask = triggermask
...
```

```
>>> hits
[<__main__.Hit at 0x10846e430>,
 <__main__.Hit at 0x108ba2100>,
 <__main__.Hit at 0x108422490>,
 <__main__.Hit at 0x1085d66a0>,
 <__main__.Hit at 0x1085d6b80>,
 <__main__.Hit at 0x1085d61c0>,
 <__main__.Hit at 0x108c366a0>,
 <__main__.Hit at 0x108c36850>,
 <__main__.Hit at 0x108c36460>,
 <__main__.Hit at 0x108c36070>,
 <__main__.Hit at 0x108c362b0>,
```

STRUCT OF ARRAYS VS ARRAY OF STRUCTS

- **Instead** of having an **array of hits**, use a **single instance with multiple arrays** of the same size, one for each field

- NumPy offers the "**recarray**" functionality for convenient attribute access

```
>>> hit_dtype = np.dtype([("pmt_id", np.uint32), ("time", np.float64), ("tot", np.uint8), ("triggermask", np.uint64)])

>>> hits = np.array([(1, 2, 3, 4), (5, 6, 7, 8), (9, 10, 11, 12)], dtype=hit_dtype).view(np.recarray)

>>> hits.time
array([ 2.,  6., 10.])

>>> hits.tot
array([ 3,  7, 11], dtype=uint8)

>>> hits[2].time
10.0
```

STRUCT OF ARRAYS VS ARRAY OF STRUCTS

- **Instead** of having an **array of hits**, use a **single instance with multiple arrays** of the same size, one for each field
- NumPy offers the "**recarray**" functionality for convenient attribute access
- Similar to the way "**Panda DataFrames**" work (columns)

```
>>> hit_dtype = np.dtype([("pmt_id", np.uint32), ("time", np.float64), ("tot", np.uint8), ("triggermask", np.uint64)])

>>> hits = np.array([(1, 2, 3, 4), (5, 6, 7, 8), (9, 10, 11, 12)], dtype=hit_dtype).view(np.recarray)

>>> hits.time
array([ 2.,  6., 10.])

>>> hits.tot
array([ 3,  7, 11], dtype=uint8)

>>> hits[2].time
10.0
```

```julia
"""
$(SIGNATURES)

Return only triggered hits.
"""
triggered(hits::Vector{T}) where {T<:AbstractHit} = filter(h->h.trigger_mask > 0, hits)
```

julia

NumPy

"NUMERIC" IN 1995, "NUMPY" IN 2006 (JUST KICKED IN WHEN I STARTED STUDYING PHYSICS)

NUMPY

NumPy is the **fundamental** package for **scientific computing** with **Python**.

- gives us a powerful **N-dimensional array** object: **ndarray**
- **broadcasting** functions
- tools for **integrating C/C++ and Fortran**
- **linear algebra**, **Fourier** transform and **random** number capabilities
- **most of the scientific libraries build upon NumPy**

NUMPY: ndarray

```
>>> a = np.arange(6)

>>> a
array([0, 1, 2, 3, 4, 5])
```

```
>>> a.dtype
dtype('int64')

>>> a.ndim
1

>>> a.shape
(6,)
```

Contiguous array in memory with a **fixed type**,
no pointer madness!

C/Fortran compatible memory layout,
so they can be passed to those
without any further efforts.

NUMPY: ARRAY OPERATIONS AND ufuncs

```
a * 23
```

```
array([  0,  23,  46,  69,  92, 115])
```

```
a**a
```

```
array([   1,    1,    4,   27,  256, 3125])
```

easy and intuitive element-wise operations

a **ufunc**, which can operate both on scalars and arrays (element-wise)

```
np.exp(a)
```

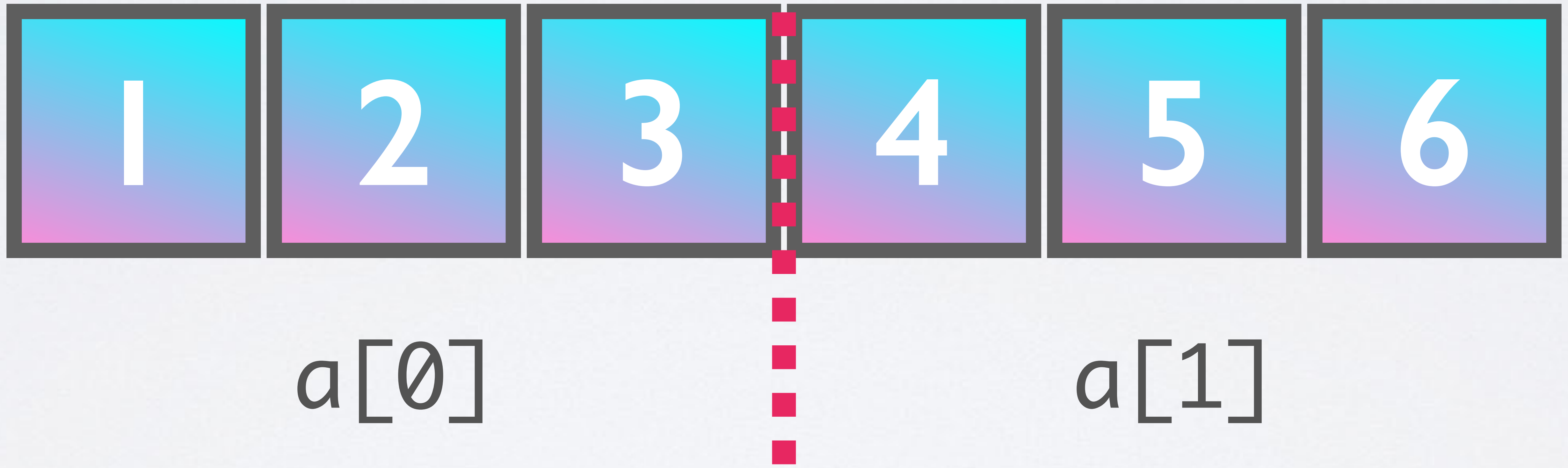
```
array([   1.        ,    2.71828183,    7.3890561 ,   20.08553692,
         54.59815003,  148.4131591 ])
```

RESHAPING ARRAYS

```
a = np.arange(6)
a
```

```
array([0, 1, 2, 3, 4, 5])
```

ndim: 1
shape: (6,)




```
a.reshape(2, 3)
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

No rearrangement of the elements in memory but **setting the iterator limits** internally!

RESHAPING ARRAYS IS CHEAP

```
>>> a = np.arange(10_000_000)

>>> %timeit a.reshape(100, 5_000, 20)
131 ns ± 2.86 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

OK, ALL FINE, NUMPY TO RULE THEM ALL?

NO...

"HIDDEN" ALLOCATIONS

```python
import resource
import sys
import numpy as np


def peak_memory_usage():
    """Return peak memory usage in MB"""
    mem = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
    factor_mb = 1 / 1024
    if sys.platform == "darwin":
        factor_mb = 1 / (1024 * 1024)
    return mem * factor_mb


def main():
    print(f"All libraries loaded: {peak_memory_usage()} MB")

    N = 1_000_000

    a = np.random.rand(N)
    print(f"'a' allocated: {peak_memory_usage()} MB")

    b = np.random.random(N)
    print(f"'b' allocated: {peak_memory_usage()} MB")

    c = 2*a + 3*b
    print(f"'c' calculated: {peak_memory_usage()} MB")


if __name__ == "__main__":
    main()
```

```
tamasgal@silentbox:tmp/missing_python_intro
00:16:38 > python3 hidden_allocations.py
All libraries loaded: 28.578125 MB
'a' allocated: 36.28125 MB
'b' allocated: 43.921875 MB
'c' calculated: 67.59375 MB
```

1000000 float64s are 8 MB

+8 MB after "a" is allocated
+8 MB after "b" is allocated
+24 MB = 8+8+8 MB after "c" is calculated

$$c = \underbrace{\underline{2*a} + \underline{3*b}}$$

AVOIDING UNNEEDED ALLOCATIONS

```python
import resource
import sys
import numpy as np


def peak_memory_usage():
    """Return peak memory usage in MB"""
    mem = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
    factor_mb = 1 / 1024
    if sys.platform == "darwin":
        factor_mb = 1 / (1024 * 1024)
    return mem * factor_mb


def main():
    print(f"All libraries loaded: {peak_memory_usage()} MB")

    N = 1_000_000

    a = np.random.rand(N)
    print(f"'a' allocated: {peak_memory_usage()} MB")

    b = np.random.random(N)
    print(f"'b' allocated: {peak_memory_usage()} MB")

    np.multiply(a, 2, out=a)
    print(f"'2*a' calculated: {peak_memory_usage()} MB")

    np.multiply(b, 3, out=b)
    print(f"'3*b' calculated: {peak_memory_usage()} MB")

    c = np.add(a, b)
    print(f"'c' calculated: {peak_memory_usage()} MB")

if __name__ == "__main__":
    main()
```

```
tamasgal@silentbox:tmp/missing_python_intro
14:53:23 > python3 numpy_out.py
All libraries loaded: 28.703125 MB
'a' allocated: 36.421875 MB
'b' allocated: 44.0625 MB
'2*a' calculated: 44.0625 MB
'3*b' calculated: 44.0625 MB
'c' calculated: 51.703125 MB
```

Reusing a and **b** with **"out="**

`a = a + b` vs. **`np.add(a, b, out=a)`**

Only a single **8 MB** allocation is needed **for c (mutating the a and b!)**

MEMORY AND PERFORMANCE PROBLEMS

- The code **may work for small sample** files
- Quickly **escalates** the **peak memory usage** if not handled with care
- Users **blindly perform** chains of **transformations**
- Trigger **redundant loops** instead using kernels and a single loop
- **Unnecessary memory allocations** for temporary data
- **Pandas** e.g. uses NumPy behind the scenes and is a **constant source** of **scaling issues** (hello Pandas concat/append/join/etc.)

MASKING/SLICING IS THE ROOT OF ALL EVIL

- **Masks** are often used in NumPy to **reduce data**
- **If Python loops were fast**, we could simply **iterate through hits** (array of structs) and select them based on a condition **=> minimal memory footprint, single loop**
- **Potential extra loops** and **allocations** which might **blow the memory**
- **Instead, masks are tempting** due do the nice and easy **syntactic sugar**

```
mask = (hits.tot > 23) & \
       (hits.time < 155667) & \
       (hits.time > 155000) & \
       (hits.triggermask == 0xFEE0101)

selected_hits = hits[mask]
```

WRITE YOUR CODE SO THAT IT ITERATES THROUGH A CONFIGURABLE SIZE OF CHUNKS!

(FOR EXAMPLE **X EVENTS** INSTEAD OF **FILE-BY-FILE**)

NUMEXPR

ROUTINES FOR THE FAST EVALUATION OF ARRAY EXPRESSIONS ELEMENT-WISE
BY USING A VECTOR-BASED VIRTUAL MACHINE.

NUMEXPR USAGE EXAMPLE

```
import numpy as np
import numexpr as ne

a = np.arange(5)
b = np.linspace(0, 2, 5)

ne.evaluate("a**2 + 3*b”)
```

```
array([  0. ,   2.5,   7. ,  13.5,  22. ])
```

NUMEXPR SPEED-UP

```
a = np.random.random(1000000)
```

NumPy:

```
2 * a**3 - 4 * a**5 + 6 * np.log(a)
```

82.4 ms ± 1.88 ms per loop

Numexpr with 4 threads:

```
ne.set_num_threads(4)

ne.evaluate("2 * a**3 - 4 * a**5 + 6 * log(a)")
```

7.85 ms ± 103 µs per loop

~10x faster (with 4 threads)

...minimal memory footprint

NUMEXPR - SUPPORTED OPERATORS

- Logical operators: **&, |, ~**
- Comparison operators:
  <, <=, ==, !=, >=, >
- Unary arithmetic operators: **-**
- Binary arithmetic operators:
  +, -, *, /, **, %, <<, >>

NUMEXPR - SUPPORTED FUNCTIONS

- **where**(bool, number1, number2): number -- number1 if the bool condition is true, number2 otherwise.
- {**sin**,**cos**,**tan**}(float|complex): float|complex -- trigonometric sine, cosine or tangent.
- {**arcsin**,**arccos**,**arctan**}(float|complex): float|complex -- trigonometric inverse sine, cosine or tangent.
- **arctan2**(float1, float2): float -- trigonometric inverse tangent of float1/float2.
- {**sinh**,**cosh**,**tanh**}(float|complex): float|complex -- hyperbolic sine, cosine or tangent.
- {**arcsinh**,**arccosh**,**arctanh**}(float|complex): float|complex -- hyperbolic inverse sine, cosine or tangent.
- {**log**,**log10**,**log1p**}(float|complex): float|complex -- natural, base-10 and log(1+x) logarithms.
- {**exp**,**expm1**}(float|complex): float|complex -- exponential and exponential minus one.
- **sqrt**(float|complex): float|complex -- square root.
- **abs**(float|complex): float|complex -- absolute value.
- **conj**(complex): complex -- conjugate value.
- {**real**,**imag**}(complex): float -- real or imaginary part of complex.
- **complex**(float, float): complex -- complex from real and imaginary parts.
- **contains**(str, str): bool -- returns True for every string in `op1` that contains `op2`.
- **sum**(number, axis=None): Sum of array elements over a given axis. Negative axis are not supported.
- **prod**(number, axis=None): Product of array elements over a given axis. Negative axis are not supported.

AVOIDING EXTRA ALLOCATIONS WITH NUMEXPR

```python
import resource
import sys
import numexpr as ne
import numpy as np


def peak_memory_usage():
    """Return peak memory usage in MB"""
    mem = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
    factor_mb = 1 / 1024
    if sys.platform == "darwin":
        factor_mb = 1 / (1024 * 1024)
    return mem * factor_mb


def main():
    print(f"All libraries loaded: {peak_memory_usage()} MB")

    N = 1_000_000

    a = np.random.rand(N)
    print(f"'a' allocated: {peak_memory_usage()} MB")

    b = np.random.random(N)
    print(f"'b' allocated: {peak_memory_usage()} MB")

    c = ne.evaluate("2*a + 3*b")
    print(f"'c' calculated: {peak_memory_usage()} MB")


if __name__ == "__main__":
    main()
```

```
tamasgal@silentbox:tmp/missing_python_intro
00:37:25 > python3 numexpr_example.py
All libraries loaded: 29.40625 MB
'a' allocated: 37.109375 MB
'b' allocated: 44.75 MB
'c' calculated: 52.59375 MB
```

Only a single **8 MB** allocation
is needed **for c**

Awkward Array

BY JIM PIVARSKI (SCIKIT-HEP)

AWKARD ARRAY

MOTIVATION

- **NumPy arrays** are **rectangular** tables or tensors: **cannot express variable-length** structures
- **Tree-like data** (very common in HEP) is difficult to express with NumPy arrays -- in an efficient way
- **Speed** and **performance** are crucial
- Easy to use and interactive interfaces for commonly used operations like cuts and aggregations

AWKARD ARRAY

- Written in **Python** and **C++**
- Has **Numba** support to take it to the next level!
- Supports arbitrary **tree representations** with as many jagged/ragged structures as you need
- Offers lots of functions to work with **ragged/jagged** data

```
In [1]: import awkward as ak

In [2]: arr = ak.Array([[1,2,3], [4,5], [6,7,8,9]])

In [3]: arr
Out[3]: <Array [[1, 2, 3], [4, 5], [6, 7, 8, 9]] type='3 * var * int64'>

In [4]: arr[:,0]
Out[4]: <Array [1, 4, 6] type='3 * int64'>

In [5]: ak.mean(arr, axis=1)
Out[5]: <Array [2, 4.5, 7.5] type='3 * ?float64'>
```

AWKARD ARRAY

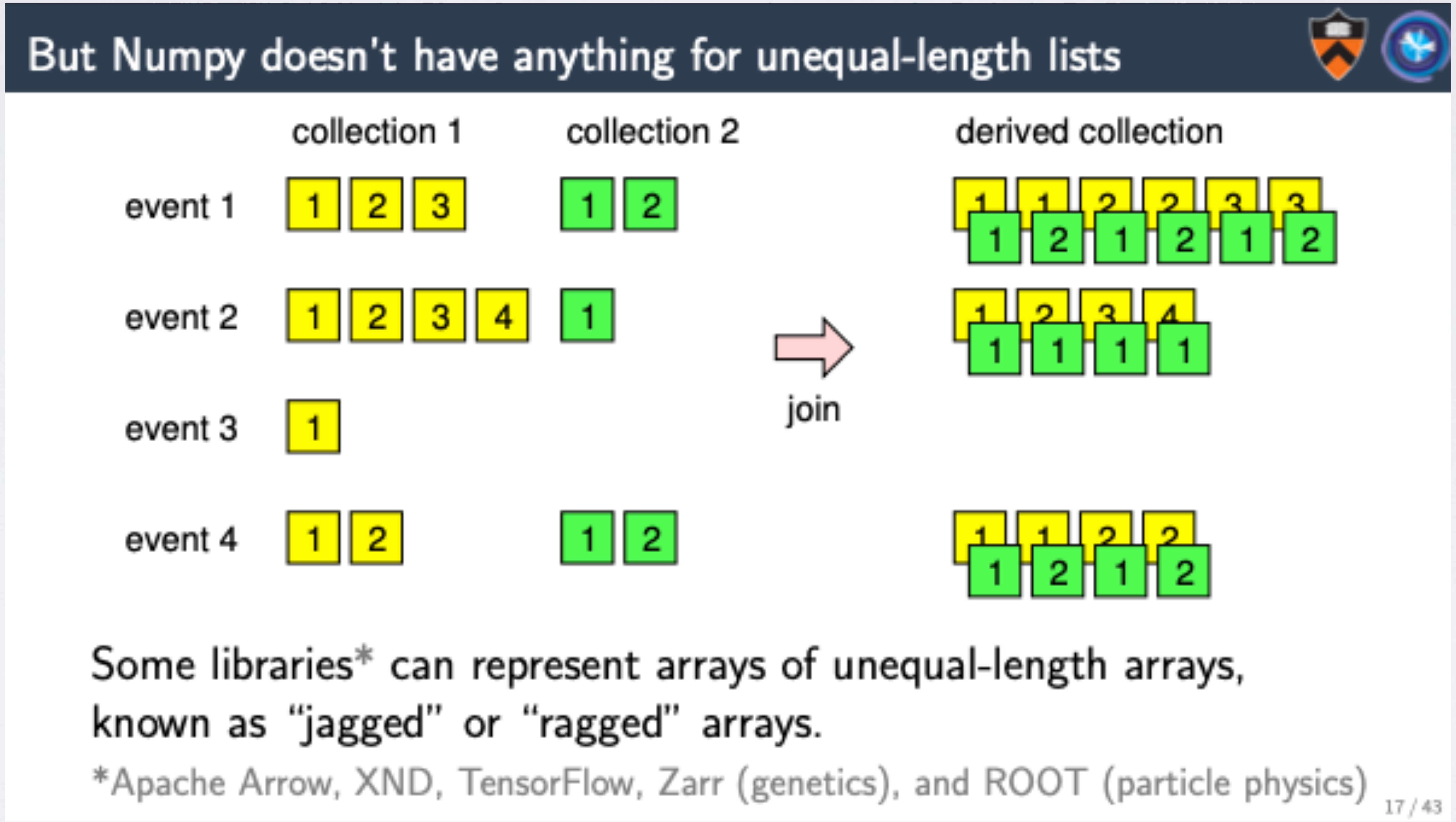
- All kinds of **multiply nested** structures are understood and "**type stable**"
- Allows **fancy indexing** over multiple nested elements
- **Database-like** operations
- Feels **like Pandas**, with awkwardly jagged arrays

```
In [1]: import awkward as ak

In [2]: arr = ak.Array([(True, 1), (False, 3), (False, 9)])

In [3]: arr
Out[3]: <Array [(True, 1), (False, 3), (False, 9)] type='3 * (bool, int64)'>
```

```
In [1]: import awkward as ak

In [2]: arr = ak.Array([{"pos_x": [3, 45, 65], "pos_y": [5, 4, 6]}, {"pos_x": [1, 3], "pos_y": [5, 6]}])

In [3]: arr
Out[3]: <Array [{pos_x: [3, 45, 65, ... pos_y: [5, 6]}] type='2 * {"pos_x": var * int64,...'>

In [4]: arr.pos_x
Out[4]: <Array [[3, 45, 65], [1, 3]] type='2 * var * int64'>
```

AWKWARD ARRAY

- A very nice **introduction** by **Jim** himself (just search for "**awkward array**" on YouTube)



- https://www.youtube.com/watch?v=2NxWpU7NArk

WHEN THINGS ARE GETTING MORE COMPLICATED THAN

2*A + 3*B

NUMBA

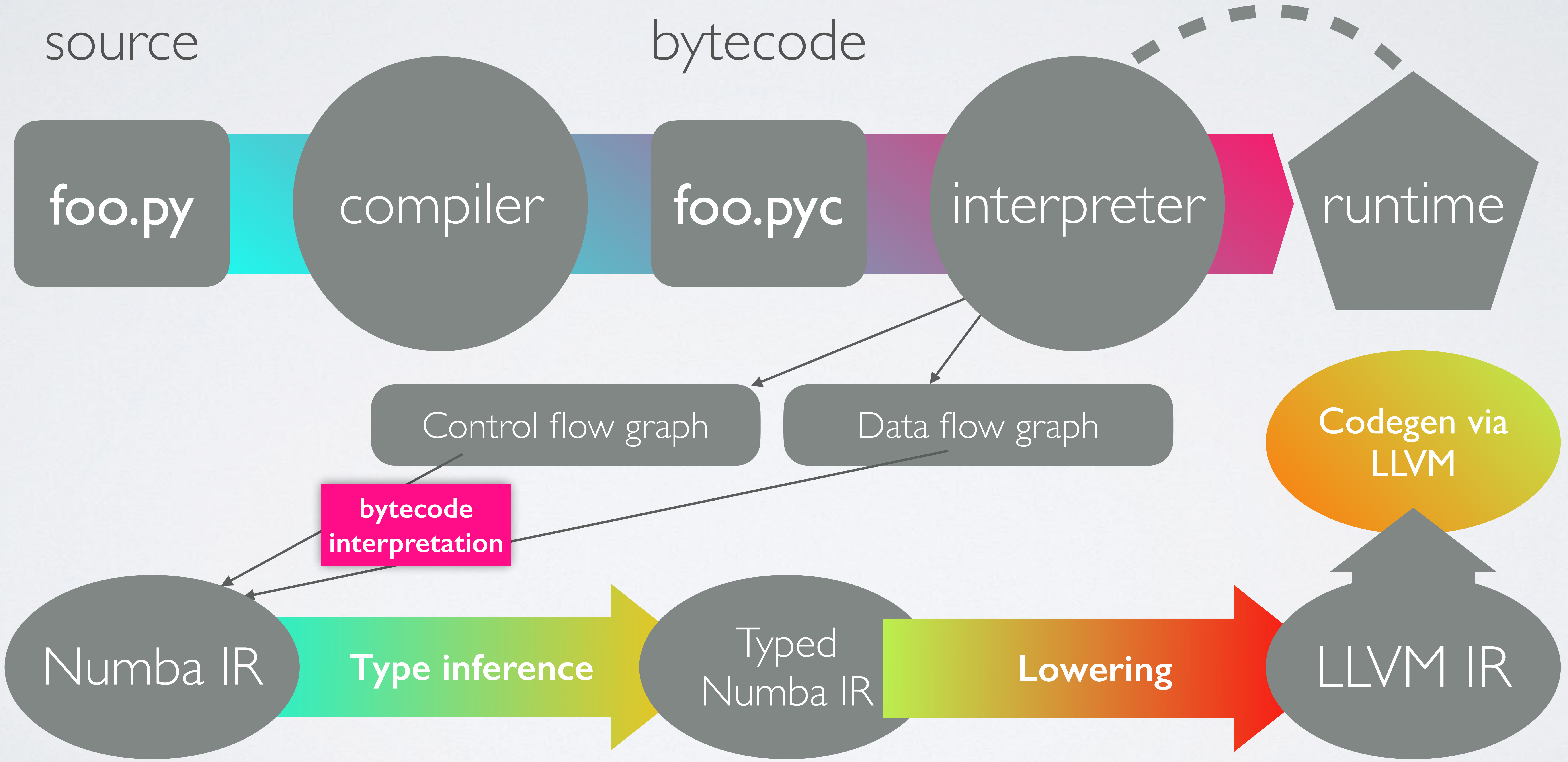
THE JIT (LLVM) COMPILER FOR PYTHON

NUMBA

Numba is a **compiler** for Python array and numerical functions that gives you the power to speed up code written directly in Python.

- uses **LLVM** to boil down pure Python code to **JIT optimised machine code**
- only **accelerates** selected **functions decorated** by yourself
- **native code** generation for **CPU** (default) and **GPU**
- **integration** with the **Python scientific software stack** (thanks to **NumPy**)
- runs side by side with regular Python code or third-party C extensions and libraries
- great **CUDA** support
- **N-core** scalability by releasing the GIL (beware: no protection from race conditions!)
- create **NumPy** `ufuncs` with the `@[gu]vectorize` decorator(s)
- **unfortunately: limited support of data structures**

FROM SOURCE TO RUNTIME



NUMBA JIT-EXAMPLE

```python
numbers = np.arange(1000000).reshape(2500, 400)
```

```python
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result
```

```python
@nb.jit
def sum2d_jit(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result
```

289 ms ± 3.02 ms per loop

2.13 ms ± 42.6 µs per loop

~135x faster, with a single line of code

NUMBA VECTORIZE-EXAMPLE

```python
a = np.arange(1000000, dtype='f8')
b = np.arange(1000000, dtype='f8') + 23
```

NumPy:

```python
np.abs(a - b) / (np.abs(a) + np.abs(b))
```

23 ms ± 845 µs per loop

Numba @vectorize:

```python
@nb.vectorize
def nb_rel_diff(a, b):
    return abs(a - b) / (abs(a) + abs(b))

rel_diff(a, b)
```

3.56 ms ± 43.2 µs per loop

~6x faster

PYTHON IS YOUR EVERYDAY **HAMMER**

IF YOU ARE HOLDING A HAMMER, EVERYTHING LOOKS LIKE A NAIL

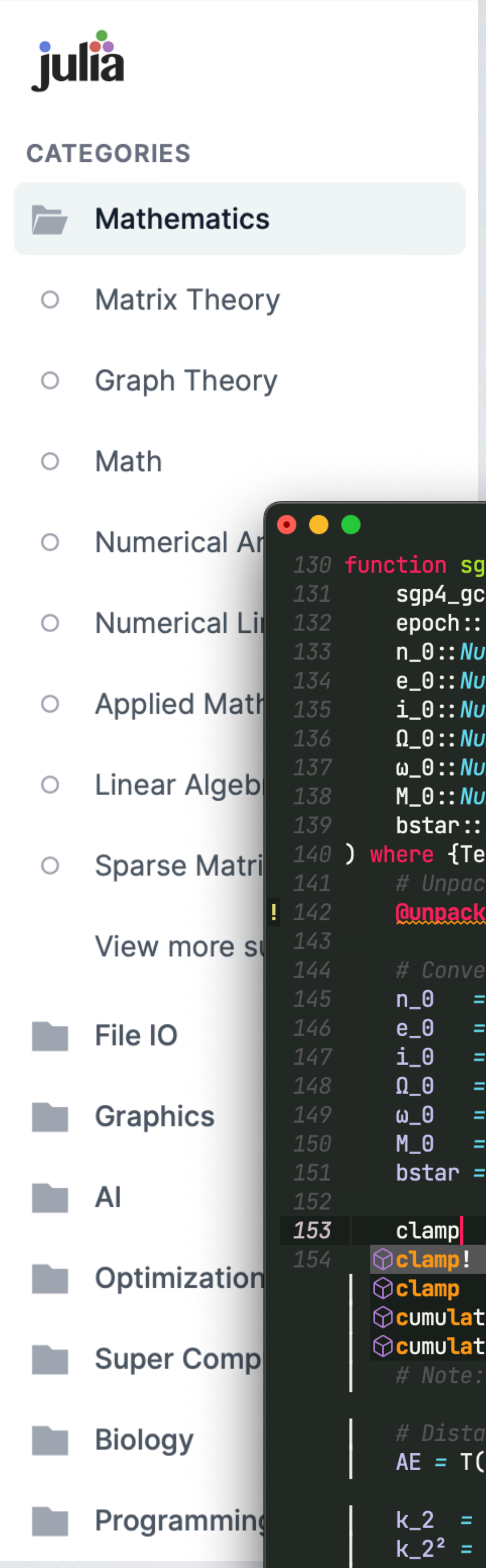
CHOOSE YOUR **TOOLS** WITH CARE

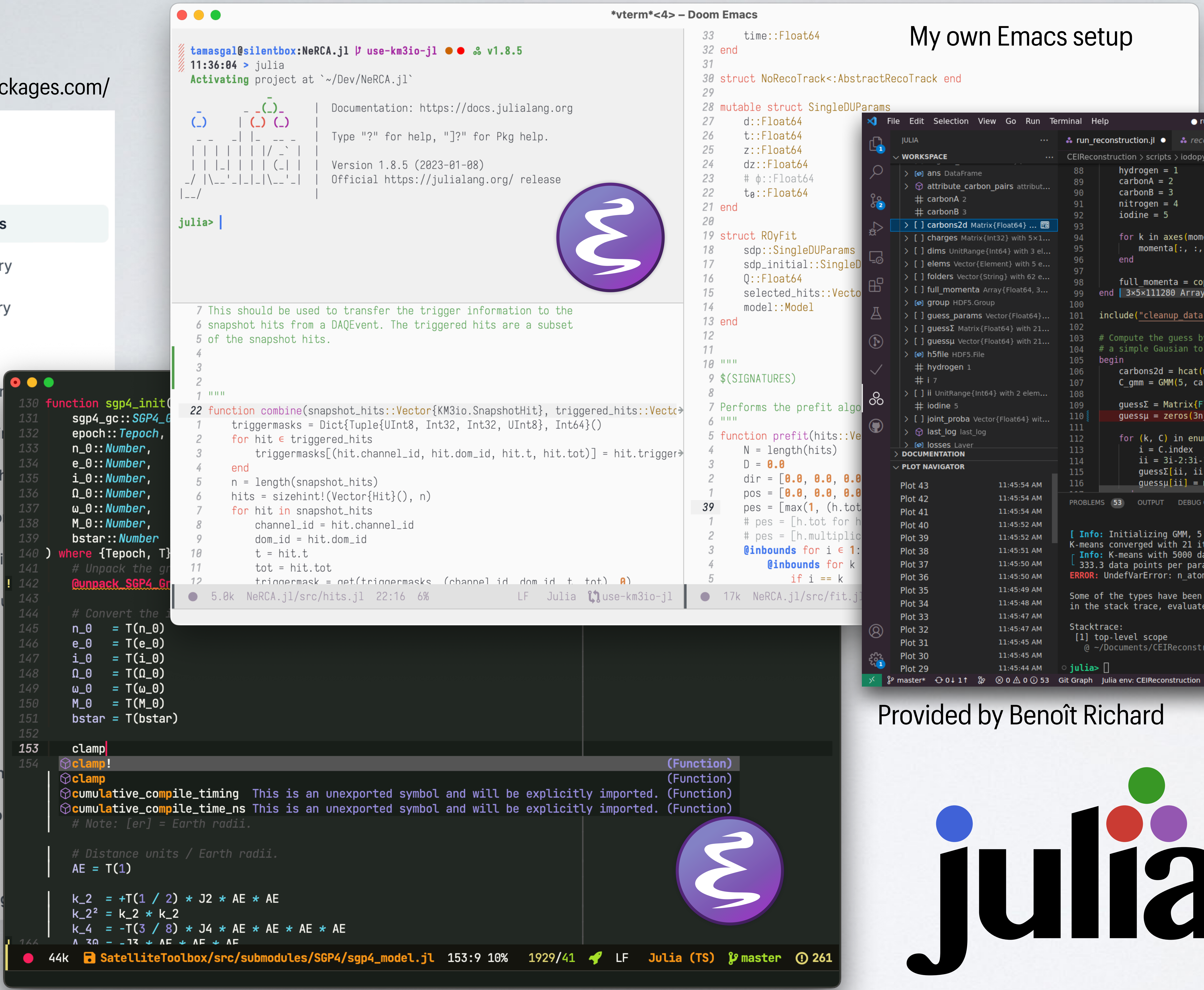
- **Python is** a **powerful** language and it can be used for many different tasks
- **However**: it's **easy to write code with horrible performance**
- It's even easier to write **code** which simply **does not scale**
- In contrary to Python's expressiveness, **high-performant Python code** is **often neither** nicely **readable**, **nor easily maintainable**
- **High-level Python APIs** can be very useful but usually act as a **barrier**
- **Low-level development** requires **multiple languages** and **technology stacks**, which increase **complexity** greatly
- You have to be **more than a Python expert** to write high-performant "Python code"
- Keep all this in mind and give **Julia** a go for scientific computing

PERSONAL REMARK

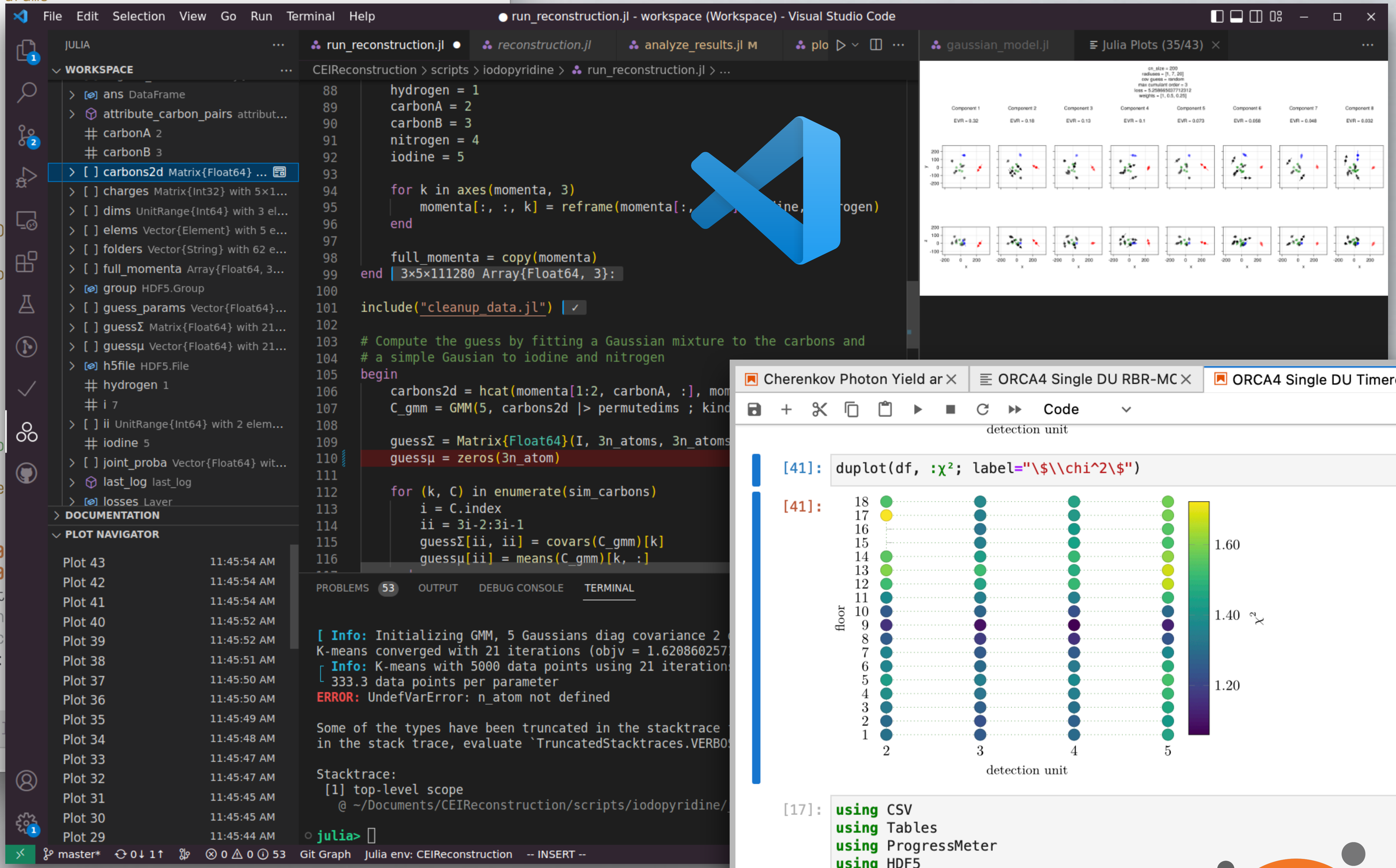
- I am a big fan of **Julia** (surprise!)
- The **Julia language** is built for **scientific computing**, it "**feels like Python** and **runs like C**".
- **No fear** of writing **for-loops**
- Define and use your **own types** and **type hierarchy** to create **expressive code**
- **Interactive** prototyping, just like in Python with a **REPL**
- Even **naively written code** is often very **close to optimal performance**
- You can still use your **Jupyter-notebooks** (fun fact: "Ju" in Jupyter stands for Julia)
- **Easy package management** and deployment (you can easily set up your own package registry)
- **Reproducible** environments are a built-in feature (crucial for reproducible **science**)

https://juliapackages.com/

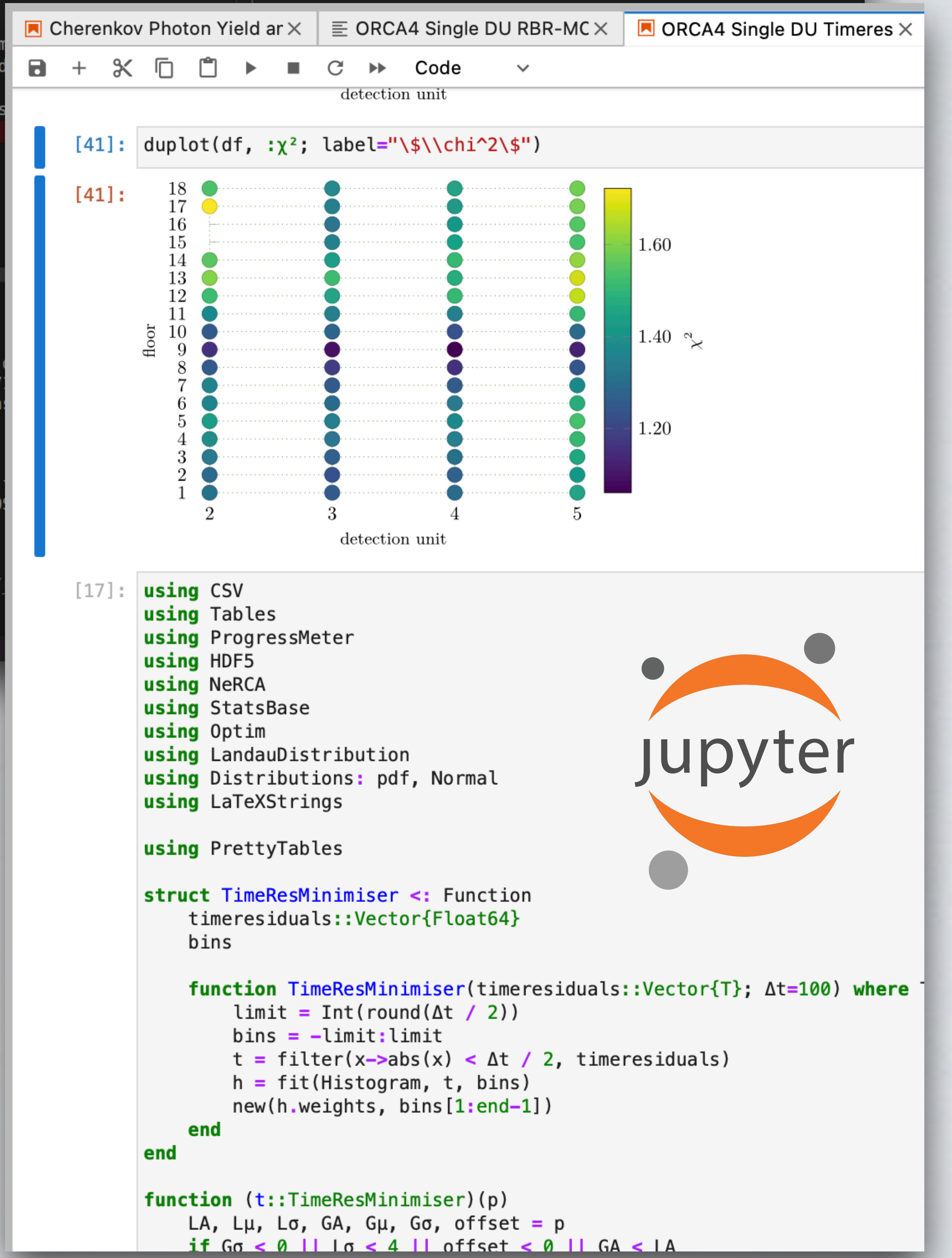




My own Emacs setup



Provided by Benoît Richard



https://discourse.julialang.org/t/help-testing-julia-tree-sitter-mode-in-emacs/92819/11

One of my Jupyter prototyping sessions

"WITH GREAT POWER COMES GREAT RESPONSIBILITY"

– UNCLE BEN

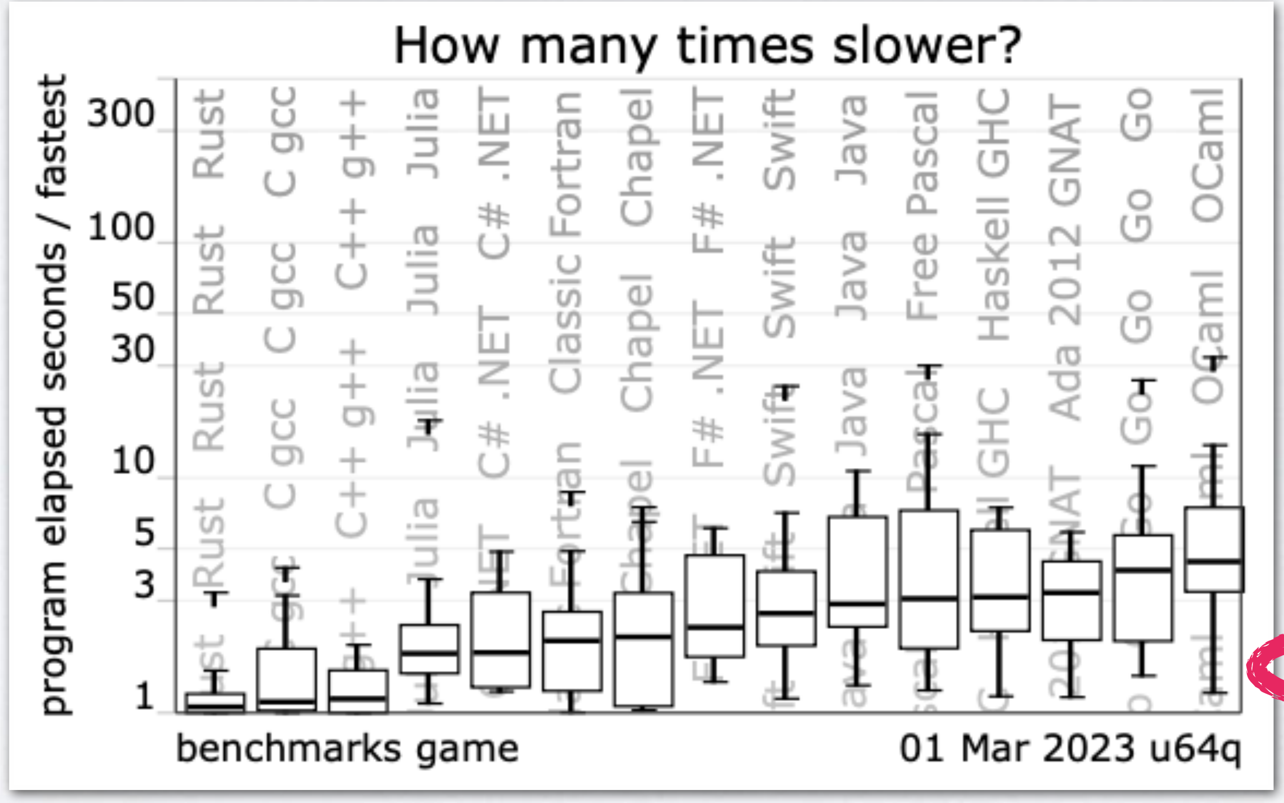
A FINAL WORD ON **GREEN** CODING/COMPUTING

- In **HPC**, we can easily **waste energy** with **inefficient code**
- It's just a matter of a **few keystrokes** to launch **thousands of computing jobs**
- It's also **your responsibility** to learn how to use these resources **efficiently** and with **care**

Table 4. Normalized global results for Energy, Time, and Memory

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

×	source	secs	mem	gz	cpu secs	cpu load
1.0	Classic **Fortran** #3	**0.71**	10,984	638	2.84	100% 98% 98% 100%
1.0	**Rust** #7	**0.72**	11,128	932	2.84	100% 100% 100% 100%
1.0	Rust #4	0.72	11,124	817	2.84	100% 100% 100% 98%
1.0	Rust #5	0.72	11,124	1055	2.84	100% 100% 100% 100%
1.0	**Chapel** #2	**0.73**	10,948	335	2.89	100% 98% 98% 98%
1.5	**Julia** #4	**1.09**	185,772	429	3.67	99% 78% 79% 79%
1.8	Julia #2	1.26	192,016	370	4.14	76% 76% 76% 98%
2.0	**Go** #4	**1.42**	11,244	548	5.67	99% 100% 99% 99%
2.0	**Swift** #3	**1.43**	11,344	601	5.68	100% 100% 99% 99%
2.0	**C** gcc #3	**1.44**	11,392	463	5.70	100% 99% 99% 99%



https://benchmarksgame-team.pages.debian.net

Source: Energy Efficiency across Programming Languages, SLE'17

THANKS

"PEOPLE ARE VERY OPEN-MINDED
ABOUT NEW THINGS — AS LONG
AS THEY'RE EXACTLY LIKE THE OLD ONES."

- CHARLES F. KETTERING