THE MISSING PYTHON INTRODUCTION 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/Shel)
  - Editor: **Vim** for ~25 years but switched to **Emacs (EVIL mode)** in 2020
  - Other: ADV motorbikes, climbing, electronics, modular synths, DIY...

@tamasgal
DISCLAIMER

The following presentation contains oversimplifications and blatant omissions due to time constraints. The viewer may also experience well-timed product placements.
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
Python is the most popular language (according to TIOBE)! ... and has beaten Java and C++! Btw. Julia is #33 ;)

Most loved languages [https://survey.stackoverflow.co/2022](https://survey.stackoverflow.co/2022)
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?

```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

\texttt{print(5 \texttt{ is } 7 - 2, 300 \texttt{ is } 302 - 2)}

\textbf{PyObject* \texttt{PyLong\_From\_Long}}(long \texttt{v})


Return a new \texttt{PyLongObject} object from \texttt{v}, or NULL on failure.

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

\texttt{"is"} is an operator which checks if \textbf{two objects are identical}: \texttt{"x is y"} is true iff \texttt{x} and \texttt{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?
PERFORMANCE OF LANGUAGES

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

Execution time

log scale
TO UNDERSTAND THE PERFORMANCE ISSUES

PYTHON INTERNALS
• **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](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

source
foo.py

bytecode
compiler
foo.pyc

interpreter
runtime

```
def add(a, b):
    "Adds two objects."
    return a + b
```

Recommended reading (Victor Skvortsov - Python behind the scenes): https://tenthousandmeters.com/blog/python-behind-the-scenes-1-how-the-cpython-vm-works/
import dis; dis.dis(compile('print(5 is 7 - 2, 300 is 302 - 2)', '', 'single'))

Python 3.6
True, False

Python 3.7
True, True
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

```python
>>> dir(42)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
 '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
 '__floor__', '__floor_div__', '__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__', '__rfloor_div__', '__rfloor_shift__', '__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

arr = [23, 5, 42]

- **Lower limit** for the size in **bytes** (on a 64bit system), oversimplified:
  \[8 + 8 + 8 + 8 + 8 + 8 + 8 + 8 \times 3 = 168\]

- "**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
single thread:

```python
>>> 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:

```python
>>> 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**

```python
>>> def mean(numbers):
...    return sum(numbers) / len(numbers)
...
>>> numbers = list(range(1000000))
>>> %timeit mean(numbers)
2.98 ms ± 21.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

**NumPy (~16x faster)**

```python
>>> import numpy as np
>>> numbers = np.random.random(1000000)
>>> %timeit np.mean(numbers)
190 µs ± 1.35 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

**Numba (~3x faster)**

```python
>>> @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(1000000)
>>> %timeit numba_mean(numbers)
1.83 ms ± 35.2 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

**Julia (~16x faster)**

```julia
julia> numbers = rand(1000000);

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

**pure Python**

```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)**

```python
>>> 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)**

```python
>>> @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 ± 0.502 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

**Julia (~150000000x faster)**

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

julia> @benchmark simple_sum(N)
BenchmarkTools.Trial: 10000 samples with 997 evaluations.
    Range (min ... max):  19.015 ns ... 489.468 ns  GC (min ... max): 0.000 MB
    Time  (median):      19.267 ns
    GC  (median):       0.000 MB
    Time  (mean ± σ):  20.046 ns ± 6.731 ns  GC (mean ± σ): 0.446 ± 0.837 MB

19 ns Histogram: log(frequency) by time 23.9 ns x
Memory estimate: 16 bytes, allocs estimate: 1.
```

⇒ \( N(N+1)/2 = 50000000050000000 \)
"WHY WE CAN'T HAVE NICE THINGS"

AKA VECTORISE FOR YOUR LIFE!
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:

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

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

```python
>>> hits
[<_main__Hit at 0x10846e430>,
 <_main__Hit at 0x108ba2100>,
 <_main__Hit at 0x108422490>,
 <_main__Hit at 0x1085d6a0>,
 <_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.

```python
>>> 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).

```python
>>> import numpy as np

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

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])
```
easy and intuitive element-wise operations

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

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

```python
a = np.arange(6)
a
array([0, 1, 2, 3, 4, 5])
a.reshape(2, 3)
array([[0, 1, 2],
       [3, 4, 5]])
```

ndim: 1
shape: (6,)

No rearrangement of the elements in memory but setting the iterator limits internally!
RESHAPING ARRAYS IS CHEAP

```python
>>> 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.getusage(resource.USAGE_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.rand(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()
```

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 = 2*a + 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("All libraries loaded: \
          {peak_memory_usage()} MB")

    N = 1_000_000
    a = np.random.rand(N)
    print("a' allocated: \{peak_memory_usage\} MB")

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

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

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

    c = np.add(a, b)
    print("c' calculated: \{peak_memory_usage\} MB")
    if __name__ == "__main__":
        main()
```

Reusing a and b with "out="

```python
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.

```python
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.
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

\[ a = \text{np.random.random}(1000000) \]

NumPy:
\[ 2 \times a^{**3} - 4 \times a^{**5} + 6 \times \text{np.log}(a) \]
82.4 ms ± 1.88 ms per loop

Numexpr with 4 threads:
\[ \text{ne.set_num_threads}(4) \]
\[ \text{ne.evaluate}("2 \times a^{**3} - 4 \times a^{**5} + 6 \times \log(a)") \]
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.
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("All libraries loaded: \{peak_memory_usage()\} MB")
    N = 1_000_000
    a = np.random.randn(N)
    print("'a' allocated: \{peak_memory_usage()\} MB")
    b = np.random.randn(N)
    print("'b' allocated: \{peak_memory_usage()\} MB")
    c = ne.evaluate("2*a + 3*b")
    print("'c' calculated: \{peak_memory_usage()\} MB")

if __name__ == "__main__":
    main()
• **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
• 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

```python
In [1]: import awkward as ak
In [2]: arr = ak.Array([[3, 4, 5], [1, 2, 3], [4, 5], [6, 7, 8, 9]])
In [3]: arr
Out[3]: <Array [[3, 4, 5], [1, 2, 3], [4, 5], [6, 7, 8, 9]] type='3 * var * int64'>
In [4]: arr[:, 0]
Out[4]: <Array [3, 1, 4, 6] type='3 * int64'>
In [5]: ak.mean(arr, axis=0)
Out[5]: <Array [2, 4.5, 7.5] type='3 * float64'>
```
AWKWARD 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

```python
In [1]: import awkward as ak
In [2]: arr = ak.Array([[True, ], (False, ), (False, 1)])
In [3]: arr
Out[3]: <Array [[True, 1], (False, 3), (False, 9)] type='3 * (bool, int64)'>

In [4]: arr.pos_x
Out[4]: <Array [[3, 45, 65], [1, 3]] type='2 * var * int64'>
```
• A very nice introduction by Jim himself (just search for "awkward array" on YouTube)

But Numpy doesn’t have anything for unequal-length lists

<table>
<thead>
<tr>
<th>event 1</th>
<th>1 2 3</th>
<th>1 2</th>
<th>1 2 1 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>event 2</td>
<td>1 2 3 4</td>
<td>1</td>
<td>1 2 1 2</td>
</tr>
<tr>
<td>event 3</td>
<td>1</td>
<td></td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>event 4</td>
<td>1 2</td>
<td>1 2</td>
<td>1 2 1 2</td>
</tr>
</tbody>
</table>

Some libraries* can represent arrays of unequal-length arrays, known as “jagged” or “ragged” arrays.

*Apache Arrow, XND, TensorFlow, Zarr (genetics), and ROOT (particle physics)

• 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

source

foo.py

compiler

control flow graph

data flow graph

Numba IR

Type inference

Typed Numba IR

Lowering

LLVM IR

interpreter

bytecode

foo.pyc

Codegen via LLVM

runtime
NUMBA JIT-EXAMPLE

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

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

@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:**
```
np.abs(a - b) / (np.abs(a) + np.abs(b))
```

23 ms ± 845 µs per loop

**Numba @vectorize:**
```
@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
• 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)
My own Emacs setup

One of my Jupyter prototyping sessions


Provided by Benoît Richard
"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

<table>
<thead>
<tr>
<th>Source</th>
<th>Energy</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Rust</td>
<td>1.03</td>
<td>1.04</td>
<td>1.05</td>
</tr>
<tr>
<td>C++</td>
<td>1.34</td>
<td>1.56</td>
<td>1.17</td>
</tr>
<tr>
<td>Ada</td>
<td>1.70</td>
<td>1.85</td>
<td>1.24</td>
</tr>
<tr>
<td>Java</td>
<td>1.98</td>
<td>1.89</td>
<td>1.34</td>
</tr>
<tr>
<td>Pascal</td>
<td>2.14</td>
<td>2.14</td>
<td>1.47</td>
</tr>
<tr>
<td>Chapel</td>
<td>2.18</td>
<td>2.83</td>
<td>1.54</td>
</tr>
<tr>
<td>Lisp</td>
<td>2.27</td>
<td>3.02</td>
<td>1.92</td>
</tr>
<tr>
<td>Ocaml</td>
<td>2.40</td>
<td>3.09</td>
<td>2.45</td>
</tr>
<tr>
<td>Fortran</td>
<td>2.52</td>
<td>3.14</td>
<td>2.57</td>
</tr>
<tr>
<td>Swift</td>
<td>2.79</td>
<td>3.40</td>
<td>2.71</td>
</tr>
<tr>
<td>Haskell</td>
<td>3.10</td>
<td>3.55</td>
<td>2.80</td>
</tr>
<tr>
<td>C#</td>
<td>3.14</td>
<td>4.20</td>
<td>2.82</td>
</tr>
<tr>
<td>Go</td>
<td>3.23</td>
<td>6.30</td>
<td>3.28</td>
</tr>
<tr>
<td>Dart</td>
<td>3.85</td>
<td>6.52</td>
<td>3.34</td>
</tr>
<tr>
<td>Fe</td>
<td>4.13</td>
<td>6.67</td>
<td>3.52</td>
</tr>
<tr>
<td>JavaScript</td>
<td>4.45</td>
<td>6.80</td>
<td>3.97</td>
</tr>
<tr>
<td>Racket</td>
<td>7.91</td>
<td>11.27</td>
<td>4.00</td>
</tr>
<tr>
<td>TypeScript</td>
<td>21.50</td>
<td>26.99</td>
<td>4.25</td>
</tr>
<tr>
<td>Hack</td>
<td>24.02</td>
<td>27.64</td>
<td>4.59</td>
</tr>
<tr>
<td>PHP</td>
<td>29.30</td>
<td>36.71</td>
<td>4.69</td>
</tr>
<tr>
<td>Erlang</td>
<td>42.23</td>
<td>43.44</td>
<td>6.01</td>
</tr>
<tr>
<td>Lua</td>
<td>45.98</td>
<td>46.20</td>
<td>6.62</td>
</tr>
<tr>
<td>Ruby</td>
<td>46.54</td>
<td>59.34</td>
<td>6.72</td>
</tr>
<tr>
<td>Perl</td>
<td>69.91</td>
<td>65.72</td>
<td>7.20</td>
</tr>
<tr>
<td>Python</td>
<td>75.88</td>
<td>71.90</td>
<td>8.64</td>
</tr>
<tr>
<td>Ruby</td>
<td>79.38</td>
<td>82.35</td>
<td>19.84</td>
</tr>
</tbody>
</table>

Source: Energy Efficiency across Programming Languages, SLE’17

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

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

- CHARLES F. KETTERING