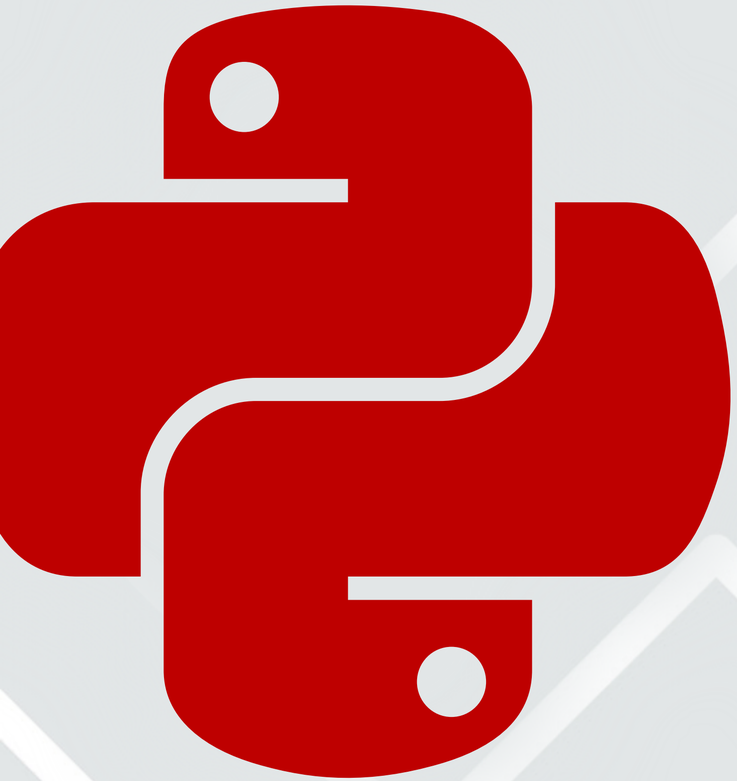


Practical Guide to Performance-Conscious Python

RMACC
HPC SYMPOSIUM

Robben E. Migacz
Scientific Consultant

CENTER FOR HIGH PERFORMANCE COMPUTING
THE UNIVERSITY OF UTAH

Why should we care about Python performance?

Isn't "performance-conscious Python" an oxymoron?

Motivations

The performance characteristics of Python are not always intuitive

The Python ecosystem is complex and ever-growing

Python is widely used, particularly among users who are new to HPC

The performance
characteristics of Python
are not always intuitive

In my case, [the] serial time [of a Python script] is 28 seconds. [The] parallel time is anywhere from ~60 seconds to 200+ seconds depending on implementation.

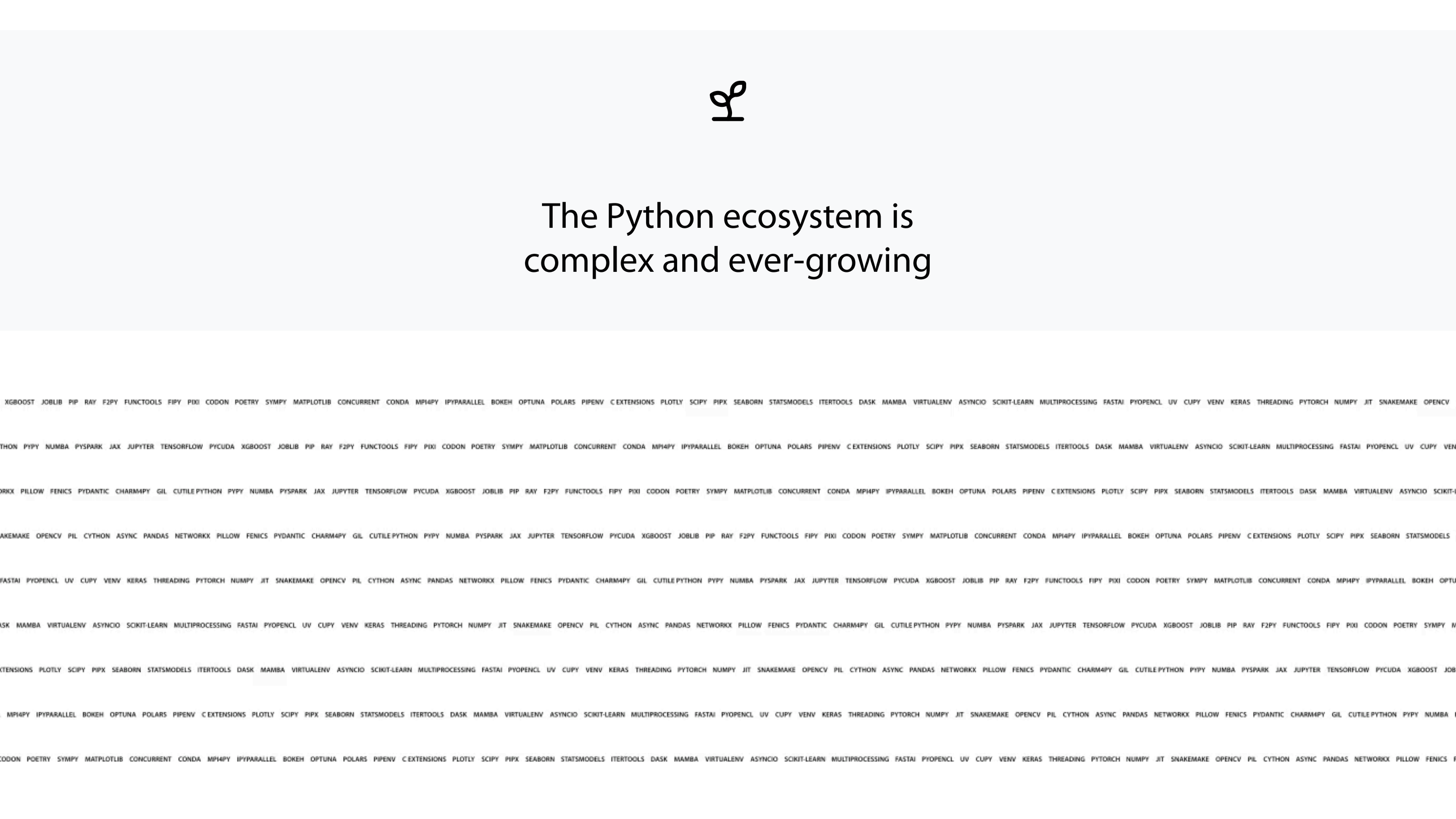
Illustration by Fast Ink on Unsplash (Unsplash License)

Motivations

The performance characteristics of Python are not always intuitive

The Python ecosystem is complex and ever-growing

Python is widely used, particularly among users who are new to HPC


The Python ecosystem is
complex and ever-growing

Motivations

The performance characteristics of Python are not always intuitive

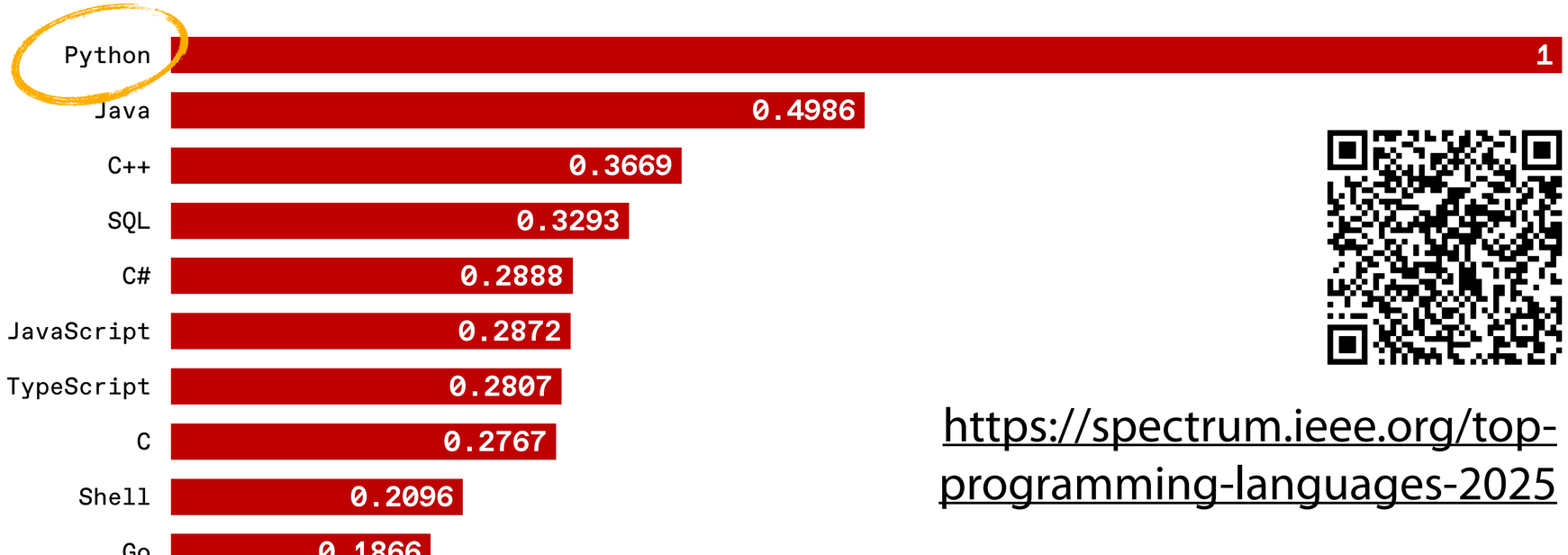
The Python ecosystem is complex and ever-growing

Python is widely used, particularly among users who are new to HPC

Python is widely used,
particularly among users
who are new to HPC

Top Programming Languages 2025

IEEE Spectrum ranking, edited by Stephen Cass (September 2025)



https://spectrum.ieee.org/top-programming-languages-2025

Python is widely used,
particularly among users
who are new to HPC

2025 Developer Survey

Stack Overflow

Python is the **#1** language among respondents in “Learning to Code”
and “Learners that Use AI” categories

---

https://survey.stackoverflow.co/2025/technology

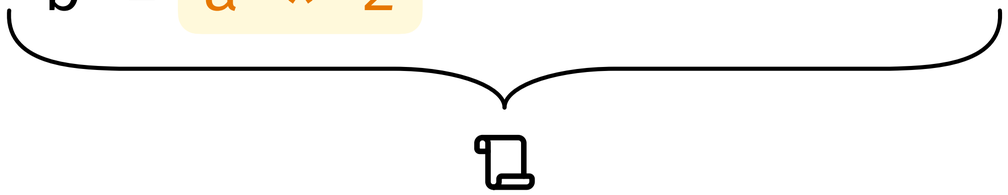
ESSENTIAL BACKGROUND:

Python is a “glue language”

ESSENTIAL BACKGROUND:

Python is a “glue language”

```
import numpy as np
a = np.array([1, 2, 3])
b = a * 2
```

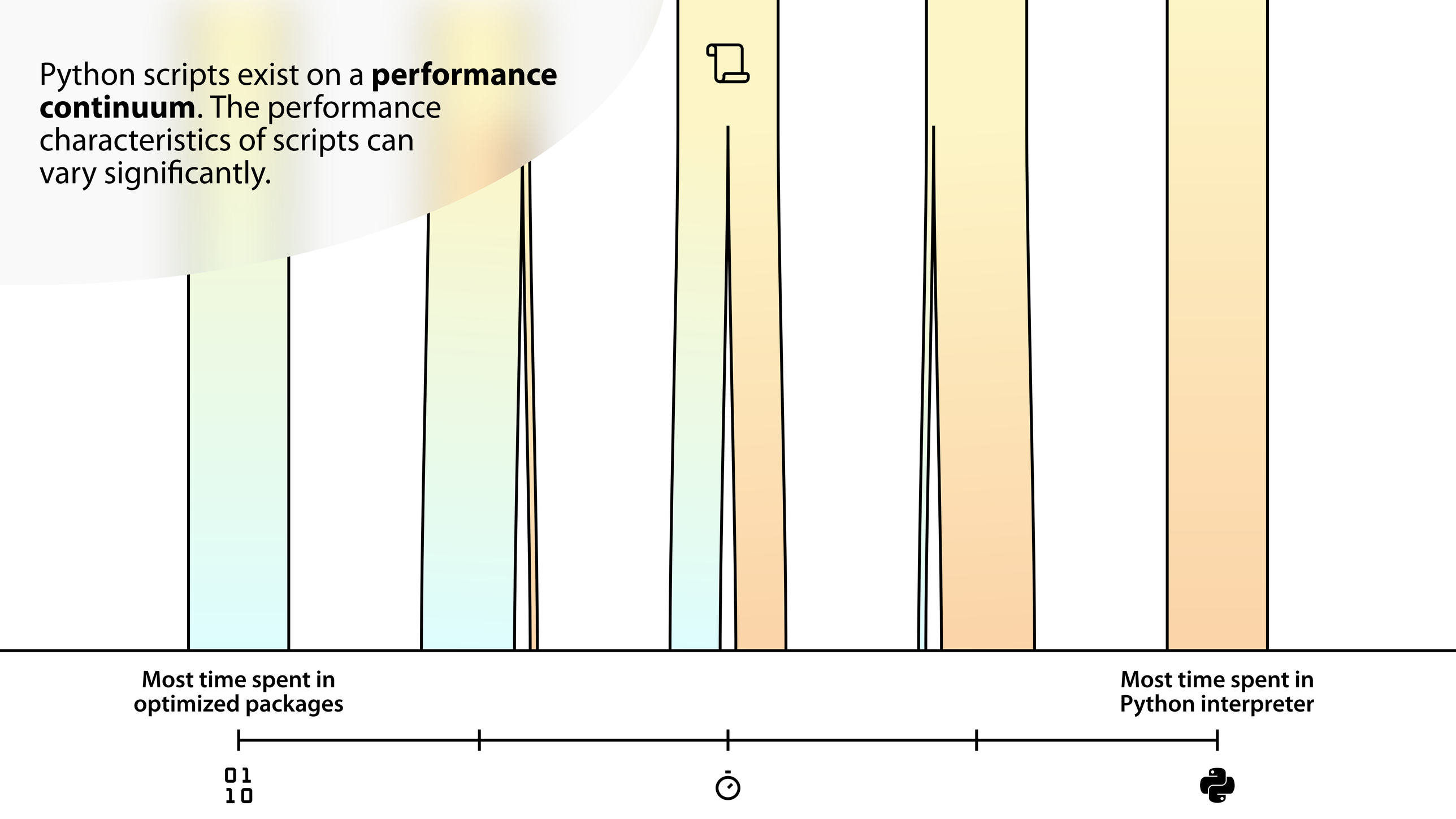


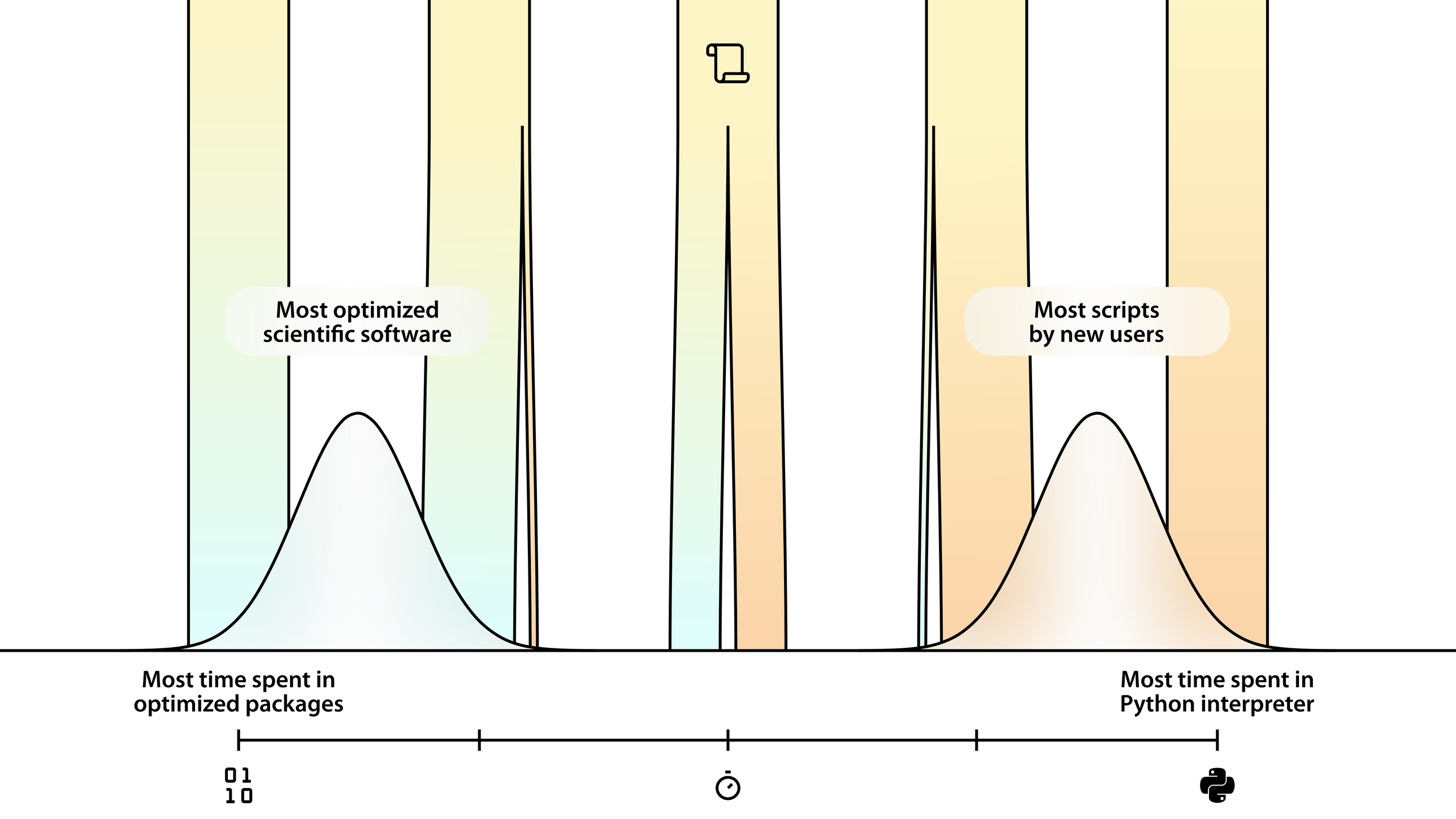
Interpreted by Python interpreter

`np.array()` is exposed in Python but **implemented in a C extension module**

`a * 2` triggers `a.__mul__(2)`, which is routed to a NumPy universal function **implemented in C**

Python scripts exist on a **performance continuum**. The performance characteristics of scripts can vary significantly.




Most optimized
scientific software
Most scripts
by new users
Most time spent in
optimized packages
Most time spent in
Python interpreter
01
10

Profiling scripts to understand performance characteristics is a critical first step for optimization.

If—upon profiling your scripts—you find yourself in the "mostly optimized packages" regime, options for optimization are limited.

If—upon profiling your scripts—you find yourself in the "mostly optimized packages" regime, options for optimization are limited.

Optimizing your Python scripts won't help much in this regime. Consider using packages that are optimized for your system or revisit your approach to the problem.

Wheels—prebuilt binary packages—are generally built for older hardware (for C standard library and CPU instruction set compatibility). Building packages yourself can yield significant performance improvements in some cases.

If—upon profiling your scripts—you find yourself in the "mostly optimized packages" regime, options for optimization are limited.

> The difference between the 2009 hardware features and … the 2019 or 2023 one[s] could be a factor of 10×, 20× in performance, depending on what you're doing.

> You can get *massive* performance increases [with optimized builds].

Panelist, *Talk Python To Me* episode 544 (April 10, 2026), ca. 17:30–18:00

In practice, you may see smaller improvements (orders of 10% to 100%), depending on software. Some packages may already have a CPU dispatcher or other features to enhance performance on newer hardware.

If you find yourself in the "mostly Python interpreter" regime, you have more room—and more options—for optimization. This will be our focus moving forward.

Start by ensuring you're using a recent version of Python and following best practices

Start by ensuring you're using a recent version of Python and following best practices

Evgenia Verbina
August 2025

48% of people are currently using Python 3.11. Upgrading to 3.13 will **make their code run ~11% faster end to end while using ~10-15% less memory**.

If they are one of the 27% still on 3.10 or older, **their code gets a whopping ~42% speed increase (with no code changes), and memory use can drop by ~20-30%!**

https://blog.jetbrains.com/pycharm/2025/08/the-state-of-python-2025/

Start by ensuring you're using a recent version of Python and following best practices

- **Preallocate memory** for objects that grow
- **Move loop-invariant statements** outside loops ("hoisting")
- **Cache results and attribute lookups** where feasible (don't run statements multiple times needlessly)

These suggestions are helpful—or, at the very least, not detrimental—even if you later use optimized packages; this is usually not wasted time.

See suggestions at https://blog.jetbrains.com/pycharm/2025/11/10-smart-performance-hacks-for-faster-python-code/

The big question for scientific Python performance:

Can you use packages optimized for numeric computation?





Can you use packages optimized for numeric computation?

Some of the most significant performance gains for Python scripts will come from using optimized packages. *Don't write Python to duplicate the effort of an existing, performance-conscious package.*

Can you use packages optimized for numeric computation?

- **NumPy ecosystem** as *de facto* standard for numeric computation in Python
- **Just-in-time compilers like Numba** or alternative interpreters or compilers
- Machine learning packages
- Data analysis tools like Polars

Can you use packages optimized for numeric computation?

- **NumPy ecosystem** as *de facto* standard for numeric computation in Python
  - Support for consistent data types, contiguous memory, vectorization, elimination of loop overhead; resolves many performance concerns with Python objects
  - CuPy (drop-in, GPU-enabled replacement) for problems with high arithmetic intensity
- **Just-in-time compilers like Numba** or alternative interpreters or compilers
- Machine learning packages
- Data analysis tools like Polars

Runtime of square operation at right reduced by **94%** with NumPy

AMD EPYC 9654P, Python 3.13

NumPy can require more careful consideration of data types; an integer can overflow here—and would have if I had selected int32

```
import numpy as np

# Generate a large dataset
size = 10_000_000
data_list = range(size)
data_array = np.arange(
    size,
    dtype=np.int64
)

# Square each element
squared_python = [
    x**2
    for x in data_list
]

# NumPy approach to calculation
squared_numpy = data_array**2
```

Can you use packages optimized for numeric computation?

- **NumPy ecosystem** as *de facto* standard for numeric computation in Python
- **Just-in-time compilers like Numba** or alternative interpreters or compilers
- Machine learning packages
- Data analysis tools like Polars

Can you use packages optimized for numeric computation?

- **NumPy ecosystem** as *de facto* standard for numeric computation in Python
- **Just-in-time compilers like Numba** or alternative interpreters or compilers
  - Compilation of numeric routines to machine code (benefits from static typing, vectorization, optimization); support for parallelism (ideally parallel problems)
  - GPU support through Numba-CUDA (NVIDIA-maintained)
- Machine learning packages
- Data analysis tools like Polars

Numba

Numba **compiles Python functions to machine code**. It is most useful if you have a hot loop (a frequently used region of your script).

A dispatcher object selects a compiled version of a function based on argument types, compiling first as needed (if the compiled function does not exist in memory or disk cache already).

Numba works best with numeric types; try avoid "object mode," which does not confer the same performance benefit.

```python
from numba import jit


def slow_sum(n):
    total = 0.0
    for i in range(n):
        total += i**0.5
    return total


N = 10_000_000
slow_sum(N)
fast_sum(N)
```

```python
@jit
def fast_sum(n):
    total = 0.0
    for i in range(n):
        total += i**0.5
    return total
```

Runtime decreases by 18% with Numba here, *even with compilation overhead*

A subsequent function call (not shown) has runtime that has decreased more than **97%** relative to standard Python

AMD EPYC 9654P, Python 3.13

If you have more complex logic and using optimized packages does not resolve performance concerns, consider parallelization.

If a problem is tightly coupled or requires high scalability, consider mpi4py

If a problem does not require significant communication, consider threading (if input- and output-heavy) or multiprocessing

mpi4py

mpi4py provides Python bindings for the Message Passing Interface (MPI) standard. It's particularly useful for highly scalable scripts.

Some methods in mpi4py have significant serialization overhead; for example, `send()` is slow (works with Python objects), while `Send()` is fast (works with NumPy arrays).

Brett Milash has an example script: https://github.com/chpc-uofu/mpi4py-examples/blob/main/laplace_mpi.py

mpi4py

This example script includes the communication of Python objects, not NumPy arrays; can you make it faster?

Brett Milash has an example script: https://github.com/chpc-uofu/mpi4py-examples/blob/main/laplace_mpi.py

If you have more complex logic and using optimized packages does not resolve performance concerns, consider parallelization.

If a problem is tightly coupled or requires high scalability, consider mpi4py

If a problem does not require significant communication, consider threading (if input- and output-heavy) or multiprocessing

BE AWARE:

Communication is *slow* with multiprocessing

SERIALIZATION

TRANSMISSION OF SEQUENCE OF BINARY OPCODES

RECONSTRUCTION OF OBJECT

EXAMPLES, IN NO PARTICULAR ORDER:

```
GET, PUT, EMPTY_LIST, TUPLE_START, APPENDS,
    INT, REDUCE, NEWTRUE, BINFLOAT, STOP
```

After optimizing, is your script *still* too slow?

If your script is *still* too slow, consider algorithmic changes or partial rewrites in low-level languages.

If your problem is tractable (doesn't necessitate algorithmic changes) but Python performance is limiting, consider writing your own low-level package in C, C++, Fortran, or Rust.

```
! add.f90
real function add_numbers(a, b)
    real, intent(in) :: a, b
    add_numbers = a + b
end function add_numbers
```

```
$ f2py -c -m add_module add.f90
$ python3
>>> import add_module
>>> print(add_module.add_numbers(3.0, 4.5))
7.5
```

```
// add.c
double add_numbers(
    double a,
    double b
) {
    return a + b;
}
```

```
$ gcc -shared -fPIC add.c -o libadd.so
$ python3
>>> import ctypes
>>> lib = ctypes.CDLL("./libadd.so")
>>> lib.add_numbers.argtypes = [ctypes.c_double, ctypes.c_double]
>>> lib.add_numbers.restype = ctypes.c_double
>>> print(lib.add_numbers(3.0, 4.5))
7.5
```

Getting started with low-level extensions is easier than ever with large language models.

Continue using Python for structure, but consider using other technologies for speed.

What about recent, native, performance-oriented Python features?

“I don’t think … [the built-in JIT compiler is] ever going to obviate the need for Numba [and similar tools]”

The internal JIT still has to be fully compatible with Python and can’t optimize like Numba and friends; “the typical performance impact of enabling it can range from 10% slower to 20% faster, depending on workload,” per https://docs.python.org/3/whatsnew/3.14.html

Free-threaded Python is “not going to have as big an impact [as it will, for example, for web developers] for this particular audience [HPC]”

Panelist, “The Future of Python on HPC Systems,” SC25

Why should we care about Python performance?

Isn't "performance-conscious Python" an oxymoron?

Python *is* a performance-oriented language

Panelist, “The Future of Python on HPC Systems,” SC25

Questions

I'd be happy to take questions.

Further questions?
helpdesk@chpc.utah.edu

I acknowledge and appreciate suggestions and comments from Martin Čuma, Wim Cardoen, and Paul Fischer at the Center for High Performance Computing during the early development of presentation concepts and materials.

Some concepts are inspired by "Python for HPC" presentations by Andrea Zonca at San Diego Supercomputer Center.