{codspeed}

Intuition vs. Reality

Surprising Truths in Python Performance

whoarewe

Arthur

Adrien

We're building

{codspeed}

{codspeed}

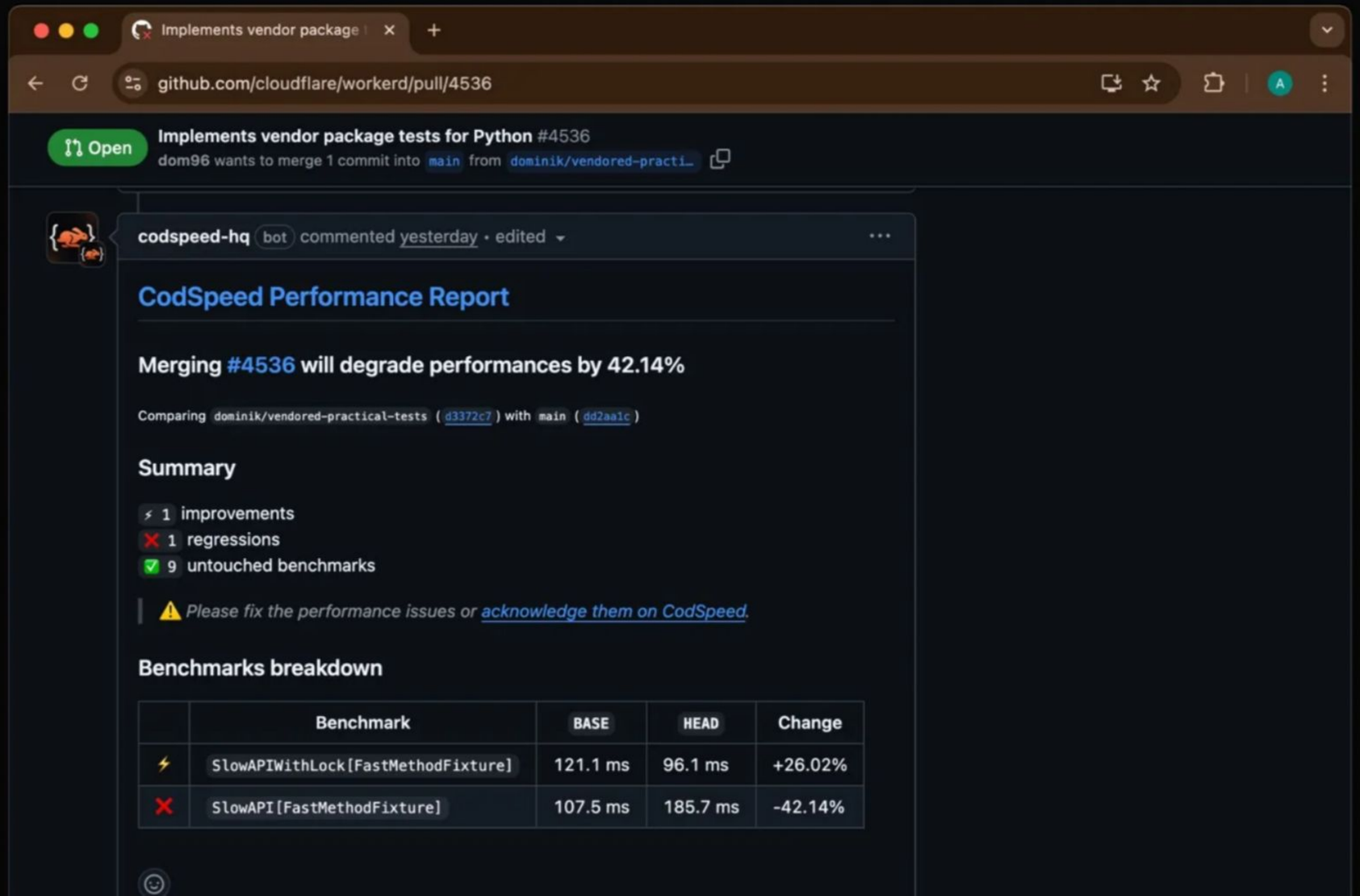
{codspeed}

test_performance.py

```python
import pytest

@pytest.mark.benchmark
def test_my_fn():
    inputs = gen_inputs()
    results = my_fn(inputs)
    assert results == "expected_result"
```

{codspeed}

{codspeed}



Implements vendor package tests for Python #4536

Comparing dominik/vendored-practical-tests (d3372c7) with main (dd2aa1c)

Improvements	Regressions	Untouched	New	Dropped	Ignored
1	1 ⚠	9	0	0	0

Benchmarks

Failed

SlowAPI[FastMethodFixture] Regression

src/workerd/tests/bench-fast-api.c++::SlowAPI[FastMethodF...

107.5 ms → 185.7 ms

Improved

SlowAPIWithLock[FastMethodFixture]

src/workerd/tests/bench-fast-api.c++::SlowAPIWithLock[Fas...

121.1 ms → 96.1 ms

Passed

request[GlobalScopeBenchmark]

src/workerd/tests/bench-global-scope.c++::request[GlobalS...

56.8 ms → 56.8 ms

Commits

Click on a commit to change the comparison range

Base main dd2aa1c

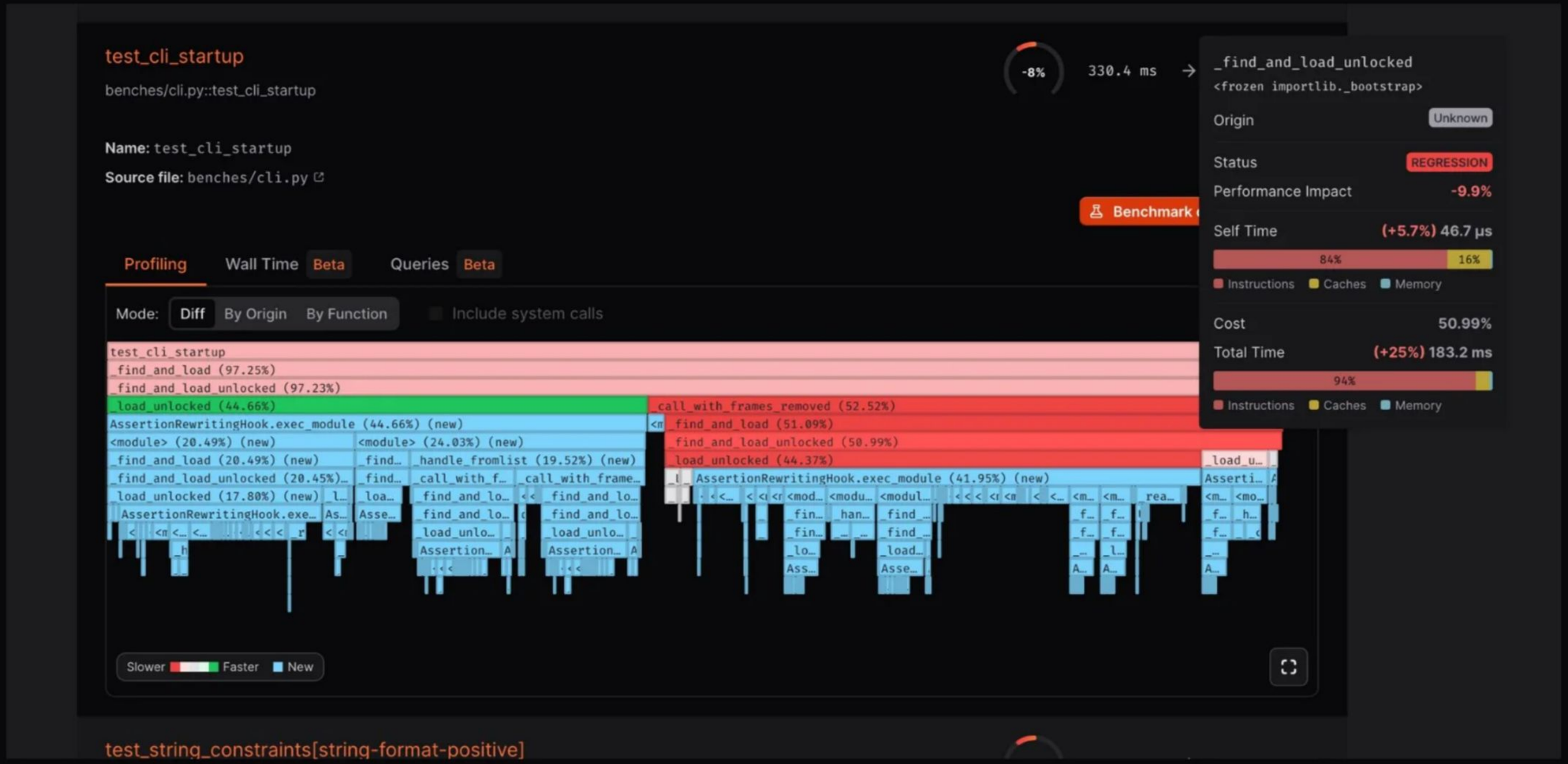
-42%

Implements vendor tests for Python

d3372c7 1 day ago by dom96

{codspeed}

{codspeed}



test_string_constraints[string-format-positive]

{codspeed}

Free for open source

Loved by performance experts:

Vercel

ASTRAL

Pydantic

CLOUDFLARE

ByteDance

LangChain

... and many more!

Why performance matters?

- User experience and revenue
  - Every 100ms faster → 1% more conversions (Mobify, earning +$380,000/yr)
- Competitive advantage
  - Performance is a feature
- Cost savings
  - Less compute for same workloads
- Sustainability
  - Reduced data center footprint

Getting performance right is hard

- Humans are not machines
- Even the best performance expert will get things wrong

Don’t trust us yet?

Let’s see!

The quiz

- There is a leaderboard counting points
- **Correct** answers earn points
- **Faster** answers earn more points
- The WINNER receives a prize 🎁

Performance is measured on CPython 3.13.3 on an x86_64 CPU

{codspeed}

Scan this QR code to join →

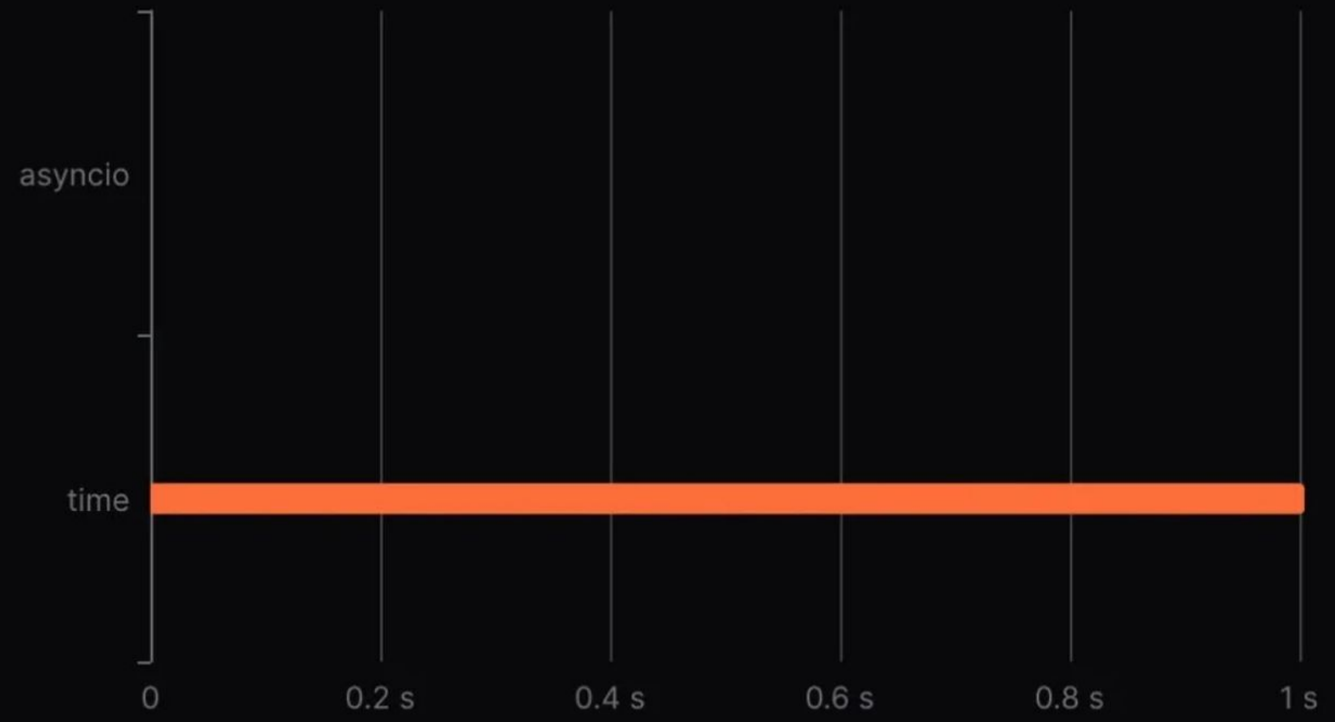
Case study #0: Sleep

```python
import time
import asyncio


def sleep_1():
    time.sleep(1)


def sleep_1000():
    asyncio.sleep(1000)
```

Which is faster?

```python
import time
import asyncio


def sleep_1():
    time.sleep(1)


def sleep_1000():
    asyncio.sleep(1000)
```

Answer




```python
import asyncio

def sleep_1000():
    asyncio.sleep(1000)
```

Measured by

{codspeed}

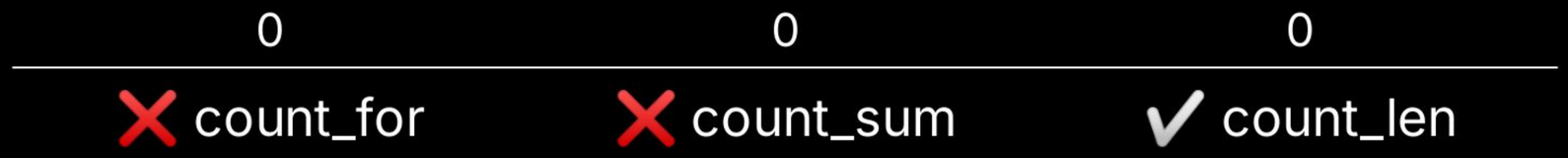
Case study #1: Counting even numbers in a list

```python
def count_for(arr: list[int]) -> int:
    even = 0
    for x in arr:
        if x % 2 == 0:
            even += 1
    return even

def count_sum(arr: list[int]) -> int:
    return sum(1 for x in arr if x % 2 == 0)

def count_len(arr: list[int]) -> int:
    return len([x for x in arr if x % 2 == 0])
```

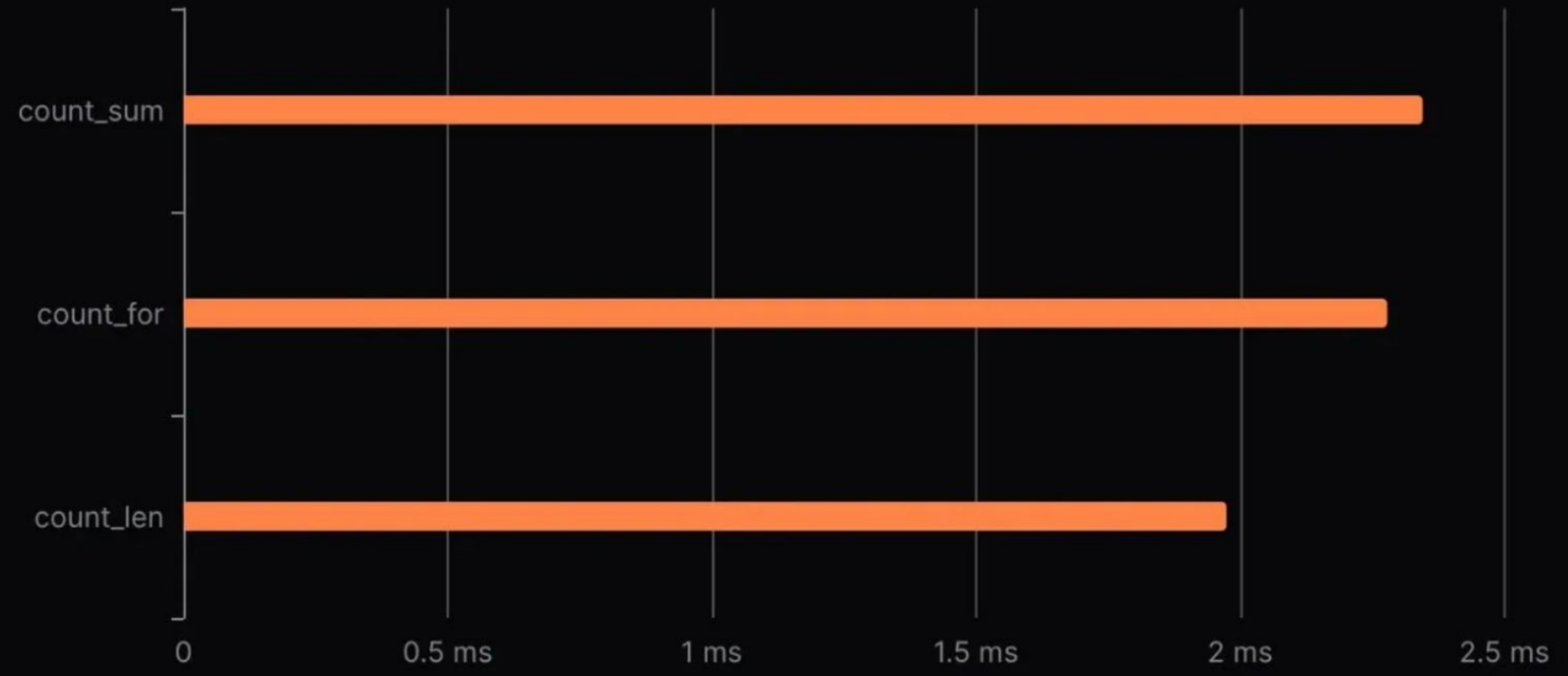
Input: list of size 10,000 with random integers

Which is faster?

```python
def count_for(arr: list[int]) -> int:
    even = 0
    for x in arr:
        if x % 2 == 0:
            even += 1
    return even

def count_sum(arr: list[int]) -> int:
    return sum(1 for x in arr if x % 2 == 0)

def count_len(arr: list[int]) -> int:
    return len([x for x in arr if x % 2 == 0])
```



Answer



`count_len` is faster

Measured by {codspeed}

Why?

```python
def count_sum(arr: list[int]) → int:
    return sum(1 for x in arr if x % 2 == 0)
```

Context switching between sum and generator

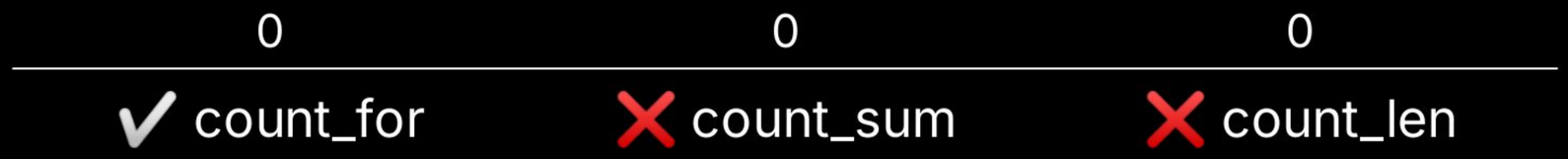
Why?

```python
def count_for(arr: list[int]) -> int:
    even = 0
    for x in arr:
        if x % 2 == 0:
            even += 1
    return even
```

The control flow overhead is bigger than using a bare `len` on a list which is O(1)

Which uses the least memory?

```python
def count_for(arr: list[int]) -> int:
    even = 0
    for x in arr:
        if x % 2 == 0:
            even += 1
    return even

def count_sum(arr: list[int]) -> int:
    return sum(1 for x in arr if x % 2 == 0)

def count_len(arr: list[int]) -> int:
    return len([x for x in arr if x % 2 == 0])
```



Memory consumption results

count_for:

Peak memory: 112 bytes

count_sum:

Peak memory: 408 bytes

count_len:

Peak memory: 4,167,400 bytes

`count_even` and `count_even_smart` are only using a few locals

`count_len` is allocating an array with a O(n) size

Measuring memory consumption

```python
import gc
import tracemalloc

def measure_memory_usage(func, *args):
    gc.collect() # Force garbage collection before measurement
    tracemalloc.start() # Start tracing

    func(*args) # Run the function

      # Get the memory usage
    current, peak = tracemalloc.get_traced_memory()

    tracemalloc.stop() # Stop tracing
    return current, peak
```

Using Python builtin `tracemalloc` to wrap functions

Learning

Execution speed sometimes comes at a **cost**,
here it is **memory**

Case study #2: String concatenation

```python
def concat_plus(strings: list[str]) -> str:
    result = ''
    for s in strings:
        result += s
    return result

def concat_join(strings: list[str]) -> str:
    return ''.join(strings)
```

Input: list of size 5,000 with random strings of size 1 to 16

Which is faster?

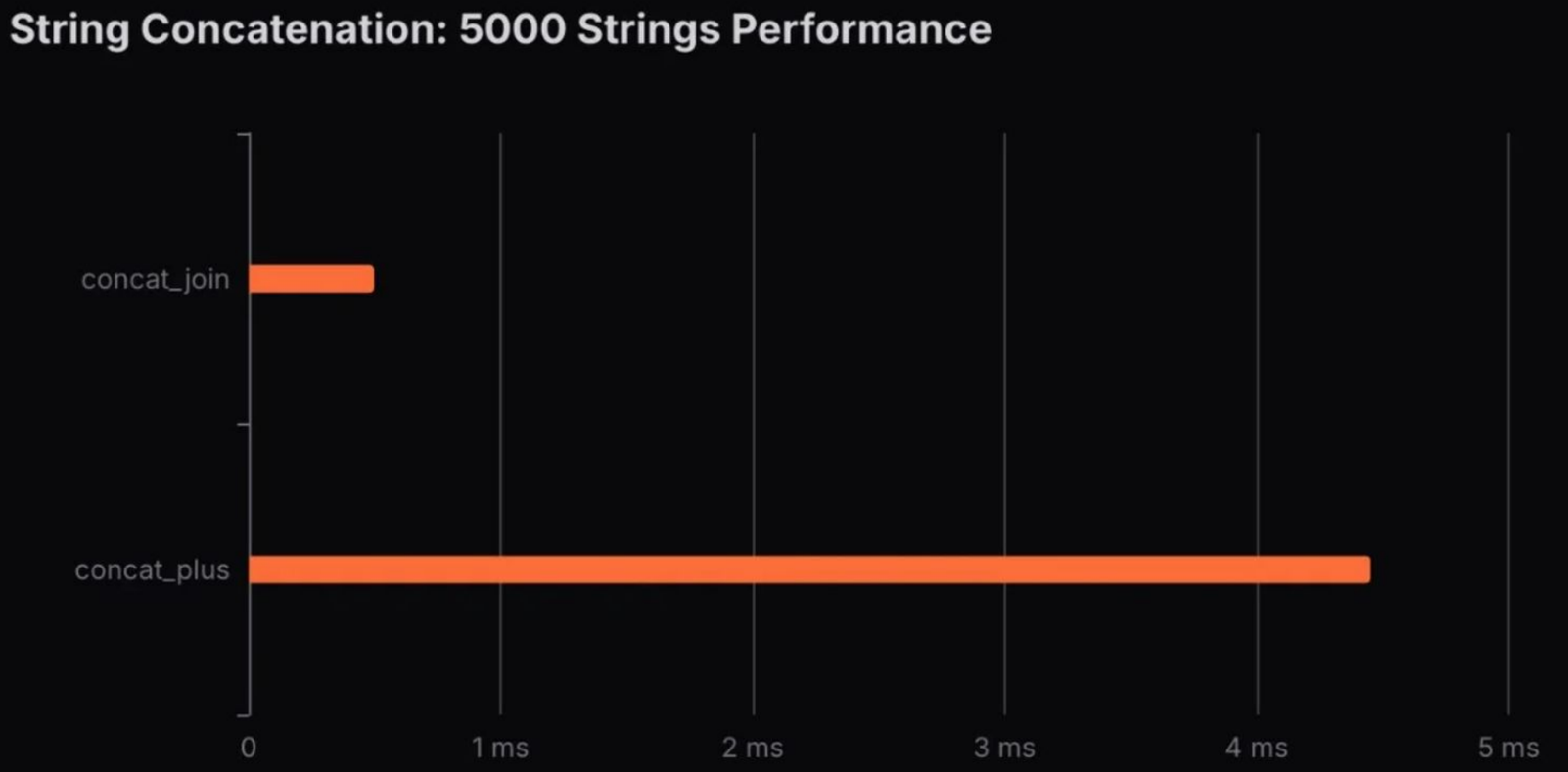
```python
def concat_plus(strings: list[str]) -> str:
    result = ''
    for s in strings:
        result += s
    return result

def concat_join(strings: list[str]) -> str:
    return ''.join(strings)
```

0	0
❌ concat_plus	✔️ concat_join

Let's measure!

{codspeed}

Answer



`concat_join` is faster

Measured by {codspeed}

Why?

```python
def concat_plus(strings: list[str]) -> str:
    result = ''
    for s in strings:
        result += s
    return result
```

*When using **+=**, the string is copied into a new larger string instance.*

Why?

```python
def concat_join(strings: list[str]) -> str:
    return ''.join(strings)
```

`join` pre-allocates the full string size and the copies each string once

Learnings

- Strings immutability leads to useless copies
- Use builtin functions when possible, they are often really well optimized

Case study #3: List comprehension vs map

```python
def list_comp(data: list[int]) -> list[int]:
    return [x * 2 for x in data]
```

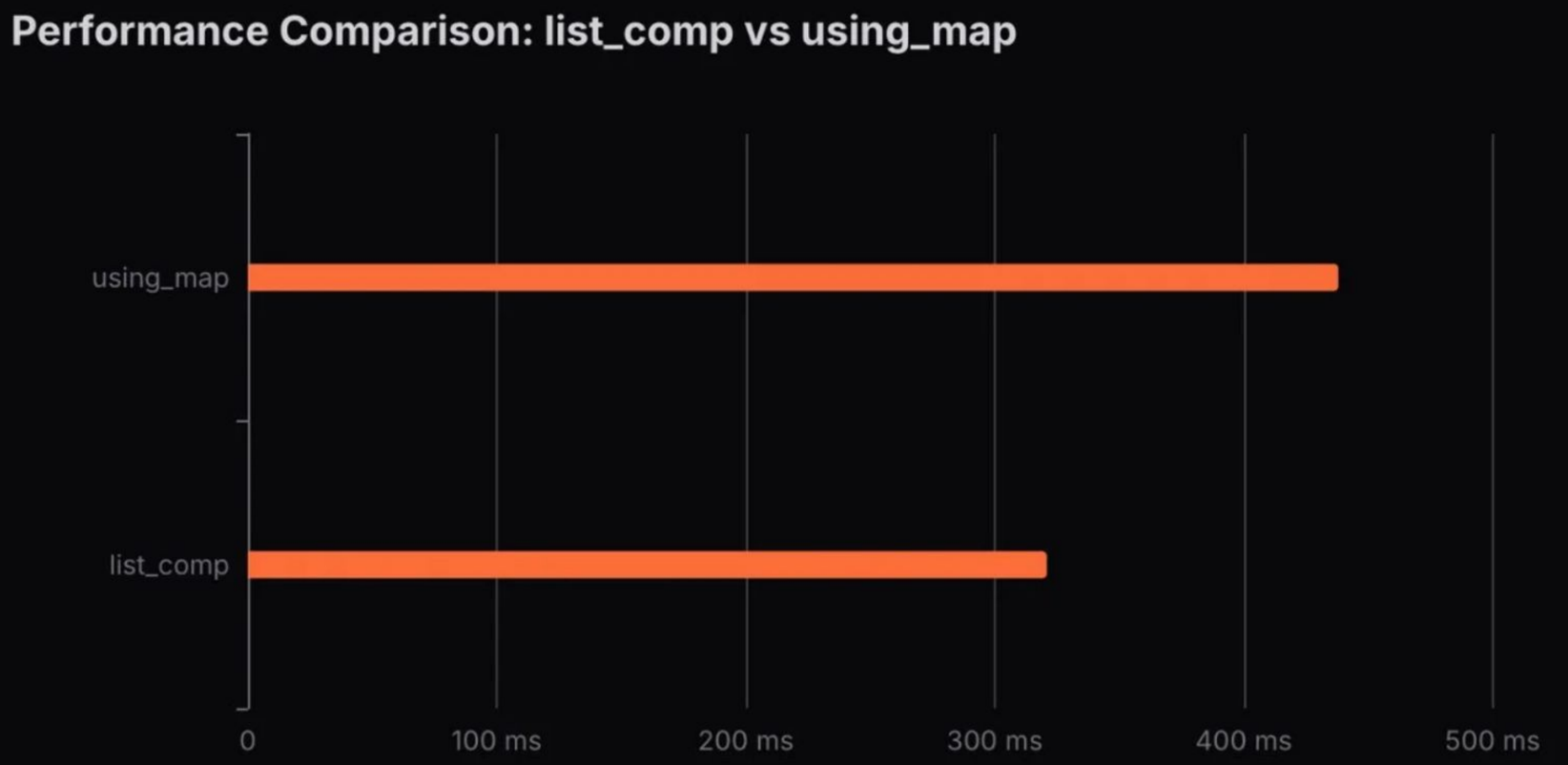
```python
def list_map(data: list[int]) -> list[int]:
    return list(map(lambda x: x * 2, data))
```

Input: list of size 1,000,000 with random ints from 0 to 9,999

Which is faster?

```python
def list_comp(data: list[int]) -> list[int]:
    return [x * 2 for x in data]


def list_map(data: list[int]) -> list[int]:
    return list(map(lambda x: x * 2, data))
```

Answer



`list_comp` is faster

Measured by

Why?

- Disassemble the code into Python Bytecode, low-level instructions passed to the interpreter

- Let’s use it to understand what it is going on

{codspeed}

Why?

```
Disassembly of list_comp:
--------------------------------------------------
  1           RESUME                   0

  2           LOAD_FAST                0 (data)
              GET_ITER
              LOAD_FAST_AND_CLEAR      1 (x)
              SWAP                     2
        L1:   BUILD_LIST               0
              SWAP                     2
        L2:   FOR_ITER                 7 (to L3)
              STORE_FAST_LOAD_FAST    17 (x, x)
              LOAD_CONST               1 (2)
              BINARY_OP                5 (*)
              LIST_APPEND              2
              JUMP_BACKWARD            9 (to L2)
        L3:   END_FOR
              POP_TOP
        L4:   SWAP                     2
              STORE_FAST               1 (x)
              RETURN_VALUE

   --   L5:   SWAP                     2
              POP_TOP

  2           SWAP                     2
              STORE_FAST               1 (x)
              RERAISE                  0
ExceptionTable:
  L1 to L4 -> L5 [2]
```

```
Disassembly of list_map:
--------------------------------------------------
  5           RESUME                   0

  6           LOAD_GLOBAL              1 (list + NULL)
              LOAD_GLOBAL              3 (map + NULL)
              LOAD_CONST               1 (<code object <lambda> at 0x10420e5d0, file
"~/project/list_map.py", line 6>)
              MAKE_FUNCTION
              LOAD_FAST                0 (data)
              CALL                     2
              CALL                     1
              RETURN_VALUE

Disassembly of <code object <lambda> at 0x10420e5d0, file "~/project/list_map.py",
line 6>:
  6           RESUME                   0
              LOAD_FAST                0 (x)
              LOAD_CONST               1 (2)
              BINARY_OP                5 (*)
              RETURN_VALUE
```

- `list_map` contains way less instructions and also call list/map both builtin methods implemented in CPython directly in C.

- Each iteration requires a context switch to the lambda(setup a new frame, pass arguments and teardown) to run its “simple” bytecode

Why?

```
Disassembly of list_comp:
--------------------------------------------------
  1           RESUME                   0

  2           LOAD_FAST                0 (data)
              GET_ITER
              LOAD_FAST_AND_CLEAR      1 (x)
              SWAP                     2
        L1:   BUILD_LIST               0
              SWAP                     2
        L2:   FOR_ITER                 7 (to L3)
              STORE_FAST_LOAD_FAST    17 (x, x)
              LOAD_CONST               1 (2)
              BINARY_OP                5 (*)
              LIST_APPEND              2
              JUMP_BACKWARD            9 (to L2)
        L3:   END_FOR
              POP_TOP
        L4:   SWAP                     2
              STORE_FAST               1 (x)
              RETURN_VALUE

   --   L5:   SWAP                     2
              POP_TOP

  2           SWAP                     2
              STORE_FAST               1 (x)
              RERAISE                  0
ExceptionTable:
  L1 to L4 -> L5 [2]
```

```
Disassembly of list_map:
--------------------------------------------------
  5           RESUME                   0

  6           LOAD_GLOBAL              1 (list + NULL)
              LOAD_GLOBAL              3 (map + NULL)
              LOAD_CONST               1 (<code object <lambda> at 0x10420e5d0, file
"~/project/list_map.py", line 6>)
              MAKE_FUNCTION
              LOAD_FAST                0 (data)
              CALL                     2
              CALL                     1
              RETURN_VALUE

Disassembly of <code object <lambda> at 0x10420e5d0, file "~/project/list_map.py",
line 6>:
  6           RESUME                   0
              LOAD_FAST                0 (x)
              LOAD_CONST               1 (2)
              BINARY_OP                5 (*)
              RETURN_VALUE
```

- `list_map` contains way less instructions and also call list/map both builtin methods implemented in CPython directly in C.

- Each iteration requires a context switch to the lambda(setup a new frame, pass arguments and teardown) to run its “simple” bytecode

Learning

- Adding a function and switching frame context creates overhead (inlining is not possible)

- Functions tend to make code easier to understand and write for humans, but they come at a cost in comparison to inlined code

Case study #4: Recursion vs Stack

```python
def recursive_factorial(n):
    if n == 0:
        return 1
    return n * recursive_factorial(n - 1)

def stack_factorial(n):
    result = 1
    stack = []
    while n > 0:
        stack.append(n)
        n -= 1
    while stack:
        result *= stack.pop()
    return result
```

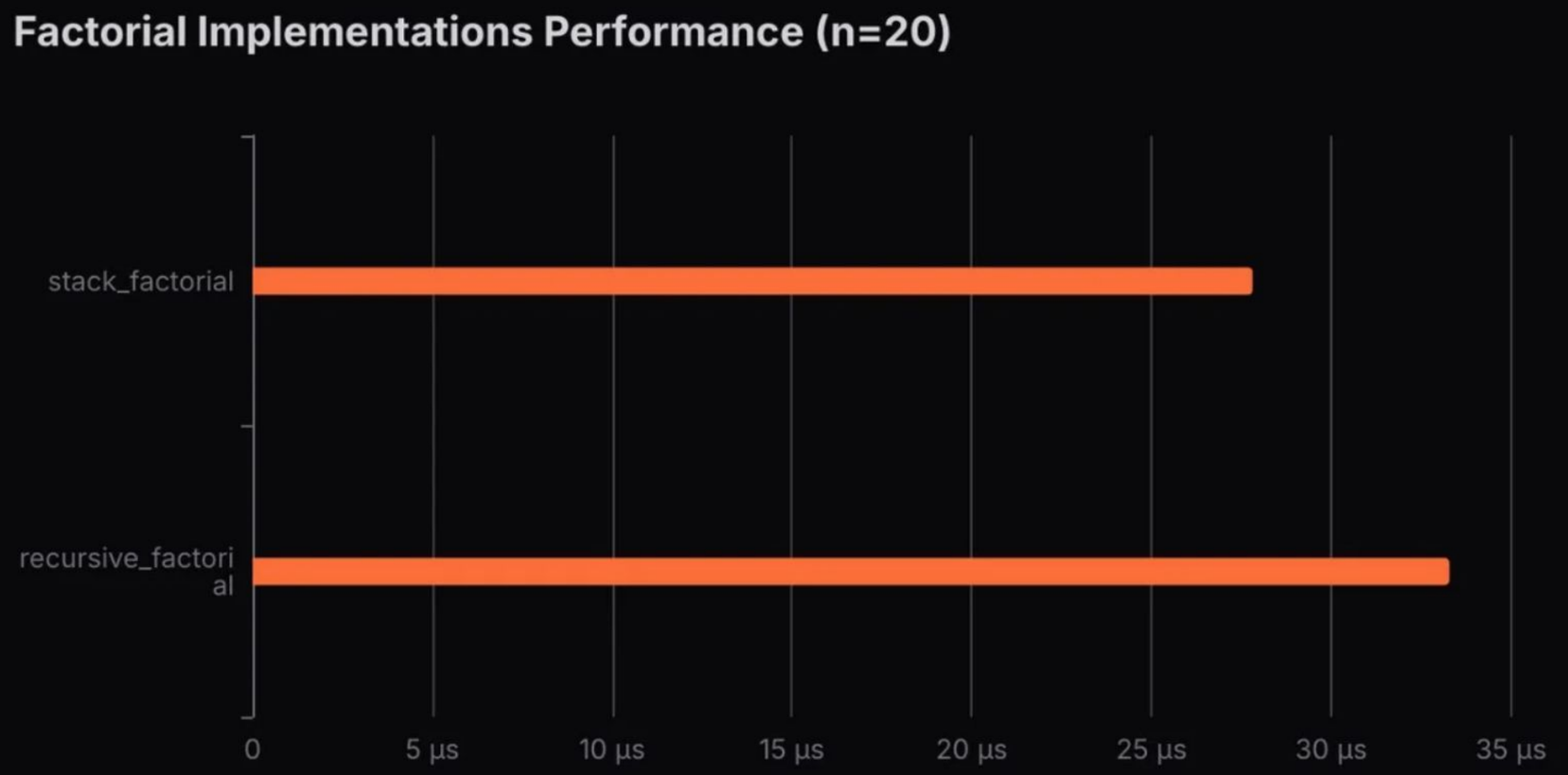
Input: n = 20

Which is faster?

```python
def recursive_factorial(n):
    if n == 0:
        return 1
    return n * recursive_factorial(n - 1)

def stack_factorial(n):
    result = 1
    stack = []
    while n > 0:
        stack.append(n)
        n -= 1
    while stack:
        result *= stack.pop()
    return result
```

{codspeed}

Answer



`stack_factorial` is faster

Measured by {codspeed}

Why?

```
Disassembly of recursive_factorial:
  1           RESUME                   0

  2           LOAD_FAST                0 (n)
              LOAD_CONST               1 (0)
              COMPARE_OP              88 (bool(==))
              POP_JUMP_IF_FALSE        1 (to L1)

  3           RETURN_CONST             2 (1)

  4     L1:   LOAD_FAST                0 (n)
              LOAD_GLOBAL              1 (recursive_factorial …)
              LOAD_FAST                0 (n)
              LOAD_CONST               2 (1)
              BINARY_OP               10 (-)
              CALL                     1
              BINARY_OP                5 (*)
              RETURN_VALUE
```

```
Disassembly of stack_factorial:
  6           RESUME                   0

  7           LOAD_CONST               1 (1)
              STORE_FAST               1 (result)

  8           BUILD_LIST               0
              STORE_FAST               2 (stack)

  9           LOAD_FAST                0 (n)
              LOAD_CONST               2 (0)
              COMPARE_OP             148 (bool(>))
              POP_JUMP_IF_FALSE       30 (to L2)

 10     L1:   LOAD_FAST                2 (stack)
              LOAD_ATTR                1 (append + NULL|self)
              LOAD_FAST                0 (n)
              CALL                     1
              POP_TOP

 11           LOAD_FAST                0 (n)
              LOAD_CONST               1 (1)
              BINARY_OP               23 (-=)
              STORE_FAST               0 (n)

  9           LOAD_FAST                0 (n)
              LOAD_CONST               2 (0)
              COMPARE_OP             148 (bool(>))
              POP_JUMP_IF_FALSE        2 (to L2)
              JUMP_BACKWARD           30 (to L1)

 12     L2:   LOAD_FAST                2 (stack)
              TO_BOOL
              POP_JUMP_IF_FALSE       27 (to L4)

 13     L3:   LOAD_FAST_LOAD_FAST      18 (result, stack)
              LOAD_ATTR                3 (pop + NULL|self)
              CALL                     0
              BINARY_OP               18 (*=)
              STORE_FAST               1 (result)

 12           LOAD_FAST                2 (stack)
              TO_BOOL
              POP_JUMP_IF_FALSE        2 (to L4)
              JUMP_BACKWARD           27 (to L3)

 14     L4:   LOAD_FAST                1 (result)
              RETURN_VALUE
```

{codspeed}

Here the difference is even bigger in term of instruction count.

But still the manual stack management is better → avoids pushing a new frame and switching context.

We can note though some calls but to builtin functions implemented in C and not new python code.

Learning

- Recursive functions can be easier to implement than functions using stack based recursion: **maintainability trade off**

- But there is an overheard of pushing frames and switching context

Case study #5: Rewriting it in Rust

- Moving Python code to Rust has been a recurring trend, most often to gain performance
- Call Rust from Python thanks to `rustimport` and PyO3

```rust
// rustimport:pyo3

use pyo3::prelude::*;

#[pyfunction]
fn square_rust(x: i32)
-> i32 {
    x * x
}
```

```python
import rustimport.import_hook
from rust.mult_rs import square_rust


def square_py(x: int) -> int:
    return x * x


def square_rs(x: int) -> int:
    return square_rust(x)
```

Input: x = 10,000

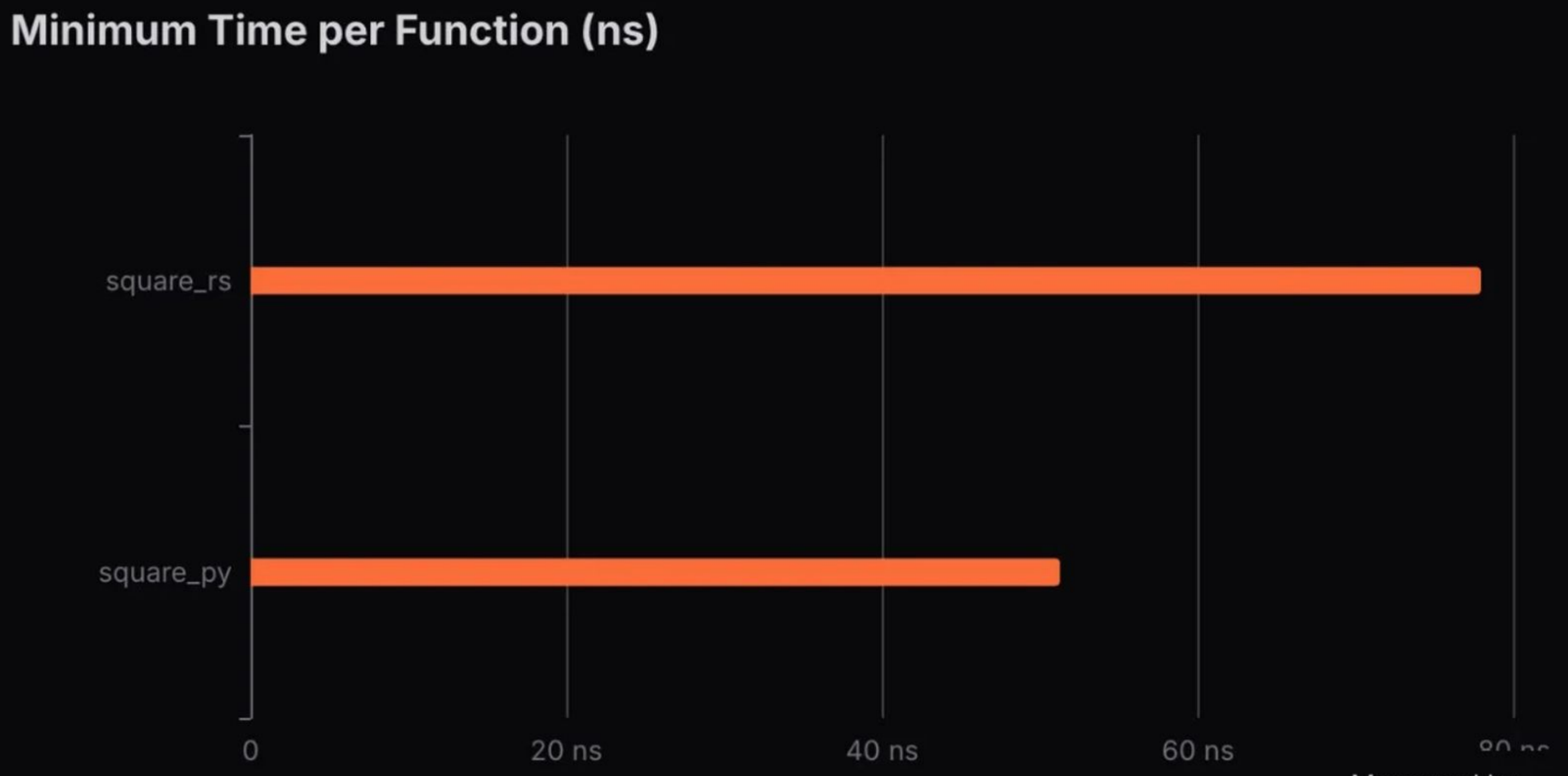
Which is faster?

```python
import rustimport.import_hook
from rust.mult_rs import square_rust


def square_py(x: int) -> int:
    return x * x


def square_rs(x: int) -> int:
    return square_rust(x)
```

0	0
✔️ Python	❌ Rust

{codspeed}

Answer



`square_py` is faster

Measured by {codspeed}

Why?

```rust
// rustimport:pyo3

use pyo3::prelude::*;

#[pyfunction]
fn square_rust(x: i32) -> i32 {
    x * x
}
```

```python
import rustimport.import_hook  # noqa: F401
from rust.mult_rs import square_rust


def square_py(x: int) -> int:
    return x * x


def square_rs(x: int) -> int:
    return square_rust(x)
```

The overhead of converting Python data to Rust types leads to slower performance for small algorithms.

Which is faster? Rust fibonacci

```rust
// rustimport:pyo3

use pyo3::prelude::*;

#[pyfunction]
fn fibonacci_rust(n: u128) -> u128 {
    if n <= 1 {
        return n;
    }

    let mut a = 0;
    let mut b = 1;

    for _ in 2..=n {
        let temp = a + b;
        a = b;
        b = temp;
    }

    b
}
```

```python
import rustimport.import_hook
from rust.fibonacci_rs import fibonacci_rust


def fibonacci_py(n: int) -> int:
    if n <= 1:
        return n

    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b

    return b


def fibonacci_rs(n: int) -> int:
    return fibonacci_rust(n)
```

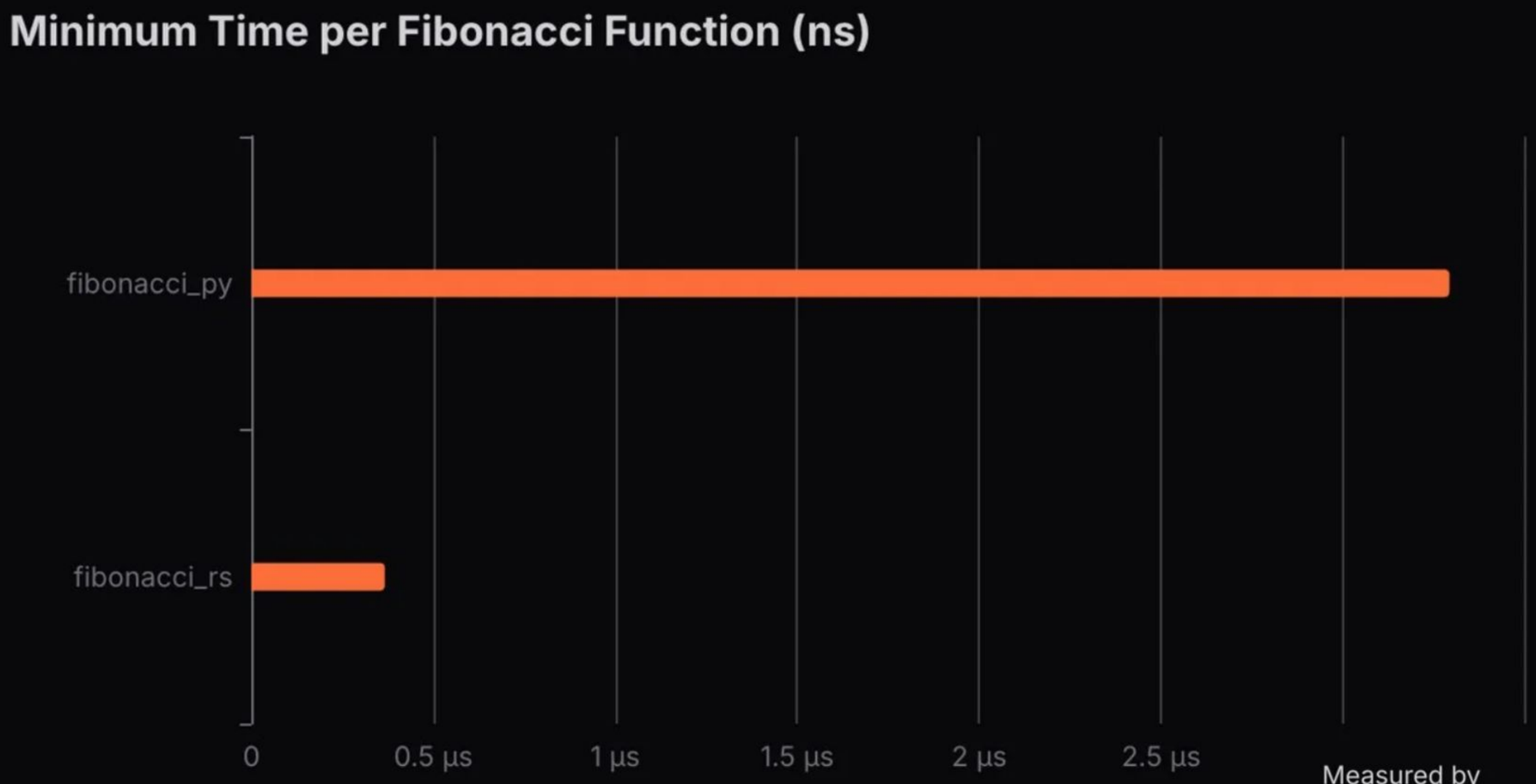
Input → n = 150

Which is faster?

```python
import rustimport.import_hook
from rust.fibonacci_rs import fibonacci_rust


def fibonacci_py(n: int) -> int:
    if n <= 1:
        return n

    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b

    return b


def fibonacci_rs(n: int) -> int:
    return fibonacci_rust(n)
```

{codspeed}

Answer



Measured by {codspeed}

`fibonacci_rs` is faster

Why?

- In Rust → Native CPU code, purely algorithmic

- Faster than the CPython interpreter bytecode execution

Which is faster? Rust binary search

```rust
// rustimport:pyo3

use pyo3::prelude::*;
use pyo3::{
    pyfunction,
    types::{PyAnyMethods, PyList, PyListMethods},
};

#[pyfunction]
fn binary_search_rust(sorted_list: Vec<u64>, target: u64) -> Option<u64> {
    let mut left = 0;
    let mut right = sorted_list.len() as u64 - 1;

    while left <= right {
        let mid = (left + right) / 2;
        let mid_idx = mid as usize;

        if sorted_list[mid_idx] == target {
            return Some(mid);
        } else if sorted_list[mid_idx] < target {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    None
```

```python
import rustimport.import_hook  # noqa: F401
from rust.binary_search_rs import binary_search_rust


def binary_search_py(sorted_list, target):
    left, right = 0, len(sorted_list) - 1

    while left <= right:
        mid = (left + right) // 2

        if sorted_list[mid] == target:
            return mid
        elif sorted_list[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1


def binary_search_rs(sorted_list, target):  # noqa: F811
    return binary_search_rust(sorted_list, target)
```

Input: list of size 10,000 with sorted random integers

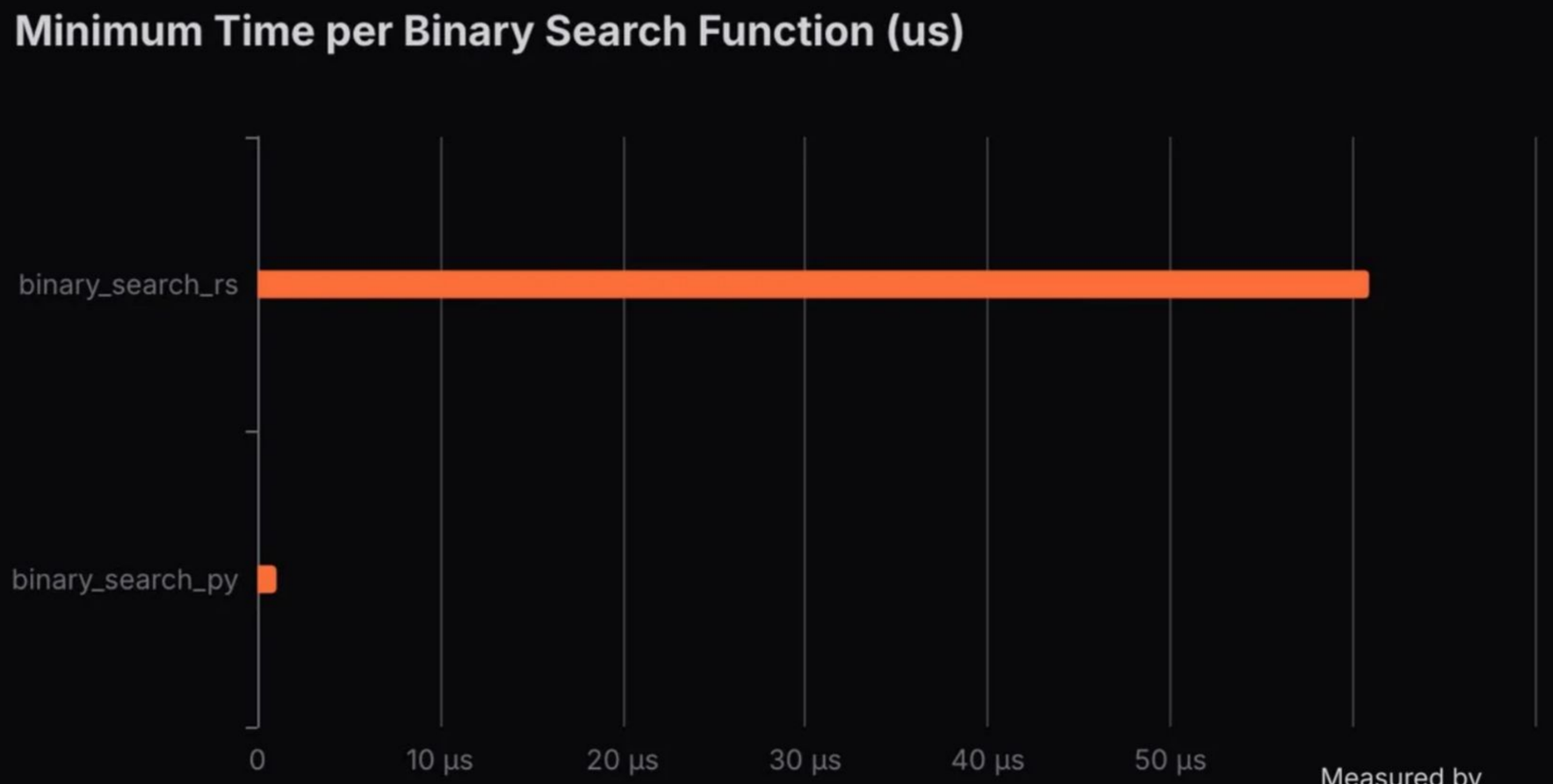
Which is faster?

```python
import rustimport.import_hook  # noqa: F401
from rust.binary_search_rs import binary_search_rust


def binary_search_py(sorted_list, target):
    left, right = 0, len(sorted_list) - 1

    while left <= right:
        mid = (left + right) // 2

        if sorted_list[mid] == target:
            return mid
        elif sorted_list[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1


def binary_search_rs(sorted_list, target):  # noqa: F811
    return binary_search_rust(sorted_list, target)
```

0	0
✔️ Python	❌ Rust

{codspeed}

Answer



`binary_search_py` is faster

Measured by {codspeed}

Why?

- Transforming a Python list into a Rust's `Vec<u64>`.
- This conversion cost is in `O(n)`, whereas the algorithm is in `O(log(n))`.

→ As the input grows, the rust implementation becomes even worse.

Using Python native types

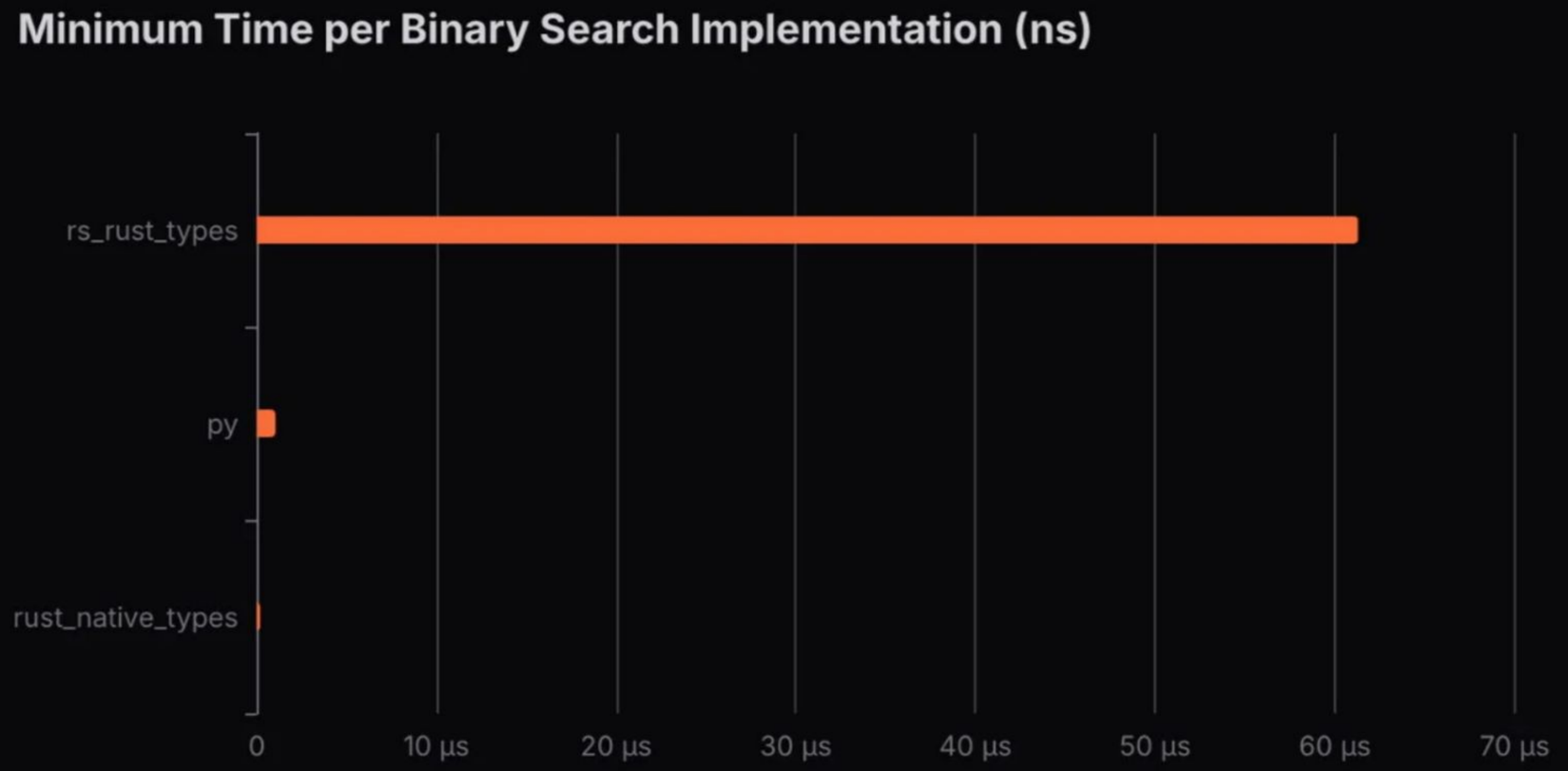
Since only log(n) elements will be accessed, we can only convert those lazily instead of the whole list.

```rust
#[pyfunction]
fn binary_search_rust_rust_types(sorted_list: Vec<u64>, target:
u64) -> Option<u64> {
    let mut left = 0;
    let mut right = sorted_list.len() as u64 - 1;

    while left <= right {
        let mid = (left + right) / 2;
        let mid_idx = mid as usize;

        let mid_value = sorted_list[mid_idx];
        if mid_value == target {
            return Some(mid);
        } else if mid_value < target {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    None
}
```

```rust
#[pyfunction]
fn binary_search_rust_native_types(
    sorted_list: &pyo3::Bound<'_, PyList>,
    target: u64,
) -> Option<u64> {
    ...
        let mid_value = sorted_list
            .get_item(mid_idx)
            .unwrap()
            .extract::<u64>()
            .unwrap();
    ...

}
```

{codspeed}

Results



rust_native_types is faster

Measured by {codspeed}

Learnings on rewriting in Rust

- For small functions or already optimized builtin functions, not necessary
- Native algorithmic code is way faster than the interpreted execution
- Be careful of the overheard of converting data from and to native types, it can become really costly

Recap

- Be careful about human intuition, always measure!
- Depending on the use case and data, some algorithms are faster or slower, measure!
- ... And do not forget to measure!

{codspeed}

Leaderboard

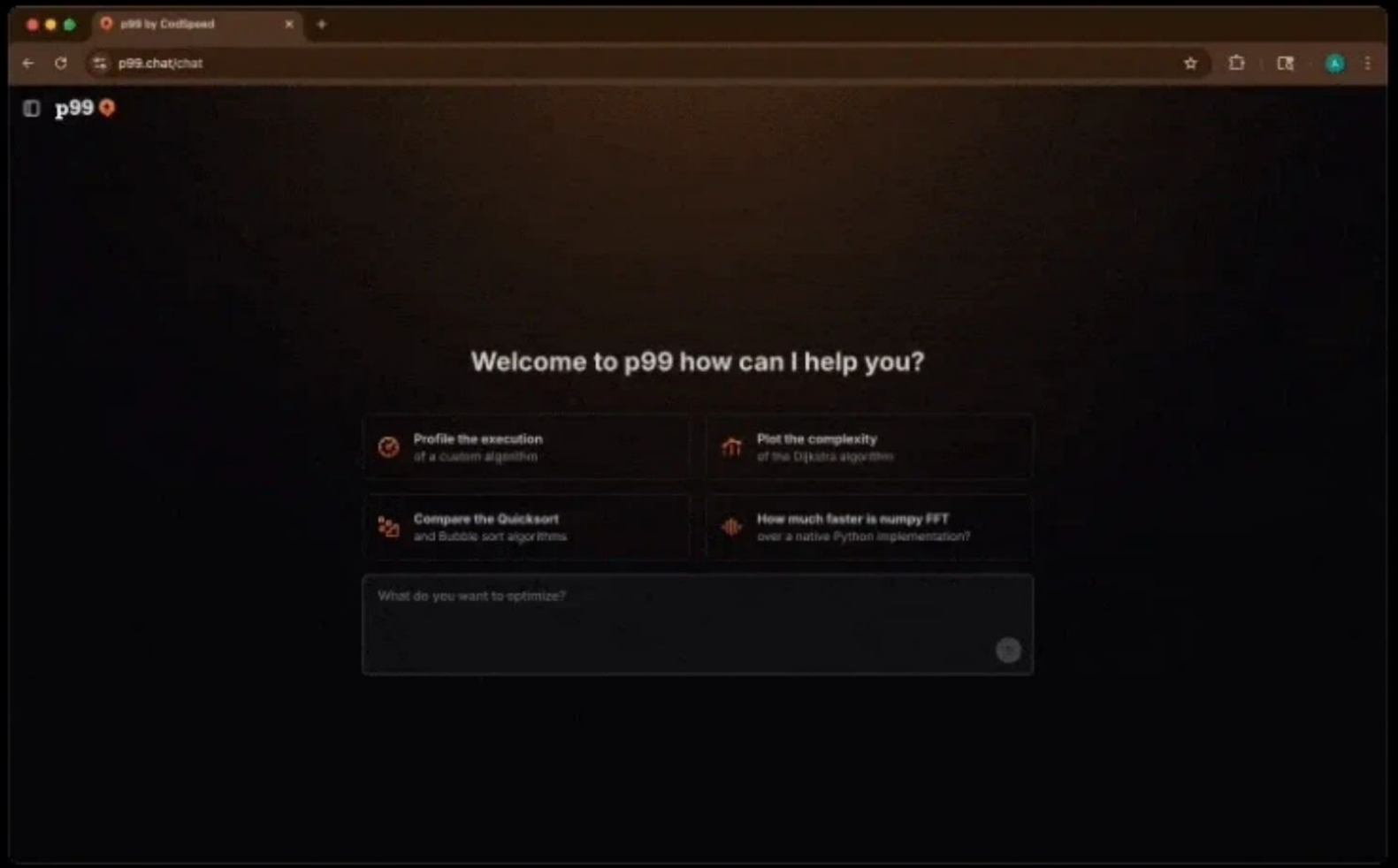
0 players

No result yet

The top Quiz players will be displayed here when there are results.

{codspeed}

We just released p99.chat: a performance lab in your browser



Try it out!

p99

Thank you!