# Performance Analysis
## of Python Runtimes

Mohana Evuri, 23B1017
CSE, IIT Bombay

Fall 2025

## Contents

**Abstract**

This report benchmarks CPython, Numba, PyPy, and Cython using a mix of custom benchmarks and some standard tests. By measuring the performances of interpreted code, compiled code and JIT execution (under identical conditions), we compare execution speed and try to identify for which workloads which runtime performs better.

# 1 Introduction

Python has several execution environments that differ in design, performance goals, and compilation strategies.

## 1.1 CPython

CPython is the reference implementation and the default **interpreter** used by most users. It compiles Python source to bytecode and executes it on a virtual machine, prioritizing portability and compatibility.

## 1.2 Numba

Numba is a **just-in-time compiler** built on LLVM. It accelerates numerical Python code by compiling selected functions to machine code at run-time, often delivering substantial speedups for array-oriented and numeric workloads with minimal code changes.
Numba is activated through decorators that wrap a function. Some of the wrappers used are `@njit`, `@jit(nopython=true)`. When the function is first called, the wrapper triggers Numba's JIT compiler, which specializes and compiles the function to machine code; after that, it runs the compiled version for all the future calls.
It is shipped as a Python library and can be imported like any other package.

## 1.3 PyPy

PyPy is an alternative Python **interpreter** that includes a **tracing just-in-time** compiler. Its goal is to improve the performance of long-running programs by optimizing frequently executed paths while remaining largely compatible with standard Python.
The difference between PyPy and Numba is that PyPy is an interpreter which tries to detect the parts of the Python code which can be optimized on its own, whereas with Numba, we mention the part of the code we want to optimize with JIT. Also, Numba is more of a numerical Python code optimizer.

### 1.4 Cython

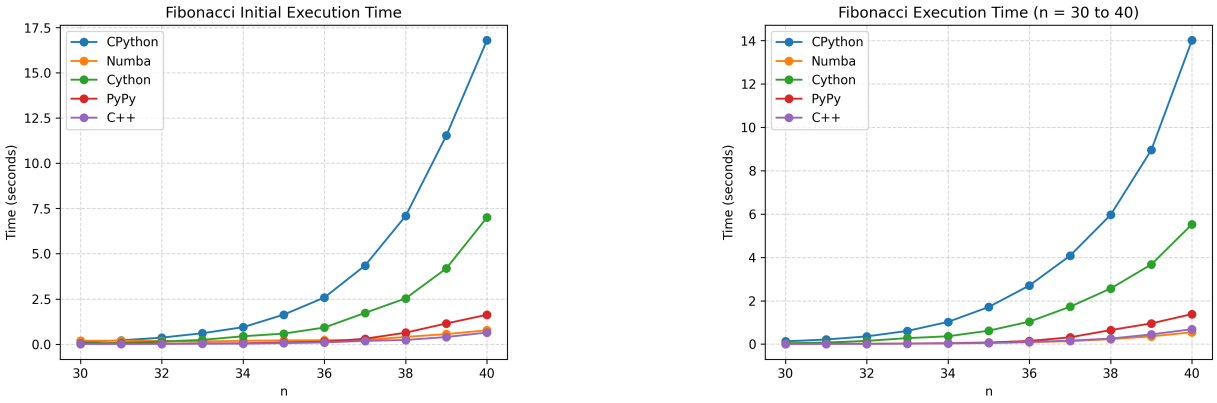
Cython is a **static compiler** that translates annotated **Python-like code** into C. By allowing optional type declarations, it produces efficient C extensions that can significantly outperform pure Python, especially in compute-heavy loops.
Cython code is invoked by compiling `.pyx` files into C and then building them as Python extension modules. This is done with build tools such as `setuptools` and `cythonize`, which generate the C source, compile it with a system compiler, and produce a loadable `.so` module. Once built, the `.pyx` module is imported and used like any regular Python module, with calls dispatched directly into the compiled C code.
It is also shipped as a Python package.
This report compares these four tools to highlight their performance differences across custom and standard benchmarks.

## 2 My Benchmarks

This section summarizes the custom benchmarks made by me, that I used to compare the performances of CPython, Numba, Cython, and PyPy. The tests cover recursion, numeric loops, nested loops, and string operations, with performance graphs included for each case.

### 2.1 Fibonacci

Calculation of the nth fibonacci number via pure recursion.
A pure recursive benchmark stressing call overhead and interpreter efficiency.



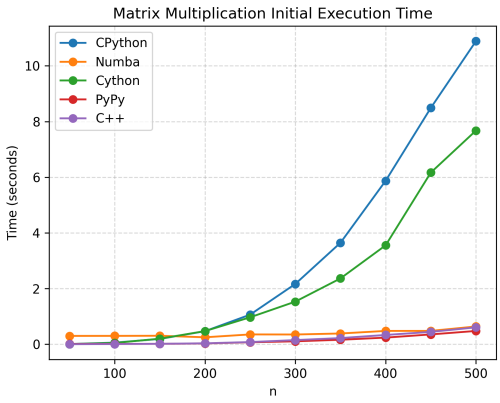

(a) Initial run

(b) Average performance

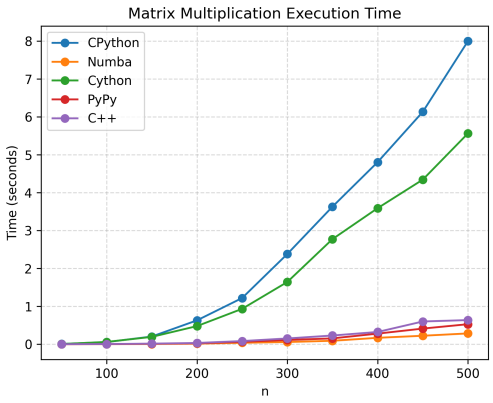
Figure 1: Fibonacci Benchmark results

### 2.2 Matrix Multiplication

Multiplying 2 matrices via nested loops.
A manual nested-loop matrix multiply stressing arithmetic throughput and cache behavior.




(a) Initial run

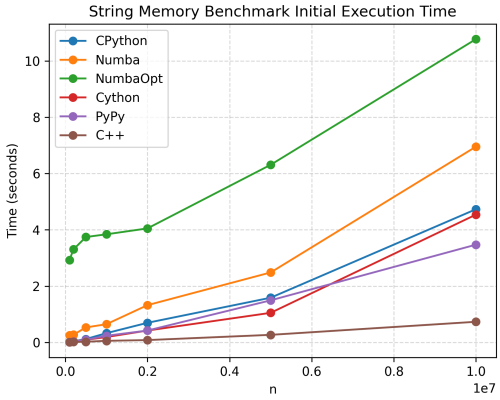



(b) Average performance

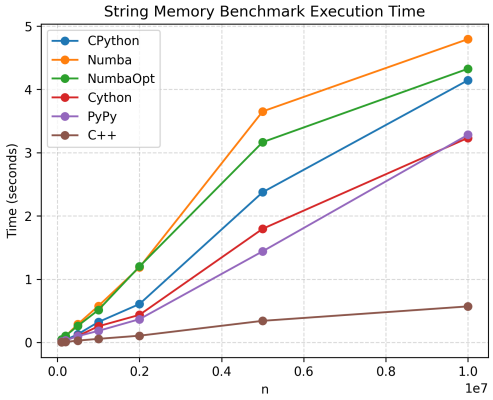
Figure 2: Matrix Multiplication Benchmark results

## 2.3 String and Memory Test

This test performs repeated string creation and manipulation to evaluate memory allocation costs and object-handling efficiency across the different runtimes.




(a) Initial run

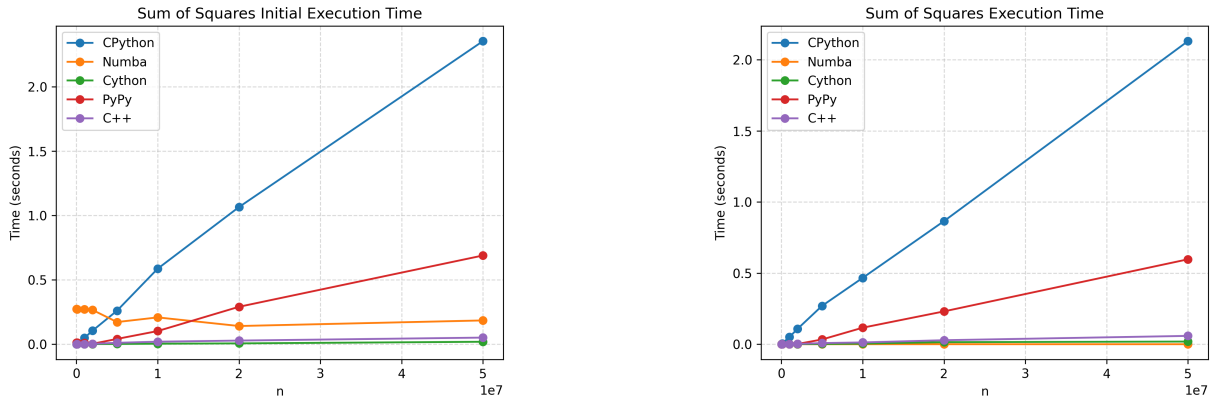



(b) Average performance

Figure 3: String and Memory Benchmark results

## 2.4 Sum of Squares

Sum of the first n squares.
A loop-heavy arithmetic benchmark that isolates basic numeric throughput.




(a) Initial run

(b) Average performance

Figure 4: Sum of Squares Benchmark results

## 2.5 Observations

After observing the results of the above mentioned custom benchmarks, we can get a few insights: The first call to the function takes significantly more time than the next few runs in JIT. This is because in the first call, the function is compiled and optimized before running, and the future runs use this compiled optimized version instead of the normal one.
Quite unexpectedly, the Cython variant is not as fast as the C++ implementation or the JIT compilation of the code in general, but is faster than the CPython interpreter. This indicates that dynamically typed languages such as Python are more suited for dynamic optimization via JIT compilers, instead of being converted into a statically typed language.
PyPy does a pretty good job of taking the same Python code (which CPython takes) and optimizing the run wherever it can. Numba is pretty good at pure mathematical operations, but when we introduce non numerical objects, such as dictionaries and strings, it loses its speed. Numba can be fine tuned with different wrappers, as seen in the Numba Optimized variant.
The code for these benchmarks is available in the GitHub repo mentioned in the references. Apart from the benchmark programs, the C code generated by the Cython compiler is also added, which can be further analysed to see the efficiency of the code conversion from Python to C.

# 3 Standard Benchmarks

Apart from the custom benchmarks, there are several benchmarking suites and tools commonly used for evaluating Python performance.

## 3.1 Benchmark suites

- **PyPerformance**: The standard benchmark suite used for comparing Python interpreters. It runs a fixed set of tests through the `pyperf` engine and requires no code modifications, making it suitable for comparing CPython and PyPy directly.
- **Numba Benchmarks**: The official benchmark suite for Numba. It targets older Python versions (notably Python 3.6) and focuses on numerical benchmarks and also some GPU benchmarks.

## 3.2 Benchmarking tools

- **pyperf**: Provides process isolation, warmups, and robust statistics. This is the central framework used by the PyPerformance benchmark suite.
- **ASV**: A benchmarking framework for Python projects. It runs user-defined benchmarks, tracks performance across commits and releases, and produces reproducible, versioned performance histories.

I couldn't find any standard benchmarking suite for Cython.

## 3.3 CPython vs PyPy

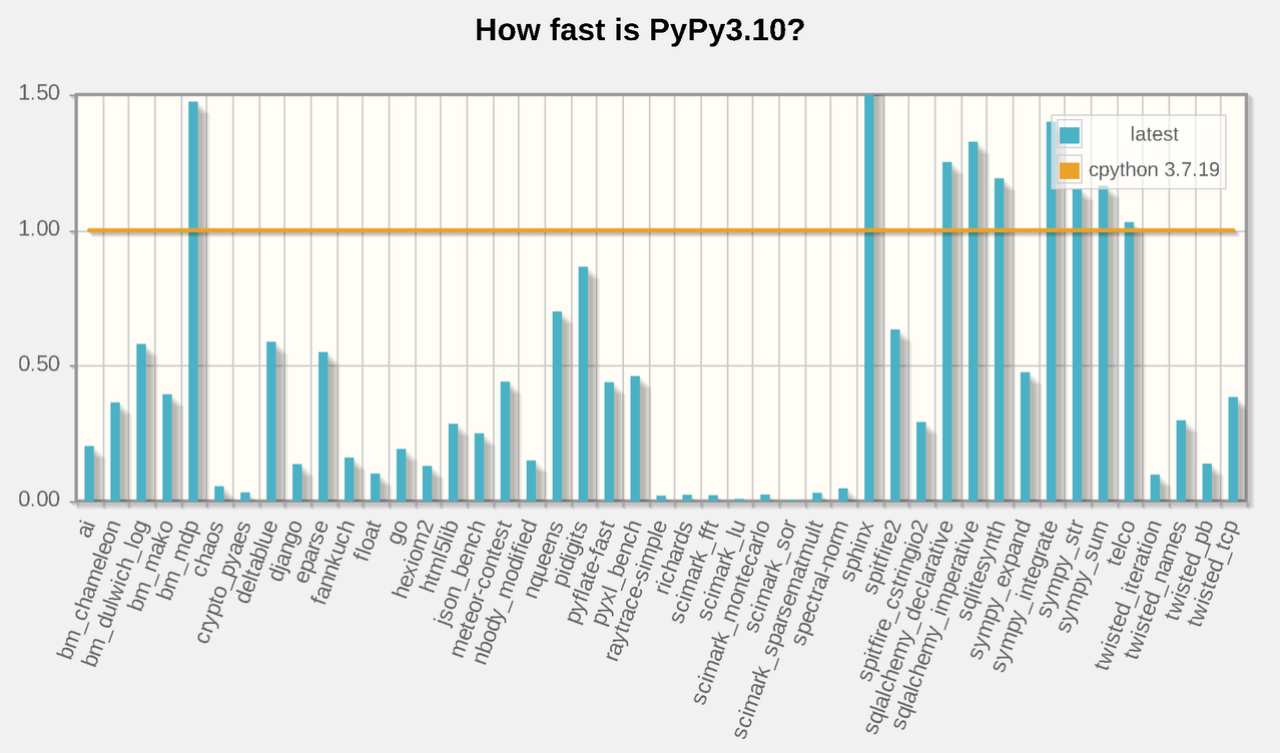



Figure 5: PyPy vs CPython3.7 - Taken from the PyPy website

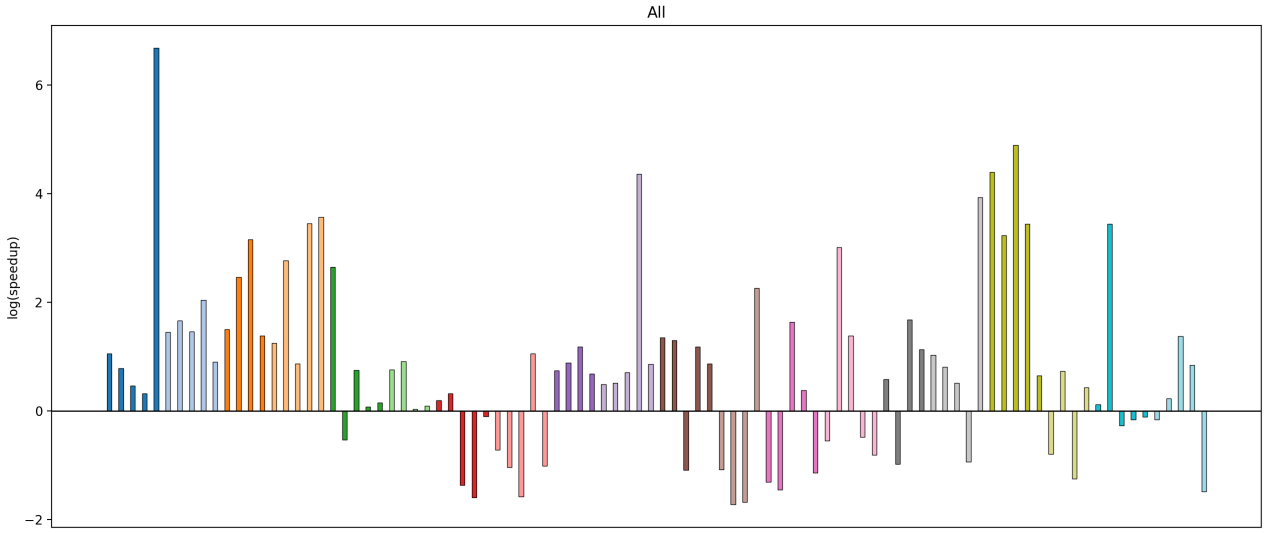



Figure 6: PyPy7.3 vs CPython3.11 - Benchmarks (without labels)

**Versions used**: CPython 3.11 (vs) PyPy 7.3 (implements Python3.11)
These results mentioned in Figure 6 are computed using the PyPerformance benchmark suite to compare the performances of the CPython and the PyPy interpreters.
The test suite was run on around 80 benchmark programs, which are grouped into the above categories such as FileIO, WebFrameworks, Concurrency, Core Language / Interpreter among others.
We will go through each of the subcategories and note down the inferences:

### 3.3.1 Core Language / Interpreter

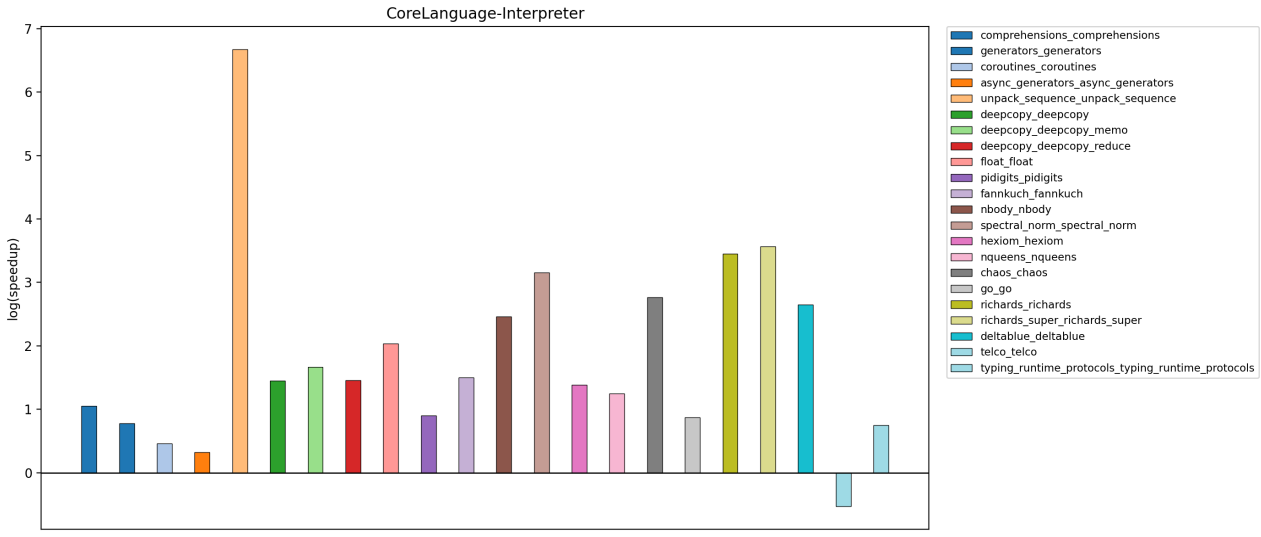
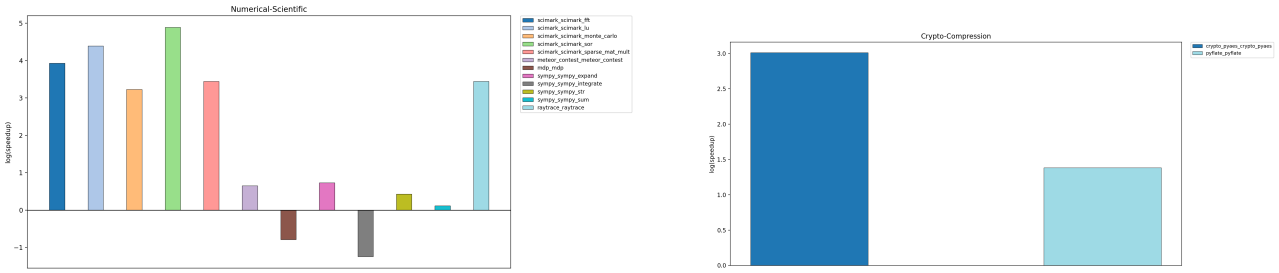



Figure 7: CoreLanguage-Interpreter results

PyPy is faster than CPython in almost every CoreLanguage-Interpreter benchmark. The improvements are typically in the range of $2 - 35\times$ faster execution for pure-Python and object-heavy workloads. These results align with PyPy's JIT compiled execution model,

which optimizes on tight numeric kernels, loops, and Object oriented workloads. This shows that PyPy performs better for pure-Python computational workloads.

### 3.3.2 Numerical




(a) Numerical-Scientific results (b) Crypto-Compression results

Figure 8: Numerical and math intensive

PyPy is faster here too: it has large gains on tight, arithmetic-heavy loops, with SciMark and ray-tracing tests seeing $25 - 130\times$ speedup and crypto/deflate benchmarks also improving sharply. So, we can tell that PyPy performs better in pure numerical and mathematical benchmarks too.

### 3.3.3 Pickle and Serialization

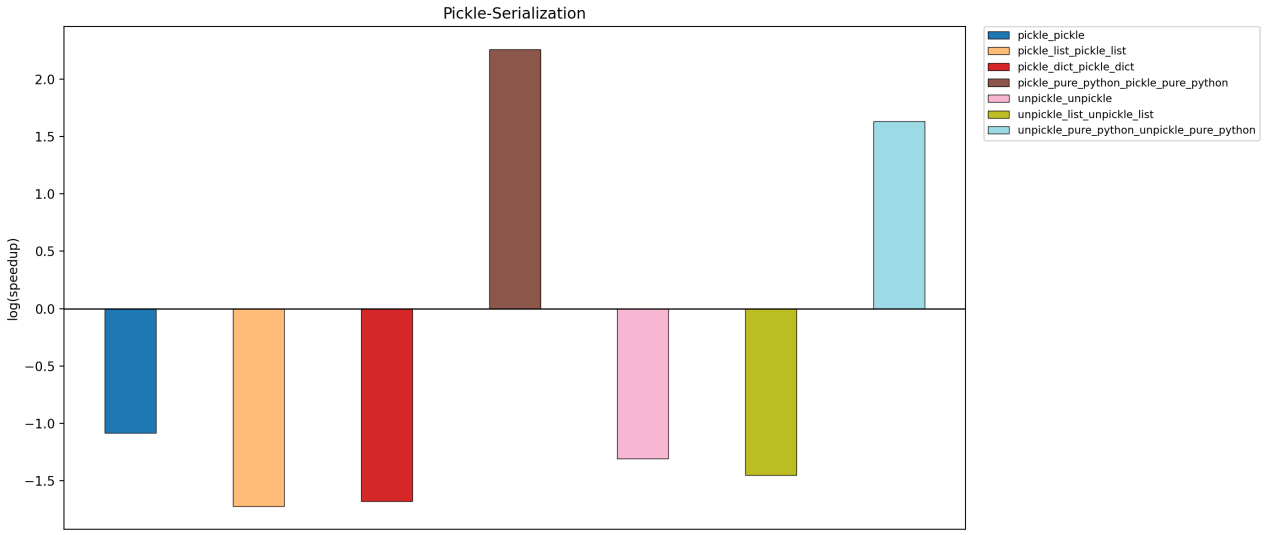



Figure 9: Pickle-Serialization results

CPython performs better on Pickle benchmarks than PyPy. PyPy is around $3 - 6\times$ slower than CPython as the latter relies on CPython's optimized C implementations for lists, dicts, and generic objects. The only cases where PyPy wins are the pure-Python fallback implementations, which its JIT can optimize better. Overall, workloads that hit CPython's C fast-paths performs better in CPython than PyPy as PyPy lacks the equivalent C accelerators, while pure-Python serialization is the opposite.

### 3.3.4 Async and Concurrency

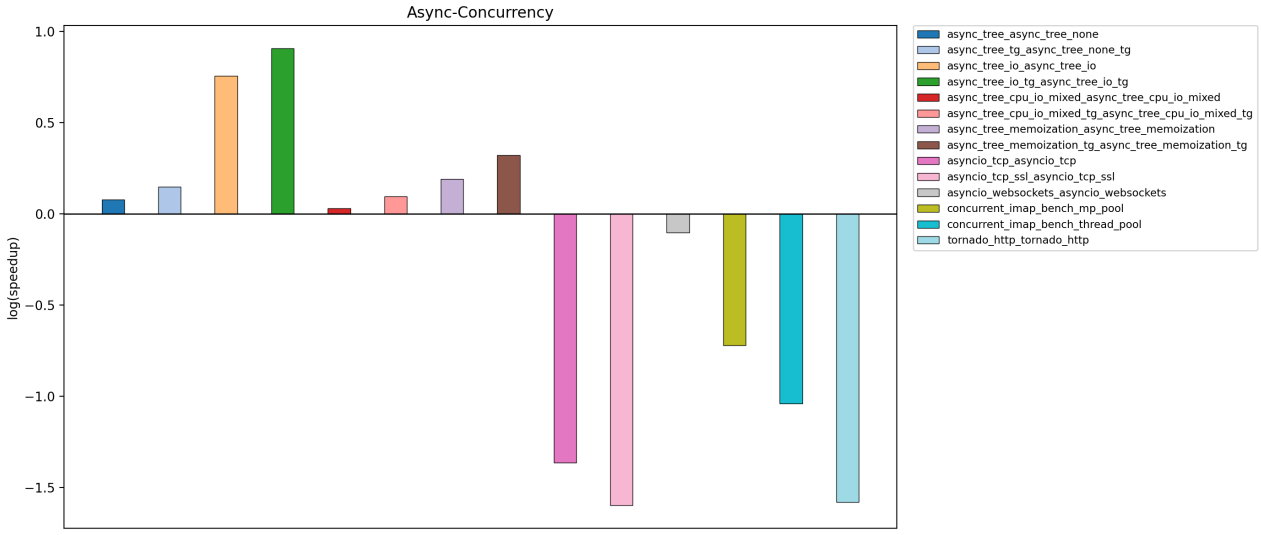



Figure 10: Async-Concurrency results

The performance in the Async and concurrency workloads are mixed but lean more towards CPython: PyPy is slightly faster up to 2.5× in pure async scheduling tests that stay within Python-level task creation and tree traversal, where its JIT can optimize coroutine-heavy traces. However, anything involving real networking, SSL, multiprocessing, thread pools, or event-loop integrations shows PyPy falling behind, often by $4-5\times$.

### 3.3.5 IO, Formatting and Parsing

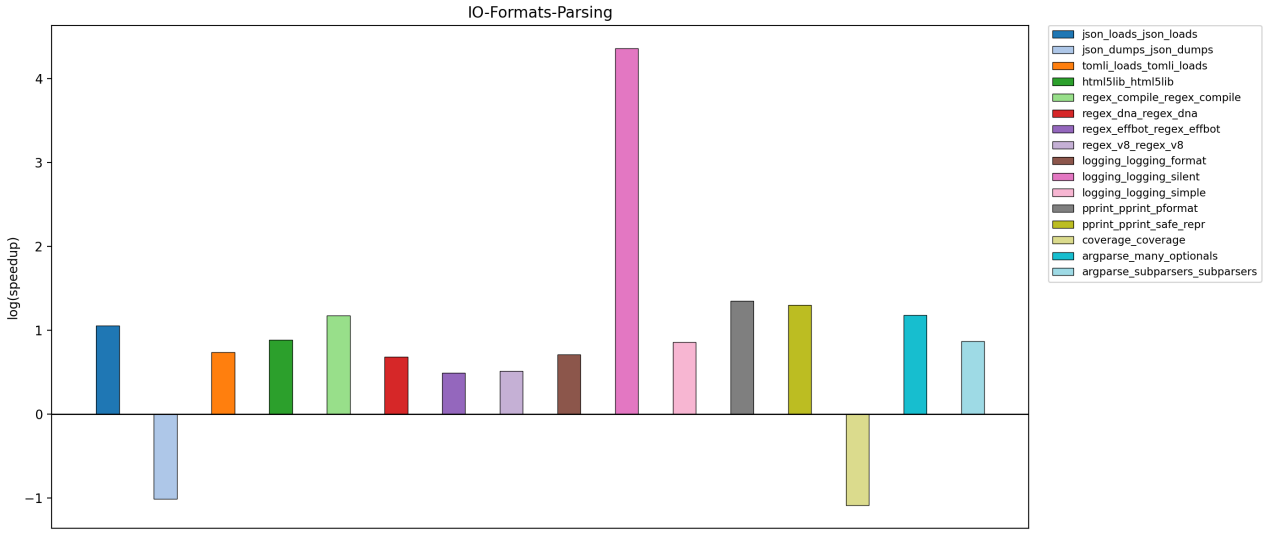


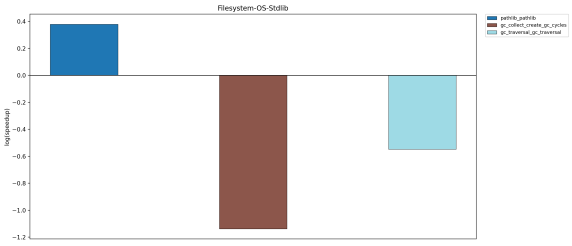

Figure 11: IO-Formats-Parsing results

PyPy performs better in IO, formats, and parsing tasks when the workload is pure Python-JSON loads, TOML parsing, HTML parsing, regex microbenchmarks, logging, pprint, and argparse. They all show $2-4\times$ speedup, with a few extreme wins on very small operations.

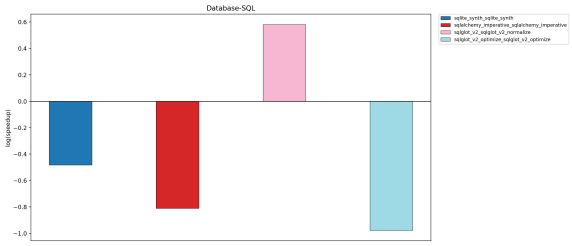
The outliers are for the benchmarks such as JSON dumps and coverage, where CPython's optimized C implementations come into the picture. Overall, Python-level parsing benefits from PyPy's JIT.

### 3.3.6 Other categories

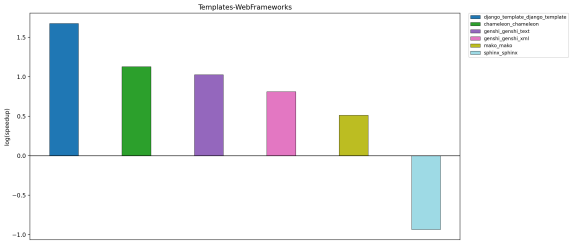
Some of the other categories are: Database / SQL, Filesystem / OS / Stdlib, Templates / Web Frameworks, ThirdParty / Domain Workloads and Tokenizers / Language Tools.



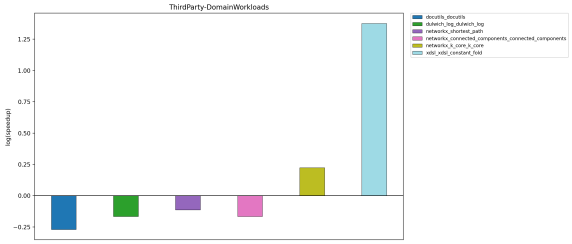

(a) Filesystem, OS and Stdlib



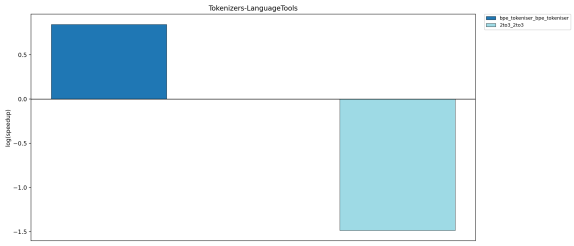

(b) Database / SQL




(c) Templates / Web Frameworks




(d) Third Party / Domain Workloads




(e) Tokenizers / Language Tools

Figure 12: Other categories

The results for the benchmarks have variations over the various benchmarks in these categories. One observation what we can get is that For the Templates / Web Frameworks benchmarks, PyPy performs slightly better than CPython in 5 out of the 6 benchmark tests, but overall, none of them have a considerable huge speedup.

# 4 References

- My GitHub code repository:
  `https://github.com/rennaMAhcuS/PythonEfficiency`
- PyPerformance benchmark suite:
  `https://github.com/python/pyperformance`
- pyperf benchmarking toolkit (used by PyPerformance):
  `https://github.com/psf/pyperf`
- Numba benchmark suite:
  `https://github.com/numba/numba-benchmark`
- Official Numba benchmark results (Python 3.6):
  `https://numba.readthedocs.io/en/stable/user/benchmarking.html`
- Airspeed Velocity (ASV) - Python benchmarking framework:
  `https://github.com/airspeed-velocity/asv`
- PyBenchmarks (microbenchmark collections):
  `https://pybenchmarks.org`