Performance Comparison of Python Runtime Systems: CPython, PyPy, and Jython

Ege Demirci, Divyanshu Bhardwaj
Winter 2025 - CMPSC 263 Runtime Systems
UC Santa Barbara

March 14, 2025

1 Introduction

Python is a high-level, dynamically typed programming language that has evolved significantly since its inception in the early 1990s by Guido van Rossum [1]. Because of its elegant syntax and readability, Python has become a language of choice in various domains—from web development and scripting to scientific computing and data analysis. As detailed in *Learning Python: Powerful Object-Oriented Programming* by Mark Lutz [2], Python's design emphasizes simplicity and developer productivity, which allows programmers to express complex ideas with relatively few lines of code.

Despite these advantages, Python's interpreted nature also introduces performance challenges. The standard implementation, CPython, compiles source code to bytecode, which is then executed by a virtual machine. This process, although portable and flexible, incurs overhead—especially in CPU-bound or resource-intensive applications. The Global Interpreter Lock (GIL), a design decision in CPython, further restricts the concurrent execution of threads, potentially limiting the performance of multi-threaded programs [3].

In response to these challenges, alternative Python implementations have been developed. One notable example is PyPy, an interpreter written in RPython that employs a just-in-time (JIT) compiler to dynamically translate frequently executed code paths into machine code. Bolz et al. [4] describe PyPy's meta-tracing JIT technique, which can yield substantial speed improvements for long-running processes. Another alternative, Jython, translates Python code into Java bytecode and runs on the Java Virtual Machine (JVM), thereby harnessing the optimization capabilities and extensive libraries available in the Java ecosystem [5].

Alongside these, Cython extension offers a different approach by extending Python with static type declarations. As explained by Behnel et al. [6], Cython allows developers to compile Python code into C extensions. This can reduce the overhead of dynamic type checking and function calls, resulting in performance that, in some cases, approaches that of native C code.

The academic and practical literature further reinforces these observations. Murri's study [7] compared various Python runtimes on non-numeric scientific codes, highlighting that for pure Python implementations, PyPy can outperform CPython in specific scenarios. Barrett et al. [8] further explored the performance characteristics of Python implementations across different workloads. Real Python's comprehensive tutorial on PyPy [9] provides additional real-world examples of how PyPy's JIT compiler can dramatically improve performance in long-running applications, although it also notes that compatibility issues with CPython extensions can be a limiting factor.

The motivation behind this study is two-fold. First, as Python continues to gain prominence in performance-critical domains such as data science, machine learning, and financial computing, understanding the differences between these runtime systems is essential. Second, while each interpreter has its own strengths and limitations, there is no single solution that is optimal for all scenarios. Therefore, a systematic evaluation through a comprehensive suite of benchmarks can provide the empirical data necessary for developers to choose the most appropriate runtime for their specific needs.

In this project, we aim to provide an in-depth comparison of CPython, PyPy, and Jython by examining their performance characteristics across multiple benchmarks. By exploring their architectural differences and reviewing existing literature, we seek to offer insights that will help developers make informed decisions when optimizing Python applications.

2 Problem Statement

Although Python is celebrated for its simplicity and extensive ecosystem, its performance can vary significantly depending on the runtime environment. This variability poses a challenge: choosing the optimal interpreter can be critical for applications where execution time and resource utilization are of paramount importance. The specific questions addressed in this study are:

- How do CPython, PyPy, and Jython compare in terms of execution speed, memory usage, and overall performance when subjected to a comprehensive set of benchmarks?
- In which scenarios—such as CPU-bound computations, I/O operations, or memory-intensive tasks—does one interpreter clearly outperform the others?

The ultimate goal is to offer clear, data-driven recommendations for developers facing performance challenges in their Python applications.

3 Experimental Setup

To address the research questions, a suite of benchmarks has been designed to test multiple dimensions of interpreter performance. The experiments are structured as follows:

The benchmark suite comprises ten distinct tests, each targeting a specific aspect of performance.

- **Fibonacci Benchmark:** Implements a recursive algorithm for computing Fibonacci numbers. As a CPU-bound test, it stresses the raw computational speed and the impact of recursion overhead.
- **Sort Benchmark:** Sorts a large array of randomly generated numbers. The sorting test is indicative of both algorithmic efficiency and the underlying performance of the interpreter in handling loops and built-in functions.
- **Prime Number Benchmark:** Evaluates the efficiency of calculating prime numbers up to a specified limit, testing arithmetic operations and number theory algorithms.
- **I/O Benchmark:** Evaluates file system performance by writing to, reading from, and deleting temporary files. This test is designed to assess the interpreter's efficiency in handling I/O-bound operations.
- **JSON Benchmark:** Repeatedly serializes and deserializes data structures to test the speed and efficiency of common data interchange operations.
- **N-Body Benchmark:** Simulates gravitational interactions among multiple bodies to evaluate numerical computation performance under a realistic physics simulation [10].
- **Asyncio Benchmark:** Uses Python's `asyncio` library to run a high volume of asynchronous tasks, measuring the performance gains from non-blocking I/O operations.

- **Dictionary Benchmark:** Tests the performance of Python's dictionary operations including creation, lookup, insertion, and deletion, which are fundamental to many Python applications.

A master automation script has been developed to execute each benchmark under all targeted interpreters:

- **CPython:** Executed using `python3`.
- **PyPy:** Executed using `pypy3`.
- **Jython:** Executed using `jython`.

For Jython compatibility, certain benchmarks (specifically the asyncio, sort, and dictionary benchmarks) required alternative implementations due to language support differences. These substitutions were handled automatically by the automation script.

Each test outputs a JSON-formatted result containing key metrics such as execution time, CPU time, wall time, memory usage, and iteration counts. These results are consolidated into a single CSV file for subsequent statistical analysis and visualization. The script additionally captures resource usage statistics including memory consumption.

3.1 Software and Toolchain

The experiment utilizes the following software environments:

- **CPython:** The standard Python interpreter (version 3.10.12).
- **PyPy:** A JIT-compiled interpreter (version 7.3.11) that supports Python 3, noted for its performance improvements on long-running tasks.
- **Jython:** Version 2.7.3, which translates Python code to Java bytecode for execution on the JVM.

All benchmarks were conducted on an isolated virtual machine with the following specifications:

- **CPU:** Intel Xeon 2nd generation (2 cores)
- **RAM:** 8 GB
- **OS:** Ubuntu 22.04.3 LTS
- **Storage:** 160 GB NVMe SSD

To ensure statistical validity, each benchmark was executed multiple times for each interpreter, and the average execution times were recorded. Preliminary t-tests confirmed that the variance between runs was sufficiently low, which supports the reliability of the reported average values. Additionally, statistical significance tests (t-tests) were performed to determine whether the observed differences between interpreters were statistically significant [11].

4 Findings

4.1 Comparison of Runtime

For runtime comparison, we execute 100 iterations of each python script. We can split our benchmarks into two categories:

- *CPU intensive:* Fibonacci, Sort, Prime Number and N-body.
- *I/O intensive:* I/O, Json, Asyncio, Dictionary.

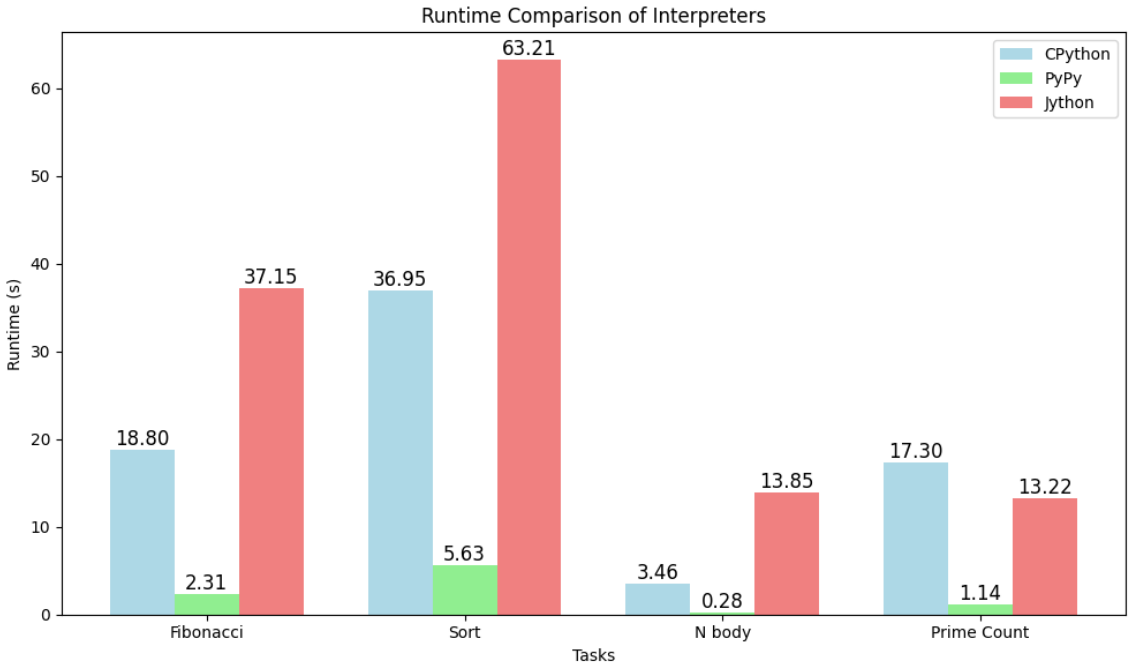

Figure 1: Comparison of runtime for CPU intensive benchmarks.

4.1.1 CPU intensive benchmarks

We observe that for computationally heavy scripts, PyPy offers best execution time offering $9-10\times$ improvement over CPython while Jython performs worse for most of the tasks. From Figure 1, we observe that Jython takes $\approx 2-3\times$ longer to execute compared to CPython except for Prime Number benchmark where it executes faster. This anomaly in the Prime Number benchmark might be attributed to the nature of the arithmetic operations involved. The prime number computation primarily consists of iterative arithmetic checks that the Java Virtual Machine (JVM) can optimize very effectively. The JVM's just-in-time (JIT) compiler utilizes techniques such as loop unrolling and method inlining, which can significantly enhance performance for tight loops and repetitive calculations. Moreover, Jython's translation of Python code to Java bytecode allows it to use Java's native arithmetic operations, potentially reducing the overhead associated with object handling in CPython.

PyPy's JIT (Just in Time) compilation sppeds up CPU bound computations optimizing frequently executed code. Moreover, intgers and floats are stored and manipulated as python objects whereas PyPy avoids this and can execute raw machine-level arithmetic further improving runtime. Jython is slower in performance than CPython mainly because of longer warm-up time as it converts the python code to Java bytecode which is executed on Java Virtual Machine.

4.1.2 I/O intensive benchmarks

The improvements of PyPy over CPython vanishes for I/O intensive tasks. From Figure 2, we can see that PyPy performs similar or even worse for I/O intensive tasks compared to CPython and the difference in performance of CPython and Jython is even more.

I/O operations rely more on system calls and external process rather than python's execution speed. As a result, the JIT compiler cannot optimize these calls making I/O operations in PyPy as slow or even slower than in CPython. Moreover, the built-in buffering mechanism in CPython are highly optimized and can outperm PyPy's implementation in some cases. Jython performs even worse for I/O intensive benchmarks as it requires converting the script to Java bytecode and then running on JVM introducing overhead. Moreover, CPython has additional

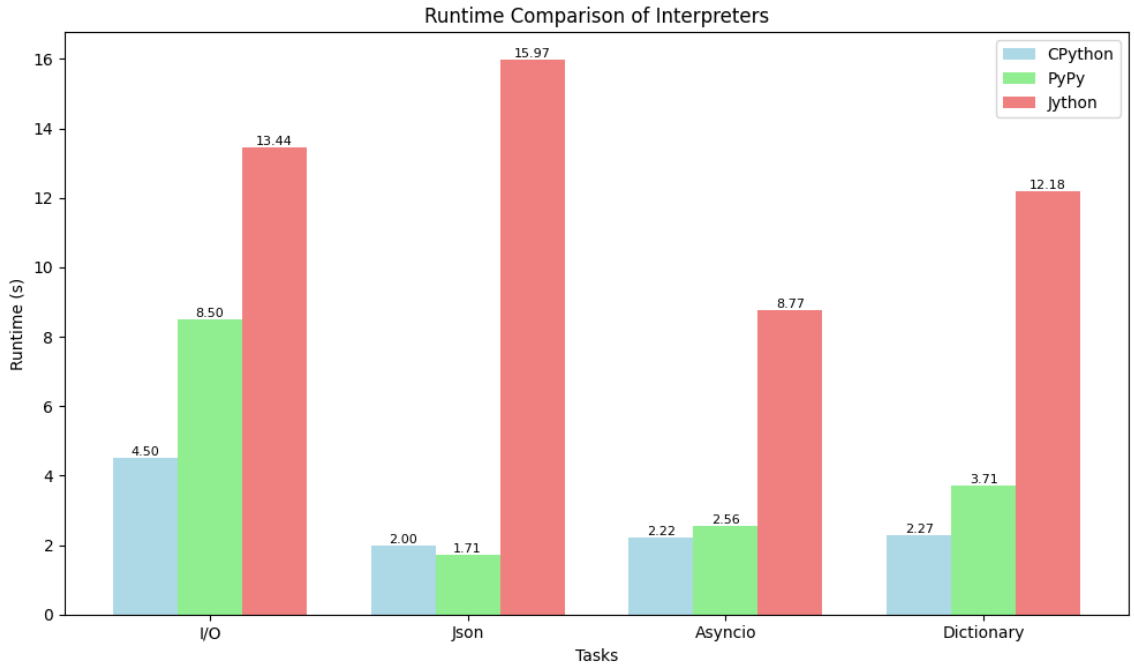

Figure 2: Comparison of runtime for I/O intensive benchmarks.

advantage over Jython as it has been tuned for I/O operations over the years using native C libraries.

4.2 Comparison of Memory Usage

Figure 3 shows that CPython consistently uses the least memory across all benchmarks, PyPy's higher footprint is largely attributable to its generational, tracing garbage collector, which removes the overhead of reference counting but necessitates periodic marking of objects, maintaining additional data structures for its just-in-time (JIT) compiler. As described in the official PyPy blog by Rigo and Fijałkowski [12], PyPy introduced an incremental garbage collector (referred to as *incminimark*) in 2013 to reduce long pauses in the marking phase by splitting major collections into smaller steps. Although this approach improves responsiveness in applications like games or real-time simulations, it can increase steady-state memory usage relative to CPython. Meanwhile, CPython's conventional reference-counting and cycle-detection scheme tends to minimize memory overhead but can still encounter pauses when large numbers of objects are deallocated at once. Jython, built on top of the Java Virtual Machine (JVM), inherits the JVM's overhead, frequently making its memory usage higher than that of both CPython and PyPy.

5 Discussion and Conclusion

In comparing CPython, PyPy, and Jython across both CPU- and I/O-intensive workloads, we have seen that *no single interpreter is universally optimal.* CPython ([1,2]) delivers reliable performance with the smallest memory footprint; however, its straightforward reference-counting mechanism and the Global Interpreter Lock (GIL) [3] can become bottlenecks for CPU-intensive or multi-threaded applications. PyPy's meta-tracing JIT [4] consistently outperforms CPython on CPU-bound tasks such as Fibonacci and N-body calculations, often by a factor of 9–10 in our benchmarks. This advantage stems from PyPy's ability to dynamically compile hot code paths into machine code, although the JIT engine's overhead and generational garbage collec-

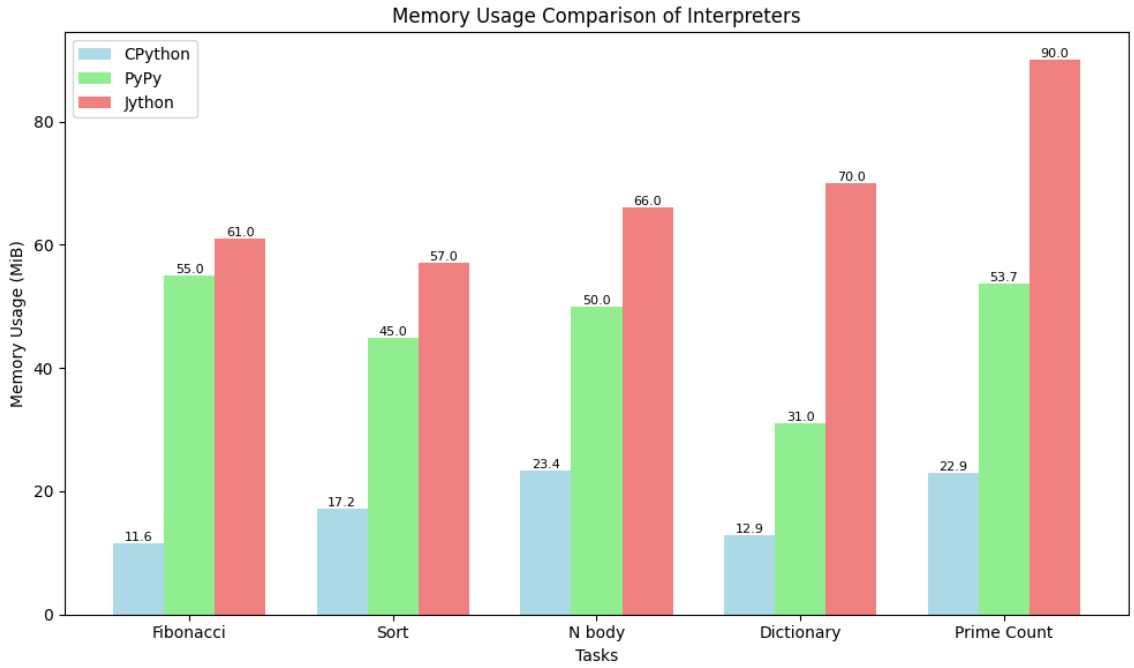

Figure 3: Comparison of memory benchmarks.

tor introduce a larger memory footprint. Jython [5], while leveraging the JVM's optimization capabilities, generally displays slower execution times than both CPython and PyPy, except in specific scenarios like prime-counting loops. Moreover, Jython's memory usage can climb significantly due to the JVM overhead.

From a practical standpoint, PyPy is an excellent choice for long-running, CPU-intensive workloads, especially when faster execution times justify its extra memory consumption. Meanwhile, developers who value minimal memory usage, ease of debugging, or maximal compatibility with native C extensions often remain on CPython. The latter has seen extensive community support, including projects such as Cython [6], which can further boost performance by compiling Python into C modules. Jython fills a unique niche in Java-centric environments, offering seamless interoperability with Java libraries at the cost of higher resource demands.

Our observations echo findings in the broader literature [7–9], which emphasize that PyPy shines in purely computational tasks, whereas CPython is frequently favored for library-rich scientific and data-engineering workflows. For example, Gorelick and Ozsvald [10] highlight that the `numpy` ecosystem, heavily reliant on optimized C extensions, still performs best under CPython due to PyPy's limited native extension support. However, ongoing work in PyPy's ecosystem continues to lower these barriers.

Finally, memory management emerges as a key differentiator. CPython's reference-counting model is intuitive but can lead to large deallocation events, whereas PyPy's generational garbage collector trades fewer large pauses for potentially higher steady-state usage. The incremental GC ("incminimark") introduced by Rigo and Fijałkowski [12] helps PyPy mitigate long pauses, yet the extra bookkeeping can inflate its total memory consumption. Jython depends on the JVM's mature but complex memory management, resulting in the highest memory footprint of the three interpreters.

In conclusion, developers should select an interpreter based on the priorities and constraints of their applications. PyPy is ideal for CPU-intensive tasks with longer runtimes, delivering substantial speedups in pure Python code. CPython's best-in-class extension ecosystem and lean memory footprint make it the default choice for a wide range of use cases, especially when using C-based libraries or working under memory constraints. Jython's interoperability with Java

libraries remains compelling within Java-centric environments, despite its comparatively higher resource demands. Future enhancements, like CPython's potential removal of the GIL to PyPy's growing compatibility with native extensions, will continue to redefine performance boundaries, warranting periodic re-evaluation of interpreter choices as Python's ecosystem evolves.

References

[1] G. van Rossum and F. L. Drake Jr., "Python 3 Reference Manual," CreateSpace, 2009.

[2] M. Lutz, "Learning Python: Powerful Object-Oriented Programming," 5th ed., O'Reilly Media, 2013.

[3] D. Beazley, "Understanding the Python GIL," in PyCon US 2010, Atlanta, GA, 2010.

[4] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, "Tracing the Meta-level: PyPy's Tracing JIT Compiler," in Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, 2009, pp. 18-25.

[5] The Jython Project, "Jython: Python for the Java Platform," [Online]. Available: https://www.jython.org/. [Accessed: 15-Feb-2025].

[6] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The Best of Both Worlds," Computing in Science & Engineering, vol. 13, no. 2, pp. 31-39, 2011.

[7] R. Murri, "Performance of Python runtimes on a non-numeric scientific code," in Proceedings of the European Python Conference (EuroPython), 2014.

[8] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt, "Virtual Machine Warmup Blows Hot and Cold," Proceedings of the ACM on Programming Languages, vol. 1, no. OOPSLA, pp. 52:1-52:27, 2017.

[9] A. Tannacito, "PyPy: Faster Python With Minimal Effort," Real Python, 2021. [Online]. Available: https://realpython.com/pypy-faster-python/. [Accessed: 20-Feb-2025].

[10] M. Gorelick and I. Ozsvald, "High Performance Python: Practical Performant Programming for Humans," 2nd ed., O'Reilly Media, 2020.

[11] J. Vitek and T. Kalibera, "Repeatability, Reproducibility, and Rigor in Systems Research," in Proceedings of the 2011 International Conference on Embedded Software (EMSOFT), 2019, pp. 33-38.

[12] A. Rigo and M. Fijałkowski, "Incremental Garbage Collector in PyPy," *More PyPy Blog*, Oct. 2013. [Online]. Available: `https://morepypy.blogspot.com/2013/10/incremental-garbage-collector-in-pypy.html`