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

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

February 28, 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 [?]. 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 [?], Python's design emphasizes simplicity and developer productivity, allowing 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, that potentially limits the performance of multi-threaded programs.

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. [?] 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 [?].

Alongside these, Cython extension offers a different approach by extending Python with static type declarations. As explained on the Cython homepage [?], 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 [?] compared various Python runtimes on non-numeric scientific codes, highlighting that for pure Python implementations, PyPy can outperform CPython in specific scenarios. Real Python's tutorial on PyPy [?] 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 twofold. 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 summary, this report aims 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 actionable 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?
- What are the underlying architectural and implementation factors (e.g., JIT compilation in PyPy, JVM optimizations in Jython, and the C-based architecture of CPython) that lead to these performance differences?
- 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.