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

Ege Demirci, Divyanshu Bhardwaj

CMPSC 263 Runtime Systems
UC Santa Barbara

Winter 2025

Outline

1. Introduction
2. Problem Statement
3. Experimental Setup
4. Findings
5. Discussion and Conclusion

Python: Advantages and Challenges

Advantages:

- High-level, dynamically typed language
- Elegant syntax and readability
- Widespread adoption across domains
- Developer productivity
- Expressive with minimal code

Challenges:

- Interpreted nature affects performance
- Overhead in CPU-bound applications
- Global Interpreter Lock (GIL) restricts concurrency
- Memory management overhead

Alternative Python Implementations

CPython Standard implementation, compiles to bytecode

- Reference implementation
- Direct C API integration
- Limited by GIL for multi-threading

PyPy Written in RPython with a Just-In-Time (JIT) compiler

- Dynamically translates code paths to machine code
- Meta-tracing JIT technique
- Potential speed improvements for long-running processes

Jython Translates Python code to Java bytecode

- Runs on the Java Virtual Machine (JVM)
- Access to Java libraries and ecosystem

Motivation for the Study

- Python increasingly used in performance-critical domains:
  - Data science
  - Machine learning
  - Financial computing
- Each interpreter has unique strengths and limitations
- No one-size-fits-all solution for all scenarios
- Need for systematic evaluation through benchmarks
- Goal: Provide empirical data for informed decisions

Problem Statement

- Performance variability depending on runtime environment
- Choosing optimal interpreter critical for time-sensitive applications
- Research questions:
  - How do CPython, PyPy, and Jython compare in terms of execution speed, memory usage, and overall performance?
  - In which scenarios does one interpreter clearly outperform the others?
    - CPU-bound computations
    - I/O operations
    - Memory-intensive tasks
- Goal: Provide data-driven recommendations for developers

Benchmark Suite Design

CPU-Intensive Benchmarks:

- **Fibonacci:** Recursive algorithm computation
- **Sort:** Large array sorting of random numbers
- **Prime Number:** Calculating primes up to a limit
- **N-Body:** Physics simulation of gravitational interactions

I/O-Intensive Benchmarks:

- **I/O:** File system operations (read/write)
- **JSON:** Serialization and deserialization
- **Asyncio:** Asynchronous task execution
- **Dictionary:** Dictionary operations (create, lookup, insert, delete)

Methodology

- Master automation script runs all benchmarks on each interpreter:
  - **CPython:** Using `python3`
  - **PyPy:** Using `pypy3`
  - **Jython:** Using `jython`
- Jython compatibility handled with alternative implementations
- Each test outputs JSON-formatted metrics:
  - Execution time
  - CPU time
  - Wall time
  - Memory usage
  - Iteration counts

Hardware and Software Environment

Software Environments:

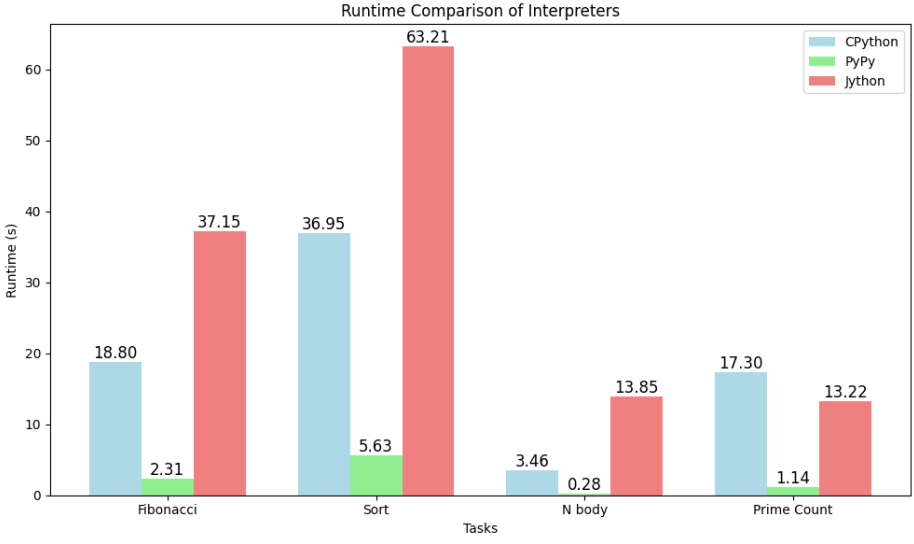
- **CPython:** Version 3.10.12
- **PyPy:** Version 7.3.11 with Python 3 support
- **Jython:** Version 2.7.3

Hardware Specifications:

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

- Multiple executions per benchmark to ensure statistical validity
- T-tests confirmed low variance between runs
- Statistical significance tests performed on interpreter differences

CPU-Intensive Benchmark Results



CPU-Intensive Performance Analysis

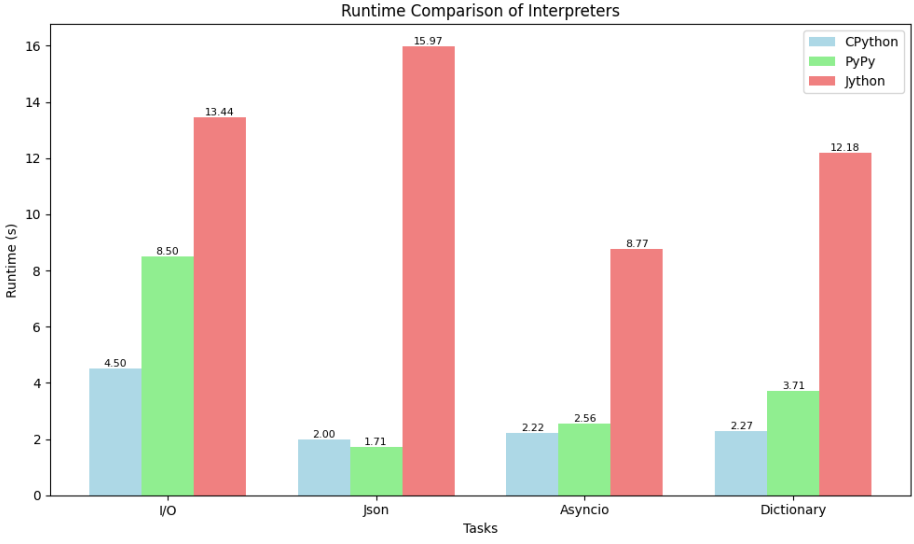
PyPy's advantages:

- JIT compilation optimizes frequently executed code
- Raw machine-level arithmetic instead of Python objects
- Effective tracing of hot paths in long-running computations

Jython's performance:

- Longer warm-up time converting Python to Java bytecode
- JVM execution introduces overhead
- Prime Number exception: JVM's optimization of arithmetic operations and loop unrolling benefits this specific workload

I/O-Intensive Benchmark Results



I/O-Intensive Performance Analysis

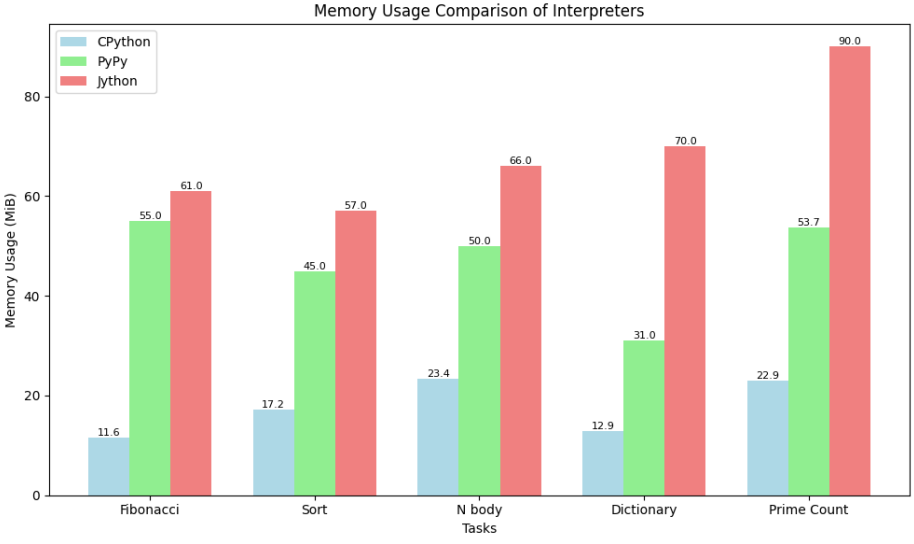
Reduced PyPy advantage:

- I/O operations rely on system calls and external processes
- JIT compiler cannot optimize these calls
- CPython's buffering mechanisms highly optimized

Jython's challenges:

- Overhead from Python-to-Java-bytecode conversion
- JVM execution adds another layer
- CPython's native C libraries for I/O operations more efficient

Memory Usage Comparison



Memory Usage Analysis

CPython:
- Efficient reference counting with cycle detection
- Minimal memory overhead
- Can have pauses during large object deallocation

PyPy:
- Generational, tracing garbage collector
- Maintains additional data structures for JIT compiler
- "Incminimark" incremental GC improves responsiveness but increases memory usage

Jython:
- Inherits JVM overhead
- Java objects typically larger than CPython objects
- JVM's garbage collection adds memory management overhead

Key Findings Summary

CPython

- Reliable baseline performance
- Smallest memory footprint
- Best library compatibility
- Limited by GIL for multi-threading

PyPy

- 9-10× faster for CPU-bound tasks
- Larger memory footprint
- JIT optimization overhead
- Limited benefit for I/O tasks

Jython

- Generally slower execution
- Highest memory usage
- Java interoperability
- No GIL limitations

No single interpreter is universally optimal

Practical Recommendations

Choose PyPy when:

- Working with long-running, CPU-intensive applications
- Memory usage is less critical than execution speed
- Application uses pure Python code (minimal C extensions)

Choose CPython when:

- Using libraries with native C extensions (numpy, scipy, etc.)
- Memory efficiency is important
- Compatibility with the Python ecosystem is essential
- Working with I/O-intensive applications

Choose Jython when:

- Integration with Java libraries is required
- Working in Java-centric environments
- Need true multi-threading (no GIL)

Future Developments

- CPython improvements:
  - Potential removal of the GIL
  - JIT compilation capabilities
  - Memory optimization efforts
- PyPy enhancements:
  - Improved compatibility with C extensions
  - Memory usage optimizations
  - Faster warm-up times
- Continued evolution of Python's ecosystem warrants periodic re-evaluation of interpreter choices

Conclusion

- Performance characteristics of Python interpreters vary significantly by workload
- For CPU-intensive tasks:
  - PyPy delivers substantial speedups (9-10$\times$)
- For I/O-intensive tasks:
  - CPython offers competitive or superior performance
- Memory usage considerations:
  - CPython most efficient
  - PyPy moderate
  - Jython highest
- Select interpreter based on application priorities and constraints

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," ICOOOLPS, 2009.
5. The Jython Project, "Jython: Python for the Java Platform," https://www.jython.org/
6. S. Behnel et al., "Cython: The Best of Both Worlds," Computing in Science & Engineering, vol. 13, no. 2, 2011.
7. A. Rigo and M. Fijałkowski, "Incremental Garbage Collector in PyPy," 2013.