

Python 3.9–3.15 인터프리터 버전에 따른 AI 에이전트, 데이터 분석, 머신러닝 워크로드 성능 비교

임재호

초록

본 연구는 CPython 3.9, 3.11, 3.13, 3.14, 3.15를 같은 빌드 체인으로 설치하고, AI 에이전트, 데이터 분석, 머신러닝을 모사한 13개 워크로드의 실행 시간을 한 대의 Apple M3 Pro 컴퓨터에서 측정하였다. 표준 라이브러리만 쓰는 8개 워크로드의 기하평균 속도 배율은 3.9 대비 3.11에서 1.24, 3.15에서 1.22였다. 즉 3.11 이후에는 추가 개선이 관측되지 않았다. 같은 라이브러리 버전을 쓴 3.14와 3.15 사이에서 NumPy, pandas, scikit-learn, Pydantic 워크로드의 차이는 2% 이내였다. 3.15의 명시적 지연 import는 쓰지 않는 무거운 모듈을 import 하는 CLI의 시작 시간을 222.98 ms에서 20.85 ms로 줄였다. 실험적 JIT는 순수 Python k-평균에서 1.24배 빨랐으나 시작 시간과 CSV 집계에서는 느려졌다. 별도 세션에서 다시 측정한 값은 최대 12.4% 달라졌으므로, 3% 미만의 차이는 구분할 수 없는 것으로 해석해야 한다.

주제어: Python, Interpreter performance, Benchmark, AI agent, Lazy import

1. 서론

Python은 AI 에이전트, 데이터 분석, 머신러닝 코드에서 널리 쓰인다. CPython 3.11은 특수화 적응형 인터프리터를 도입하였고[1], 공식 문서는 표준 벤치마크에서 3.10 대비 평균 1.25배 빨라졌다고 보고한다[2]. 이후 3.13은 GIL을 끌 수 있는 빌드를[3], 실험적 JIT를[4] 추가하였다. 2026년 10월 9일 출시된 3.15는 명시적 지연 import를 추가하였다[5, 6].

공식 수치는 범용 벤치마크의 평균이다. 개발자가 버전을 고를 때 필요한 정보는 자신의 작업 유형에서 얼마나 달라지는가이다. 특히 실제 연산 대부분을 C 확장이 맡는 데이터 분석과 머신러닝에서 인터프리터 개선이 얼마나 전달되는지는 범용 평균만으로 알 수 없다.

본 연구의 질문은 다음과 같다. 같은 기기와 같은 빌드 체인에서, 3.9부터 3.15까지의 인터프리터 버전이 에이전트, 데이터 분석, 머신러닝 워크로드의 실행 시간을 얼마나 바꾸는가. 기여는 다음과 같다.

- 세 영역을 모사한 13개 워크로드를 표준 라이브러리층과 C 확장층으로 나누어 설계하였다(절 3).
- 다섯 버전의 실행 시간과 배율을 워커 프로세스 단위 부트스트랩 구간과 함께 보고하였다(절 5).
- 3.15의 지연 import와 실험적 JIT의 효과를 따로 측정하고, 3.15에서 설치할 수 없는 주요 패키지를 확인하였다(절 5).

2. 관련 연구

PEP 659는 실행 중에 바이트코드를 자료형에 맞게 특수화하는 인터프리터를 제안하였고, 10–60%의 속도 향상을 예상하였다[1]. 3.11 공식 문서는 GCC로

빌드한 Ubuntu Linux에서 pyperformance 기준 3.10 대비 평균 25% 빨라졌고, 작업에 따라 10–60%라고 적는다[2]. 또한 인터프리터 시작이 10–15% 빨라졌다고 적는다[2].

PEP 703은 GIL을 끄는 빌드를 제안하였고, 단일 스레드에서 5–6%의 실행 부담을 보고하였다[3]. PEP 744는 실험적 JIT가 대부분의 플랫폼에서 기존 특수화 인터프리터와 비슷한 속도라고 적고, 실험 단계를 벗어나는 조건으로 약 5%의 개선을 제시한다[4]. PEP 810은 lazy 키워드로 모듈 로딩을 첫 사용 시점까지 미루며, 명령줄 도구에서 시작 시간을 50–70% 줄일 수 있다고 주장한다[5].

이 수치들은 각 제안자가 각자의 환경에서 보고한 것이다. 본 연구는 한 기기에서 다섯 버전을 같은 코드로 측정하여, 영역별 차이를 한 표에서 비교한다.

3. 측정 방법

3.1 가설

측정 전에 다음 세 가설과 기각 조건을 정하였다.

- H1 표준 라이브러리만 쓰는 워크로드는 3.9에서 3.11로 갈 때 크게 빨라지고, 3.13에서 3.15로 갈 때의 개선 폭은 그보다 작다. 3.11 대비 3.15의 개선이 3.9 대비 3.11의 개선보다 크면 기각한다.
- H2 C 확장이 연산을 맡는 워크로드는 같은 라이브러리 버전에서 인터프리터 버전에 따른 차이가 작다. 배율의 95% 구간 전체가 1에서 5% 넘게 벗어나면 기각한다.
- H3 3.15의 지연 import는 쓰지 않는 무거운 모듈을 import 하는 CLI의 시작 시간을 줄인다. 배율의 95% 구간이 1을 포함하면 기각한다.

3.2 워크로드

표 1은 13개 워크로드를 보인다. L1층은 표준 라이브러리만 쓰며 다섯 버전에서 같은 코드로 돈다. L2층은 NumPy, pandas, scikit-learn, Pydantic을 쓰며, 각 버전에 설치할 수 있는 최신 라이브러리 버전을 쓴다. 모든 입력은 시드를 고정한 합성 데이터이다.

표 1. 워크로드 구성

ID	층	내용
A1	L1	빈 인터프리터 시작 (-c pass)
A2	L1	표준 라이브러리 20개 import 후 종료
A3	L1	툴 호출 메시지 2,000개 JSON 직렬화와 파싱
A4	L1	asyncio 코루틴 2,000개 gather와 큐
A5	L1	문서 500개 프롬프트 조립, 정규식, 토큰 분할
D1	L1	CSV 50,000행 파싱과 그룹 집계
M1	L1	순수 Python k-평균 (점 5,000개, $k = 8, 10$ 회)
M2	L1	순수 Python 로지스틱 회귀 SGD ($3,000 \times 10$, 3세대)
A6	L2	Pydantic 모델 검증 10,000건
D2	L2	pandas groupby와 merge (1,000,000행)
M3	L2	NumPy 행렬곱 (2048×2048)
M4	L2	scikit-learn 로지스틱 회귀 ($20,000 \times 20$)
M5	L2	scikit-learn 랜덤 포레스트 (나무 30개, 1스레드)

ID의 첫 글자는 영역을 뜻한다. A는 에이전트, D는 데이터 분석, M은 머신러닝이다. 3.15 전용으로 두 측정을 더 하였다. 첫째, NumPy, pandas, Pydantic 등 7개 모듈을 import한 뒤 버전 출력 경로에서 바로 끝나는 CLI를 일반 import와 lazy import로 각각 실행하였다. 둘째, L1 워크로드를 PYTHON_JIT=1로 다시 측정하였다.

3.3 측정 절차와 통계

측정 절차는 그림 1과 같다. 측정 도구는 pyperf 2.10.0이다[7]. pyperf는 반복 횟수를 보정하는 프로세스 1개를 먼저 실행한 뒤, 워커 프로세스 20개를 차례로 실행한다. 각 워커는 워밍업 1회를 버리고 값 3개를 남긴다[7]. 따라서 워크로드와 버전마다 값은 60개이다. 버전 실행 순서는 층마다 시드 고정 서플로 정하였다.

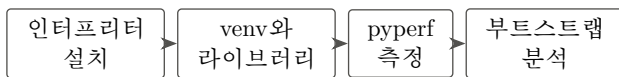


그림 1. 측정 절차

대표값은 60개 값의 중앙값 $\tilde{t}$ 이다. 기준 버전 r 에 대한 버전 v 의 속도 배율은

$$S_{v/r} = \tilde{t}_r / \tilde{t}_v \quad (1)$$

이며, 1보다 크면 v 가 빠르다. L1층 k 개 워크로드의 종합값은 기하평균 $G_{v/r} = (\prod_{i=1}^k S_{v/r}^{(i)})^{1/k}$ 이다. 같은

프로세스 안의 값은 서로 독립이 아니므로, 95% 구간은 워커 프로세스를 재표본하는 부트스트랩 10,000회로 구하였다. 이상치는 제거하지 않았다.

4. 실험 환경

표 2는 실험 환경을 보인다. 다섯 인터프리터는 모두 python-build-standalone 빌드이며[8], PGO와 LTO가 켜져 있다. 3.9.25는 20251031 릴리스이고 나머지는 20261009 릴리스이다. 3.13 이상 빌드에는 JIT가 기본 꺼짐 상태로 들어 있다. 운영체제에 들어 있는 Python과 Homebrew Python은 빌드 조건이 달라 쓰지 않았다.

표 2. 실험 환경

항목	값
CPU, 메모리	Apple M3 Pro 12코어, 36 GB
운영체제	macOS 26.6.2, AC 전원
인터프리터	3.9.25, 3.11.17, 3.13.16, 3.14.8, 3.15.0
3.9 라이브러리	NumPy 2.0.2, pandas 2.3.3, scikit-learn 1.6.1, Pydantic 2.13.5
3.11 라이브러리	NumPy 2.4.6, pandas 3.0.6, scikit-learn 1.9.1, Pydantic 2.14.0
3.13-3.15 라이브러리	NumPy 2.5.4, pandas 3.0.6, scikit-learn 1.9.1, Pydantic 2.14.0
측정 시각	2026-10-11 23:31 - 2026-10-12 00:05

3.13, 3.14, 3.15는 라이브러리 버전이 같다. 따라서 이 세 버전 사이의 L2 차이는 인터프리터 차이로 볼 수 있다. 3.9와 3.11은 라이브러리 버전이 달라 L2 차이를 인터프리터 효과로 나눌 수 없다. 측정 시작 시 1분 부하 평균은 4.99였다. 다른 프로세스가 동시에 실행되고 있었다는 뜻이다.

5. 실험 결과

5.1 버전별 실행 시간

표 3은 워크로드별 중앙값을 보인다. L1층 기하평균 배율은 3.9 대비 3.11에서 1.24 (95% 구간 1.24-1.25), 3.13에서 1.15 (1.14-1.16), 3.14에서 1.19 (1.18-1.20), 3.15에서 1.22 (1.21-1.23)였다(그림 2).

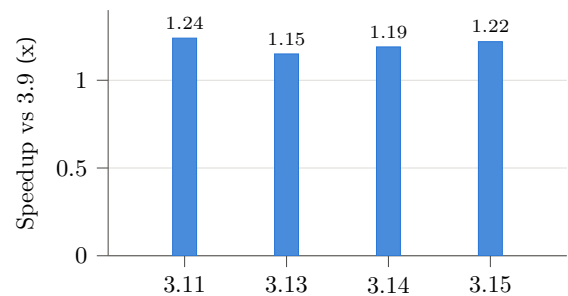


그림 2. L1층 8개 워크로드의 3.9 대비 기하평균 배율 (95% 구간은 본문)

표 3. 워크로드별 실행 시간 중앙값 (ms, 값 60개)

ID	3.9	3.11	3.13	3.14	3.15
A1	12.34	9.19	11.43	10.80	10.83
A2	42.19	42.58	49.54	52.91	50.02
A3	7.94	7.18	7.08	7.29	6.74
A4	9.88	6.59	6.10	5.15	5.15
A5	3.58	3.53	3.71	3.92	3.75
D1	62.95	47.33	52.10	49.06	48.13
M1	36.98	24.55	27.75	25.01	23.66
M2	9.30	7.22	8.03	7.75	7.94
A6	8.81	8.56	8.68	8.61	8.75
D2	70.35	65.03	60.89	67.31	66.15
M3	46.44	46.14	45.68	47.00	46.54
M4	7.25	6.96	6.89	6.93	6.88
M5	1365.66	1026.74	998.13	1003.11	1018.47

5.2 3.15의 배율

표 4과 그림 3은 식 (1)의 배율을 3.15에 대해 3.9 기준과 3.14 기준으로 보인다. 3.9 대비 가장 크게 빨라진 워크로드는 `asyncio` 팬아웃(A4)으로 1.92배였다. 이 개선은 3.14에서 이미 나타났고, 3.14 대비 3.15는 1.00배였다. 순수 Python k-평균(M1)은 1.56배, CSV 집계(D1)는 1.31배였다.

반대로 느려진 워크로드도 있다. 표준 라이브러리 20개 import(A2)는 3.9 대비 0.84배, 프롬프트 조립(A5)은 0.95배였다. A2에서 import 후 로드된 모듈 수는 3.9에서 209개, 3.15에서 227개였다. 모듈 수는 8.6% 늘었고 시간은 18.6% 늘었다. 본 연구는 이 차이의 원인을 분해하지 않았다.

표 4. 3.15의 속도 배율과 95% 부트스트랩 구간

ID	3.15 / 3.9	3.15 / 3.14
A1	1.14 [1.10, 1.19]	1.00 [0.98, 1.02]
A2	0.84 [0.82, 0.86]	1.06 [1.03, 1.09]
A3	1.18 [1.15, 1.20]	1.08 [1.06, 1.11]
A4	1.92 [1.90, 1.95]	1.00 [0.99, 1.02]
A5	0.95 [0.94, 0.96]	1.04 [1.03, 1.08]
D1	1.31 [1.28, 1.33]	1.02 [1.00, 1.05]
M1	1.56 [1.55, 1.58]	1.06 [1.05, 1.07]
M2	1.17 [1.16, 1.19]	0.98 [0.97, 0.99]
A6	1.01 [0.98, 1.03]	0.98 [0.96, 1.01]
D2	1.06 [1.03, 1.08]	1.02 [0.97, 1.09]
M3	1.00 [0.99, 1.01]	1.01 [1.00, 1.03]
M4	1.05 [1.03, 1.08]	1.01 [1.00, 1.03]
M5	1.34 [1.32, 1.37]	0.98 [0.97, 1.00]

C 확장층에서 3.15와 3.14의 배율은 0.98–1.02였다. 모든 95% 구간은 0.96–1.09 안에 있었다. 3.9 대비로는 랜덤 포레스트(M5)가 1.34배 빨랐다. 그러나 3.9에서는 `scikit-learn` 1.6.1을, 3.15에서는 1.9.1을 썼으므로 이 차이를 인터프리터 효과로 볼 수 없다.

5.3 3.15 신기능: 지연 import와 JIT

일반 import CLI의 실행 시간 중앙값은 222.98 ms (IQR 207.49–243.37)였다. `lazy import` CLI는

20.85 ms (IQR 20.30–21.43)였다. 배율은 10.69 (95% 구간 10.19–11.20)이다.

표 5과 그림 4은 3.15에서 JIT를 켜었을 때의 배율이다. 순수 Python 루프인 M1은 1.24배, M2는 1.10배, `asyncio`(A4)는 1.11배 빨라졌다. 반대로 인터프리터 시작(A1, A2)과 CSV 집계(D1)는 0.87–0.91배로 느려졌다. 8개 배율의 기하평균은 1.01이다(점추정만 계산).

표 5. 3.15에서 JIT를 켜었을 때의 실행 시간 (ms)과 배율

ID	JIT 끄	JIT 켜	배율 [95% 구간]
A1	10.83	11.94	0.91 [0.85, 0.92]
A2	50.02	57.58	0.87 [0.85, 0.91]
A3	6.74	6.44	1.05 [1.02, 1.07]
A4	5.15	4.64	1.11 [1.08, 1.13]
A5	3.75	3.94	0.95 [0.94, 0.97]
D1	48.13	52.78	0.91 [0.89, 0.93]
M1	23.66	19.10	1.24 [1.23, 1.25]
M2	7.94	7.18	1.10 [1.10, 1.11]

5.4 패키지 가용성

표 6은 2026년 10월 11일 PyPI 최신 릴리스에 macOS arm64 wheel이 있는지를 보인다. 3.15에서 PyTorch와 DuckDB는 wheel이 없다. 3.9에서는 11개 중 lxml만 최신 릴리스 wheel이 있다.

표 6. 최신 릴리스의 macOS arm64 wheel 유무 (2026-10-11 조회, ○ 있음, – 없음)

패키지	3.9	3.11	3.13	3.14	3.15
NumPy 2.5.4	–	–	○	○	○
SciPy 1.18.1	–	–	○	○	○
pandas 3.0.6	–	○	○	○	○
scikit-learn 1.9.1	–	○	○	○	○
pydantic-core 2.50.0	–	○	○	○	○
orjson 3.13.0	–	○	○	○	○
PyArrow 26.0.0	–	○	○	○	○
Polars 2.0.0	–	○	○	○	○
lxml 6.1.3	○	○	○	○	○
PyTorch 2.14.1	–	○	○	○	–
DuckDB 1.5.6	–	○	○	○	–

5.5 재현성

3.14의 L1층을 별도 세션에서 다시 측정하였다. 연산 워크로드(A3–A5, D1, M1, M2)의 중앙값 차이는 –1.5%에서 2.4% 사이였다. 시작 시간 워크로드는 A1이 12.4%, A2가 3.1% 달랐다. 8개 중 4개만 첫 측정의 95% 구간 안에 들었다. 따라서 보고한 구간은 세션 안의 변동만 반영하며, 세션 간 변동은 반영하지 않는다.

6. 고찰

6.1 가설 판정

H1의 앞부분은 지지되었다. 3.9 대비 3.11의 L1 기하평균 배율은 1.24였다. 뒷부분은 더 강한 형태로 나타났다. 3.15의 배율은 1.22로 3.11보다 높지 않았고,

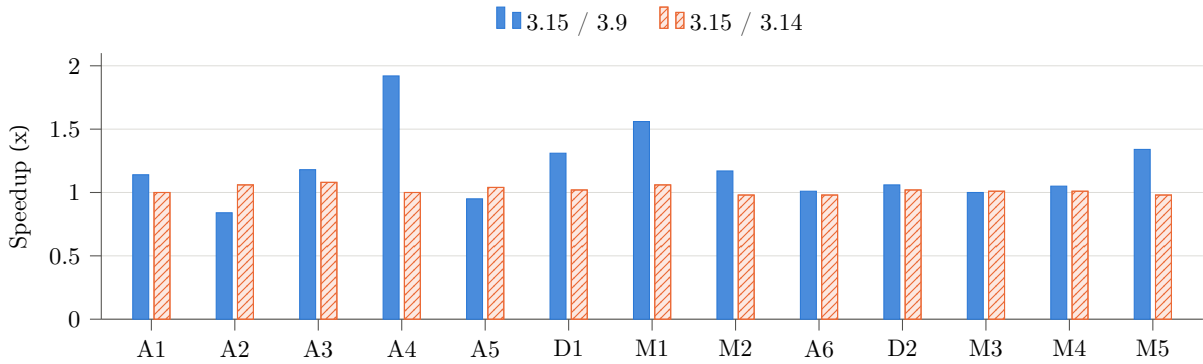


그림 3. 워크로드별 3.15의 속도 배율 (값과 95% 구간은 표 4)

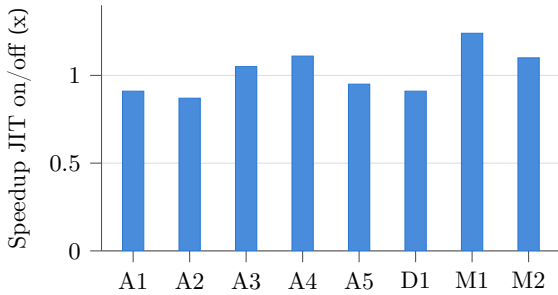


그림 4. 3.15 JIT의 워크로드별 배율 (표 5)

3.13은 1.15로 낮아졌다. 즉 이 워크로드 묶음에서는 3.11 이후의 순 개선이 관측되지 않았다.

H2는 기각되지 않았다. 라이브러리 버전이 같은 3.14와 3.15 사이에서 C 확장층의 배율은 0.98~1.02였고, 95% 구간 전체가 1에서 5% 넘게 벗어난 워크로드는 없었다.

H3은 기각되지 않았다. 지연 import의 배율 구간은 10.19~11.20으로 1을 포함하지 않는다. 다만 이 CLI는 쓰지 않는 무거운 모듈을 일부러 넣어 효과가 크게 나오도록 설계하였다. PEP 810이 주장하는 실사용 범위(50~70%)와 직접 비교할 수 없다[5].

6.2 영역별 해석

에이전트 영역에서 효과가 큰 것은 asyncio 오케스트레이션이다. 같은 코드가 3.9 대비 1.92배 빨랐다. 반면 시작 시간은 3.11에서 가장 짧았고(A1 9.19ms), 표준 라이브러리를 많이 import 하면 3.9보다 느렸다. 짧게 실행되는 에이전트 CLI에서는 버전 업그레이드보다 지연 import가 시작 시간에 더 큰 영향을 준다.

데이터 분석과 머신러닝 영역에서는 연산이 어디서 일어나는가가 결과를 갈랐다. 순수 Python 루프(D1, M1, M2)는 1.17~1.56배 빨라졌다. C 확장이 연산을 맡는 행렬곱(M3)은 1.00배였다. 이 영역에서 버전을 고를 때는 속도보다 표 6의 패키지 가용성이 먼저 제약이 된다. 예를 들어 3.15는 2026년 10월 11일 기준으로 PyTorch를 설치할 수 없다.

실험적 JIT의 효과는 워크로드에 따라 방향이 달랐다. 순수 Python 수치 루프에서는 빨라졌지만, 시작과 CSV 처리에서는 느려졌다. 기하평균 1.01은 PEP

744의 기술, 즉 대부분의 경우 기존 인터프리터와 비슷하다는 서술과 어긋나지 않는다[4].

6.3 한계와 타당성 위협

- 한 대의 Apple M3 Pro와 macOS 한 종류에서만 측정하였다. Linux, x86-64, 다른 컴파일러로 일반화하지 않는다.
- 측정 중 부하 평균이 4.99였고, 세션 간 차이는 최대 12.4%였다. 3% 미만의 차이는 구분할 수 없는 것으로 본다. 이 3% 기준은 측정 후에 정하였다.
- L2층에서 3.9와 3.11은 라이브러리 버전이 달라, 3.9 대비 배율에는 라이브러리 변경 효과가 섞여 있다.
- 입력은 합성 데이터이며 크기를 한 가지로만 두었다. 입력 크기에 따른 변화는 측정하지 않았다.
- 실제 LLM API 호출, 네트워크 지연, 메모리 사용량, GIL 없는 빌드는 측정하지 않았다.
- 버전 실행 순서는 총마다 한 번 섞었을 뿐 교차 반복하지 않았다. 열 상태 변화가 순서와 섞었을 수 있다.

7. 결 론

본 연구는 CPython 3.9~3.15를 한 기기와 같은 빌드 체인에서 비교하였다. 표준 라이브러리 워크로드는 3.11에서 3.9 대비 1.24배 빨라졌고, 3.15는 1.22배로 그 이상 빨라지지 않았다. C 확장 워크로드는 같은 라이브러리 버전에서 버전 간 차이가 2% 이내였다. 3.15에서 가장 큰 차이는 지연 import로, 무거운 모듈을 쓰지 않는 CLI의 시작 시간을 10.69배 줄였다. 따라서 3.15로 올리는 근거는 연산 속도보다 지연 import와 같은 기능에 있다. 후속 연구로 Linux 서버, 입력 크기 변화, GIL 없는 빌드, 실제 LLM 호출을 포함한 측정이 필요하다.

워크로드 코드, pyperf 원본, 분석 스크립트는 bench/와 paper/data, paper/scripts에 있다. 같

은 기기에서 `bash bench/run_all.sh`와 `python paper/scripts/analyze.py paper/data`로 다시 실행할 수 있다.