Style Conventions Override Performance Predictions in Coding LLMs

Matthew Kotzbauer [1]

Abstract

When a coding model considers which of two equivalent programs will run faster, what is it comparing? We build a benchmark to test when performance reasoning is able to overcome pretrained style convention given the two contradict. The benchmark contains 92 pairs of equivalent Python snippets across 16 idiom families, where the idiomatic version is the measured slower one by $\geq$1.05×. We query ten models across five providers in a forced-choice format with a confidence score. Every model scores near-zero on at least one family with high confidence, with each model's failures concentrated on different families, indicating the existence of model-specific style biases. A logistic classifier over 16 syntax features reaches 0.924 accuracy, beating every frontier LLM, the best of which reaches 0.793. The results suggest that style conventions learned during training may interfere with causal performance reasoning in coding models.

1. Introduction

Python development is governed by a well-documented set of canonical idioms, which are codified in style guides and heavily represented in open-source training corpora (Alexandru et al., 2018; Zhang et al., 2022). Since software is highly repetitive and language models capture its statistical regularities (Hindle et al., 2016), and since LLM accuracy correlates directly with how frequently relevant patterns appear in pretraining data (Kandpal et al., 2023; McCoy et al., 2023), we should expect coding models to exhibit a strong prior toward producing and endorsing idiomatic code.

This prior is usually helpful, but creates a specific blind spot when the canonical idiom is slower than the non-idiomatic alternative. In CPython 3.13, for instance, `enumerate` allocates a fresh tuple on every iteration while a plain index loop does not—a constant overhead that produces a 7–47% slowdown, despite `enumerate` being the conventional choice for indexed iteration in Python. We build a benchmark to test this across 16 idiom families: 92 pairs of equivalent Python snippets where the idiomatic version is the measured slower one by $\geq$1.05×. Ten models across five providers judge each pair, picking the faster snippet and reporting a confidence score. Every model scores 0% on at least one family with high confidence. A supervised logistic classifier over 16 syntax features reaches 0.924 accuracy, while the best LLM reaches 0.793. This gap points to a style bias that persists through LLM training and likely requires targeted correction to close.

2. Benchmark

Construction. The benchmark contains 92 pairs of equivalent Python snippets across 16 idiom families. Each pair shares setup code and solves the same task; any pair whose outputs disagree is discarded. We label which snippet is idiomatic and measure which one wins under `timeit`. All 92 pairs are drawn from the subset where the idiomatic snippet is the measured loser by $\geq$1.05×. This restriction makes the task diagnostic: choosing the idiomatic snippet is always wrong under the measured runtime.

Runtime measurement. We calibrate a `timeit` batch size toward 0.05s of wall time per batch, run one warm-up batch, then record 9 timed batches and report the median per-iteration time. The slower median divided by the faster gives the speedup. All measurements ran on CPython 3.13.11 on an Intel i7-12700H. Speedups across the 92 cases range from 1.05× to 1.47×. These measurements define the faster snippet used in each forced-choice label.

Families. The 92 pairs span 16 idiom families. Five contain ten or more pairs and drive most of the variance in model accuracy:

- **eager_vs_lazy** (17 cases): `sum([f(x) for x in data])` vs. `sum(f(x) for x in data)`. The list comprehension wins because CPython's `sum` iterates a concrete list faster than stepping through a generator one element at a time. Speedups: 1.05–1.43×.

[1]Harvard University, Cambridge, Massachusetts, USA. Correspondence to: Matthew Kotzbauer <matthewkotzbauer@college.harvard.edu>.

Accepted to the 1st Workshop on Combining Theory and Benchmarks, CTB@ICML 2026, Seoul, South Korea.

Faster (non-canonical)	Slower (canonical)	Speedup
`for i in range(len(data)): out.append((i, data[i]))`	`for i, x in enumerate(data): out.append((i, x))`	1.13×
`sum([x > 1 for x in data])`	`sum(x > 1 for x in data)`	1.39×
`total += d[q]`	`total += d.get(q)`	1.20×

Table 1. Three representative pairs from the benchmark, measured on CPython 3.13.11 (Intel i7-12700H). The non-idiomatic form is faster in each case due to a small per-iteration cost in the canonical idiom: tuple allocation (`enumerate`), generator dispatch (`sum` genexpr), and method lookup (`.get`).

Model	`index` (16)	`eager/lazy` (17)	`any/all` (11)	`map+reduce` (26)	All 92
GPT-5.5	0.250	1.000	1.000	1.000	0.793
GPT-5.4	0.000	0.706	1.000	1.000	0.717
Sonnet 4.6	0.000	1.000	0.636	1.000	0.696
DeepSeek V3.2	0.438	0.353	0.727	1.000	0.641
GPT-4.1	0.500	0.000	0.545	1.000	0.630
GLM-5	0.312	0.176	0.636	1.000	0.630
DeepSeek R1	0.188	0.412	0.455	1.000	0.598
Kimi K2 Thinking	0.188	0.000	0.636	0.962	0.533
Haiku 4.5	0.000	0.000	0.455	1.000	0.511
Kimi K2.5	0.000	0.000	0.455	0.962	0.478
Static baseline		(leave-one-family-out)			0.924
Canonical idiom	0.000	0.000	0.000	0.000	0.000

Table 2. Accuracy on the five largest idiom families (70 of 92 pairs). `map+reduce` combines mapping_lookup and reduce_vs_loop (26 pairs). Models sorted by overall accuracy over all 92 pairs.

- **index_iteration** (16 cases): `for i in range(len(data)):` with `data[i]` vs. `for i, x in enumerate(data):`. The index loop wins because `enumerate` allocates a fresh tuple on every iteration. Speedups: 1.07–1.47×.
- **mapping_lookup** (16 cases): `d[k]` vs. `d.get(k)` with guaranteed-present keys. Direct indexing is faster because `.get` goes through method lookup. Speedups: 1.05–1.20×.
- **any_all_genexpr** (11 cases): `any`/`all` fed a list comprehension vs. a generator expression. The list wins for the same reason as the `sum` family, offset somewhat by short-circuit behavior. Speedups: 1.05–1.25×.
- **reduce_vs_loop** (10 cases): `functools.reduce` vs. an explicit accumulator loop. The loop wins by avoiding a Python-level function call on every element. Speedups: 1.06–1.37×.

The remaining 22 cases span 11 smaller families: `pathlib` vs. `os.path`, `contextlib.suppress` vs. `try/except`, `operator.itemgetter` vs. lambda in sorts, `namedtuple` attribute access vs. index, `@property` vs. direct attribute, and others.

Examples. Table 1 shows one pair from each of the three hardest families. Each canonical idiom carries a small per-iteration cost absent from the non-idiomatic form: tuple allocation (`enumerate`), generator dispatch (`sum` with a genexpr), method lookup (`.get`). Most models fail on the first two and pass on the third.

Protocol. The prompt shows setup code and both snippets (presentation order randomized), then requests a single-line JSON response: {`"choice":"A"|"B"`, `"confidence":0.5..1.0`}. The model picks a winner and self-reports a confidence score; it does not generate or modify code. We report clean zero-shot runs where all 92 responses parsed successfully for ten models across five providers: Claude Sonnet 4.6, Claude Haiku 4.5 (Anthropic); GPT-4.1, GPT-5.4, GPT-5.5 (OpenAI); DeepSeek V3.2, DeepSeek R1 (DeepSeek); GLM-5 (ZhipuAI); Kimi K2.5, Kimi K2 Thinking (Moonshot).

3. Results

Every model scores ≥0.96 on the mapping and reduce families (Table 2). On `index_iteration` and `eager_vs_lazy`, every model scores 0% on at least one of the two at high confidence—but which one differs by model.

Complementary failures. The failure pattern differs by model. Sonnet scores 0/16 on `index_iteration` and 17/17 on `eager_vs_lazy`; GPT-4.1 is the mirror image: 8/16 on index, 0/17 on eager/lazy. GPT-5.4 fails on index (0/16) but recovers on eager/lazy (12/17) and aces any/all (11/11). DeepSeek R1 and Kimi K2 Thinking—both chain-of-thought models—each score 0% on at least one family, matching their non-reasoning counterparts.

On `any_all_genexpr` accuracy is more spread: GPT-5.5 and GPT-5.4 get all 11; four models (Sonnet, DeepSeek V3.2, GLM-5, Kimi K2 Thinking) get 7/11; the remaining four get 5/11.

Confidence. Confidence does not drop on families where accuracy is zero. GPT-4.1, for instance, averages 0.95 on the 17 eager/lazy cases it gets wrong. Haiku and Kimi K2.5 average 0.76–0.84 across the 33 cases (index + eager/lazy) where they score 0%. Kimi K2 Thinking averages 0.92 on the 17 eager/lazy cases it gets wrong—the same failure mode as its non-reasoning counterpart, with comparable confidence.

				Worst family	
Model	Acc.	Conf.	ECE†	Acc.	Conf.
GPT-5.5	0.793	0.874	0.133	0.250	0.762
GPT-5.4	0.717	0.852	0.137	0.000	0.721
Sonnet 4.6	0.696	0.714	0.067	0.000	0.665
DeepSeek V3.2	0.641	0.801	0.159	0.353	0.794
GPT-4.1	0.630	0.873	0.263	0.000	0.950
GLM-5	0.630	0.898	0.284	0.176	0.926
DeepSeek R1	0.598	0.844	0.262	0.188	0.781
Kimi K2 Thinking	0.533	0.892	0.378	0.000	0.920
Haiku 4.5	0.511	0.827	0.315	0.000	0.756
Kimi K2.5	0.478	0.882	0.404	0.000	0.838

Table 3. Per-model calibration over all 92 pairs. †ECE = expected calibration error (lower is better). "Worst family" is the family each model scores lowest on; confidence on that family does not drop relative to overall.

Logistic baseline. A logistic classifier over 16 simple code features achieves 0.924 under leave-one-family-out cross-validation, trained to predict which snippet is faster. The features are AST node counts plus token-presence indicators for common constructs in the benchmark: subscripting, generator expressions, list comprehensions, calls, attributes, loops, lambdas, conditional expressions, `enumerate`, `range`, `len`, `sum`, `any`, `all`, and `.get`. Top positive weights: `Subscript` node (+2.09), `has_subscript` (+1.68); top negative: `GeneratorExp` node (-1.15), `enumerate` token (-1.01), `.get` token (-0.71). The classifier has no knowledge of Python execution semantics; it outperforms every frontier model by at least 0.13 points, including the chain-of-thought models.

4. Related Work

Work on calibration for code models shows that LLM confidence signals often require post-hoc correction even when predictions are directionally useful (Spiess et al., 2025). On verbalized confidence more broadly, zero-shot reports from LLMs are usually overconfident, with partial improvement from reasoning augmentation (Lin et al., 2022; Xiong et al., 2024; Yoon et al., 2025). On code performance, BigO(Bench) tests asymptotic complexity reasoning (Chambon et al., 2025), KernelBench evaluates GPU kernel generation (Ouyang et al., 2025), and Omniwise predicts kernel runtimes (Wang et al., 2025). Our setting differs: we test constant-factor performance where the answer depends on CPython implementation details rather than algorithmic structure, and we use a forced-choice format that contrasts idiom convention against measured performance without requiring direct code generation.

5. Discussion

The failures are localized: each model scores 0% on a specific family while passing others at high accuracy. This is not a uniform bias toward idiomatic code—it is per-idiom and differs by model. Sonnet 4.6 scores 0% on `index_iteration` and 100% on `eager_vs_lazy`; GPT-4.1 reverses both. We use "style bias" in this behavioral sense: when a conventional form and the measured winner conflict, model choices remain systematically family-specific rather than tracking runtime uniformly. The implication is that models carry distinct per-idiom priors that operate more like categorical defaults than a general preference for idiomatic style.

Chain-of-thought does not override these defaults. DeepSeek R1 and Kimi K2 Thinking each score 0% on at least one family at high confidence, matching their non-reasoning counterparts. Models that reason explicitly still report high confidence in the wrong direction on the families they fail—suggesting the style prior is determining the output and the reasoning is rationalizing it after the fact.

The correct answer on each pair is determined by measured execution time, making this a natural fit for reinforcement learning with verifiable rewards (RLVR)—the paradigm used in recent work to train code models against execution-based reward signals (Jiang et al., 2025; Fan et al., 2025). Preference pairs where the idiomatic snippet is the measured loser provide a supervision signal requiring no human annotation, and the per-idiom structure of failures keeps the training scope narrow.

The benchmark is scoped to constant-factor CPython idioms; results may not generalize to asymptotic complexity or other runtimes. This focus isolates the specific cases where stylistic convention and measurable performance directly contradict.

References

Alexandru, C. V., Merchante, J. J., Panichella, S., Proksch, S., Gall, H. C., and Robles, G. On the usage of pythonic idioms. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pp. 1–11. ACM, 2018.

Chambon, P., Roziere, B., Sagot, B., and Synnaeve, G. Bigo(bench) – can LLMs generate code with controlled time and space complexity?, 2025. URL https://arxiv.org/abs/2503.15242.

Fan, L., Zhang, Y., Chen, M., and Liu, Z. Recode: Reinforcing code generation with reasoning-process rewards. *arXiv preprint arXiv:2508.05170*, 2025.

Hindle, A., Barr, E. T., Gabel, M., Su, Z., and Devanbu, P. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

Jiang, X., Dong, Y., Liu, M., Deng, H., Wang, T., Tao, Y., Cao, R., Li, B., Jin, Z., Jiao, W., Huang, F., Li, Y., and Li, G. Coderl+: Improving code generation via reinforcement with execution semantics alignment. *arXiv preprint arXiv:2510.18471*, 2025.

Kandpal, N., Deng, H., Roberts, A., Wallace, E., and Raffel, C. Large language models struggle to learn long-tail knowledge. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, 2023.

Lin, S., Hilton, J., and Evans, O. Teaching models to express their uncertainty in words. *Transactions on Machine Learning Research*, 2022. URL https://openreview.net/forum?id=8s8K2UZGTZ3.

McCoy, R. T., Yao, S., Friedman, D., Hardy, M., and Griffiths, T. L. Embers of autoregression: Understanding large language models through the problem they are trained to solve, 2023. URL https://arxiv.org/abs/2309.13638.

Ouyang, A., Guo, S., Arora, S., Zhang, A. L., Hu, W., Re, C., and Mirhoseini, A. Kernelbench: Can LLMs write efficient GPU kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415, 2025.

Spiess, C., Gros, D., Pai, K. S., Pradel, M., Rabin, R., Alipour, A., Jha, S., Devanbu, P., and Ahmed, T. Calibration and correctness of language models for code. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*, 2025.

Wang, Z., Ramos, C., Awad, M. A., and Lowery, K. Omniwise: Predicting GPU kernels performance with LLMs, 2025. URL https://arxiv.org/abs/2506.20886.

Xiong, M., Hu, Z., Lu, X., Li, Y., Fu, J., He, J., and Hooi, B. Can LLMs express their uncertainty? an empirical evaluation of confidence elicitation in LLMs. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=gjeQKFxFpZ.

Yoon, D., Kim, S., Yang, S., Kim, S., Kim, S., Kim, Y., Choi, E., Kim, Y., and Seo, M. Reasoning models better express their confidence, 2025. URL https://arxiv.org/abs/2505.14489.

Zhang, Z., Xing, Z., Xia, X., Xu, X., and Zhu, L. Making python code idiomatic by automatic refactoring non-idiomatic python code with pythonic idioms. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2022.