Verifying Python Algorithms: A Three-Layer Model of Contracts, Tests, and Complexity Evidence

Helen Jacinto

Abstract

A runnable implementation is not yet a trustworthy algorithm. Trust requires explicit claims about behavior, evidence that challenges those claims, and performance measurements whose design supports the conclusion drawn from them. This conceptual-methodological article uses A Practical Python Approach to Data Structures and Algorithms as a curricular foundation for a verification-oriented approach to Python implementations of arrays, linked lists, stacks, queues, sorting routines, and trees. The source is examined as an instructional artifact; no learner data or benchmark results are inferred. The proposed framework joins three forms of evidence: correctness contracts, systematic tests, and empirical complexity checks. Contracts state preconditions, postconditions, and representation rules. Example-based tests target boundaries and failure modes, while property-based and metamorphic relations provide reusable oracles when exact expected outputs are costly to enumerate. Complexity claims are examined through input families, scale, repeated measurements, environment control, and plots interpreted alongside asymptotic analysis. Two practical tools are presented: a test-oracle matrix for core DSA components and a protocol for producing defensible timing evidence. The article distinguishes functional correctness from structural validity, deterministic results from measurement variability, and observed growth from proof of asymptotic complexity. It argues that students and practitioners should be taught to report what was claimed, how it was challenged, what remains uncertain, and which evidence would change the conclusion. The framework is intended for laboratory instruction, code review, and future empirical study; it makes no unsupported claim that a particular implementation or teaching intervention is effective.

Keywords: algorithm testing; property-based testing; metamorphic testing; empirical complexity; Python; reproducible benchmarking

Introduction

The statement ‘the code works’ compresses several different claims. It may mean that the program ran without an exception, matched one expected output, preserved a data structure after an update, met a functional specification for a defined input domain, or performed within an acceptable resource budget. These claims require different evidence. Confusing them creates fragile software and weak algorithm instruction: a sorting function can return an ordered sequence while losing elements, a tree can find inserted keys while violating its ordering policy elsewhere, and a benchmark can appear fast because the tested inputs never exercise the difficult case.

Software correctness begins with a relation between a program and a specification. Hoare's (1969) axiomatic account formalized reasoning from preconditions through program execution to postconditions. Dijkstra (1972) emphasized intellectual control of programs rather than confidence based on demonstration alone. Modern testing

complements—not replaces—such reasoning by searching for executions that contradict a claim. The oracle problem remains central: for many tests, deciding the expected behavior is itself difficult (Barr et al., 2015).

A Practical Python Approach to Data Structures and Algorithms moves from Python syntax and algorithm analysis to arrays, linked lists, stacks, queues, sorting, and trees, with explanations, code, and self-paced exercises (Bajao, n.d.). This breadth makes the guide a useful basis for asking what evidence should accompany an implementation. The present article does not rerun or evaluate its examples. It constructs a verification and benchmarking layer that can be applied to comparable Python DSA tasks without inventing performance values or learner outcomes.

Purpose and Scope

The guiding question is: What minimum evidence should accompany a Python DSA implementation before its correctness and efficiency claims are considered trustworthy? The article develops a three-part answer: specify a behavioral and structural contract, challenge the contract through complementary testing strategies, and evaluate performance claims through a controlled empirical protocol interpreted against analytic complexity. The framework is pedagogical and methodological; it is not a certification standard or an empirical comparison of algorithms.

Analytical Method

The source guide's principal structures and algorithms were analyzed as testable components. For each component, the analysis identified its input domain, observable result, mutable state, structural validity conditions, boundary cases, likely oracle, and expected resource-growth claim. These categories were compared with established ideas in specification, structural testing, property-based testing, metamorphic testing, and performance evaluation. The resulting matrices state what evidence a laboratory report or code review could provide. No source code was modified, no timings were collected, and no claim about the guide's implementation quality is made.

Correctness Begins with a Contract

Preconditions and Postconditions

A precondition identifies the states for which an operation promises a result. Binary search, for example, requires an ordered search space under a defined comparison relation. A postcondition identifies what must be true after execution: the returned index denotes the target, or an absence result is produced under the stated convention. Without these conditions, a test result is ambiguous. Running binary search on unsorted input does not necessarily reveal a defect in the implementation; it may violate the operation's domain. The report must state whether invalid inputs are rejected, normalized, or outside scope.

Representation Invariants

Mutable data structures add internal obligations. A queue's logical order must agree with its front and rear representation. A linked list must not lose reachable nodes during insertion. A binary search tree must maintain its ordering relation throughout each subtree, not only between a parent and immediate children. An AVL tree must maintain both search-tree ordering and its height or balance policy. A method's visible output can be correct while the internal structure has already been damaged. Tests must

therefore inspect permitted structural properties, either through a validation method or through public operations whose combined behavior exposes the structure.

Failure Behavior

A contract also includes failure. Popping an empty stack, dequeuing an empty queue, comparing incompatible keys, or requesting an invalid index should have a documented outcome. Returning a sentinel, raising an exception, or refusing an operation are different interfaces. Tests should verify the chosen behavior and confirm that failure leaves the structure valid. Exception coverage is not ancillary; it is part of the algorithm's observable semantics.

Testing Strategies as Complementary Evidence

Examples and Boundaries

Example-based tests are indispensable when selected for a reason. Empty and singleton structures expose initialization and termination. Duplicate keys reveal unstated ordering or stability policies. Already sorted and reverse-sorted sequences differentiate algorithm paths. Repeated enqueue and dequeue operations can expose stale indices or storage growth. Extremely unbalanced insertions can challenge recursive depth and worst-case tree behavior. A test plan should connect each case to a risk rather than treat coverage as a count of examples (Ammann & Offutt, 2016).

Structural Coverage

Statement coverage can show that lines executed, but execution alone says little about whether decisions and paths were adequately challenged. McCabe (1976) linked control-flow structure to a measure of program complexity that can guide independent path selection. For student-scale DSA programs, the practical lesson is modest: exercise both outcomes of conditions, loop termination at zero and multiple iterations, recursive base and recursive cases, and each rotation or deletion case that the implementation claims to handle. Coverage data should direct questions, not operate as proof of correctness.

Property-Based Testing

Property-based testing generates many inputs and checks a relation rather than listing every expected output. QuickCheck established a practical pattern in which programmers express properties and generators produce test cases, with failing cases reduced to smaller counterexamples (Claessen & Hughes, 2000). In Python DSA tasks, sorting can be checked for order and preservation of the input multiset; stack behavior can be checked by comparing sequences of pushes and pops with a simple model; a search-tree traversal can be checked for order and membership. A property must be logically strong. Checking only that a sort's output is ordered would accept an empty result for every input.

Metamorphic Relations

When a direct oracle is unavailable or expensive, metamorphic testing compares related executions. If a list is sorted and then sorted again, the result should be unchanged. Adding a constant to every numeric key should preserve the relative ordering produced by a comparison sort. Searching after inserting a previously absent key should change the membership result under the structure's defined duplicate policy. A survey by Segura et al. (2016) shows how metamorphic relations address oracle

limitations across domains. In instruction, the technique encourages learners to state lawful transformations rather than guess individual outputs.

Table 1

Correctness contracts and candidate test oracles

Component	Contract or invariant	Boundary case	Property or metamorphic relation
Stack	Pop returns the latest unremoved push; failure behavior is defined.	Empty, singleton, repeated push/pop	Pushing x then popping returns x and restores prior model state.
Queue	Dequeue follows arrival order and preserves remaining order.	Empty, one item, alternating enqueue/dequeue	Enqueueing a sequence then draining reproduces that sequence.
Linked list	All stored nodes are reachable exactly as the API specifies.	Insert/delete at head, tail, and singleton	Insert then delete the same position restores observable contents.
Comparison sort	Output is ordered and is a permutation of input.	Empty, duplicates, sorted, reverse sorted	Sorting twice is idempotent; monotone translation preserves rank order.
Binary search	If present, returned position contains target; domain is ordered.	First, last, absent, duplicate target	Searching after lawful insertion changes absence to presence.
Binary search tree	Subtrees obey the declared ordering and membership policy.	Monotone inserts, duplicate keys, root removal	In-order traversal is ordered and matches inserted multiset policy.
AVL tree	Search-tree ordering and allowed balance factor both hold.	Each single/double rotation pattern	Insertion preserves membership and restores the balance condition.

Note. A property is useful only when it is independent enough to detect plausible defects in the implementation.

From Analytic Complexity to Empirical Evidence

What Timing Can and Cannot Establish

Asymptotic analysis characterizes growth under a model; timing records behavior of a particular program, interpreter, machine, input family, and measurement procedure. Observing a near-linear trend over a limited range does not prove an O(n) bound. Conversely, small inputs can hide an asymptotic advantage behind constants, allocation, caching, or interpreter overhead. Trustworthy reporting keeps analytic and empirical claims separate, then asks whether measured behavior is compatible with the proposed explanation.

Input Families and Scale

A performance experiment needs a defined size variable and input family. For sorting, n may be the number of elements, while distribution, initial order, duplicates, and key-comparison cost must also be controlled or varied deliberately. For search trees, insertion order influences shape. For a queue, the selected Python representation can make front removal costly. Multiple sizes should span a range large enough for growth to become visible without exceeding resource limits. The range and stopping rule should be reported, not selected after viewing a preferred pattern.

Repetition and Environmental Control

Single timings are vulnerable to scheduler activity, warm-up effects, garbage collection, and clock granularity. Python's timeit facility repeats small statements and uses a high-resolution timer while avoiding some common mistakes (Python Software Foundation, 2026). It does not eliminate the need to report setup, repetition counts, input regeneration, Python version, hardware, operating system, and whether measurements include construction or copying. Feitelson (2015) distinguishes repeatability and reproducibility as related but different goals; a report should provide enough detail for both same-environment repetition and independent reconstruction.

Statistical Interpretation

Performance data are distributions, not decorations. Georges et al. (2007) showed the need for statistically rigorous evaluation in managed-language systems, and Kalibera and Jones (2013) proposed methods for determining repetitions and confidence in benchmarking. For a classroom study, the reporting need not be elaborate, but it should avoid cherry-picking the fastest run. Raw observations or summaries, a stated aggregation rule, variability, and uncertainty should accompany plots. Log-log slopes or normalized ratios can support interpretation, provided the report does not convert an empirical fit into a proof.

Table 2

Protocol for defensible empirical complexity evidence

Claim element	Measurement design	Major confound	Reporting requirement
Operation and size	Define the exact callable, setup, and input-size variable.	Timing construction or copying unintentionally	Code version, setup boundary, and units
Input family	Vary size and intentionally select order/distribution cases.	One easy family presented as typical	Generator, seed or data source, and case rationale
Scale range	Use multiple predeclared sizes spanning a meaningful range.	Tiny inputs dominated by fixed overhead	All tested sizes and stopping rule
Repetition	Repeat within and across fresh inputs or processes as appropriate.	Clock noise, warm-up, caching, garbage collection	Trial structure and complete aggregation rule
Environment	Hold configuration stable and record relevant software/hardware.	Unreported interpreter or machine differences	Python version, platform, processor, memory
Analysis	Plot observations and uncertainty; compare with analytic prediction.	Fitting only the visually preferred region	Raw or reusable data, model, residual/uncertainty evidence
Conclusion	State compatibility, anomalies, and alternative explanations.	Equating observed growth with mathematical proof	Bounded claim tied to tested conditions

Note. The protocol scales from student laboratories to formal studies; rigor should match the strength of the claim.

A Verification-Oriented Laboratory Report

A concise laboratory report can organize evidence in six parts. First, state the operation and its contract. Second, describe the representation and validation checks. Third, list risk-based examples and boundaries. Fourth, state independent properties or metamorphic relations and explain why they detect plausible defects. Fifth, present analytic complexity and an empirical protocol if performance is claimed. Sixth, report failures, revisions, and residual uncertainty. The last part is crucial: trustworthy work records counterevidence and explains what the current tests do not establish.

The report should preserve the difference between a test oracle and the implementation under test. Reusing the same logic twice can produce agreement without independence. A simple reference model is often preferable to a second optimized algorithm. For example, a stack can be checked against a straightforward Python list model; a tree's membership can be compared with a set under an explicitly matched duplicate policy. Zeller (2009) treats systematic debugging as a search for causes, and small counterexamples are especially useful because they connect a failure to a violated assumption.

Pedagogical and Review Implications

Assessment rubrics should separate specification quality, test adequacy, structural validity, functional behavior, performance method, and interpretation. A learner who supplies many redundant examples has not necessarily built a strong test suite; a learner who identifies one high-value property may demonstrate deeper understanding. Code review can ask whether a claimed invariant is checked, whether failures preserve state, and whether performance evidence matches the operation actually analyzed.

The framework also discourages fabricated certainty. If an operation has not been tested under duplicates, the report says so. If timing variability is large, it is reported rather than hidden. If an observed curve is compatible with more than one growth model over the tested range, the conclusion remains qualified. This stance aligns technical communication with the evidence available and gives future reviewers a clear path for replication or challenge.

Discussion

The proposed framework links three practices that are often taught separately. Contracts define what success means. Tests attempt to contradict that definition across examples, paths, properties, and transformations. Benchmarking examines resource claims under controlled conditions. Their combination is stronger than any one alone: specification without testing may overlook implementation defects, testing without a contract may celebrate undefined behavior, and timing without analysis may confuse environmental effects with algorithmic growth.

Python is well suited to this work because functions, assertions, data generation, and timing can be expressed compactly. That convenience does not reduce the need for methodological judgment. Built-in operations have their own semantics and costs; mutation can complicate repeated tests; random generators need reproducible control; and interpreter overhead affects small measurements. These details belong in the explanation because trustworthy evidence is contextual.

Research Agenda

Future studies can evaluate whether contract-first laboratories improve defect detection, explanation quality, and transfer across structures. Experimental or design-based work should use seeded defects with known categories, independently scored specifications, and tasks not practiced during instruction. Research can also examine which combinations of examples, properties, and metamorphic relations novices generate without prompting and how benchmark reporting changes after explicit methodological instruction. Any conclusions should distinguish learning outcomes

from the quality of submitted programs and should publish materials needed for replication.

Conclusion

Trust in an algorithm is an evidence problem. A successful run answers a narrow question; it does not establish the full behavioral contract, structural validity, or resource claim. Python DSA instruction can make this distinction concrete by requiring contracts, risk-based examples, independent properties, metamorphic relations, and carefully bounded performance evidence.

The source guide provides accessible implementations and topics on which this verification layer can operate. The two matrices in this article turn that layer into a practical reporting framework. No benchmark values or learner effects are asserted. The contribution is a disciplined method for saying what is claimed, how it was challenged, and what evidence still remains to be gathered.

References

Bajao, N. A. (n.d.). A practical Python approach to data structures and algorithms: With self-paced exercises [Unpublished instructional guide].

Ammann, P., & Offutt, J. (2016). Introduction to software testing (2nd ed.). Cambridge University Press.

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2015). The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering, 41(5), 507-525. https://doi.org/10.1109/TSE.2014.2372785

Claessen, K., & Hughes, J. (2000). QuickCheck: A lightweight tool for random testing of Haskell programs. In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (pp. 268-279). ACM. https://doi.org/10.1145/351240.351266

Dijkstra, E. W. (1972). The humble programmer. Communications of the ACM, 15(10), 859-866. https://doi.org/10.1145/355604.361591

Feitelson, D. G. (2015). From repeatability to reproducibility and correlability. ACM SIGOPS Operating Systems Review, 49(1), 3-11. https://doi.org/10.1145/2723872.2723875

Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous Java performance evaluation. In Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (pp. 57-76). ACM. https://doi.org/10.1145/1297027.1297033

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. Communications of the ACM, 12(10), 576-580, 583. https://doi.org/10.1145/363235.363259

Kalibera, T., & Jones, R. (2013). Rigorous benchmarking in reasonable time. In Proceedings of the 2013 International Symposium on Memory Management (pp. 63-74). ACM. https://doi.org/10.1145/2491894.2464160

McCabe, T. J. (1976). A complexity measure. IEEE Transactions on Software Engineering, SE-2(4), 308-320. https://doi.org/10.1109/TSE.1976.233837

Python Software Foundation. (2026). timeit—Measure execution time of small code snippets. Python 3 documentation. https://docs.python.org/3/library/timeit.html

Segura, S., Fraser, G., Sanchez, A. B., & Ruiz-Cortes, A. (2016). A survey on metamorphic testing. IEEE Transactions on Software Engineering, 42(9), 805-824. https://doi.org/10.1109/TSE.2016.2532875

Zeller, A. (2009). Why programs fail: A guide to systematic debugging (2nd ed.). Morgan Kaufmann.