

Operationalizing Misconception Catalogs for LLM Distractor Generation in Introductory Python

Bryan Hollingsworth^{1,*}, Peter Brusilovsky¹ and Jaromir Savelka²

¹University of Pittsburgh, USA

²Carnegie Mellon University, USA

Abstract

Misconception catalogs in CS education describe how novice programmers typically go wrong. Prior catalogs are descriptive registries: useful for instructors, but not directly actionable as input to automated assessment generation. We present a Python-specific master catalog of 34 misconceptions across 9 syntactic and semantic categories, hand-curated from peer-reviewed catalogs, CS1 textbooks, and the Python language specification, augmented with operational metadata: each entry pairs the misconception description with a verified one-line Python example demonstrating how the misconception surfaces in code. As a work-in-progress contribution, we further introduce an automated catalog-construction pipeline distilled from this manual curation work, intended to make future catalog construction reproducible at scale. We use the curated catalog to drive an LLM distractor-generation pipeline and evaluate the resulting MCQs against a catalog-free baseline using a cross-model rating procedure. Both conditions use identical single-invoke generation architecture so the experimental variable is solely catalog-vs.-no-catalog. We report misconception-fidelity rates across two generator LLMs (one commercial, one open-weight): the curated catalog suppresses distractors the validator approved but that run and produce the expected output to near-zero on both generators, while the uncurated conditions (naive baseline and pipeline-output catalog) produce them at 13–25%, despite scoring higher on the validator’s surface approve rate. The contribution of this paper is threefold: the curated master catalog itself, the in-progress pipeline that automates its construction, and an experimental design that isolates the contribution of catalog grounding from generation-pipeline architecture and from rater-judge biases documented in the LLM-as-a-judge literature.

Keywords

Distractor generation, programming misconceptions, multiple-choice questions, Large Language Models, CS1, computer science education

1. Introduction

CS education at scale relies on multiple-choice questions (MCQs) for formative assessment, and the diagnostic value of an MCQ depends largely on its distractors. A well-designed distractor is the manifestation of a specific student misconception: when a student picks it, the instructor learns *why* the student was wrong, not merely *that* they were wrong. Distractor design varies widely in CS1 practice, but the recent body of work on automated distractor generation (Section 2) has not been architected around catalog discipline: distractors are typically evaluated for plausibility or student-engagement metrics, but rarely for whether they actually instantiate a documented misconception. Programming misconception catalogs—Sorva’s dissertation [1], the curated progmiscon.org inventory [2], the Qian & Lehman literature review [3], and the empirical studies they reference [4, 5, 6, 7, 8, 9]—have spent two decades documenting the recurring faulty mental models novices hold about variables, control flow, scope, and object semantics. These catalogs are descriptive registries, not operational specifications: they tell you *what* the misconception is, but not *how* to manifest it as a single buggy line of code that a generation pipeline can produce on demand.

Existing approaches to automated distractor generation have largely worked around this gap rather than closing it [10, 11, 12, 13]. This paper closes the gap with three contributions. First, a Python-

CSEDM’26: 10th Educational Data Mining in Computer Science Education (CSEDM) Workshop, June 28, 2026, Seoul, South Korea

*Corresponding author.

✉ BRH369@pitt.edu (B. Hollingsworth); peterb@pitt.edu (P. Brusilovsky); jsavelka@andrew.cmu.edu (J. Savelka)

🌐 <https://www.pitt.edu/> (B. Hollingsworth); <https://sites.pitt.edu/~peterb> (P. Brusilovsky)

🆔 0009-0008-1885-019X (B. Hollingsworth); 0000-0002-1902-1464 (P. Brusilovsky); 0000-0002-3674-5456 (J. Savelka)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

specific *master catalog* of 34 hand-curated misconceptions across 9 syntactic and semantic categories (Section 3), synthesized from six peer-reviewed catalogs (Sorva [1]; Chiodini et al. [2]; Sirkiä & Sorva [5]; Kaczmarczyk et al. [4]; Žanko et al. [8]; Johnson et al. [9]), CS1 textbook and language-specification sources, with each entry paired with a verified one-line Python example that demonstrates how the misconception manifests in code. The catalog was constructed via human-AI collaboration, with each entry’s source recorded and audited (Section 3); we treat the master catalog as a contribution in its own right, not merely as input to the pipeline.

Second, an *in-progress* automated catalog-construction pipeline (Section 3) distilled from the manual curation work above. The pipeline operationalizes the lessons learned during hand-curation: candidate extraction, LLM-judge validation, manifestation classification, and per-entry executable-example generation with retry-with-feedback. Its maturation toward reproducible catalog generation is the work-in-progress we report here.

Third, an experimental framework (Sections 4–5) comprising (a) a catalog-grounded LLM distractor-generation pipeline that uses the catalog as an executable specification, picking a misconception per problem and writing the buggy line a student holding it would produce while preserving the catalog identifier and citations through to the MCQ artifact; (b) a catalog-free *naïve* pipeline of identical generation architecture as a baseline, in which the generator draws on its own internal misconception priors; and (c) a cross-model rating protocol that scores every distractor on a four-axis rubric (catalog grounding, claim fidelity, code validity, distinctness; Section 4), addressing LLM-as-a-judge self-preference concerns documented in the evaluation literature [14, 15]. We instantiate the framework with two generator models (Claude Opus 4.7 and gemma4:31b; one commercial, one open-weight), an in-loop judge (Claude Haiku 4.5) that is a distinct model from any generator, and an independent post-hoc cross-validator (GPT-OSS 20B).

The remainder of the paper proceeds as follows. Section 2 reviews related work in CS misconception cataloging and distractor generation. Section 3 describes the master catalog and the in-progress pipeline distilled from its construction. Section 4 describes the distractor-generation architecture. Section 5 reports the internal validation experiment and results. Section 6 discusses limitations and future work.

2. Related work

2.1. Distractor Generation

Automated MCQ generation has received a lot of focus from researchers since manually authoring MCQs is labor intensive. While there has been attempts addressing the task as a whole [16, 17, 18, 19, 20, 21, 22] most of the work focuses on a specific component of an MCQ—the stem, the key (correct answer), or the distractors (incorrect choices; this work). Many approaches to generate high-quality distractors have been proposed and studied [23, 24, 25, 26, 27, 28, 29]. One of the foundational work in this area is Gierl et al. [30] who proposed guidelines for high-quality distractors related to, e.g., their number, ordering, or plausibility. Specifically, plausibility is often being cited as one of the most important properties of distractors. Further, distractors must be unambiguously incorrect [24, 23, 31], related to the studied problems [32], and similar to (but not to be confused with) a correct answer [33]. There is no consensus in terms of the ideal number of distractors or their ordering. Some researchers suggest that two distractors are enough and there is little benefit from adding more [30, 32]. Gierl et al. [30] state that the most common practical recommendation related to distractors is to focus on identifying common misconceptions related to thinking, reasoning, and solving the problem [34, 35, 36, 37]. This recommendation provides the fundamental motivation for our approach in this work.

In computing education research, existing approaches to automated distractor generation have generally avoided explicit use of misconceptions in the process. Smith & Zilles [10] mutate the AST of correct solutions to produce Parsons-problem distractors—syntactically diverse, but with no necessary connection to a documented misconception. Lee, Kim & Jo [13] train a model to predict which distractors students are most likely to choose, optimizing for *plausibility* without committing to *fidelity*: the resulting distractors are good lures but blind to what they reveal. Hassany et al. [12, 11] generate

code-completion distractors with LLMs and benchmark them against human-instructor distractors; their work touches catalog-grounded generation but is not architected around it. They report that across multiple LLMs the model’s internal misconception priors outperformed prompts grounded in an unedited general-purpose misconception catalog (Sorva’s misconception inventory [1]), with OpenAI o1 the most striking case (catalog-grounded prompts produced *fewer* functional distractors than a simpler baseline prompt) [12]. We read this not as evidence against catalog grounding per se but as a finding about catalog *quality*: when the supplied catalog is general-purpose and unedited, an LLM’s internal misconception representations are a stronger signal—motivating our hypothesis that the missing ingredient is operational metadata (*how* the misconception surfaces as a code change) and curation discipline, rather than the presence of a catalog itself. Across this literature, catalog discipline is rare: distractors are evaluated for plausibility or for student-engagement metrics, but rarely for whether they actually instantiate the misconception they claim to demonstrate. We position our four-axis distractor rubric (catalog grounding, claim fidelity, code validity, distinctness; Section 4) within the established multiple-choice item-design tradition reviewed by Gierl et al. [30], which synthesizes decades of guidance on plausibility, distractor distinctness, and misconception-sourcing for human item-writers; our rubric adapts these constructs from the human-author setting to one where the writer and the judge are both LLMs. Albrecht & Grabowski [38] document the inverse problem in student-error analysis—principled misconceptions are routinely conflated with sloppy slips—which sharpens the need for fidelity-graded evaluation on the generation side.

2.2. Cataloging Misconceptions in CS Education

A programming misconception is a stable, incorrect understanding of a programming concept that produces predictable wrong code. Misconception catalogs aggregate these across studies into reusable inventories: each entry pairs a description of the faulty model with the errors it produces and citations to the empirical work that observed it. Catalogs differ along three dimensions that matter for downstream use: the language they target, how they were collected (interview / think-aloud, large-scale log mining, or expert synthesis), and the granularity at which entries are described (concept-level, error-pattern-level, or both). Our master catalog (Section 3) draws on the catalogs reviewed below; for each contributing source we record the citation alongside the entries it informs.

Two sources anchor the catalog. Sorva’s 2012 doctoral thesis on visual program simulation [1] consolidates 162 concept-level misconceptions about programming notional machines and remains the most widely cited inventory for CS1. Chiodini et al.’s `progmiscon.org` [2] is a curated, web-hosted registry with stable IDs and citation chains back to original studies—the only source on this list explicitly designed as a structured registry. The historical anchor is Pea [39], who identified language-independent conceptual “bugs” traceable to a default-strategy “superbug” in novice programming.

Smaller catalogs complement these anchors. Kaczmarczyk et al. [4] identify four recurring CS1 misconception themes (variables, assignment, language elements, references) from student think-alouds; Sirkiä & Sorva [5] mine student mistakes from program-simulation logs as empirical grounding for Sorva’s catalog; Qian & Lehman [3] synthesize prior work into a structured taxonomy that we treat as a meta-source; Altadmri & Brown [6] rank the most frequent error types in 37 million Java compilations as an empirical-attestation sanity check. Python-specific and younger-cohort evidence comes from Swidan, Hermans & Smit [7], Žanko, Mladenović & Krpan [8], and Johnson, McQuistin & O’Donnell [9]. None of these sources, individually, is structured to drive automated MCQ generation: all describe *what* the misconception is without specifying *how* it surfaces as a single buggy line of code. Bridging that gap is the purpose of the master catalog described in Section 3.

3. The Master Catalog and Construction Pipeline

The catalog underlying this paper was built first by hand, and the lessons learned during that hand-curation work motivated the in-progress pipeline that the present paper reports as a work in progress.

We describe both, in that order, since chronologically and methodologically the manual catalog came first.

Manual catalog. Starting from a prior internal v1 catalog whose breadth had outgrown its rigor, the authors hand-curated a second-generation catalog by extracting candidate misconceptions from the peer-reviewed sources reviewed in Section 2 together with the LLM-distractor-generation work of Hassany et al. [11, 12] and Smith & Zilles [10], then pairing each retained entry with a verified one-line Python example demonstrating how the misconception manifests in code. The result is a 34-entry catalog spanning 9 syntactic and semantic CS1 categories, with the largest densities in lists and mutability ($n = 11$), variables and assignment ($n = 8$), and functions and scope ($n = 5$); counts allow multi-tag membership.¹ Every entry traces to at least one peer-reviewed published source (the catalog contains no instructor-attested entries and no generic-web placeholders); the full catalog is listed in Table 3 alongside its Python example.

The closest published comparison is Hassany et al. [12], who used Sorva’s misconception inventory [1] unedited. Our catalog differs in being Python-specific, in pairing each entry with a verified executable example, and in restricting scope to code-writing misconceptions: our experimental MCQ format is line-substitution, so the catalog admits only entries whose misconception can be encoded as a single buggy line. The pipeline’s later extraction quantifies this restriction (see in-progress pipeline paragraph below): of 138 candidate entries, only 67 (49%) carry a code-writing manifestation, while the remaining 71 (51%) are exclusively semantic-prediction entries describing how a student misreads program behavior rather than how they write a buggy line. This manifestation distinction is orthogonal to the tracing-vs-writing skill axis [40, 41]: that axis concerns the student task being assessed; ours concerns whether a misconception can be encoded as a single buggy substitution at all. Table 1 illustrates the catalog with six entries selected to span its semantic range, organized by category and labeled (a)–(f). Each row pairs the kebab-case `misconception_id` with its catalog-source citation, a one-sentence misconception description, the verified one-line Python example, and the runtime behavior of that example — the failure signal the runtime classifier in Section 5 captures and that an LLM judge alone cannot see. The same six entries drive the distractors in Figure 1 (Section 5), where each distractor is labeled with its (a)–(f) letter from this table.

In-progress automated pipeline. The hand-curation work above made clear which steps were repetitive and could be automated, and which required human judgment that no single LLM call could replicate. We are now building a multi-stage pipeline that operationalizes the automatable portion: per-source extraction (an LLM reads polished source text and emits candidate misconception entries with source provenance), id-keyed merging across sources, entry-level validation (an LLM judge filters out entries that do not represent real CS1 misconceptions or that are clear aliases of another entry), manifestation classification (whether the misconception surfaces in code-writing, in semantic prediction of program behavior, or both), and a per-entry generate-and-validate retry loop that attaches a single canonical Python example to each code-writing entry with up to three regeneration attempts. The pipeline produces a 138-entry intermediate catalog with three projections: a 67-entry code-writing subset (28 cw-only entries plus 39 dual-tagged with semantic-prediction); a 44-entry `v2-cw-thin` subset (the 67 filtered to entries with a runtime-verified Python example); and a 71-entry exclusively-semantic-prediction complement (51%). Reconciling the pipeline output against the manual catalog—and ultimately replacing the manual hand-curation step with an automated one informed by the same human-judgment criteria—is open work; the present paper reports this pipeline as the work-in-progress contribution. The hand-curated catalog of the previous paragraph is the primary experimental input in Section 4; the pipeline’s `v2-cw-thin` output is included alongside as a catalog-quality manipulation. The full pipeline-output catalog is given in Appendix A (Table 4).

¹Two entries present at generation time were removed in a final citation review; the distractor-generation experiment (Section 5) ran on the 36-entry pre-removal catalog, but neither removed entry was ever selected by any generator, so the reported results are unaffected.

ID	Misconception	Python example	Runtime
<i>Variables and assignment</i>			
(a) assignment-direction-reversed [5, 7]	Students believe the assignment statement transfers the value from the left-hand side into the right-hand variable, reversing the actual semantics.	<code>5 = x</code>	<code>SyntaxError</code> — assignment LHS must be a name.
<i>For-loops and iteration</i>			
(b) c-style-for-loop-in-python [42]	Students with prior C or Java exposure write C-style for loops (parentheses, type declaration, increment expression) in Python contexts, producing non-Python syntax instead of the <code>for i in range(n):</code> idiom.	<code>for (int i = 0; i < n; i++)</code>	<code>SyntaxError</code> — <code>int</code> and <code>i++</code> are not Python.
(c) for-loop-control-variable-semantics [1]	Students misunderstand the semantics of a for-loop’s iteration variable: how it is automatically updated each iteration, whether it is accessible or modifiable inside the loop body, and what its values constrain.	<code>for i in range(5): i = i + 1</code>	Runs without error; rebinding <code>i</code> is overwritten next iteration.
<i>Lists and mutability</i>			
(d) mmr5-off-by-one-array-length [4]	Students believe a list or array of length N has valid indices 0 through N inclusive ($N+1$ slots), rather than 0 through $N-1$.	<code>[1, 2, 3][3]</code>	<code>IndexError</code> — length-3 list has valid indices 0–2.
<i>Meta-cognitive (programmer-intent)</i>			
(e) computer-infers-programmer-intent [1]	Students believe the computer can deduce what the programmer meant and fill in gaps or correct errors, instead of executing the written code literally.	<code>total = a and b</code>	Runs without error; and returns an operand by truthiness, not a sum.
(f) omitted-code-inferred [39]	Students omit necessary statements, values, or logic because they tacitly assume the computer can fill in what was meant from context, just as a human reader infers intent from partial natural-language instructions.	<code>result = max()</code>	<code>TypeError</code> — <code>max()</code> requires at least one argument.

Table 1

Six master-catalog entries spanning the catalog’s semantic range, labeled (a)–(f). The full 34-entry catalog is in Table 3.

4. Applying Misconceptions: Generating Distractors

Inputs and architecture. Distractor generation runs over eight CS1-Python problems (variable assignment, comparisons, sequence iteration, conditional branching, logical operators, range-based iteration, accumulator patterns, and `while`-loops). Each problem record carries a statement, a canonical solution (5–8 lines), one canonical buggy variant with the misconception it targets (used as a generator hint and as a feasibility check that the misconception admits a one-line manifestation), three increasingly specific hints, and one to four LLM-proposed test inputs whose distinguishing power is mechanically verified by running both the canonical and buggy variants in a Python subprocess. For a given (problem, generator) cell, the pipeline issues a single structured-output call to the generator producing a set of three distractors, then a second call to a distinct in-loop judge (Claude Haiku 4.5) that emits a set-level verdict (`approve` / `fix` / `reject`) together with a per-distractor issue string. On non-approve, the next attempt sees the validator’s per-distractor feedback verbatim and surgically replaces only the failing distractors; the loop runs at most three attempts.

Catalog and naive arms. Both arms share this generation architecture; the only difference is whether the catalog (Section 3) is supplied to the generator and validator. In the *catalog* arm, the generator selects a `misconception_id` from the supplied catalog and writes the buggy line that a student holding that misconception would produce; the validator scores the set on four axes (catalog grounding, claim fidelity, code validity, distinctness). In the *naive* arm no catalog is supplied; the generator draws on its internal misconception priors and emits a free-form kebab-case label, and the catalog-grounding axis collapses to the remaining three. After each generation call, a deterministic post-processing step normalises indentation, merges the catalog entry’s prose and citations into the artifact (no LLM involved in this transcription), and executes each distractor in a Python subprocess against the verified inputs, classifying its runtime outcome as `runs_wrong`, `runs_correct` (false-positive), `crashes`, `syntax_error`, or `timeout`. Every artifact carries provenance: pipeline-script and runtime git SHAs, generator and validator model identifiers, sampling temperature, and a hash of the catalog file at run time.

Generators and cross-judge audit. We instantiate the framework with two generators spanning the commercial / open-weight axis: `gemma4:31b` (Google, served locally) and `claude-opus-4-7` (Anthropic). Each generator runs three conditions: catalog-curated (the hand-curated catalog as it stood at generation time, 36 entries; two were later removed in citation review and were never selected by any generator), catalog-pipeline (the 44-entry `v2-cw-thin` pipeline output, included to test whether catalog *quality* interacts with the catalog effect), and naive (no catalog), yielding six (generator, condition) cells, each run at n_1 (low temperature) and n_2 (high temperature). Generator-model separation is enforced: no generator rates its own output. The in-loop judge is Claude Haiku 4.5, a distinct model from the Claude Opus 4.7 generator though served on the same Anthropic backend. Every distractor is re-graded post-hoc by an independent open-weight cross-validator (GPT-OSS 20B) using the per-arm rubric the in-loop validator received; we report agreement (Section 5) as an external check on in-loop validator reliability [14, 15].

5. Internal Validation

We evaluate the framework of Section 4 as a 2×3 factorial: two generator models (`gemma4:31b` open-weight, Claude Opus 4.7 commercial) crossed with three catalog conditions (curated, pipeline, naive), over eight CS1-Python problems. Each (generator, condition, problem) cell runs at n_1 (low temperature) and n_2 (high temperature).² The design supports within-model contrasts across the three catalog conditions (per generator) and a cross-model contrast at each catalog condition. Distractors are evaluated along four LLM-judge axes (catalog grounding³, claim fidelity, code validity, distinctness; both in-loop and post-hoc cross-judge) and one program-execution axis (runtime validity, computed against per-problem test inputs). Our planned analyses are: paired approve-rate and runtime-validity comparisons across catalog conditions within each generator, the same comparison across generators within each condition, and in-loop vs cross-judge agreement on the common item pool.

Catalog ablation and headline finding. Table 2 reports validator approve rate and the *trojan-horse* rate (briefly *trojan*: a distractor the validator approved but that runs and produces the expected output) across the 2×3 design. Naive leads on validator approve rate on both generators, reproducing the directional finding of Hassany et al. [12]; the runtime classifier reverses the picture for both. Hand-curation excluded entries judged likely trojan-prone: 0% trojan on the open-weight generator and 5% on the commercial one, against 13–25% for the four uncured cells. The filter is invisible to the validator and is recovered only by executing each substituted line. The commercial generator produces

²Pooled across cells, approve rate rose from 81% at n_1 to 88% at n_2 with trojan-horse counts flat (16 vs 17). No reversal of the headline contrasts at either temperature; we report combined rates throughout.

³Catalog grounding collapses to a vacuous axis in the naive condition; for the catalog conditions, it is folded into the set-level approve verdict.

higher trojan rates without catalog grounding (21% naive, 25% pipeline) than the open-weight one (13%, 15%); ungrounded generation does not mitigate the failure mode at higher model capability in our two-generator sample.

Condition	Approve		Trojan		xJudge agree	
	gemma	opus	gemma	opus	gemma	opus
curated	69%	88%	0/33	2/42	12%	38%
pipeline	81%	75%	6/39	9/36	44%	69%
naive	94%	100%	6/45	10/48	56%	81%

Table 2

Validator set-approve rate⁴ (n=16 sets per cell), trojan rate (runs-correct distractors among validator-approved distractors), and post-hoc cross-judge agreement (GPT-OSS 20B vs in-loop Haiku, set-level verdict match). Trojan denominators vary because the validator gates approval; expressed as percent of 48 generated distractors per cell, the trojan rates are 0%, 13%, 13%, 4%, 19%, 21% (curated–pipeline–naive, gemma then opus), preserving the qualitative pattern. Curated catalog produces zero or near-zero trojan distractors on both generators; ungrounded trojan rates are higher on the larger generator in our sample; cross-judge agreement is lowest on the strictest condition (curated, where the catalog-grounding axis adds a way for judges to disagree).

Coverage interacts with arm. The eight problems cover concepts unevenly represented in the catalog (`vars` and `ifelse` have many catalog entries; `while` and `accumulator-pattern` have none). Splitting by coverage class, naive’s trojan problem is heaviest on strong-to-mid coverage on both generators: gemma’s 6 trojans cluster on `ifelse` (4) with `vars` contributing zero, and opus shows a similar pattern at higher absolute counts (`ifelse` 5; `cmp` 2, `logical` 2; `vars` 0), with one outlier on none-coverage `while`. On weakly-covered concepts (`forseq`, `range`) the curated catalog’s cost differs across generators: 3 of 12 distractors clear the validator on gemma vs 9 of 12 on opus, while naive clears the validator on 12 of 12 distractors on both. Curation discipline transfers cleanly across none-coverage cells (zero curated-arm trojans on `while` and `accum` for both generators); opus curated’s only trojan failures (2 of 42 approved distractors) both fall on mid-coverage `cmp`. At n=12 distractors per (generator, coverage-class) cell, these are descriptive contrasts rather than tests of an interaction; the pattern is consistent with the cost of forcing ill-fitting catalog entries falling more heavily on the smaller generator.

MCQ Examples. Figure 1 shows two MCQs produced by the catalog-curated arm, one per generator (`gemma4:31b` and `claude-opus-4-7`). Each item reports the stem with a blank, the keyed answer, and the three distractors with the catalog `misconception_id` that drove generation. Both items are validator-approved, all six distractors are non-trojan (none of them execute and produce the expected output), and the catalog IDs span three distinct misconceptions per item, demonstrating the catalog’s role as a generation specification rather than a single template. To make the connection between distractors and misconceptions more clear, all six misconceptions used to generate distractors for the examples shown in Figure 1 can be found in Table 1 along with the runtime classification of the substituted line. We used (a)–(f) letters to cross-reference misconceptions listed in Table 1 with distractors shown in Figure 1.

⁴The in-loop validator (Claude Haiku 4.5) shares a model family with one of the two generators (Claude Opus 4.7). Pooled across conditions, Haiku approves Opus sets at 88% and Gemma sets at 81%; while a directional gap of this kind would be consistent with documented LLM-as-judge self-preference [15], the post-hoc GPT-OSS cross-judge agrees with Haiku more often on Opus (62% pooled) than on Gemma (38%) — the opposite direction self-preference would predict if Haiku were inflating Opus approves out of bias. The 7-point Haiku gap is therefore not consistent with simple self-preference; we rely on the runtime classifier (judge-independent) as the actual backstop for the catalog-effect interpretation, with the GPT-OSS cross-judge providing a second, correlated LLM-judge view rather than a fully independent one.

Example 1. Concept: range-based-iteration. Generator: gemma4:31b, n_1 .

Stem.

```
daily_goal = 5000
days = 7
_____ # fill the blank
total = daily_goal * day
print(day, total)
```

Keyed answer:

```
for day in range(1, days + 1):
```

Distractors:

(b) c-style-for-loop-in-python

```
for (int day = 1; day <= days; day++)
```

syntax_error

(d) mmr5-off-by-one-array-length

```
for day in range(1, days):
```

runs_wrong

(c) for-loop-control-variable-semantics

```
for day in range(1, days + 1): day = day + 1
```

syntax_error

Example 2. Concept: accumulator-pattern. Generator: claude-opus-4-7, n_1 .

Stem.

```
scores = [85, 92, 77, 88]
total = 0
for score in scores:
    _____ # fill the blank
print("Total:", total)
```

Keyed answer:

```
total = total + score
```

Distractors:

(a) assignment-direction-reversed

```
total + score = total
```

syntax_error

(e) computer-infers-programmer-intent

```
total = total and score
```

runs_wrong

(f) omitted-code-inferred

```
total + score
```

runs_wrong

Figure 1: Two MCQ examples from the catalog-curated arm, one per generator (gemma4:31b and claude-opus-4-7, both at n_1). Each distractor’s (a)–(f) letter cross-references its row in Table 1 (Section 3).

6. Conclusion

We presented two contributions toward operational misconception catalogs for CS education: a 34-entry hand-curated Python misconception catalog augmented with verified one-line manifestation examples, and a distractor-generation pipeline that uses the catalog as an executable specification. Across two generators (gemma4:31b and Claude Opus 4.7) and three catalog conditions (curated, pipeline-output, naive baseline), the curated catalog suppressed trojan distractors to 0% on the open-weight generator and 5% on the commercial one, against 13–25% for the four uncured cells. The validator’s set-approve rate favors the catalog-free baseline, reproducing the directional finding of Hassany et al. [12]; the runtime classifier reverses the picture, indicating that the catalog’s contribution is precisely to suppress the failure mode the LLM judge cannot see. As a work-in-progress contribution, we also reported an automated catalog-construction pipeline distilled from the manual curation work, whose 44-entry v2-cw-thin output was included as a third experimental condition and behaves more like the naive baseline than the hand-curated catalog, locating curation discipline (not catalog *presence*) as the operative variable.

Two boundaries scope the work. First, validation in this paper is internal: we measure misconception-instantiation quality via cross-model rating, not student outcomes. Second, the master catalog was constructed via human-AI collaboration; for entries where no peer-reviewed catalog source could be located, we honestly cite the CS1 textbook chapter, language specification, or instructor-attestation that informed the entry, rather than over-stating the catalog’s published-source coverage.

Our future work runs in two directions. The first is reducing the human-in-the-loop dependence of the catalog-construction pipeline so that catalog expansion (to additional CS1 topics and eventually beyond Python) becomes reproducible at scale rather than rate-limited by manual curation. The second is broadening the question types the catalog can drive: the present paper exercises the catalog only through fill-in-the-blank distractor generation, but the same catalog entries naturally support code-tracing items (predict program output) and find-the-buggy-line items (locating the manifestation in a longer program), each of which recruits more catalog entries into testable form. Misconception catalogs also support other interventions: detecting misconceptions in student code [43, 44], providing remedial feedback [45], and generating erroneous examples [46].

Table 3: Hand-curated master catalog (34 entries) used as the catalog-curated experimental input. Each entry pairs a one-sentence misconception description with a verified one-line Python example demonstrating how the misconception manifests in code; citations under each id record the peer-reviewed catalog source(s) the entry was synthesized from.

ID	Misconception	Python example
aliasing-invisible [9]	Students do not understand that two variable names can refer to the same underlying list object, so they fail to predict that mutating through one name changes the value seen through the other name.	<code>a=[1,2]; b=a; b.append(3); print(a)</code>
and-operator-as-double-ampersand [42, 2]	Students who have prior experience with Java or JavaScript believe that '&&' is valid Python syntax for logical AND, when in fact Python uses the keyword 'and' and '&&' is a syntax error.	<code>if x > 0 && y > 0:</code>
append-no-aliasing-effect [9]	Students lack a robust mental model of '.append()' as a mutating method call on an object, and therefore fail to predict that appending a list and later modifying that list will change the container it was appended into.	<code>a=[]; b=[a]; a.append(1); print(b)</code>
assignment-copies-object-not-reference [2, 5, 1]	Students believe a variable stores an object itself rather than a reference to it, so assigning one variable to another duplicates the object's state instead of creating a second reference to the same object.	<code>a=[1]; b=a; a.append(2); print(b)</code>
assignment-direction-reversed [5, 7, 1]	Students believe the assignment statement transfers the value from the left-hand side into the right-hand variable, reversing the actual semantics.	<code>5 = x</code>
assignment-equality-operator-confusion [6, 2]	Students conflate the assignment operator with the equality-comparison operator, writing '=' where a comparison is needed (e.g., 'if x = 5:') or '==' where a value must be stored.	<code>if x = 5:</code>
assignment-moves-value [1]	Students believe that 'y = x' moves the value out of x into y, leaving x empty or undefined, rather than copying the value while leaving x unchanged.	<code>x=5; y=x; print(x)</code>
augmented-assign-rebind-equivalence [9]	Students believe 'a += b' is always semantically equivalent to 'a = a + b', not realising that for lists '+=' mutates in place and affects all aliases, whereas 'a = a + b' rebinds 'a' to a new object.	<code>a=[1]; b=a; b+=[2]; print(a)</code>
c-style-for-loop-in-python [42]	Students with prior C or Java exposure write C-style for loops (parentheses, type declaration, increment expression) in Python contexts, producing non-Python syntax instead of the 'for i in range(n):' idiom.	<code>for (int i = 0; i < n; i++)</code>
chained-comparison-left-associative [42, 2]	Students expect 'a == b == c' in Python to evaluate as '(a == b) == c' (the Java/JavaScript semantics), rather than as the mathematical chaining '(a == b) and (b == c)' that Python actually implements.	<code>assert 1 == 2 == False</code>
class-instance-relationship-confusion [1]	Students misconceive the relationship between a class and its objects, treating the class as either a collection containing its instances or as a supertype of which each object is a subtype, rather than as a blueprint or type that instances are created from.	<code>5 in int</code>
computer-infers-programmer-intent [1]	Students believe the computer can deduce what the programmer meant and fill in gaps or correct errors, instead of executing the written code literally.	<code>total = a and b</code>
dot-operator-only-for-methods [1]	Students believe the dot notation (e.g., 'obj.x') can only be used to call methods, not to access data attributes of an object.	<code>(1).real()</code>
equality-assignment-identity-operator-confusion [6, 1]	Students conflate the operators for assignment ('='), value equality ('=='), and object identity ('is'), using them interchangeably in conditionals and assignments.	<code>if x = 10:</code>
equals-sign-operator-confusion [6, 8]	Students misread the '=' symbol's role: writing comparison operators as '=>/'='<' instead of '>=/'<=', and treating assignment '=' as bidirectional mathematical equality rather than a directed store.	<code>if x => 10:</code>
for-loop-control-variable-semantics [1]	Students misunderstand the semantics of a for-loop's iteration variable: how it is automatically updated each iteration, whether it is accessible or modifiable inside the loop body, and what its values constrain.	<code>for i in range(5): i = i + 1</code>
func1-def-executes-body [5]	Students believe that a 'def' statement begins executing the function body right away, rather than binding a new function object to a name in memory without running any of the body's code.	<code>def f(): print("hi")</code>
func6-return-value-not-captured [5, 1]	Students believe a returned value is automatically deposited into the caller's variable without an explicit assignment, failing to assign the result of a call like 'text = ask_euros()'.	<code>len([1,2,3]); print(result)</code>

ID	Misconception	Python example
function-argument-and-return-type-mismatch [6]	Students invoke functions with arguments whose types do not match what the function expects, or assign a function's return value to a variable used as an incompatible type.	<code>len(42)</code>
function-mutates-input-to-output [1]	Students believe a function works by transforming the variable passed in, so the argument variable itself becomes the result rather than a separate return value being produced.	<code>n = -5; abs(n); print(n)</code>
list-append-api-confusion [10]	Students fail to add an element to a list by calling <code>list.append(item)</code> correctly, instead using attribute assignment (<code>list.append = value</code>), the wrong method (<code>list.insert</code> without an index), or a method from another collection type (<code>list.add</code>).	<code>[1, 2, 3].add(4)</code>
list-callable-instead-of-method [10]	Students write <code>'result = lst(value)'</code> — calling the list object directly as though it were a function — instead of <code>'result = lst.count(value)'</code> .	<code>[1, 2, 3](2)</code>
list-plus-commutative [9]	Students assume that <code>'+'</code> is commutative for lists and strings, predicting that <code>'[6,7] + [4,5]'</code> and <code>'[4,5] + [6,7]'</code> produce the same result when they do not.	<code>assert [1, 2] + [3, 4] == [3, 4] + [1, 2]</code>
list-vs-element-confusion [1]	Students confuse the entire list/array object with a single cell or element within it, leading to errors when indexing, assigning, or passing lists to functions.	<code>[1, 2, 3] + 10</code>
method-call-syntax-confusion-brackets-or-missing-parens [10]	Students invoke list methods using subscript brackets (e.g., <code>'lst.append[x]'</code> , <code>'lst.count[item]'</code> , <code>'lst.remove[v]'</code>) or omit parentheses entirely (e.g., <code>'lst.count'</code>), treating method invocation as indexing or as bare attribute access.	<code>[1, 2, 3].append[4]</code>
misc4-conditional-stores-into-tested-var [5]	Students believe that evaluating a comparison expression like <code>'divisor == 0'</code> writes the resulting boolean value back into the variable that was tested (<code>'divisor'</code>), as if the comparison were an assignment.	<code>divisor=5; divisor == 0; print(divisor)</code>
mmr5-off-by-one-array-length [4]	Students believe a list or array of length N has valid indices 0 through N inclusive (N+1 slots), rather than 0 through N-1.	<code>[1, 2, 3][3]</code>
no-reserved-words [2]	Incorrect: Every sequence of letters and digits starting with a letter or an underscore can be used as an identifier.	<code>return = 5</code>
omitted-code-inferred [39]	Students omit necessary statements, values, or logic because they tacitly assume the computer can fill in what was meant from context, just as a human reader infers intent from partial natural-language instructions.	<code>result = max()</code>
pop-vs-remove-list-method-confusion [10]	Students use <code>list.pop()</code> when <code>list.remove()</code> is required, treating the two methods as interchangeable and conflating index-based removal with value-based removal.	<code>[1, 2, 3].pop(2)</code>
same-state-same-object [1]	Students believe that two objects of the same class with identical attribute values must be the same object (i.e., equality implies identity), conflating <code>'=='</code> with <code>'is'</code> .	<code>assert [1, 2] is [1, 2]</code>
string-number-representation-confusion [1]	Students treat the string <code>"456"</code> as equivalent to the integer <code>'456'</code> , not understanding that the type distinction affects what operations are valid and how Python handles them.	<code>result = "456" + 1</code>
variable-declaration-memory-and-value-confusion [4]	Students hold inconsistent mental models about what happens in memory when a variable is declared, instantiated, or left unassigned, conflating naming, allocation, and value initialization.	<code>print(x)</code>
variable-name-string-literal-confusion [8]	Students conflate variable references with string literals, predicting that <code>'print(a)'</code> outputs the literal text <code>'a'</code> and that <code>'print('number')'</code> outputs the value stored in a variable named <code>'number'</code> .	<code>print('result')</code>

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant HCC-2426839.

Declaration on Generative AI

During the preparation of this work, the authors used Claude (Anthropic) for: Drafting content, Paraphrase and reword, and Grammar and spelling check. The construction of the master catalog

(Section 3) was a human-AI collaboration disclosed and audited as described in that section; that disclosure stands separately from this declaration, which covers the writing of the present manuscript. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

- [1] J. Sorva, Visual Program Simulation in Introductory Programming Education, Ph.D. dissertation, Aalto University, 2012.
- [2] L. Chiodini, I. Moreno Santos, A. Gallidabino, A. Tafliovich, A. L. Santos, M. Hauswirth, A curated inventory of programming language misconceptions, in: Proceedings of 26th ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE'21, ACM Press, 2021, pp. 380 – 386. doi:10.1145/3430665.3456343.
- [3] Y. Qian, J. Lehman, Students' misconceptions and other difficulties in introductory programming: A literature review, ACM Transactions on Computing Education 18 (2017) 1–24. doi:10.1145/3077618.
- [4] L. C. Kaczmarczyk, E. R. Petrick, J. P. East, G. L. Herman, Identifying student misconceptions of programming, in: Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10), 2010, pp. 107–111. doi:10.1145/1734263.1734299.
- [5] T. Sirkiä, J. Sorva, Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises, in: Proceedings of the 12th Koli Calling International Conference on Computing Education Research, ACM, Koli, Finland, 2012, pp. 19–28.
- [6] A. Altadmri, N. C. C. Brown, 37 million compilations: Investigating novice programming mistakes in large-scale student data, in: Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15), 2015, pp. 522–527. doi:10.1145/2676723.2677258.
- [7] A. Swidan, F. Hermans, M. Smit, Programming misconceptions for school students, in: Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18), 2018, pp. 151–159. doi:10.1145/3230977.3230995.
- [8] Z. Žanko, M. Mladenović, D. Krpan, Analysis of school students' misconceptions about basic programming concepts, Journal of Computer Assisted Learning 38 (2022) 738–747. doi:10.1111/jcal.12643.
- [9] F. Johnson, S. McQuistin, J. O'Donnell, Analysis of student misconceptions using python as an introductory programming language, in: Proceedings of the 4th Conference on Computing Education Practice (CEP '20), 2020, pp. 1–4. doi:10.1145/3372356.3372360.
- [10] D. H. Smith IV, C. Zilles, Discovering, autogenerating, and evaluating distractors for python parsons problems in cs1, in: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, ACM, 2023, pp. 924 – 930. doi:10.1145/3545945.3569801.
- [11] M. Hassany, K. Akhuseyinoglu, A. B. Lekshmi Narayanan, A. Agarwal, J. Savelka, P. Brusilovsky, Generating distractors for code completion problems: Can LLM assist instructors?, in: Proceedings of the Thirty-Eighth International Florida Artificial Intelligence Research Society Conference (FLAIRS-38), 2025. doi:10.32473/flairs.38.1.138995.
- [12] M. Hassany, P. Brusilovsky, J. Savelka, A. B. Lekshmi Narayanan, K. Akhuseyinoglu, A. Agarwal, R. A. Hendrawan, Generating effective distractors for introductory programming challenges: LLMs vs humans, in: Proceedings of the 15th International Learning Analytics and Knowledge Conference, ACM, 2025, p. 484–493. doi:10.1145/3706468.3706529.
- [13] Y. Lee, S. Kim, Y. Jo, Generating plausible distractors for multiple-choice questions via student choice prediction, in: Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2025, p. 23669–23692.
- [14] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, I. Stoica, Judging LLM-as-a-judge with MT-Bench and Chatbot Arena, in:

Advances in Neural Information Processing Systems 36 (NeurIPS 2023) Datasets and Benchmarks Track, 2023. [arXiv:2306.05685](#).

- [15] A. Panickssery, S. R. Bowman, S. Feng, LLM evaluators recognize and favor their own generations, 2024. [arXiv:2404.13076](#).
- [16] R. Rodriguez-Torrealba, E. Garcia-Lopez, A. Garcia-Cabot, End-to-end generation of multiple-choice questions using text-to-text transfer transformer models, *Expert Systems with Applications* 208 (2022) 118258.
- [17] K. Vachev, M. Hardalov, G. Karadzhov, G. Georgiev, I. Koychev, P. Nakov, Leaf: Multiple-choice question generation, in: *European Conference on Information Retrieval*, Springer, 2022, pp. 321–328.
- [18] D. R. Ch, S. K. Saha, Automatic multiple choice question generation from text: A survey, *IEEE Transactions on Learning Technologies* 13 (2018) 14–25.
- [19] V. R. Madri, S. Meruva, A comprehensive review on mcq generation from text, *Multimedia Tools and Applications* 82 (2023) 39415–39434.
- [20] B. H. H. Cheung, G. K. K. Lau, G. T. C. Wong, E. Y. P. Lee, D. Kulkarni, C. S. Seow, R. Wong, M. T. H. Co, Chatgpt versus human in generating medical graduate exam questions—an international prospective study, *medRxiv* (2023) 2023–05.
- [21] N. Nasution, Using artificial intelligence to create biology multiple choice questions for higher education, *Agricultural and Environmental Education* 2 (2023) 4–8.
- [22] J. Doughty, Z. Wan, A. Bompelli, J. Qayum, T. Wang, J. Zhang, Y. Zheng, A. Doyle, P. Sridhar, A. Agarwal, et al., A comparative study of AI-generated (GPT-4) and human-crafted MCQs in programming education, in: *Proceedings of the 26th Australasian Computing Education Conference*, 2024, pp. 114–123.
- [23] C. Liang, X. Yang, N. Dave, D. Wham, B. Pursel, C. L. Giles, Distractor generation for multiple choice questions using learning to rank, in: *Proceedings of the thirteenth workshop on innovative use of NLP for building educational applications*, 2018, pp. 284–290.
- [24] S. Ren, K. Q. Zhu, Knowledge-driven distractor generation for cloze-style multiple choice questions, in: *Proceedings of the AAAI conference on artificial intelligence*, volume 35, 2021, pp. 4339–4347.
- [25] W. Feng, J. Lee, H. McNichols, A. Scarlatos, D. Smith, S. Woodhead, N. Ornelas, A. Lan, Exploring automated distractor generation for math multiple-choice questions via large language models, in: *Findings of the Association for Computational Linguistics: NAACL 2024*, 2024, pp. 3067–3082.
- [26] H. McNichols, W. Feng, J. Lee, A. Scarlatos, D. Smith, S. Woodhead, A. Lan, Automated distractor and feedback generation for math multiple-choice questions via in-context learning, 2023. [arXiv:2308.03234](#).
- [27] J. Shin, Q. Guo, M. J. Gierl, Multiple-choice item distractor development using topic modeling approaches, *Frontiers in psychology* 10 (2019) 825.
- [28] D. Traynor, J. P. Gibson, Synthesis and analysis of automatic assessment methods in CS1: generating intelligent MCQs, *ACM SIGCSE Bulletin* 37 (2005) 495–499.
- [29] Z. Wu, D. H. Smith IV, Evaluating micro parsons problems as exam questions, in: *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, 2024, pp. 674–680.
- [30] M. J. Gierl, O. Bulut, Q. Guo, X. Zhang, Developing, analyzing, and using distractors for multiple-choice tests in education: A comprehensive review, *Review of Educational Research* 87 (2017) 1082–1116. doi:10.3102/0034654317726529.
- [31] C. Grévisse, Comparative quality analysis of gpt-based multiple choice question generation, in: *International Conference on Applied Informatics*, Springer, 2023, pp. 435–447.
- [32] A. P. Kumar, A. Nayak, M. Shenoy, S. Goyal, et al., A novel approach to generate distractors for multiple choice questions, *Expert Systems with Applications* 225 (2023) 120022.
- [33] S. Jiang, J. S. Lee, Distractor generation for chinese fill-in-the-blank items, in: *Proceedings of the 12th workshop on innovative use of NLP for building educational applications*, 2017, pp. 143–148.
- [34] S. Lane, M. R. Raymond, T. M. Haladyna, et al., *Handbook of test development*, volume 2, Routledge New York, NY, 2016.
- [35] R. Moreno Rodríguez, R. J. Martínez, J. Muñoz Fernández, et al., Guidelines based on validity

- criteria for the development of multiple choice items, *Psicothema*, 27 (4) (2015).
- [36] S. M. Case, D. B. Swanson, *Constructing written test questions for the basic and clinical sciences*, National Board of Medical Examiners Philadelphia, 1998.
 - [37] M. Tarrant, J. Ware, A. M. Mohammed, An assessment of functioning and non-functioning distractors in multiple-choice questions: a descriptive analysis, *BMC medical education* 9 (2009) 40.
 - [38] E. Albrecht, J. Grabowski, Sometimes it’s just sloppiness - studying students’ programming errors and misconceptions, in: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE ’20)*, 2020, pp. 340 – 345. doi:10.1145/3328778.3366862.
 - [39] R. D. Pea, Language-independent conceptual “bugs” in novice programming, *Journal of Educational Computing Research* 2 (1986) 25–36. doi:10.2190/689T-1R2A-X4W4-29J2.
 - [40] M. Lopez, J. Whalley, P. Robins, R. Lister, Relationships between reading, tracing and writing skills in introductory programming, in: *Proceedings of the Fourth International Workshop on Computing Education Research (ICER ’08)*, 2008, pp. 101–112. doi:10.1145/1404520.1404531.
 - [41] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, L. Thomas, A multi-national study of reading and tracing skills in novice programmers, in: *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, 2004, pp. 119–150. doi:10.1145/1044550.1041673.
 - [42] N. Shrestha, C. Botta, T. Barik, C. Parnin, Here we go again: Why is it difficult for developers to learn another programming language?, in: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE ’20*, ACM, 2020, pp. 691–701. URL: <https://doi.org/10.1145/3377811.3380352>.
 - [43] Z. C. Liding, M. Osmolovskiy, H. Lanka, N. Roy, R. Borela, AI-augmented instruction: Real-time misconception detection, in: *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.2*, 2026, pp. 1421 – 1422. doi:10.1145/3770761.3777326.
 - [44] E. Al-Hossami, R. Bunescu, Mcmining: Automated discovery of misconceptions in student code, in: *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics, Volume 2: Short Papers*, 2026. URL: <https://aclanthology.org/2026.eacl-short.10.pdf>.
 - [45] R. Niousha, S. Smith, A. O’Neill, J. Zamfirescu-Pereira, J. DeNero, N. Norouzi, Misconception-aware LLM programming tutor: Lessons learned from student-tutor interactions, in: *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.2*, 2026, pp. 1455 – 1456. doi:10.1145/3770761.3777289.
 - [46] Y. Chen, C. Zhao, K. Feng, J. Zhang, V. Malhotra, M. Silva, Exploring LLMs for generating erroneous examples in cs1, in: *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.2*, 2026, pp. 1261 – 1262. doi:10.1145/3770761.3777210.

A. Catalog tables

Table 4: Pipeline-output catalog (44 entries, v2-cw-thin) used as the catalog-pipeline experimental input. Generated by the in-progress automated catalog-construction pipeline (Section 3); citations under each id record the peer-reviewed catalog source(s) the extraction pipeline attributed the entry to.

ID	Misconception	Python example
actual-vs-formal-parameter-confusion [1]	Students conflate the arguments passed at a function call site with the formal parameters in the function’s definition, leading to errors such as believing call-site and parameter names must differ or being unable to trace where parameter values originate.	<code>f(x)</code>
and-operator-as-double-ampersand__chiadini2021 [42, 2]	Students who have prior experience with Java or JavaScript believe that ‘&&’ is valid Python syntax for logical AND, when in fact Python uses the keyword ‘and’ and ‘&&’ is a syntax error.	<code>if x > 5 && y < 10:</code>

ID	Misconception	Python example
assignment-equality-operator-confusion [6, 2]	Students conflate the assignment operator with the equality-comparison operator, writing '=' where a comparison is needed (e.g., 'if x = 5:') or '==' where a value must be stored.	if x = 5:
assignment-reversed-direction__sorva2012 [1]	Students believe that 'x = 5' stores x into 5 (right-to-left reading of the target), reversing the actual left-hand-target semantics of assignment.	5 = x
bracket-or-missing-parens-on-list-method-call [10]	Students invoke list methods like .append(), .remove(), and .count() using subscript brackets (e.g., lst.remove[value]) or by omitting parentheses entirely (e.g., lst.count), conflating method invocation with indexing or attribute access.	n = items.count[x]
c-style-for-loop-in-python [42]	Students with prior C or Java exposure write C-style for loops (parentheses, type declaration, increment expression) in Python contexts, producing non-Python syntax instead of the 'for i in range(n):' idiom.	for (int i = 0; i < n; i++):
class-instance-relationship-confusion [1]	Students misconceive the relationship between a class and its objects, treating the class as either a collection containing its instances or as a supertype of which each object is a subtype, rather than as a blueprint or type that instances are created from.	x = cls[0]
comparison-with-bool-literal__progmiscon-web [2]	Incorrect: To test whether an expression is True or False, one must compare it to True or to False.	if (x > 5) == True:
computer-infers-programmer-intent [1]	Students believe the computer can deduce what the programmer meant and fill in gaps or correct errors, instead of executing the written code literally.	x = "10" + 5 # Trusting Python to understand I want numeric addition
dot-operator-only-for-methods__sorva2012 [1]	Students believe the dot notation (e.g., 'obj.x') can only be used to call methods, not to access data attributes of an object.	type('C', (), {'x': 5})().x()
equality-assignment-identity-operator-confusion [6, 1]	Students conflate the operators for assignment ('='), value equality ('=='), and object identity ('is'), using them interchangeably in conditionals and assignments.	if x = 5:
equals-sign-operator-confusion [6, 8]	Students misread the '=' symbol's role: writing comparison operators as '>='/'<=' instead of '>='/'<=', and treating assignment '=' as bidirectional mathematical equality rather than a directed store.	if x => 5:
for-loop-control-variable-semantics [1]	Students misunderstand the semantics of a for-loop's iteration variable: how it is automatically updated each iteration, whether it is accessible or modifiable inside the loop body, and what its values constrain.	for i in range(10): i = 20 if i == 5 else i; print(i)
func6-return-value-not-captured__sirkia2012 [5]	Students believe a returned value is automatically deposited into the caller's variable without an explicit assignment, failing to assign the result of a call like 'text = ask_euros()'.	get_value() print(result)
function-argument-and-return-type-mismatch [6]	Students invoke functions with arguments whose types do not match what the function expects, or assign a function's return value to a variable used as an incompatible type.	x = len(5)
function-argument-return-type-mismatch [6]	Students call functions with arguments of the wrong type or assign function return values to variables of an incompatible type, reflecting a mental model in which functions accept and produce values regardless of type.	result = len(42)
function-mutates-input-to-output__sorva2012 [1]	Students believe a function works by transforming the variable passed in, so the argument variable itself becomes the result rather than a separate return value being produced.	y = 5; increment(y); print(y)
init-misunderstood-as-object-creator [1, 2]	Students confuse __init__ with object creation itself, believing it allocates the object, must return it, must contain non-trivial code, or is limited to plain attribute assignments.	def __init__(self, x): return self
instance-var-vs-object-ref-confusion__sorva2012 [1]	Students conflate an object's instance variable (e.g., an attribute called 'name') with the variable that holds a reference to the object itself.	def foo(person): return name
list-append-api-confusion [10]	Students fail to add an element to a list by calling list.append(item) correctly, instead using attribute assignment (list.append = value), the wrong method (list.insert without an index), or a method from another collection type (list.add).	items.append = x
list-callable-instead-of-method__smith2023 [10]	Students write 'result = lst(value)' — calling the list object directly as though it were a function — instead of 'result = lst.count(value)'.	result = lst(value)

ID	Misconception	Python example
list-plus-commutative__johnson2020 [9]	Students assume that '+' is commutative for lists and strings, predicting that '[6,7] + [4,5]' and '[4,5] + [6,7]' produce the same result when they do not.	[6,7] + [4,5] == [4,5] + [6,7]
list-vs-element-confusion__sorva2012 [1]	Students confuse the entire list/array object with a single cell or element within it, leading to errors when indexing, assigning, or passing lists to functions.	total = total + items
method-call-syntax-confusion-brackets-or-missing-parens [10]	Students invoke list methods using subscript brackets (e.g., 'lst.append[x]', 'lst.count[item]', 'lst.remove[v]') or omit parentheses entirely (e.g., 'lst.count'), treating method invocation as indexing or as bare attribute access.	x = items.count[value]
misc3-false-condition-returns-false__sirkia2012 [5]	Students conflate the boolean value of a conditional expression with the function's return value, believing that a condition evaluating to False causes the function to immediately return False.	return x if x > 0 else x > 0
mmr5-off-by-one-array-length__kaczmarczyk2010 [4]	Students believe a list or array of length N has valid indices 0 through N inclusive (N+1 slots), rather than 0 through N-1.	x = items[len(items)]
no-atomic-expression__chiodini2021 [2]	Students believe that a single literal or variable name is not a valid expression in Python; they think an expression must always combine multiple pieces (e.g., an operator and operands).	def foo(value + 0):
no-reserved-words__progmiskon-web [2]	Incorrect: Every sequence of letters and digits starting with a letter or an underscore can be used as an identifier.	return = 5
no-short-circuit__progmiskon-web [2]	Incorrect: and/or always evaluate both operands.	x > 0 and items.append(value)
objects-know-their-methods__sorva2012 [1]	Students model the relationship as the object being aware of and directing method calls, rather than understanding that a method call selects an object to operate on via 'self'.	MyClass.method()
objects-self-create__sorva2012 [1]	Students believe objects come into existence automatically by virtue of a class being defined, without needing an explicit instantiation call such as 'MyClass()'.	MyClass.method()
oo5-method-call-without-object-lookup__sirkia2012 [5]	Students believe a method can be invoked by name alone without first resolving the object reference through a variable, skipping the step of accessing the target instance before dispatching the call.	obj = SomeClass(); method()
pop-vs-remove-list-method-confusion [10]	Students use list.pop() when list.remove() is required, treating the two methods as interchangeable and conflating index-based removal with value-based removal.	arr.pop(20)
recursion-is-magic__sorva2012 [1]	Students treat recursion as a syntactic template to copy-paste for certain problem types without understanding how the runtime call stack actually executes, viewing the behavior as unexplainable.	def f(n): return f(n-1)
return-value-need-not-be-stored__sorva2012 [1]	Students believe a function's return value is available for later use even if it was not captured in a variable at the point of the call.	func(); print(result)
same-state-same-object__sorva2012 [1]	Students believe that two objects of the same class with identical attribute values must be the same object (i.e., equality implies identity), conflating '==' with 'is'.	if MyClass(1) is MyClass(1):
static-vs-instance-call__altadmri2015 [6]	Students call an instance method directly on the class name rather than on an object instance, conflating the class with its instances.	my_list = [3, 1, 2]; list.sort()
string-number-representation-confusion__sorva2012 [1]	Students treat the string "456" as equivalent to the integer '456', not understanding that the type distinction affects what operations are valid and how Python handles them.	total = "100" - "50"
subprogram-accesses-caller-vars__sorva2012 [1]	Students believe a called function can freely read and write variables that belong to its calling function, not understanding Python's scoping rules that prevent this.	x = 5; def f(): x += 1; f()
superbug-hidden-mind__pea1986 [39]	Underlying parallelism, intentionality, and egocentrism errors is a tacit assumption that a hidden, intelligent interpreter lives inside the language and can infer intent, understand context across lines, and benevolently fill in what the programmer left implicit.	count = count + 1
type-as-constraint__sorva2012 [1]	Students model types as mere validation rules that restrict which values are allowed, missing the richer concept of types as determining what operations are meaningful and how values are represented.	result = x + y if isinstance(x, int) and isinstance(y, str) else None
variable-name-string-literal-confusion [8]	Students conflate variable references with string literals, predicting that 'print(a)' outputs the literal text 'a' and that 'print('number')' outputs the value stored in a variable named 'number'.	x = 5; print('x')

ID	Misconception	Python example
vps3-function-used-as-operand__ sirkia2012 [5]	Students believe a function such as 'float' can serve directly as a numeric operand in an arithmetic expression without first being called and having its return value substituted, treating the function object as a number.	5 * float
wrong-method-to-append-to-list [10]	Students add an element to the end of a list using something other than 'list.append(item)' — assigning to 'list.append', calling 'list.insert(item)', or calling 'list.add(item)'.	items.add(value)