Classroom 50 Incremental Autograding

Quick Start Guide

A compact companion to Building Incremental Custom Autograders for Classroom 50

Goal: build a small autograder that gives students useful progress feedback on every push instead of waiting for the entire program to be finished.

1. The workflow

1. Define the observable program contract.
2. Write a known-correct reference solution.
3. Break the behavior into small, incremental capabilities.
4. Build and test a local harness first.
5. Test partial solutions and deliberately broken solutions.
6. Convert the validated tests into Classroom 50 result.json output.
7. Install autograder.py under the assignment slug.
8. Verify the real student workflow with partial and complete code.

2. Tiny example: Number Analyzer

Contract: the program prints a title, asks for one integer, and reports whether the value is positive, negative, or zero.

```
Number Analyzer
Enter an integer: 7
Result: positive
```

Use exact, deterministic output for anything you intend to test automatically. For this example, assume the input is always a valid integer.

Reference solution

```
def main():
    print("Number Analyzer")
    number = int(input("Enter an integer: "))

    if number > 0:
        result = "positive"
    elif number < 0:
        result = "negative"
    else:
        result = "zero"

    print(f"Result: {result}")


if __name__ == "__main__":
    main()
```

Design rule: Test the observable contract, not whether the student reproduced the instructor solution.

3. Design incremental tests

For this example, use only four tests:

Test	Capability
Program interface	Correct title and input prompt
Positive number	Correctly identifies a positive integer
Negative number	Correctly identifies a negative integer
Zero	Correctly identifies zero

A student who has only the interface should pass 1/4. Adding positive-number handling should raise that to 2/4. The earlier interface test should continue to pass after later features are added.

Monotonicity: When practical, adding correct later functionality should not make an earlier capability test fail.

4. Build the local harness first

Run the student program as a subprocess so the harness can simulate stdin, capture stdout/stderr, detect crashes, and impose a timeout.

```
import subprocess, sys

TIMEOUT_SECONDS = 3

def run_program(program, user_input):
    return subprocess.run(
        [sys.executable, program],
        input=user_input,
        text=True,
        capture_output=True,
        timeout=TIMEOUT_SECONDS,
    )

def test_interface(program):
    r = run_program(program, "7\n")
    expected = "Number Analyzer\nEnter an integer: "
    return r.returncode == 0 and r.stdout.startswith(expected)

def test_result(program, value, expected):
    r = run_program(program, f"{value}\n")
    return r.returncode == 0 and f"Result: {expected}" in r.stdout

TESTS = [
    ("Program interface", lambda p: test_interface(p)),
    ("Positive number", lambda p: test_result(p, 7, "positive")),
    ("Negative number", lambda p: test_result(p, -4, "negative")),
    ("Zero", lambda p: test_result(p, 0, "zero")),
]
```

Your full local harness can add friendly PASS/FAIL output and timeout handling. The important point is to validate the test logic locally before adding Classroom 50 integration.

Validate the validator

- Reference solution: expect 4/4.
- Interface-only partial solution: expect 1/4.
- Positive-only partial solution: expect 2/4.
- Mutate `number > 0` to `number >= 0`: the Zero test should fail.
- Change the title: the Program interface test should fail.

5. Convert the tests to Classroom 50

Keep the same test logic. Replace local PASS/FAIL printing with a Classroom 50 result.json file. Each failed student test is recorded in JSON; the autograder itself still exits successfully if it completed its job.

```
import datetime, json, os
from pathlib import Path


# Assume TESTS contains (name, function) pairs and each
# function returns True/False after running main.py.

def main():
    test_results = []

    for name, test_function in TESTS:
        try:
            passed = bool(test_function("main.py"))
        except Exception:
            passed = False

        test_results.append({
            "test-name": name,
            "passed": passed,
            "score": 1 if passed else 0,
            "max-score": 1,
        })

    result = {
        "schema": "classroom50/result/v1",
        "classroom": os.environ["CLASSROOM"],
        "assignment": os.environ["ASSIGNMENT"],
        "submission": os.environ["SUBMISSION_TAG"],
        "commit": os.environ["COMMIT_URL"],
        "release": os.environ["RELEASE_URL"],
        "review": os.environ.get("REVIEW_URL")
                or os.environ["COMMIT_URL"],
        "datetime": datetime.datetime.now(
            datetime.timezone.utc
        ).strftime("%Y-%m-%dT%H:%M:%SZ"),
        "score": sum(t["score"] for t in test_results),
        "max-score": sum(t["max-score"] for t in test_results),
        "tests": test_results,
    }

    Path("result.json").write_text(
        json.dumps(result, indent=2), encoding="utf-8"
    )
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

Important: A student test failure belongs in result.json. A nonzero autograder exit should generally mean the grading infrastructure itself failed.

6. Install it in the Classroom 50 configuration repository

For an assignment-specific custom grader, use the exact Classroom 50 assignment slug:

```
classroom50/
└── <classroom>/
    └── autograders/
        └── <assignment-slug>/
            └── autograder.py
```

If you do not know the slug, check the classroom's assignments.json. Do not guess it.

Use Classroom 50's built-in autograding infrastructure. The assignment-specific autograder.py is an override executed by that infrastructure; it is not a replacement GitHub Actions workflow.

7. Verify the real student workflow

Use a student/test account. Push several development states and confirm Classroom 50 reports the expected progression:

```
Interface only          -> 1/4
+ positive handling       -> 2/4
Complete reference solution -> 4/4
```

If the local harness behaves correctly but Classroom 50 does not, you now know the remaining problem is integration rather than test design.

8. Rules worth keeping

Define the contract first. Autograding cannot repair an ambiguous assignment specification.

Test capabilities, not milestones. A test should correspond to a meaningful observable behavior.

Keep tests narrow and diagnostic. A failing test name should tell the student what behavior to investigate.

Prefer monotonic early tests. Later correct functionality should not unnecessarily invalidate earlier passes.

Test boundaries. Use values around thresholds, not only typical values.

Mutation-test the suite. Deliberately introduce likely bugs and verify that the intended tests fail.

Use timeouts. Student programs can accidentally loop forever.

Do not overfit to the reference solution. Equivalent implementations should pass if they satisfy the contract.

Separate correctness from quality. Autograder points can represent tests passed; style, architecture, documentation, Git history, and understanding may still require instructor review.

Verify end-to-end. The suite is not finished until it works through the same Classroom 50 workflow students will use.

9. Reusable checklist

- ☐ Observable output is explicit and deterministic.
- ☐ Reference solution passes every intended test.
- ☐ Partial solutions receive useful partial success.
- ☐ Likely incorrect solutions fail the appropriate tests.
- ☐ Local test harness is stable before Classroom 50 conversion.
- ☐ autograder.py writes result.json and returns 0 after ordinary student test failures.
- ☐ autograder.py is stored under autograders/<assignment-slug>/.
- ☐ Real student/test repository has been checked with partial and complete solutions.

References and version note

Classroom 50 project: https://github.com/foundation50/classroom50

Teacher guide: https://github.com/gh-cli-for-education/hello-c50/blob/main/teacher-guide.md

Autograder architecture discussion: https://github.com/foundation50/classroom50/discussions/206

Current as of September 2026. Classroom 50 is under active development; verify current paths and interface wording if the platform changes.