Building Incremental Custom Autograders for Classroom 50

Prepared for Austin Community College Software Development & Computer Science Department

Alexander Katrompas, PhD
September 5th, 2026

Table of Contents

Introduction 2
Start With the Observable Program Contract 2
    Why define the contract first? 3
Build a Reference Solution 3
Think in Capabilities, Not Just Final Correctness 4
Make Incremental Tests Monotonic When Possible 5
Build a Standalone Test Script First 6
Run the Local Tests 10
Test the Tests Incrementally 10
Mutation-Test the Test Suite 11
Avoid Testing Implementation Unless Implementation Is the Requirement 12
Convert the Validated Tests to Classroom 50 12
Classroom 50 Version of the Number Analyzer Grader 13
Why the Autograder Exits Successfully Even When Tests Fail 17
Understanding result.json 18
Install the Custom Autograder 19
Classroom 50 Assignment Configuration 19
Test the Real Classroom 50 Workflow 20
Why Local Testing Before Classroom 50 Matters 20
Do Not Make Every Test Depend on the Finished Program 21
Test One Idea at a Time 21
Test Boundaries, Not Just Typical Values 22
Give Every Test a Purpose 23
Use Timeouts 23
Run Student Programs as Separate Processes 23
Do Not Overfit the Tests to the Reference Solution 24
Keep the Autograder Out of the Student's Way 24
Updating an Autograder 25
Scaling This Pattern to Larger Assignments 25
A Recommended Workflow for Every New Assignment 26
Final Principles 26
Classroom 50 References 27

Introduction

Classroom 50 supports automated testing of programming assignments through GitHub. A custom autograder can run whenever a student pushes code, evaluate the current state of the program, and report individual test results back to the student.

A useful instructional autograder should do more than determine whether the final program is correct.

It should also support the **development process**.

A student who has correctly completed the first part of an assignment should be able to see evidence that the first part works, even if the rest of the program has not yet been written. As the student develops the program incrementally, additional tests should begin to pass.

This guide develops that approach from beginning to end using a deliberately simple Python program.

The process is:

1. define the program's observable contract;
2. create a known-correct reference solution;
3. identify incremental capabilities worth testing;
4. build a standalone local test script;
5. test the test script against partial, correct, and deliberately incorrect programs;
6. convert the validated tests into a Classroom 50 custom autograder;
7. install the autograder in the Classroom 50 configuration repository; and
8. verify the complete process using an actual assignment repository.

Classroom 50 is a free, open-source system for managing and autograding GitHub-based programming assignments.

> ***Note:*** *Classroom 50 is under active development. The paths and autograder interface in this guide reflect a working configuration as of September 2026. Consult the current Classroom 50 documentation if the software changes.*

Start With the Observable Program Contract

Before writing tests, define exactly what the program is required to do **from the outside**.

For this guide, we will use a small program called **Number Analyzer**.

The program asks the user for one integer and determines whether that number is positive, negative, or zero.

The program must begin by displaying exactly:

```
Number Analyzer
```

It must then ask:

```
Enter an integer:
```

There must be one space after the colon before the user enters a value.

After reading the integer, the program must display exactly one of:

```
Result: positive

Result: negative
```

```
Result: zero
```

For example:

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

For -4:

```
Number Analyzer
Enter an integer: -4
Result: negative
```

For 0:

```
Number Analyzer
Enter an integer: 0
Result: zero
```

For this example, assume the user always enters a valid integer. Input validation is deliberately outside the scope of the problem.

Why define the contract first?

The autograder needs a deterministic target.

If the assignment merely says:

> *Ask the user for a number and say whether it is positive or negative.*

then many questions remain:

- What should the prompt say?
- Is zero positive, negative, or something else?
- What capitalization should be used?
- What output should the program produce?
- Is extra text allowed?

Those questions become especially important when programs are evaluated automatically.

A good assignment distinguishes between:

Observable requirements, which the autograder can verify, and **implementation decisions**, which students may be expected to make themselves.

For Number Analyzer, the autograder cares about the title, prompt, and classification results. It does not need to know how the student implemented the decision-making logic.

Build a Reference Solution

Before building the tests, create a known-correct implementation.

For Number Analyzer:

```
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()
```

The reference solution serves two purposes.

First, it gives us a program that should pass every test.

Second, writing the reference solution often reveals ambiguity in the assignment itself. If the instructor cannot implement the specification without making unstated assumptions, the specification should probably be clarified before automated tests are written.

The reference solution is therefore not merely an answer key. It is also a test of the **assignment design**.

Think in Capabilities, Not Just Final Correctness

A common first approach to autograding is to run the completed program and compare all of its output against the expected output.

That works for final verification, but it is poor feedback during development.

Suppose a student has written only:

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


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

This is not a finished assignment.

But it is also not entirely wrong.

The student has correctly:

- created the required program structure;
- displayed the correct title; and
- implemented the required input interface.

A useful instructional autograder should recognize that progress.

For this small assignment, we will define 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

Test	Capability
Zero	Correctly identifies zero

With this structure, the partial program above should receive:

```
[PASS] Program interface
[FAIL] Positive number
[FAIL] Negative number
[FAIL] Zero

1/4 tests passed
```

If the student then adds:

```
if number > 0:
    print("Result: positive")
```

the student could now see:

```
[PASS] Program interface
[PASS] Positive number
[FAIL] Negative number
[FAIL] Zero

2/4 tests passed
```

Eventually:

```
4/4 tests passed
```

This leads to an important design principle:

> ***An instructional autograder should provide useful evidence of progress during development, not merely pronounce the finished program correct or incorrect.***

Make Incremental Tests Monotonic When Possible

There is a subtle testing issue here.

Suppose the first test required the program's entire output to be exactly:

```
Number Analyzer
Enter an integer:
```

The partial program would pass.

But once the student correctly added:

```
Result: positive
```

the first test would fail because the program now produces additional output.

That is undesirable.

A legitimate improvement to the program should not normally cause an earlier development-stage test to stop passing.

Therefore, the first test should check only the capability it owns.

For example:

```
expected_prefix = (
    "Number Analyzer\n"
    "Enter an integer: "
)

output.startswith(expected_prefix)
```

The unfinished program passes.

The finished program also passes.

This property is sometimes useful to think of as **monotonic testing**:

> *Once a correctly implemented capability passes its test, adding later required capabilities should not normally make that earlier test fail.*

Not every test suite can achieve perfect monotonicity, but it is an excellent design goal for educational autograders.

Build a Standalone Test Script First

Do not begin by debugging the tests inside Classroom 50.

First build a normal local program that tests the assignment.

For this example, create:

```
test_number_analyzer.py
```

The test script will:

- run main.py as another process;
- provide simulated keyboard input;
- capture the program's output;
- check one capability at a time;
- detect crashes; and
- stop programs that fail to terminate.

Here is a complete test script:

```
#!/usr/bin/env python3

import re
import subprocess
import sys
from pathlib import Path


TIMEOUT_SECONDS = 3

TITLE = "Number Analyzer"
PROMPT = "Enter an integer: "


def resolve_program_path(argument):
    path = Path(argument).expanduser().resolve()

    if path.is_dir():
        path = path / "main.py"
```

```
    return path


def run_program(program, user_input):
    try:
        completed = subprocess.run(
            [sys.executable, program.name],
            input=user_input,
            text=True,
            capture_output=True,
            cwd=program.parent,
            timeout=TIMEOUT_SECONDS,
        )

        return {
            "stdout": completed.stdout.replace("\r\n", "\n"),
            "stderr": completed.stderr.replace("\r\n", "\n"),
            "returncode": completed.returncode,
            "timed_out": False,
        }

    except subprocess.TimeoutExpired:
        return {
            "stdout": "",
            "stderr": "",
            "returncode": None,
            "timed_out": True,
        }


def execution_problem(result):
    if result["timed_out"]:
        return (
            f"program did not finish within "
            f"{TIMEOUT_SECONDS} seconds"
        )

    if result["returncode"] != 0:
        details = result["stderr"].strip()

        if details:
            return (
                f"program exited with code "
                f"{result['returncode']}: {details}"
            )

        return (
            f"program exited with code "
            f"{result['returncode']}"
        )

    return None


def results_from(output):
```

```
    return re.findall(
        r"Result: (positive|negative|zero)",
        output,
    )


def test_interface(program):
    result = run_program(program, "7\n")
    problem = execution_problem(result)

    if problem:
        return False, problem

    expected_prefix = (
        f"{TITLE}\n"
        f"{PROMPT}"
    )

    if not result["stdout"].startswith(expected_prefix):
        return False, "title or input prompt is incorrect"

    return True, ""


def test_classification(program, value, expected):
    result = run_program(program, f"{value}\n")
    problem = execution_problem(result)

    if problem:
        return False, problem

    actual = results_from(result["stdout"])

    if actual != [expected]:
        return False, (
            f"expected {expected!r}, "
            f"got {actual}"
        )

    return True, ""


def test_positive(program):
    return test_classification(
        program,
        7,
        "positive",
    )


def test_negative(program):
    return test_classification(
        program,
        -4,
        "negative",
    )
```

```
def test_zero(program):
    return test_classification(
        program,
        0,
        "zero",
    )


TESTS = [
    ("Program interface", test_interface),
    ("Positive number", test_positive),
    ("Negative number", test_negative),
    ("Zero", test_zero),
]


def main():
    argument = (
        sys.argv[1]
        if len(sys.argv) > 1
        else "main.py"
    )

    program = resolve_program_path(argument)

    if not program.is_file():
        print(f"ERROR: cannot find {program}")
        return 2

    passed = 0

    for name, test_function in TESTS:
        try:
            ok, details = test_function(program)

        except Exception as exc:
            ok = False
            details = (
                f"test script error: "
                f"{type(exc).__name__}: {exc}"
            )

        if ok:
            passed += 1
            print(f"[PASS] {name}")

        else:
            print(f"[FAIL] {name}")

            if details:
                print(f"       {details}")

    print()
    print(f"{passed}/{len(TESTS)} tests passed")

    return 0 if passed == len(TESTS) else 1
```

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

Run the Local Tests

If the test script and main.py are in the same directory:

```
python3 test_number_analyzer.py
```

You can also test a program elsewhere:

```
python3 test_number_analyzer.py /path/to/main.py
```

or an entire student repository:

```
python3 test_number_analyzer.py /path/to/student-repository
```

The script will locate main.py inside that directory.

The correct reference solution should produce:

```
[PASS] Program interface
[PASS] Positive number
[PASS] Negative number
[PASS] Zero

4/4 tests passed
```

Test the Tests Incrementally

Do not test only the finished solution.

We want to know whether the **test suite itself** behaves correctly during development.

Start with:

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


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

Expected:

```
[PASS] Program interface
[FAIL] Positive number
[FAIL] Negative number
[FAIL] Zero

1/4 tests passed
```

Then add:

```
if number > 0:
    print("Result: positive")
```

Expected:

```
[PASS] Program interface
[PASS] Positive number
[FAIL] Negative number
[FAIL] Zero

2/4 tests passed
```

Finally test the complete reference solution:

```
4/4 tests passed
```

This verifies that the autograder reflects legitimate incremental development.

Mutation-Test the Test Suite

A correct program passing every test is necessary, but it is not enough.

You should also intentionally create **wrong programs** and verify that the tests detect the errors.

This is commonly called mutation testing.

For example, change:

```
if number > 0:
```

to:

```
if number >= 0:
```

Now zero will probably be classified as positive.

The zero test should fail.

Change:

```
elif number < 0:
```

to:

```
elif number > 0:
```

The negative test should fail.

Change:

```
print("Number Analyzer")
```

to:

```
print("Number Analyser")
```

The interface test should fail.

Change:

```
print(f"Result: {result}")
```

to:

```
print(f"The result is {result}")
```

The classification test should fail because the observable contract is no longer satisfied.

Mutation testing answers an important question:

> ***Does the test suite actually detect the mistakes it was designed to detect?***

A test suite that passes the correct solution but also passes several obviously incorrect solutions is not yet trustworthy.

Avoid Testing Implementation Unless Implementation Is the Requirement

Suppose the assignment requires students to make a decision based on the value.

The autograder should usually test:

```
Result: positive
```

rather than searching the source code for:

```
if
```

There are many reasons for this.

A source-code search can produce false results. The word if could appear in a comment. A student may implement equivalent behavior differently. Source inspection also couples the grader to implementation instead of behavior.

If the assignment explicitly requires a particular technique, that requirement may need separate evaluation.

For example:

- use a for loop;
- do not use break;
- use a specific function signature;
- use recursion;
- use an abstract data type;
- maintain appropriate Git history.

Those may be important assignment requirements, but they are not necessarily well represented by behavioral autograding.

A useful distinction is:

> ***Automated tests establish behavioral correctness. They do not automatically establish software quality, design quality, or student understanding.***

Those may still require instructor review.

Convert the Validated Tests to Classroom 50

Only after the local test suite behaves correctly should it be converted into a Classroom 50 custom autograder.

Classroom 50 stores custom autograders in the **instructor's Classroom 50 configuration repository** rather than permanently embedding the grader itself into every student repository. Student repositories contain a workflow shim that retrieves the current instructor-defined grader. This allows an instructor to update a grader without editing every student's repository.

For a per-assignment custom grader, Classroom 50 supports:

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

A classroom can also have a default autograder.py, but an assignment-specific directory provides an override for that assignment.

For example:

```
classroom50/
└── cosc-1336/
    ├── classroom.json
    ├── assignments.json
    ├── ...
    └── autograders/
        └── number-analyzer/
            └── autograder.py
```

The directory must use the assignment's actual Classroom 50 **slug**.

Do not guess the slug if you are uncertain. Check the classroom's assignments.json.

Classroom 50 Version of the Number Analyzer Grader

The actual testing logic can remain almost identical.

The major difference is that instead of printing:

```
[PASS]
```

and:

```
[FAIL]
```

for local use, the Classroom 50 version builds a structured result.json file.

Here is a complete example:

```python
#!/usr/bin/env python3

import datetime
import json
import os
import re
import subprocess
import sys
from pathlib import Path


TIMEOUT_SECONDS = 3

TITLE = "Number Analyzer"
PROMPT = "Enter an integer: "
PROGRAM = Path("main.py")


def run_program(user_input):
    if not PROGRAM.is_file():
```

```
        return {
            "stdout": "",
            "stderr": "main.py was not found",
            "returncode": None,
            "timed_out": False,
            "missing": True,
        }

    try:
        completed = subprocess.run(
            [sys.executable, str(PROGRAM)],
            input=user_input,
            text=True,
            capture_output=True,
            timeout=TIMEOUT_SECONDS,
        )

        return {
            "stdout": completed.stdout.replace(
                "\r\n",
                "\n",
            ),
            "stderr": completed.stderr.replace(
                "\r\n",
                "\n",
            ),
            "returncode": completed.returncode,
            "timed_out": False,
            "missing": False,
        }

    except subprocess.TimeoutExpired:
        return {
            "stdout": "",
            "stderr": "",
            "returncode": None,
            "timed_out": True,
            "missing": False,
        }


def execution_problem(result):
    if result.get("missing"):
        return "main.py was not found"

    if result["timed_out"]:
        return (
            f"program did not finish within "
            f"{TIMEOUT_SECONDS} seconds"
        )

    if result["returncode"] != 0:
        details = result["stderr"].strip()

        if details:
```

```
            return (
                f"program exited with code "
                f"{result['returncode']}: {details}"
            )

        return (
            f"program exited with code "
            f"{result['returncode']}"
        )

    return None


def results_from(output):
    return re.findall(
        r"Result: (positive|negative|zero)",
        output,
    )


def test_interface():
    result = run_program("7\n")
    problem = execution_problem(result)

    if problem:
        return False, problem

    expected_prefix = (
        f"{TITLE}\n"
        f"{PROMPT}"
    )

    if not result["stdout"].startswith(
        expected_prefix
    ):
        return False, (
            "title or input prompt is incorrect"
        )

    return True, ""


def test_classification(value, expected):
    result = run_program(f"{value}\n")
    problem = execution_problem(result)

    if problem:
        return False, problem

    actual = results_from(result["stdout"])

    if actual != [expected]:
        return False, (
            f"expected {expected!r}, "
            f"got {actual}"
        )
```

```
    return True, ""


def test_positive():
    return test_classification(
        7,
        "positive",
    )


def test_negative():
    return test_classification(
        -4,
        "negative",
    )


def test_zero():
    return test_classification(
        0,
        "zero",
    )


TESTS = [
    ("Program interface", test_interface),
    ("Positive number", test_positive),
    ("Negative number", test_negative),
    ("Zero", test_zero),
]


def main():
    test_results = []

    for name, test_function in TESTS:
        try:
            passed, details = test_function()

        except Exception as exc:
            passed = False
            details = (
                f"{type(exc).__name__}: {exc}"
            )

        result = {
            "test-name": name,
            "passed": passed,
            "score": 1 if passed else 0,
            "max-score": 1,
        }

        if details:
            result["details"] = details

        test_results.append(result)
```

```
    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(
            test["score"]
            for test in test_results
        ),
        "max-score": sum(
            test["max-score"]
            for test 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())
```

Why the Autograder Exits Successfully Even When Tests Fail

Notice that the script ends with:

```
return 0
```

even when the student fails tests.

That is intentional.

There are two different ideas involved:

The student's program failed a test

and:

The autograder itself failed to execute

Those are not the same thing.

A student's incorrect classification should be represented inside result.json:

```
{
  "test-name": "Zero",
  "passed": false,
  "score": 0,
  "max-score": 1
}
```

The grader itself still successfully completed its job.

A nonzero process exit should generally indicate that the **grading infrastructure** failed, not simply that the student's program was incorrect.

Understanding result.json

Classroom 50 needs an overall result plus the individual tests.

Conceptually, our four-test result might look like:

```
{
  "score": 2,
  "max-score": 4,
  "tests": [
    {
      "test-name": "Program interface",
      "passed": true,
      "score": 1,
      "max-score": 1
    },
    {
      "test-name": "Positive number",
      "passed": true,
      "score": 1,
      "max-score": 1
    },
    {
      "test-name": "Negative number",
      "passed": false,
      "score": 0,
      "max-score": 1
    },
    {
      "test-name": "Zero",
      "passed": false,
      "score": 0,
      "max-score": 1
    }
  ]
}
```

The actual grader also includes Classroom 50 metadata supplied by the runner.

The individual tests become the student's visible feedback.

The score therefore represents:

```
tests passed
```

It does not necessarily have to represent the student's complete assignment grade.

An instructor may still separately evaluate:

- source-code quality;
- comments;
- documentation;
- required programming techniques;
- Git history;
- testing discipline;
- written reflections;
- design decisions; or
- other course objectives.

Install the Custom Autograder

Locate the instructor's Classroom 50 configuration repository.

A Classroom 50 classroom is represented as a directory in the organization's classroom50 repository. The classroom can contain assignment-specific autograder directories under autograders/<slug>/.

For example:

```
my-course/
└── autograders/
    └── number-analyzer/
        └── autograder.py
```

Put the Classroom 50 version of the grader into that directory and commit it.

The final filename must be:

```
autograder.py
```

Do **not** place this file into the student starter repository merely to make autograding work. Classroom 50's current architecture intentionally keeps the instructor's grader in the Classroom 50 configuration repository and has the student-side workflow retrieve it.

This also has an important operational benefit: updates to the grader can take effect without individually modifying previously created student repositories.

Classroom 50 Assignment Configuration

In the workflow used in this guide, the assignment should use Classroom 50's built-in autograding infrastructure.

The custom autograder.py is an override executed by that infrastructure; it is **not** the same thing as replacing Classroom 50 with your own GitHub Actions workflow.

For a fully custom grader like this example, the assignment does not need separate declarative test cases defined in the Classroom 50 interface. The test definitions live in autograder.py.

Because Classroom 50 is evolving rapidly, confirm the current web-interface wording against the current version if the options change.

Test the Real Classroom 50 Workflow

Do not stop after installing the grader.

Use a student or test account to accept the assignment and test the actual end-to-end workflow.

Start with the partial program:

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


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

Commit and push.

The expected Classroom 50 result should be:

```
1/4
```

with only:

```
Program interface
```

passing.

Then implement positive-number handling and push again.

Expected:

```
2/4
```

Finally submit the complete reference solution.

Expected:

```
4/4
```

At this point, the system has been tested at three different levels:

```
Reference solution
        ↓
Local test script
        ↓
Classroom 50 autograder
        ↓
Real student repository
```

Each layer confirms the next one.

Why Local Testing Before Classroom 50 Matters

Debugging three things simultaneously is difficult:

- student program behavior;
- test logic; and
- Classroom 50 integration.

The standalone test script separates those concerns.

Before Classroom 50 is involved, you already know:

- the reference solution is correct;
- the test cases behave as expected;
- partial implementations receive partial success;
- incorrect implementations are rejected; and
- timeouts and crashes are handled.

When the test logic is later moved into Classroom 50, most remaining problems are integration problems rather than testing-design problems.

This dramatically reduces the debugging surface.

Do Not Make Every Test Depend on the Finished Program

Consider this poor test:

```
expected = """
Number Analyzer
Enter an integer: Result: positive
"""

assert output == expected
```

This might be a reasonable **final interface test**, but it is a poor first development test.

It requires:

- title;
- prompt;
- input;
- decision logic;
- result generation;
- formatting; and
- complete program execution.

A student who correctly completed four of those six things receives no evidence of success.

Instead, divide the assignment into meaningful observable capabilities.

For a more complicated assignment, this could look like:

```
PASS Title
PASS Input
PASS Input validation
PASS Repetition
FAIL Classification
FAIL Accumulation
FAIL Summary
FAIL Boundary conditions
```

That feedback tells the student something about the state of the software.

Test One Idea at a Time

Tests should be narrow enough that their names mean something.

A test called:

```
Basic functionality
```

is not very informative.

A test called:

```
Zero classification
```

is.

When that test fails, both the instructor and student know what behavior should be investigated.

This becomes increasingly important as assignments grow.

A 15-test suite should not merely be one giant correctness test repeated with 15 sets of input.

Each test should have a reason to exist.

Test Boundaries, Not Just Typical Values

Number Analyzer is too small to contain many interesting numeric boundaries, but even here we deliberately test:

```
positive
negative
zero
```

rather than only:

```
7
```

For larger assignments, boundaries are often the most valuable tests.

If a specification says:

```
90 or greater
```

then useful cases include:

```
89.99
90
90.01
```

If a condition says:

```
more than half
```

then test:

```
less than half
exactly half
more than half
```

If a condition is:

```
A or B
```

try to construct one test where only A is true and another where only B is true.

Good testing tries to distinguish similar but incorrect implementations.

Give Every Test a Purpose

Before writing a test, be able to answer:

> *What specific incorrect program is this test intended to detect?*

For example:

Positive number

Detects programs that do not recognize values greater than zero.

Negative number

Detects programs that handle positive input but omit or incorrectly implement negative input.

Zero

Detects common mistakes such as:

```
if number >= 0:
```

when zero is supposed to have its own classification.

Program interface

Detects programs that do not conform to the required external interface.

If two tests are intended to detect exactly the same mistakes, one may be redundant.

Use Timeouts

Student programs can accidentally contain infinite loops.

Without a timeout, an autograder can hang indefinitely.

This guide uses:

```
TIMEOUT_SECONDS = 3
```

and:

```
subprocess.run(
    ...,
    timeout=TIMEOUT_SECONDS,
)
```

The appropriate timeout depends on the assignment.

A tiny command-line program should finish almost immediately.

A large data-processing or compiled assignment may reasonably need longer.

The goal is not to make execution artificially difficult. It is to keep an erroneous program from preventing the grading process from completing.

Run Student Programs as Separate Processes

The examples in this guide use:

```
subprocess.run(...)
```

rather than importing the student's Python module directly.

For simple console assignments, this has several advantages.

It lets the grader:

- simulate stdin;
- capture stdout;
- capture stderr;
- detect the program's exit status;
- impose a timeout; and
- evaluate the program approximately as a user would run it.

This is especially useful when the assignment contract is based on command-line behavior.

Other assignments may be better tested by importing functions, using pytest, compiling programs, or using language-specific test frameworks.

The principle remains the same:

> *Choose a testing mechanism that matches the assignment's actual interface.*

Do Not Overfit the Tests to the Reference Solution

The reference solution is a known-correct implementation.

It is **not** necessarily the only correct implementation.

For example, this:

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

and another logically equivalent implementation might both satisfy the assignment.

The autograder should not reject a student's program merely because the student's internal organization differs from the instructor's.

Test the contract.

Do not accidentally test:

> *Did the student reproduce my solution?*

unless reproducing a particular structure is itself an explicit learning objective.

Keep the Autograder Out of the Student's Way

The student's repository should primarily contain the assignment and the student's work.

The instructor's hidden or instructor-controlled test harness does not need to become another file the student edits.

Classroom 50's architecture supports this separation: the instructor-controlled grader resides in the Classroom 50 configuration repository, while student repositories contain the infrastructure necessary to invoke it.

That is both cleaner and easier to maintain.

Updating an Autograder

One advantage of Classroom 50's current architecture is that the autograder is not permanently frozen inside every generated student repository.

If an instructor discovers:

- a bad expected value;
- an incorrect boundary;
- a test that is too strict;
- a typo;
- or a missing case,

the grader in the instructor's configuration repository can be corrected.

The student-side workflow retrieves the current grading harness rather than requiring the instructor to manually patch every student repository.

Of course, changing tests after students have begun an assignment should still be done carefully. Test changes can affect previously reported results.

Scaling This Pattern to Larger Assignments

The Number Analyzer example has only four tests.

A larger assignment might naturally grow to:

```
Program interface
Basic input
Invalid input recovery
Repetition
Basic decision logic
Boundary 1
Boundary 2
Counter behavior
Accumulator behavior
Summary calculation
Special case A
Special case B
Final interface
```

The important idea is not the number of tests.

The important idea is the structure.

Early tests should represent early capabilities.

Later tests should represent increasingly complete behavior.

A student who develops the program incrementally should see the test results evolve with the software.

For example:

```
Commit 1:  2/13
Commit 2:  4/13
Commit 3:  6/13
Commit 4:  8/13
Commit 5: 11/13
Commit 6: 13/13
```

The exact progression will vary by student, and it does not need to be perfectly linear.

What matters is that the autograder provides **useful feedback during construction**, not only after construction.

A Recommended Workflow for Every New Assignment

For future assignments, use the following process:

1. Write the behavioral specification.
2. Make all automatically tested output sufficiently precise.
3. Write the instructor reference solution.
4. Identify independently meaningful capabilities.
5. Order them roughly according to likely development progression.
6. Write small tests for those capabilities.
7. Avoid making early tests depend unnecessarily on later features.
8. Run the tests against the reference solution.
9. Run them against partial solutions.
10. Deliberately introduce bugs and confirm the appropriate tests fail.
11. Only then convert the suite into autograder.py.
12. Install it under the exact Classroom 50 assignment slug.
13. Test an actual student repository with partial work.
14. Test the complete solution.
15. Review what the autograder does **not** assess and retain those items for instructor evaluation.

Final Principles

A strong instructional autograder is not simply a machine that says:

```
correct
```

or:

```
incorrect
```

It is part of the development environment.

The most important principles are:

Define the observable contract first.
Testing cannot repair an ambiguous specification.

Build a reference solution before building the grader.
Know what correct behavior looks like.

Test incrementally.
Students should receive evidence of legitimate progress.

Make tests narrow and diagnostic.
A failed test should communicate something useful.

Make early tests monotonic where practical.
Adding correct functionality should not destroy previously earned evidence of correctness.

Test the tests.
Use partial solutions and intentionally incorrect solutions.

Separate behavior from design quality.
A passing autograder does not prove good architecture, good style, understanding, documentation, or appropriate development practice.

Develop locally before integrating with Classroom 50.
Validate the test logic before adding platform complexity.

Verify the real student workflow.
A test suite is not finished until it has run successfully through the same mechanism the students will use.

The result is an autograder that does more than assign points.

It becomes a source of continuous, structured feedback while the student is building software.

Classroom 50 References

- Classroom 50 project and announcement: https://github.com/foundation50/classroom50/discussions/46
- Classroom 50 teacher guide: https://github.com/gh-cli-for-education/hello-c50/blob/main/teacher-guide.md
- Classroom 50 autograder architecture discussion: https://github.com/foundation50/classroom50/discussions/206
- Classroom 50 repository: https://github.com/foundation50/classroom50