Test Case Writing Guide for Faculty

This guide explains how to write test cases for the AutoGrader system. Test cases are used to automatically evaluate student submissions and assign grades.

Table of Contents

How Test Cases Work

The AutoGrader system combines your test cases with student code to create a complete program that runs and evaluates the submission. Each test case:

- Is executed independently
- Has a point value you assign
- Can be marked as visible (students see results) or hidden (only you see results)
- Automatically reports pass/fail status

When a student submits code, the system: 1. Takes the student's code 2. Combines it with your test cases 3. Executes the combined program 4. Reports which tests passed or failed 5. Calculates the grade based on point values

---

Supported Languages

The system currently supports the following programming languages:

- **Python** - Uses Python assertions (`assert`)
- **Java** - Uses Java assertions (`assert`) with Solution class
- **C++** (also accepts `cpp` or `gcc`) - Uses C++ assertions (`assert()`)
- JavaScript, C, C#, Go, Rust, Ruby, PHP - Generic template support

---

Python Test Cases

Format

Python test cases are written as simple assertion statements. The system automatically wraps each test case in a try-except block, so if an assertion fails, the test is marked as failed without crashing the entire test suite.

Basic Structure

Each test case should contain Python code with `assert` statements:

```
assert add(2, 3) == 5
assert add(10, 20) == 30
```

Examples

Example 1: Simple Function Test

```
assert calculate_area(5, 10) == 50
```

Example 2: Multiple Assertions

```
assert add(2, 3) == 5
assert add(-5, -3) == -8
assert add(0, 0) == 0
```

Example 3: Testing with Lists

```
result = reverse_list([1, 2, 3, 4])
assert result == [4, 3, 2, 1]
assert len(result) == 4
```

Example 4: Testing String Operations

```
assert uppercase("hello") == "HELLO"
assert uppercase("WORLD") == "WORLD"
assert uppercase("") == ""
```

Example 5: Testing Error Conditions

```
try:
    divide(10, 0)
    assert False, "Should have raised ZeroDivisionError"
except ZeroDivisionError:
```

```
    assert True
```

Important Notes for Python

- ■ Use standard Python `assert` statements
- ■ Multiple assertions in one test case are allowed
- ■ Tests are automatically wrapped in try-except blocks
- ■ If any assertion fails, the entire test case fails
- ■ You can use any Python code (variables, loops, conditionals) in test cases

---

Java Test Cases

Format

Java test cases use Java `assert` statements. The system automatically converts these to proper exception-throwing code. **Important:** Student code must be written in a class called `Solution`.

Basic Structure

You must create an instance of the `Solution` class and test its methods:

```
Solution s = new Solution();
assert s.add(2, 3) == 5;
```

Examples

Example 1: Simple Method Test

```
Solution s = new Solution();
assert s.calculateArea(5, 10) == 50;
```

Example 2: Testing Multiple Methods

```
Solution s = new Solution();
assert s.add(2, 3) == 5;
assert s.subtract(10, 3) == 7;
assert s.multiply(4, 5) == 20;
```

Example 3: Testing with Arrays

```
Solution s = new Solution();
int[] input = {1, 2, 3, 4};
int[] result = s.reverseArray(input);
assert result.length == 4;
assert result[0] == 4;
assert result[3] == 1;
```

Example 4: Testing String Methods

```
Solution s = new Solution();
assert s.uppercase("hello").equals("HELLO");
assert s.uppercase("WORLD").equals("WORLD");
```

Example 5: Testing Boolean Returns

```
Solution s = new Solution();
assert s.isEven(4) == true;
assert s.isEven(5) == false;
assert s.isEven(0) == true;
```

Important Notes for Java

- ■ Student code must be in a class named `Solution`
- ■ Always create a `Solution` instance: `Solution s = new Solution();`
- ■ Use Java `assert` statements (they're automatically converted)
- ■ Use `.equals()` for string comparisons, not `==`
- ■ Tests are automatically wrapped in try-catch blocks

C++ Test Cases

Format

C++ test cases use `assert()` statements. The system automatically converts these to `test_assert()` macros that throw exceptions instead of aborting, allowing proper error handling.

Basic Structure

Use standard C++ `assert()` statements to test functions:

```
assert(add(2, 3) == 5);
assert(add(10, 20) == 30);
```

Examples

Example 1: Simple Function Test

```
assert(calculateArea(5, 10) == 50);
```

Example 2: Multiple Assertions

```
assert(add(2, 3) == 5);
assert(add(-5, -3) == -8);
assert(add(0, 0) == 0);
```

Example 3: Testing with Vectors

```
std::vector<int> input = {1, 2, 3, 4};
std::vector<int> result = reverseVector(input);
assert(result.size() == 4);
assert(result[0] == 4);
assert(result[3] == 1);
```

Example 4: Testing String Operations

```
assert(uppercase("hello") == "HELLO");
assert(uppercase("WORLD") == "WORLD");
assert(uppercase("") == "");
```

Example 5: Testing Float Comparisons

```
#include <cmath>
double result = divide(10.0, 3.0);
assert(std::abs(result - 3.333) < 0.001);  // Use tolerance for floats
```

Important Notes for C++

- ■ Use standard C++ `assert()` statements
- ■ The system converts `assert()` to `test_assert()` automatically
- ■ Tests are automatically wrapped in try-catch blocks
- ■ Use `std::abs()` and tolerance for floating-point comparisons
- ■ Include necessary headers if needed (they're available)

Best Practices

1. Write Clear, Specific Test Cases

- ■ Test one concept or behavior per test case
- ■ Use descriptive variable names
- ■ Include multiple assertions when testing related behaviors

Good:

```
assert add(2, 3) == 5
assert add(-5, -3) == -8
assert add(0, 0) == 0
```

Less Clear:

```
assert x == y  # What are x and y?
```

2. Set Appropriate Point Values

- ■ Distribute points based on complexity and importance
- ■ Common distribution: 10-20 points per test case
- ■ Reserve higher point values for critical functionality

Example: - Basic functionality: 10 points each - Edge cases: 5 points each - Complex algorithms: 20-30 points each

3. Test Edge Cases

Always include tests for: - ■ Empty inputs (`""`, `[]`, `null`, etc.) - ■ Zero values - ■ Negative numbers (when applicable) - ■ Boundary conditions - ■ Error conditions (division by zero, null pointers, etc.)

4. Use Visibility Settings

- **Visible Tests**: Students see results immediately after submission
- Use for: Basic functionality, learning objectives

  Helps students understand what's expected

  Hidden Tests: Only visible to faculty

- Use for: Edge cases, advanced scenarios, preventing hardcoding
- Prevents students from gaming the system

5. Test Incrementally

- ■ Start with simple, basic tests
- ■ Progress to more complex scenarios

- ■ End with edge cases and error conditions

Common Pitfalls

Python Pitfalls

■ Don't use function decorators

```
# This won't work in our system
@points(10)
def test_add():
    assert add(2, 3) == 5
```

■ Instead, just write assertions

```
assert add(2, 3) == 5
```

■ Don't define test functions

```
# This won't work
def test_add():
    assert add(2, 3) == 5
```

■ Write assertions directly

```
assert add(2, 3) == 5
```

Java Pitfalls

■ Don't forget to create Solution instance

```
// This won't work - no Solution instance
assert add(2, 3) == 5;
```

■ Always create Solution instance first

```
Solution s = new Solution();
assert s.add(2, 3) == 5;
```

■ Don't use == for string comparison

```
// Wrong for strings
assert s.uppercase("hello") == "HELLO";
```

■ **Use .equals() for strings**

```
assert s.uppercase("hello").equals("HELLO");
```

C++ Pitfalls

■ **Don't use == for floating-point comparison**

```
// Wrong - floating point imprecision
assert(divide(10.0, 3.0) == 3.333);
```

■ **Use tolerance for floating-point**

```
#include <cmath>
double result = divide(10.0, 3.0);
assert(std::abs(result - 3.333) < 0.001);
```

■ **Don't forget to handle exceptions**

```
// This might abort if function throws
assert(riskyFunction() == expected);
```

■ **The system handles this automatically** - your `assert()` statements are converted to exception-safe `test_assert()`

General Pitfalls

■ **Don't hardcode test data that students can see**

```
# If this test is visible, students might hardcode for this input
assert calculateGrade(85, 90, 88) == 87.67
```

■ **Use hidden tests for specific inputs** or vary the test data

■ **Don't test implementation details**

```
# Don't test internal variables
assert student_code.internal_counter == 5
```

■ **Test behavior and results**

```
result = student_code.process()
assert result == expected_output
```

Examples Reference

For complete working examples, see: - `manual_test_files/calculator_demo/tests/calculator_basic.py` - Python test examples - Create assignments and experiment with the test case interface to see how your tests execute

---

Getting Help

If you encounter issues writing test cases:

1. Test your test cases using the "Test Route" feature in the system
2. Check that student code matches expected function/class names
3. Verify point values are set correctly
4. Ensure test visibility settings match your intentions

Remember: Test cases should validate that student code produces the correct output, not how they implemented it internally.