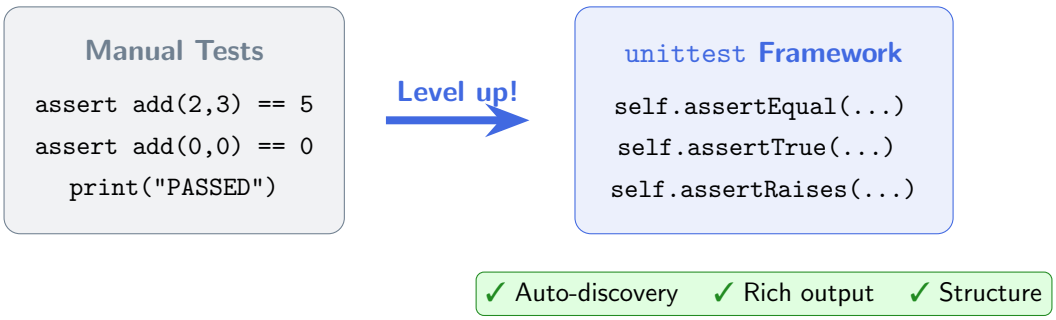
Supplementary Material

Python *unittest*

A Handout & Walkthrough Tutorial



Comp 111 — Forman Christian College

Spring 2026

Estimated Time: ∼**1.5 hours**

How to Use This Handout

This handout is a **supplementary reference**—it extends the testing skills you learned in Lab 02 (Section 6). If you are comfortable writing test functions using `assert`, you are ready for this.

- **Read** the short explanations and compare the "before and after" examples.
- **Type** every code listing yourself. Muscle memory matters.
- **Run** every example and check that your output matches.
- **Complete** the exercises, graded by difficulty:
  - ★ Easy — Immediate practice. Follow the pattern.
  - ★★ Medium — Requires some thinking. Combines concepts.
  - ★★★ Hard — Stretch goal. It's OK to need help!

Before You Begin

This handout assumes you completed **Lab 02, Sections 6–9**. You should be comfortable writing functions, writing test functions with `assert`, and running Python files from the terminal.

1 From assert to unittest

In Lab 02 you wrote tests like this:

```python
# Manual test approach (Lab 02)
from math_tools import add

def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0
    assert add(0, 0) == 0
    print("test_add: ALL PASSED")

test_add()
```

This works, but it has limitations:

- You have to **call each test function yourself** at the bottom of the file.
- When a test fails, `assert` only says `AssertionError`—it doesn't tell you what the actual value was.
- There is no summary telling you "5 tests ran, 4 passed, 1 failed."
- If one test fails, the rest don't run at all.

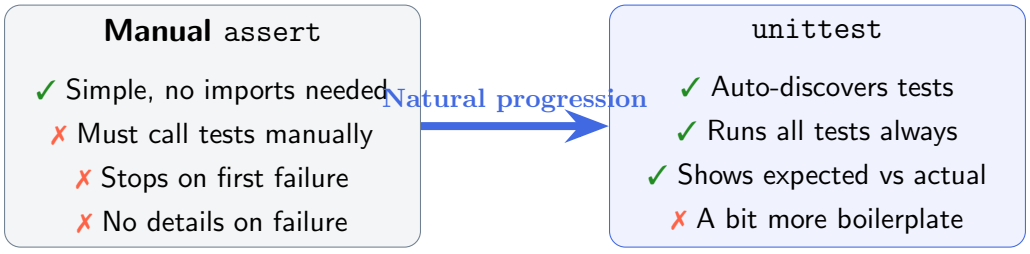
Python's built-in `unittest` module solves all of these problems.

Why Use a Test Framework?

A test framework gives you:

1. **Auto-discovery**—it finds and runs all your tests automatically.
2. **Rich assertions**—instead of just pass/fail, it shows what went wrong.
3. **Independence**—one failing test doesn't stop the others from running.
4. **Structured reports**—a clear summary of results.

You already understand the *concept* of testing. `unittest` gives you better *tools*.



2 Writing Your First unittest Test

2.1 The Anatomy of a unittest Test

Every unittest test file follows the same structure:

```
import unittest                          # 1. Import the module

class TestAdd(unittest.TestCase):        # 2. Create a test class
    def test_positive(self):             # 3. Write test methods
        self.assertEqual(add(2, 3), 5)

if __name__ == "__main__":               # 4. Run when executed
    unittest.main()
```

Let's break that down:

Part	What it does
`import unittest`	Loads Python's built-in testing framework
`class TestAdd(unittest.TestCase)`	A test class—groups related tests together. Must inherit from `unittest.TestCase`
`def test_positive(self)`	A single test. The method name **must** start with `test_`
`self.assertEqual(a, b)`	Checks that `a == b`. Replaces `assert a == b`
`unittest.main()`	Discovers and runs all test methods in the file

Two Rules to Remember

1. Your test class must inherit from `unittest.TestCase`.
2. Every test method must start with `test_`. Methods without this prefix are **ignored**.

2.2 Walkthrough: Converting Lab 02 Tests

Remember `math_tools.py` from Lab 02? Let's convert its tests to use `unittest`. First, make sure you have `math_tools.py`:

```
# math_tools.py (same as Lab 02)

def add(a, b):
    return a + b

def multiply(a, b):
    return a * b

def is_even(n):
    return n % 2 == 0
```

Now create `test_math_tools_v2.py` in the same folder:

```
# test_math_tools_v2.py
```

```python
import unittest
from math_tools import add, multiply, is_even

class TestAdd(unittest.TestCase):
    def test_positive_numbers(self):
        self.assertEqual(add(2, 3), 5)

    def test_negative_numbers(self):
        self.assertEqual(add(-1, 1), 0)

    def test_zeros(self):
        self.assertEqual(add(0, 0), 0)

class TestMultiply(unittest.TestCase):
    def test_positive(self):
        self.assertEqual(multiply(3, 4), 12)

    def test_negative(self):
        self.assertEqual(multiply(-2, 5), -10)

    def test_by_zero(self):
        self.assertEqual(multiply(0, 100), 0)

class TestIsEven(unittest.TestCase):
    def test_even(self):
        self.assertTrue(is_even(4))

    def test_odd(self):
        self.assertFalse(is_even(7))

    def test_zero(self):
        self.assertTrue(is_even(0))

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

Run it from the terminal:

```
python -m unittest test_math_tools_v2 -v
```

```
test_negative_numbers (test_math_tools_v2.TestAdd) ... ok
test_positive_numbers (test_math_tools_v2.TestAdd) ... ok
test_zeros (test_math_tools_v2.TestAdd) ... ok
test_negative (test_math_tools_v2.TestMultiply) ... ok
test_positive (test_math_tools_v2.TestMultiply) ... ok
test_by_zero (test_math_tools_v2.TestMultiply) ... ok
test_even (test_math_tools_v2.TestIsEven) ... ok
test_odd (test_math_tools_v2.TestIsEven) ... ok
test_zero (test_math_tools_v2.TestIsEven) ... ok

----------------------------------------------------------------------
```

```
Ran 9 tests in 0.001s

OK
```

The -v Flag

The `-v` flag stands for **verbose**. It shows each test by name instead of just dots. Without `-v`, you'd see `.........` (one dot per passing test) followed by `OK`.

2.3 What Happens When a Test Fails?

Let's see the difference. Temporarily change one test to something wrong:

```python
def test_positive_numbers(self):
    self.assertEqual(add(2, 3), 999)    # wrong on purpose
```

Run again:

```
python -m unittest test_math_tools_v2 -v
```

```
test_positive_numbers (test_math_tools_v2.TestAdd) ... FAIL
test_negative_numbers (test_math_tools_v2.TestAdd) ... ok
test_zeros (test_math_tools_v2.TestAdd) ... ok
...

======================================================================

FAIL: test_positive_numbers (test_math_tools_v2.TestAdd)
----------------------------------------------------------------------

Traceback (most recent call last):
  File "test_math_tools_v2.py", line 7, in test_positive_numbers
    self.assertEqual(add(2, 3), 999)
AssertionError: 5 != 999

----------------------------------------------------------------------

Ran 9 tests in 0.001s

FAILED (failures=1)
```

Notice two things:

1. The failure message says `5 != 999`—it tells you the **actual** value (5) and the **expected** value (999). With plain `assert`, you'd only get `AssertionError` with no details.
2. **All 9 tests ran.** The other 8 still passed. With manual `assert`, the program would have crashed on the first failure.

Remember to change the test back to the correct value before continuing!

2.4 Common Assert Methods

`unittest.TestCase` provides many assertion methods. Here are the most useful:

Method	Checks that...
`self.assertEqual(a, b)`	`a == b`
`self.assertNotEqual(a, b)`	`a != b`
`self.assertTrue(x)`	`x` is `True`
`self.assertFalse(x)`	`x` is `False`
`self.assertIn(a, b)`	`a in b` (e.g. item in list, char in string)
`self.assertIsNone(x)`	`x is None`
`self.assertAlmostEqual(a, b)`	`a` ≈ `b` (for floating-point comparison)
`self.assertRaises(Error)`	A specific exception is raised (see Section 3)

assertEqual vs assertTrue

Prefer `self.assertEqual(add(2,3), 5)` over `self.assertTrue(add(2,3) == 5)`. Why? When `assertEqual` fails, it shows you both values (`5 != 999`). When `assertTrue` fails, it only says `False is not true`—much less helpful!

2.5 Exercises: First `unittest` Tests

1. ★ Easy Add a function `subtract(a, b)` to `math_tools.py`. Write a new test class `TestSubtract` in `test_math_tools_v2.py` with at least three test methods. Run with `-v` and verify all pass.

2. ★ Easy Run your tests *without* the `-v` flag: `python -m unittest test_math_tools_v2`. What does the output look like? Count the dots—does the count match the number of tests?

3. ★★ Medium Write a function `contains_vowel(s)` that returns `True` if the string contains at least one vowel. Write a `TestContainsVowel` class with tests for: a word with vowels, a word with no vowels (e.g., `"rhythm"`), an empty string, and a single vowel character. Use both `assertTrue` and `assertFalse`.

4. ★★ Medium Write a function `safe_divide(a, b)` that returns `a / b`, or `None` if `b` is zero. Write tests that use `assertEqual` for normal division and `assertIsNone` for the zero case.

3 Test Organization: setUp, tearDown, and assertRaises

3.1 Shared Setup with setUp

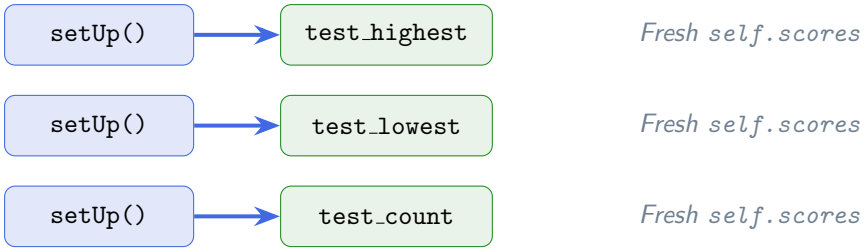
Sometimes several tests need the same starting data. Instead of repeating it in every test method, you can use `setUp`:

```
class TestGrades(unittest.TestCase):
    def setUp(self):
        """Runs before EACH test method."""
        self.scores = [90, 85, 70, 60, 95]

    def test_highest(self):
        self.assertEqual(max(self.scores), 95)

    def test_lowest(self):
        self.assertEqual(min(self.scores), 60)

    def test_count(self):
        self.assertEqual(len(self.scores), 5)
```



setUp Runs Before Each Test

`setUp()` is called before **every single test method**, not just once for the whole class. This means each test starts with a clean, fresh copy of the data. One test cannot accidentally affect another.

There is also `tearDown()`, which runs *after* each test. It is used for cleanup (closing files, resetting state). You won't need it often at this stage, but it's good to know it exists.

3.2 Walkthrough: Grade Calculator with setUp

Let's rewrite the grade calculator tests from Lab 02 Section 9 using `unittest` and `setUp`. Make sure you have the **fixed** version of `grades.py`:

```
# grades.py (fixed version from Lab 02)

def letter_grade(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
```

```
        return "C"
    elif score >= 60:
        return "D"
    else:
        return "F"

def class_average(scores):
    total = 0
    for score in scores:
        total += score
    return total / len(scores)

def highest_score(scores):
    best = scores[0]
    for score in scores:
        if score > best:
            best = score
    return best
```

Now create test_grades_v2.py:

```
# test_grades_v2.py
import unittest
from grades import letter_grade, class_average, highest_score

class TestLetterGrade(unittest.TestCase):
    def test_a_grade(self):
        self.assertEqual(letter_grade(95), "A")
        self.assertEqual(letter_grade(90), "A")

    def test_b_grade(self):
        self.assertEqual(letter_grade(85), "B")
        self.assertEqual(letter_grade(80), "B")

    def test_f_grade(self):
        self.assertEqual(letter_grade(50), "F")
        self.assertEqual(letter_grade(0), "F")

    def test_boundary_values(self):
        self.assertEqual(letter_grade(90), "A")
        self.assertEqual(letter_grade(89), "B")
        self.assertEqual(letter_grade(80), "B")
        self.assertEqual(letter_grade(79), "C")

class TestClassAverage(unittest.TestCase):
    def setUp(self):
        self.mixed_scores = [100, 80, 60]
        self.uniform_scores = [90, 90, 90]

    def test_mixed(self):
        self.assertEqual(class_average(self.mixed_scores),
            80.0)
```

```
31
32      def test_uniform(self):
33          self.assertEqual(class_average(self.uniform_scores),
                90.0)
34
35      def test_single_score(self):
36          self.assertEqual(class_average([50]), 50.0)
37
38  class TestHighestScore(unittest.TestCase):
39      def test_normal(self):
40          self.assertEqual(highest_score([70, 85, 90, 60]), 90)
41
42      def test_single(self):
43          self.assertEqual(highest_score([50]), 50)
44
45      def test_negative_scores(self):
46          self.assertEqual(highest_score([-10, -20, -5]), -5)
47
48  if __name__ == "__main__":
49      unittest.main()
```

Run it:

```
python -m unittest test_grades_v2 -v
```

```
test_a_grade (test_grades_v2.TestLetterGrade) ... ok
test_b_grade (test_grades_v2.TestLetterGrade) ... ok
test_boundary_values (test_grades_v2.TestLetterGrade) ... ok
test_f_grade (test_grades_v2.TestLetterGrade) ... ok
test_mixed (test_grades_v2.TestClassAverage) ... ok
test_single_score (test_grades_v2.TestClassAverage) ... ok
test_uniform (test_grades_v2.TestClassAverage) ... ok
test_negative_scores (test_grades_v2.TestHighestScore) ... ok
test_normal (test_grades_v2.TestHighestScore) ... ok
test_single (test_grades_v2.TestHighestScore) ... ok

----------------------------------------------------------------------

Ran 10 tests in 0.001s

OK
```

3.3 Testing for Exceptions with `assertRaises`

Sometimes you want to verify that your code correctly raises an error for bad input. Use `assertRaises` with a `with` block:

```
1  class TestClassAverageEdgeCases(unittest.TestCase):
2      def test_empty_list_raises_error(self):
3          with self.assertRaises(ZeroDivisionError):
```

```
4                 class_average([])
```

This test **passes** if `class_average([])` raises a `ZeroDivisionError`, and **fails** if no exception is raised.

When to Use `assertRaises`

Use `assertRaises` when you want to test that invalid inputs are handled correctly. For example: dividing by zero, accessing an empty list, or passing the wrong type.

3.4 Exercises: Test Organization

1. ★★ Medium Add a `lowest_score(scores)` function to `grades.py`. Write a `TestLowestScore` class that uses `setUp` to create a shared list of scores. Write at least three test methods.
2. ★★ Medium Write a function `grade_report(scores)` that returns a list of letter grades (one per score). For example: `grade_report([95, 72, 55])` returns `["A", "C", "F"]`. Write a test class using `setUp` and `assertIn` to check specific grades appear in the result.
3. ★★★ Hard Write a function `validate_score(score)` that raises a `ValueError` if the score is negative or greater than 100. Write a test class with:
   - Tests for valid scores (should *not* raise an error)
   - Tests using `assertRaises(ValueError)` for scores like `-5` and `101`

4 Running Tests & Reading Failures

4.1 Ways to Run unittest

There are several ways to run your tests:

Command	What it does
`python -m unittest`	Discover and run *all* test files in the current directory
`python -m unittest test_grades_v2`	Run all tests in a specific module
`python -m unittest test_grades_v2 -v`	Same, but with verbose output
`python -m unittest test_grades_v2.TestLetterGrade`	Run only one test class
`python -m unittest test_grades_v2.TestLetterGrade.test_a_grade`	Run only one test method
`python test_grades_v2.py`	Works because of `unittest.main()` at the bottom

Auto-Discovery

When you run just `python -m unittest` with no arguments, Python automatically finds every file matching `test*.py` in the current directory and runs all the tests inside. This is one of the biggest advantages of using `unittest`—you never have to manually list which tests to run.

4.2 Walkthrough: Reading a Failure

Let's practice reading failure output. Create a file `test_demo_fail.py`:

```
# test_demo_fail.py
import unittest

def double(n):
    return n * 3    # Bug! Should be n * 2

class TestDouble(unittest.TestCase):
    def test_positive(self):
        self.assertEqual(double(5), 10)

    def test_zero(self):
        self.assertEqual(double(0), 0)

    def test_negative(self):
        self.assertEqual(double(-3), -6)

```

```
17 if __name__ == "__main__":
18     unittest.main()
```

Run it:

```
python -m unittest test_demo_fail -v
```

```
test_negative (test_demo_fail.TestDouble) ... FAIL
test_positive (test_demo_fail.TestDouble) ... FAIL
test_zero (test_demo_fail.TestDouble) ... ok

======================================================================

FAIL: test_negative (test_demo_fail.TestDouble)
----------------------------------------------------------------------

Traceback (most recent call last):
  File "test_demo_fail.py", line 14, in test_negative
    self.assertEqual(double(-3), -6)
AssertionError: -9 != -6

======================================================================

FAIL: test_positive (test_demo_fail.TestDouble)
----------------------------------------------------------------------

Traceback (most recent call last):
  File "test_demo_fail.py", line 8, in test_positive
    self.assertEqual(double(5), 10)
AssertionError: 15 != 10

----------------------------------------------------------------------

Ran 3 tests in 0.001s

FAILED (failures=2)
```

Let's read this output step by step:

1. **Summary line:** `test_positive ... FAIL` tells you *which* test failed.
2. **Traceback:** Shows the exact *file*, *line number*, and *code* that failed.
3. **AssertionError:** `15 != 10` — the function returned 15 but you expected 10.
4. **Final summary:** `Ran 3 tests ... FAILED (failures=2)` — 2 out of 3 failed.

From the failure message `15 != 10`, you can immediately see that `double(5)` returned 15 instead of 10. That's $5 \times 3 = 15$—the function is tripling instead of doubling!

⚠ Read the Actual Value First

When you see `AssertionError: X != Y`, the **left** side (`X`) is what your code *actually* returned. The **right** side (`Y`) is what you *expected*. The bug is in your code, not your test (usually!).

4.3 Exercises: Running and Reading

1. ★ Easy Fix the bug in `test_demo_fail.py` (change `n * 3` to `n * 2`). Run the tests again and confirm all pass.

2. ★★ Medium Put `test_math_tools_v2.py` and `test_grades_v2.py` in the same folder. Run `python -m unittest -v` with no file specified. Does it find and run tests from *both* files?

3. ★★★ Hard Write a function `fizzbuzz(n)` that returns `"Fizz"` if `n` is divisible by 3, `"Buzz"` if divisible by 5, `"FizzBuzz"` if divisible by both, and the number as a string otherwise. Write a `TestFizzBuzz` class with at least 6 test methods covering all cases. Deliberately introduce a bug, run the tests, read the failure output, and then fix it.

Quick Reference

unittest Cheat Sheet

Test File Template:

```python
import unittest
from my_module import my_function

class TestMyFunction(unittest.TestCase):
    def setUp(self):
        # Shared setup (optional)
        self.data = [1, 2, 3]

    def test_basic(self):
        self.assertEqual(my_function(2), 4)

    def test_edge_case(self):
        self.assertIsNone(my_function(None))

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

Assert Methods:

Method	Replaces	Better because...
`assertEqual(a, b)`	`assert a == b`	Shows both values on failure
`assertNotEqual(a, b)`	`assert a != b`	Shows both values on failure
`assertTrue(x)`	`assert x`	Clearer intent
`assertFalse(x)`	`assert not x`	Clearer intent
`assertIn(a, b)`	`assert a in b`	Shows what was searched
`assertIsNone(x)`	`assert x is None`	Clearer failure message
`assertAlmostEqual(a, b)`	(no equivalent)	Handles floating-point errors
`assertRaises(Err)`	`try/except + assert`	Clean, readable syntax

Running Tests:

Command	Scope
`python -m unittest`	All test files in the directory
`python -m unittest -v`	All tests, verbose output
`python -m unittest test_file`	One file
`python -m unittest test_file.TestClass`	One class
`python -m unittest test_file.TestClass.test_method`	One method

`unittest` gives you structure and better tools.
`assert` gave you the concept.

You already know *how* to test—now
you know how to test *professionally*.