



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Chapter 03

UNIT TESTING AND CLEAN CODE

Where We Are Now

Your Progress So Far:

- ✅ **Chapter 02 (Code Quality in Practice):** PEP8 compliance with Ruff
- ✅ **Chapter 02 (Feature Branch Development):** Feature branches & Pull Requests
- ✅ **Chapter 02 (Automation and CI/CD):** Automated CI/CD checks
- ✅ **Chapter 02 (Refactoring):** Refactored to modular structure

Your CI is green! Your code is clean! 🎉

But... does your code actually work?

The Shocking Discovery

Your clean, modular code is deployed to production...

```
# A user enters a vertical camera angle
angle = 90.0

# Your code crashes:
def find_intersection(x_road, y_road, angle_degrees, ...):
    angle_rad = -np.deg2rad(angle_degrees)
    slope = np.tan(angle_rad) # tan(90°) = ∞
    # ZeroDivisionError or infinite loop!
```

-  Ruff: "Code is formatted nicely!"
-  Pylint: "Types look good!"
-  Nobody tested if the code actually works

Today's Learning Objectives

By the end of this lecture, you will:

1.  **Understand the difference** between code quality (style) and code correctness (logic)
2.  **Master the testing pyramid** - unit, module, and end-to-end tests
3.  **Write effective unit tests** using the AAA pattern
4.  **Use pytest** to test your modular Python code
5.  **Apply clean code principles** to make tests maintainable
6.  **Understand why exhaustive testing is impossible** (and what to do about it)

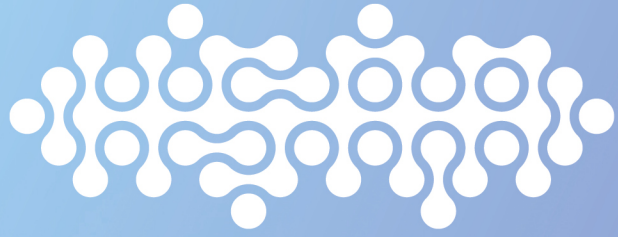
What You WILL and WON'T Learn Today

What you WILL learn:

-  How to catch bugs before users do
-  How to write tests that give you confidence
-  Why the testing pyramid matters
-  Clean code principles for testing

What you WON'T learn (yet):

-  Test-driven development (TDD) - that's Chapter 03 (TDD and CI)
-  Mocking and fixtures - advanced topics for later
-  UI testing - complex and out of scope
-  How to use LLMs for testing - that's Chapter 03 (Boundary Analysis)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Why Testing Matters

1

The Bug That CI Missed

Your `find_intersection()` function:

```
def find_intersection(x_road, y_road, angle_degrees, ...):  
    angle_rad = -np.deg2rad(angle_degrees)  
  
    # Handle vertical ray  
    if np.abs(np.cos(angle_rad)) < 1e-10:  
        return None, None, None  
  
    slope = np.tan(angle_rad)  
    # ... rest of function
```

What Happens With Different Inputs?

Angle	Expected Behavior	Actual Behavior	CI Status
-10°	Find intersection	✓ Works	✓ Green
-45°	Find intersection	✓ Works	✓ Green
0°	Horizontal ray	✓ Works	✓ Green
90°	Vertical ray	✗ Returns None	✓ Green
Empty array	Handle empty data	✗ IndexError	✓ Green

CI is blind to logic bugs!

Interactive Question

Think about your own projects:

Have you ever had a bug make it to production (or a demo) even though your CI was green?

Common scenarios:

- "It worked on my machine!"
- "But I tested it manually..."
- "The types were all correct!"

Let's discuss: What kinds of bugs slip through?

Code Quality vs. Code Correctness

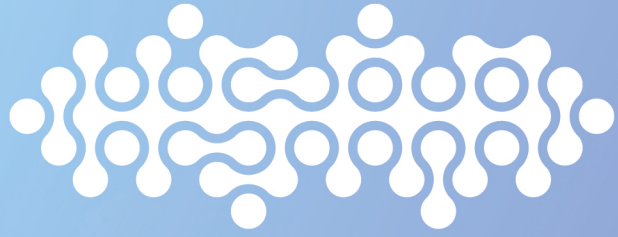
Code Quality Tools → "Does this look professional?"

- ✓ Formatted nicely?
- ✓ Type hints present?
- ✓ No unused imports?

Tests → "Does this actually work?"

- ✓ Correct results?
- ✓ Handles edge cases?
- ✓ Fails gracefully?

Both are essential for professional software development.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

The Testing Pyramid

2

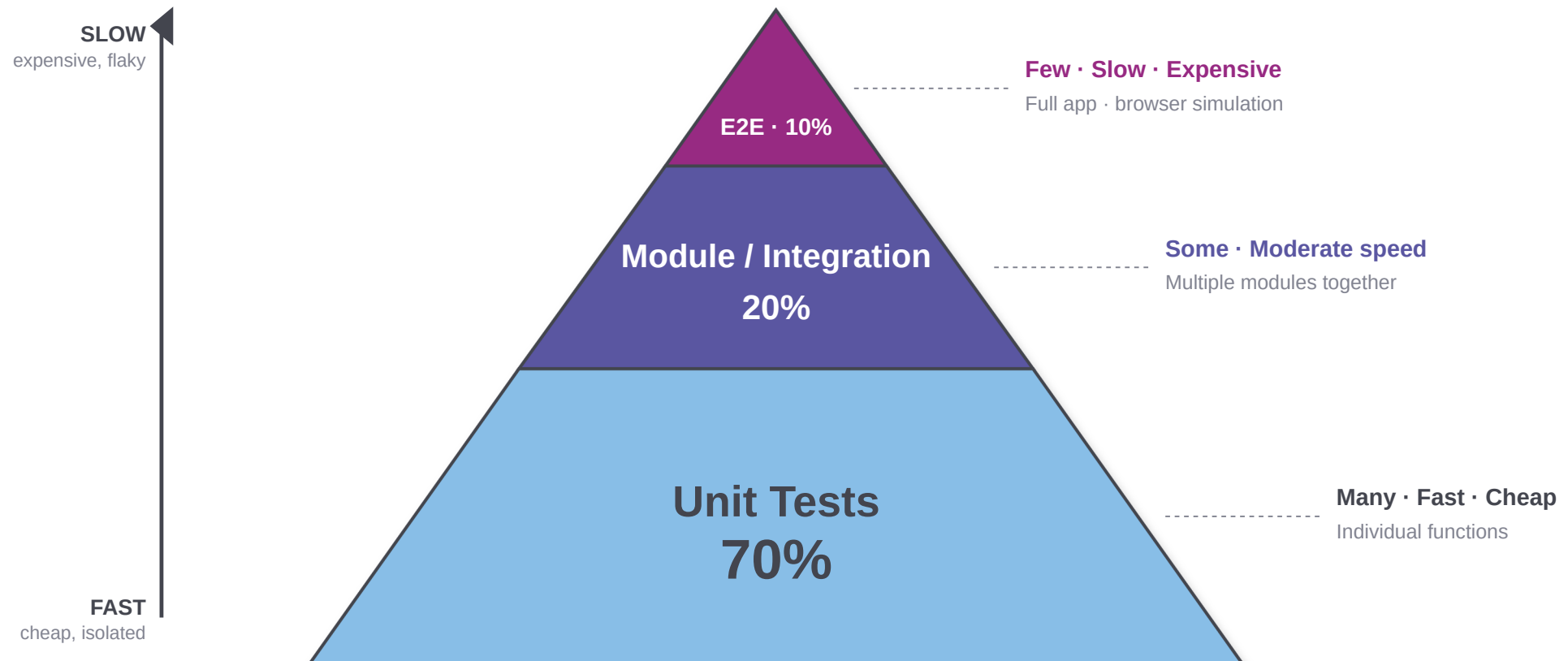
How Do We Test Software Systematically?

Three levels of testing, each with different purposes:

1. **Unit Tests** - Test individual functions
2. **Module/Integration Tests** - Test modules working together
3. **End-to-End Tests** - Test entire application

But how much of each should we write?

What is the Testing Pyramid?



Why the Pyramid Shape?

Recommended ratio:

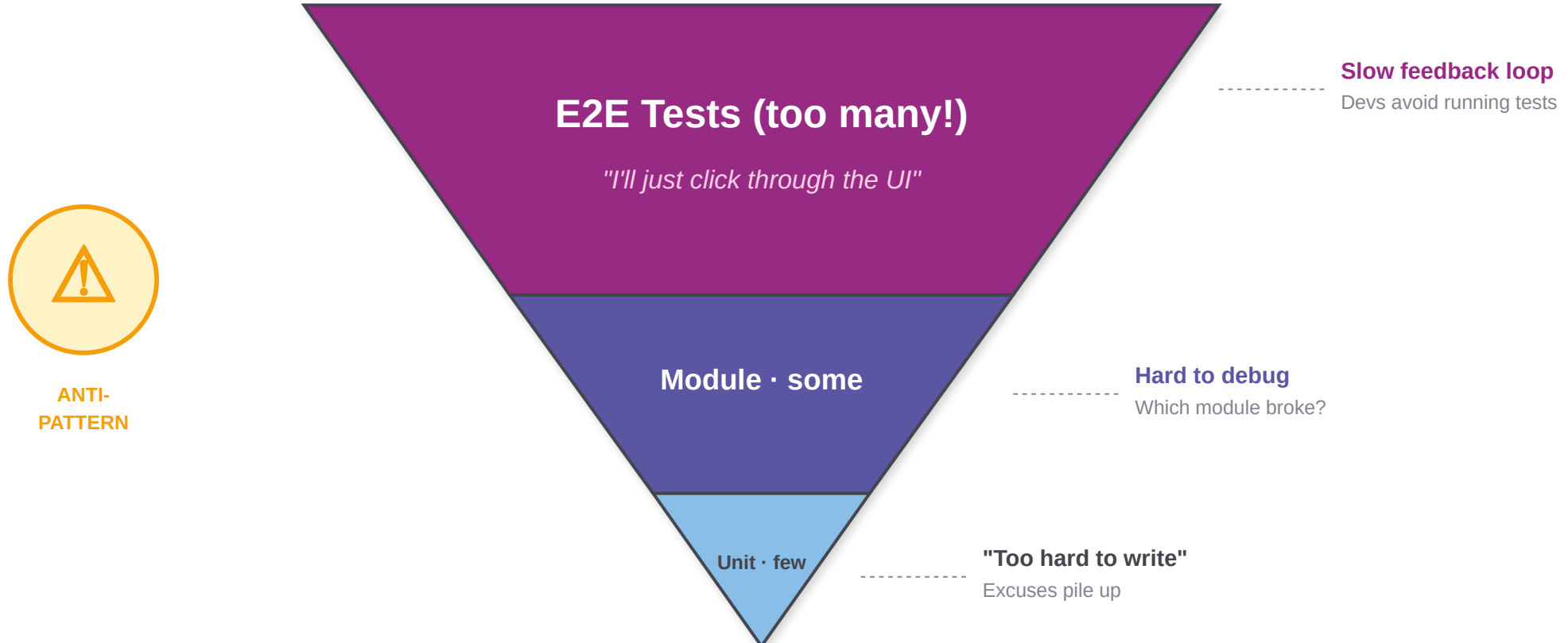
- **70%** Unit tests
- **20%** Module/Integration tests
- **10%** End-to-End tests

Why? Speed matters for the feedback loop:

If you change one line in `find_intersection()`:

-  50 unit tests run in **0.5 seconds** → instant feedback
-  50 E2E tests run in **250 seconds** → you go get coffee, lose flow

The Anti-Pattern: The Test Cone



This is backwards! Same colors, inverted shape — every layer becomes a liability.

Problems With the Test Cone

1. Slow feedback:

- Every change requires running slow E2E tests
- Developers avoid running tests → bugs pile up

2. Hard to debug:

- E2E test fails → which of 10 modules caused it?
- Unit test fails → you know exactly which function is broken

3. Brittle:

- UI changes break all E2E tests
- Unit tests are unaffected by UI changes

4. Vicious cycle:

- Tests are slow → Developers avoid running them → More bugs



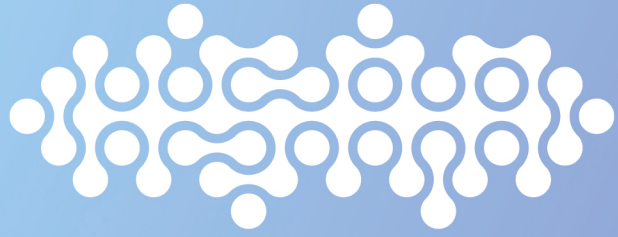
Interactive Discussion

Why do you think developers create test cones?

Common reasons:

- "I don't know how to write unit tests"
- "Writing test setup is tedious"
- "The UI is easy to click through manually"
- "Unit tests feel like extra work"

Today's goal: Make unit testing so easy you'll prefer it!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Unit Testing Deep Dive

3

From WHY to HOW

Now that we know WHY unit tests matter, let's learn HOW to write them.

Key topics:

1. What is a unit test?
2. The AAA pattern (Arrange-Act-Assert)
3. Why exhaustive testing is impossible
4. Why we need pytest
5. Writing your first unit test
6. Clean code principles for testing

What is a Unit Test?

Definition:

A unit test verifies that a **single unit** (function, method, class) works correctly **in isolation**.

Key characteristics:

- ✓ Tests ONE function or method
- ✓ No dependencies on external systems (databases, APIs, UI)
- ✓ Fast (runs in milliseconds)
- ✓ Clear what's being tested
- ✓ Independent (can run in any order)

Example: Test `find_intersection()` without starting Dash app

Example Unit Test

```
def test_find_intersection_normal_angle():  
    """Test that find_intersection() works with a normal downward angle."""  
  
    # Arrange: Set up test data  
    x_road = np.array([0, 10, 20, 30])  
    y_road = np.array([0, 2, 3, 4])  
    angle = -10.0  
  
    # Act: Call the function  
    x, y, dist = find_intersection(x_road, y_road, angle)  
  
    # Assert: Verify the result  
    assert x is not None, "Should find an intersection"  
    assert dist > 0, "Distance should be positive"
```

That's it! Simple, focused, fast.

The AAA Pattern: Arrange-Act-Assert

Every good unit test follows this structure:

```
def test_example():  
    # ARRANGE: Set up the test data and conditions  
    input_data = prepare_test_data()  
    expected_result = calculate_expected()  
  
    # ACT: Call the function being tested  
    actual_result = function_under_test(input_data)  
  
    # ASSERT: Verify the result matches expectations  
    assert actual_result == expected_result
```

Why this pattern? Clear structure makes tests readable, easy to see what's being tested, easy to debug when tests fail.

AAA Pattern in Practice

```
def test_find_intersection_finds_intersection_for_normal_angle():
    # ARRANGE: Create simple road going upward
    x_road = np.array([0, 10, 20, 30], dtype=np.float64)
    y_road = np.array([0, 2, 4, 6], dtype=np.float64)
    angle = -10.0
    camera_x = 0.0
    camera_y = 10.0 # Camera above road

    # ACT: Find intersection
    x, y, dist = find_intersection(x_road, y_road, angle, camera_x, camera_y)

    # ASSERT: Should find an intersection
    assert x is not None
    assert dist > 0
    assert 0 <= x <= 30 # Within road bounds
```

Testing Frameworks: Why We Need pytest

Question: Can't we just write tests as regular Python functions?

```
# Without a testing framework:
def manual_test():
    x, y, dist = find_intersection(x_road, y_road, -10.0)
    if x is None:
        print("❌ FAILED: x should not be None")
        return False
    if dist <= 0:
        print("❌ FAILED: distance should be positive")
        return False
    print("✅ PASSED")
    return True

# Run test manually
manual_test()
```

Problems With Manual Testing

Manual testing has many issues:

1. ✗ No automatic test discovery (you have to call each function manually)
2. ✗ No organized test reports (just prints)
3. ✗ No test isolation (one failure might affect others)
4. ✗ No helpful error messages (you have to write all the checks)
5. ✗ No test fixtures (setup/teardown code)
6. ✗ No parallel execution (slow!)
7. ✗ No integration with CI/CD

Enter pytest: The industry standard testing framework

What Makes pytest Great

```
# Automatic test discovery - finds all test_*.py files  
$ uv run pytest
```

```
# Run with detailed output  
$ uv run pytest -v
```

```
# Run specific test file  
$ uv run pytest tests/test_geometry.py
```

```
# Run tests matching a pattern  
$ uv run pytest -k "intersection"
```

```
# Stop at first failure  
$ uv run pytest -x
```

pytest Features

What makes pytest the industry standard:

1.  **Simple syntax:** Just use `assert` (no special methods)
2.  **Auto-discovery:** Finds all `test_*.py` files automatically
3.  **Rich output:** Shows exactly what failed and why
4.  **Fixtures:** Share setup code across tests
5.  **Parametrization:** Run same test with different inputs
6.  **Plugins:** Extend functionality (coverage, benchmarking, etc.)
7.  **CI/CD integration:** Works seamlessly with GitHub Actions

pytest Output Example

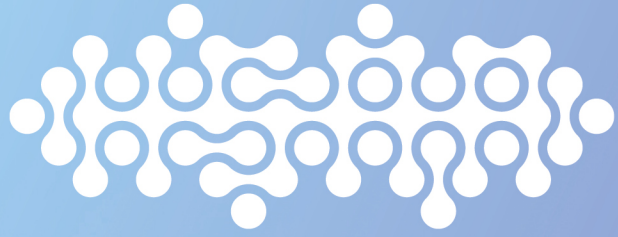
```
$ uv run pytest tests/test_geometry.py -v

tests/test_geometry.py::test_normal_angle PASSED      [ 20%]
tests/test_geometry.py::test_empty_array FAILED      [ 60%]

    def test_empty_array():
>         x, y, dist = find_intersection(x_road, y_road, -10.0)
E         IndexError: index 0 is out of bounds

1 failed, 4 passed in 0.12s
```

Shows exactly which test failed, the error, and pass/fail count



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Hands-On Practice

4

Writing Your First Unit Test

Let's write a test together for `find_intersection()`

Goal: Test the normal downward angle case

Step 1: Create test file structure

```
$ mkdir tests  
$ touch tests/__init__.py  
$ touch tests/test_geometry.py
```

Step 2: Write the test

Step 3: Run it and see it pass!

Your First Unit Test

```
# tests/test_geometry.py
import numpy as np
from road_profile_viewer.geometry import find_intersection

def test_find_intersection_finds_intersection_for_normal_angle():
    # Arrange
    x_road = np.array([0, 10, 20, 30], dtype=np.float64)
    y_road = np.array([0, 2, 4, 6], dtype=np.float64)

    # Act
    x, y, dist = find_intersection(x_road, y_road, -10.0, 0.0, 10.0)

    # Assert
    assert x is not None
    assert dist > 0
```

Run Your First Test

```
$ uv run pytest tests/test_geometry.py -v

===== test session starts =====
collected 1 item

tests/test_geometry.py::test_find_intersection_finds_intersection_for_normal_angle PASSED [100%]

===== 1 passed in 0.05s =====
```

✓ **Your first unit test passes!**

Congratulations! You've written your first pytest unit test.



Let's Write Another Test Together!

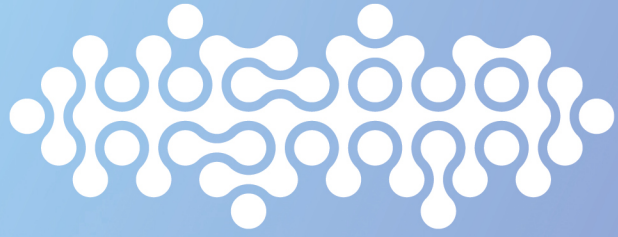
Interactive exercise: Let's write a test for `angle = 90°` (vertical angle)

What should happen?

- The function returns `None, None, None` (can't find intersection for vertical ray)

Your turn: What would the test look like?

Hint: Use the AAA pattern!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

The Limits of Testing

5

The Impossibility of Exhaustive Testing

Question: Why can't we just test every possible input?

Let's do the math for `find_intersection()`:

```
def find_intersection(x_road, y_road, angle_degrees,  
                     camera_x=0, camera_y=1.5):  
    """Find intersection between camera ray and road profile."""  
    # ...
```

Parameters:

- `angle_degrees`: float (-180° to 180°)
- `camera_x`: float
- `camera_y`: float
- `x_road`, `y_road`: arrays (variable length and values)

The Math: Testing Just One Parameter

Just `angle_degrees` (a float):

- **Possible values:** ~1,000,000 distinct values (conservative estimate)
- **Time per test:** ~1 millisecond (very fast!)

Total time to test just `angle_degrees`:

$1,000,000 \text{ tests} \times 0.001 \text{ seconds} = 1,000 \text{ seconds} \approx 17 \text{ minutes}$

Not too bad! But we have 3 float parameters...

The Math: Testing All Combinations

Testing ALL combinations of 3 float parameters:

$1,000,000^3 = 1$ quintillion tests

At 1ms per test = 32 BILLION YEARS 🤖

Age of the universe: ~14 billion years

Conclusion: Exhaustive testing is literally impossible

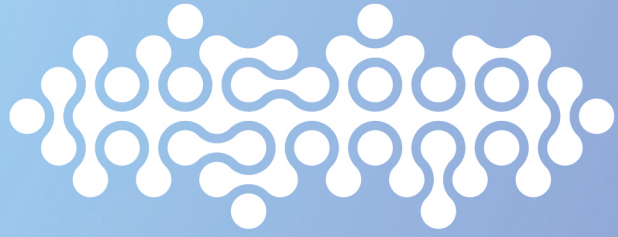
What Do We Do Instead?

Since exhaustive testing is impossible, we need smart testing strategies:

1. **Equivalence Classes** - Group similar inputs (Chapter 03 (Boundary Analysis))
2. **Boundary Value Analysis** - Test edge cases (Chapter 03 (Boundary Analysis))
3. **Risk-Based Testing** - Focus on critical functionality
4. **Code Coverage** - Ensure all code paths are tested (Chapter 03 (TDD and CI))

Today: We'll learn the fundamentals (AAA pattern, pytest, clean code)

Next time: We'll learn smart testing strategies (equivalence classes, boundaries)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Clean Code for Tests

6

Clean Code Principles for Testing

Three key principles from Robert C. Martin's "Clean Code":

1. **One Concept Per Test** - Each test verifies ONE behavior
2. **One Assert Per Test** (mostly) - Multiple OK if same concept
3. **Descriptive Test Names** - `test_[function]_[scenario]_[expected]()`

Why? When a test fails, you immediately know what's broken.

Principle 1: One Concept Per Test

✗ Bad: Testing multiple concepts

```
def test_find_intersection_everything():  
    x, y, dist = find_intersection(x_road, y_road, -10.0, 0.0, 10.0)  
    assert x is not None # Concept 1: downward angle  
  
    x2, y2, dist2 = find_intersection(x_road, y_road, 90.0)  
    assert x2 is None # Concept 2: vertical angle  
  
    x3, y3, dist3 = find_intersection(np.array([]), np.array([]), -10.0)  
    assert x3 is None # Concept 3: empty arrays
```

Problem: Which concept failed?

One Concept Per Test: Good Example

✓ Good: Three separate tests

```
def test_find_intersection_
    returns_intersection_
    for_downward_angle():
    x, y, dist = find_intersection(
        x_road, y_road,
        -10.0, 0.0, 10.0)
    assert x is not None

def test_find_intersection_
    returns_none_for_vertical_angle():
    x, y, dist = find_intersection(
        x_road, y_road, 90.0)
    assert x is None
```

Third test:

```
def test_find_intersection_
    returns_none_for_empty_arrays():
    x, y, dist = find_intersection(
        np.array([]),
        np.array([]),
        -10.0)
    assert x is None
```

Benefits:

- ✓ Test name tells exactly what failed
- ✓ Each test is focused and easy to understand
- ✓ Clear which scenario broke

Principle 2: One Assert Per Test

Ideally, each test should have **ONE** assert statement.

But there are reasonable exceptions!

When ONE assert is ideal:

```
def test_find_intersection_returns_positive_distance():
    """Test that distance is always positive when intersection found."""
    x_road = np.array([0, 10, 20], dtype=np.float64)
    y_road = np.array([0, 2, 4], dtype=np.float64)
    x, y, dist = find_intersection(x_road, y_road, -10.0, 0.0, 10.0)

    assert dist > 0 # One concept: distance is positive
```

When Multiple Asserts Are OK

✓ Multiple asserts OK when testing SAME concept:

```
def test_find_intersection_returns_valid_coordinates():
    x_road = np.array([0, 10, 20, 30], dtype=np.float64)
    y_road = np.array([0, 2, 4, 6], dtype=np.float64)
    x, y, dist = find_intersection(x_road, y_road, -10.0, 0.0, 10.0)

    # All verify SAME concept: valid coordinate bounds
    assert x is not None
    assert y is not None
    assert 0 <= x <= 30
    assert 0 <= y <= 6
```

Acceptable: All asserts test coordinate validity (one concept)

When to Split Tests

✗ Bad: Different concepts

```
def test_find_intersection_
  coordinates_and_distance():
    x, y, dist = find_intersection(
      x_road, y_road,
      -10.0, 0.0, 10.0)
    assert x > 0      # Concept 1
    assert dist > 0   # Concept 2
```

Problem: Testing two different concepts in one test

✓ Good: Separate tests

```
def test_find_intersection_
  returns_positive_x():
    x, y, dist = find_intersection(
      x_road, y_road,
      -10.0, 0.0, 10.0)
    assert x > 0

def test_find_intersection_
  returns_positive_distance():
    x, y, dist = find_intersection(
      x_road, y_road,
      -10.0, 0.0, 10.0)
    assert dist > 0
```

Principle 3: Descriptive Test Names

Test names should describe:

1. **What** is being tested
2. **What scenario**
3. **What the expected result** is

Pattern: `test_[function]_[scenario]_[expected_behavior]()`

✗ Bad Test Names:

```
def test_1(): # What does this test?
def test_intersection(): # Too vague
def test_angle(): # Which angle? What about it?
def test_edge_case(): # Which edge case?
```

Descriptive Test Names: Good Examples

✓ Good Test Names:

```
def test_find_intersection_returns_none_for_empty_arrays():  
    """Clear: function + scenario + expected result"""  
  
def test_find_intersection_finds_intersection_for_downward_angle():  
    """You know exactly what this tests"""  
  
def test_find_intersection_returns_none_when_ray_misses_road():  
    """Scenario and outcome are explicit"""  
  
def test_find_intersection_handles_vertical_angle_gracefully():  
    """Clear edge case handling"""
```

Why Descriptive Names Matter

Test output when failure occurs:

```
✗ FAILED test_1  
# What does this tell you? Nothing!  
  
✓ FAILED test_find_intersection_returns_none_for_empty_arrays  
# Immediately clear: empty array handling is broken
```

You immediately know what's broken without reading any code!



Interactive Exercise

What would you name a test for this scenario?

Test that `find_intersection()` returns None when the camera is below the road and the ray points downward.

Think about it:

- Function: `find_intersection`
- Scenario: Camera below road, ray points down
- Expected: Returns None (ray misses road)

What's your test name?

Clean Code Principles: Summary

Principle	Guideline	Why It Matters
One Concept Per Test	Each test verifies ONE behavior	When test fails, you immediately know what's broken
One Assert Per Test	Ideally one; multiple OK if same concept	Failures are unambiguous
Descriptive Names	<code>test_[function]_[scenario]_[expected]()</code>	Test output is self-documenting
AAA Pattern	Arrange-Act-Assert	Tests are readable and consistent

These principles make your tests easy to read, debug, and maintain.

Summary: What We've Learned

Key Takeaways:

1.  **Code quality $\neq$ code correctness** - CI checks style, tests check logic
2.  **Testing pyramid** - 70% unit, 20% module, 10% E2E
3.  **Avoid the test cone** - unit tests are faster and easier to debug
4.  **AAA pattern** - Arrange-Act-Assert makes tests readable
5.  **Exhaustive testing is impossible** - need smart strategies
6.  **pytest is the industry standard** - simple syntax, powerful features
7.  **Clean code principles** - one concept, descriptive names, focused tests

What's Next?

Chapter 03 (Boundary Analysis) will cover:

-  Equivalence class partitioning (smart testing strategies)
-  Boundary value analysis (finding edge cases)
-  LLM-assisted test writing (using AI to accelerate testing)
-  Human-AI collaboration for testing

Chapter 03 (TDD and CI) will cover:

-  Test-Driven Development (TDD)
-  Test coverage metrics
-  Adding pytest to CI/CD pipeline
-  Making tests mandatory

But first: Practice what you learned today!