



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Kapitel 03

UNIT TESTING UND CLEAN CODE

Wo wir jetzt stehen

Ihr bisheriger Fortschritt:

- ✓ **Kapitel 02 (Code-Qualität in der Praxis):** PEP8-Konformität mit Ruff
- ✓ **Kapitel 02 (Feature-Branch-Entwicklung):** Feature Branches & Pull Requests
- ✓ **Kapitel 02 (Automatisierung und CI/CD):** Automatisierte CI/CD-Checks
- ✓ **Kapitel 02 (Refactoring):** Refactoring zu modularer Struktur

Ihre CI ist grün! Ihr Code ist sauber! 🎉

Aber... funktioniert Ihr Code tatsächlich?

Die schockierende Entdeckung

Ihr sauberer, modularer Code wird in Produktion deployed...

```
# Ein Benutzer gibt einen vertikalen Kamerawinkel ein
angle = 90.0

# Ihr Code stürzt ab:
def find_intersection(x_road, y_road, angle_degrees, ...):
    angle_rad = -np.deg2rad(angle_degrees)
    slope = np.tan(angle_rad) # tan(90°) = ∞
    # ZeroDivisionError oder Endlosschleife!
```

-  Ruff: "Code ist schön formatiert!"
-  Pylint: "Typen sehen gut aus!"
-  **Niemand hat getestet, ob der Code funktioniert**

Heutige Lernziele

Am Ende dieser Vorlesung werden Sie:

1. ☒ **Den Unterschied verstehen** zwischen Code-Qualität (Stil) und Code-Korrektheit (Logik)
2. ☒ **Die Testpyramide beherrschen** - Unit-, Modul- und End-to-End-Tests
3. ☒ **Effektive Unit-Tests schreiben** mit dem AAA-Pattern
4. ☒ **pytest verwenden** zum Testen Ihres modularen Python-Codes
5. ☒ **Clean Code Prinzipien anwenden** für wartbare Tests
6. ☒ **Verstehen, warum erschöpfendes Testen unmöglich ist** (und was man dagegen tun kann)

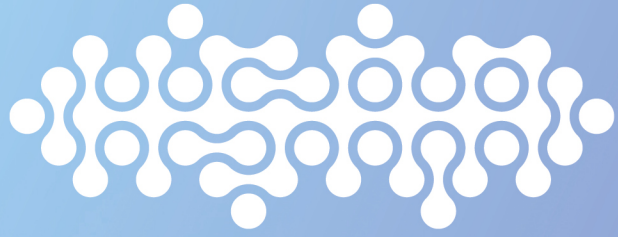
Was Sie HEUTE lernen und NICHT lernen

Was Sie lernen werden:

-  Wie man Bugs findet, bevor es die Benutzer tun
-  Wie man Tests schreibt, die Vertrauen geben
-  Warum die Testpyramide wichtig ist
-  Clean Code Prinzipien für Tests

Was Sie NICHT lernen werden (noch nicht):

-  Test-Driven Development (TDD) - das ist Kapitel 03 (TDD und CI)
-  Mocking und Fixtures - fortgeschrittene Themen für später
-  UI-Testing - komplex und außerhalb des Umfangs
-  Wie man LLMs zum Testen verwendet - das ist Kapitel 03 (Grenzwertanalyse)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Warum Testen wichtig ist

1

Der Bug, den die CI übersehen hat

Ihre `find_intersection()` Funktion:

```
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
```

Was passiert mit verschiedenen Eingaben?

Winkel	Erwartetes Verhalten	Tatsächliches Verhalten	CI-Status
-10°	Schnittpunkt finden	✓ Funktioniert	✓ Grün
-45°	Schnittpunkt finden	✓ Funktioniert	✓ Grün
0°	Horizontaler Strahl	✓ Funktioniert	✓ Grün
90°	Vertikaler Strahl	✗ Gibt None zurück	✓ Grün
Leeres Array	Leere Daten behandeln	✗ IndexError	✓ Grün

CI ist blind für Logikfehler!



Interaktive Frage

Denken Sie an Ihre eigenen Projekte:

Hatten Sie jemals einen Bug in Produktion (oder einer Demo), obwohl Ihre CI grün war?

Häufige Szenarien:

- "Es hat auf meinem Rechner funktioniert!"
- "Aber ich habe es manuell getestet..."
- "Die Typen waren alle korrekt!"

Diskutieren wir: Welche Arten von Bugs rutschen durch?

Code-Qualität vs. Code-Korrektheit

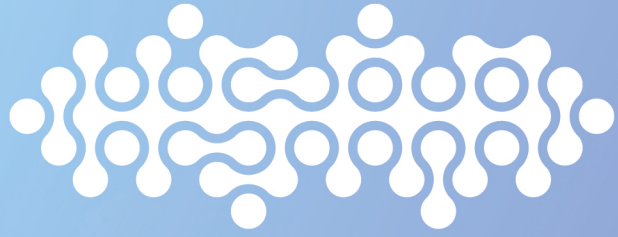
Code-Qualitäts-Tools → "Sieht das professionell aus?"

- ✓ Schön formatiert?
- ✓ Type Hints vorhanden?
- ✓ Keine ungenutzten Imports?

Tests → "Funktioniert das tatsächlich?"

- ✓ Korrekte Ergebnisse?
- ✓ Behandelt Randfälle?
- ✓ Fehler werden abgefangen?

Beide sind essentiell für professionelle Softwareentwicklung.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Die Testpyramide

2

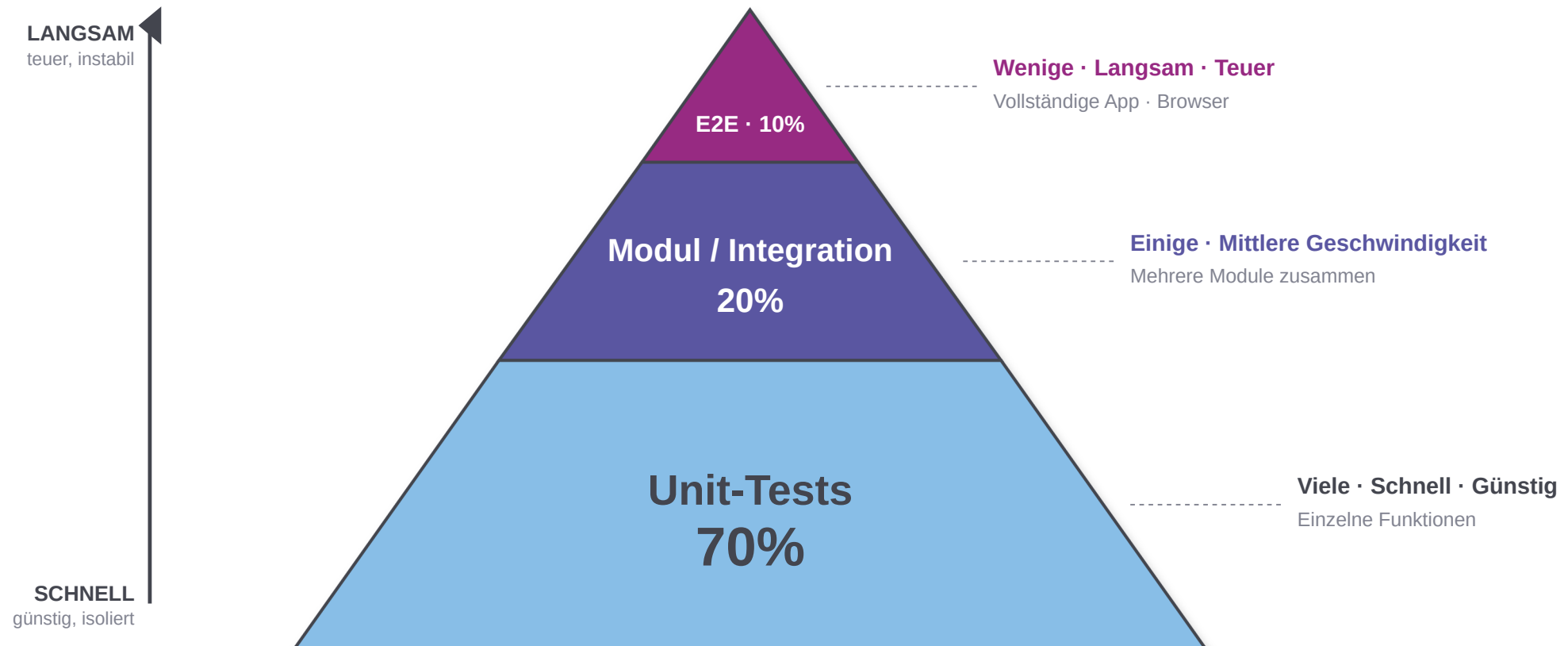
Wie testet man Software systematisch?

Drei Ebenen des Testens, jede mit unterschiedlichen Zwecken:

1. **Unit-Tests** - Testen einzelne Funktionen
2. **Modul-/Integrationstests** - Testen Module, die zusammenarbeiten
3. **End-to-End-Tests** - Testen die gesamte Anwendung

Aber wie viel von jedem sollten wir schreiben?

Was ist die Testpyramide?



Die Form ist wichtig! Mehr Tests unten (Unit), weniger oben (E2E).

Warum die Pyramidenform?

Empfohlenes Verhältnis:

- **70%** Unit-Tests
- **20%** Modul-/Integrationstests
- **10%** End-to-End-Tests

Warum? Geschwindigkeit ist wichtig für die Feedback-Schleife:

Wenn Sie eine Zeile in `find_intersection()` ändern:

-  50 Unit-Tests laufen in **0,5 Sekunden** → sofortiges Feedback
-  50 E2E-Tests laufen in **250 Sekunden** → Sie holen sich Kaffee, verlieren den Flow

Die Feedback-Schleife ist kritisch

Schnelle Tests:

Oft laufen lassen → Bugs sofort finden → Schnell beheben → Im Flow bleiben

Langsame Tests:

Selten laufen lassen → Bugs häufen sich → Schwer zu debuggen → Kontextwechsel

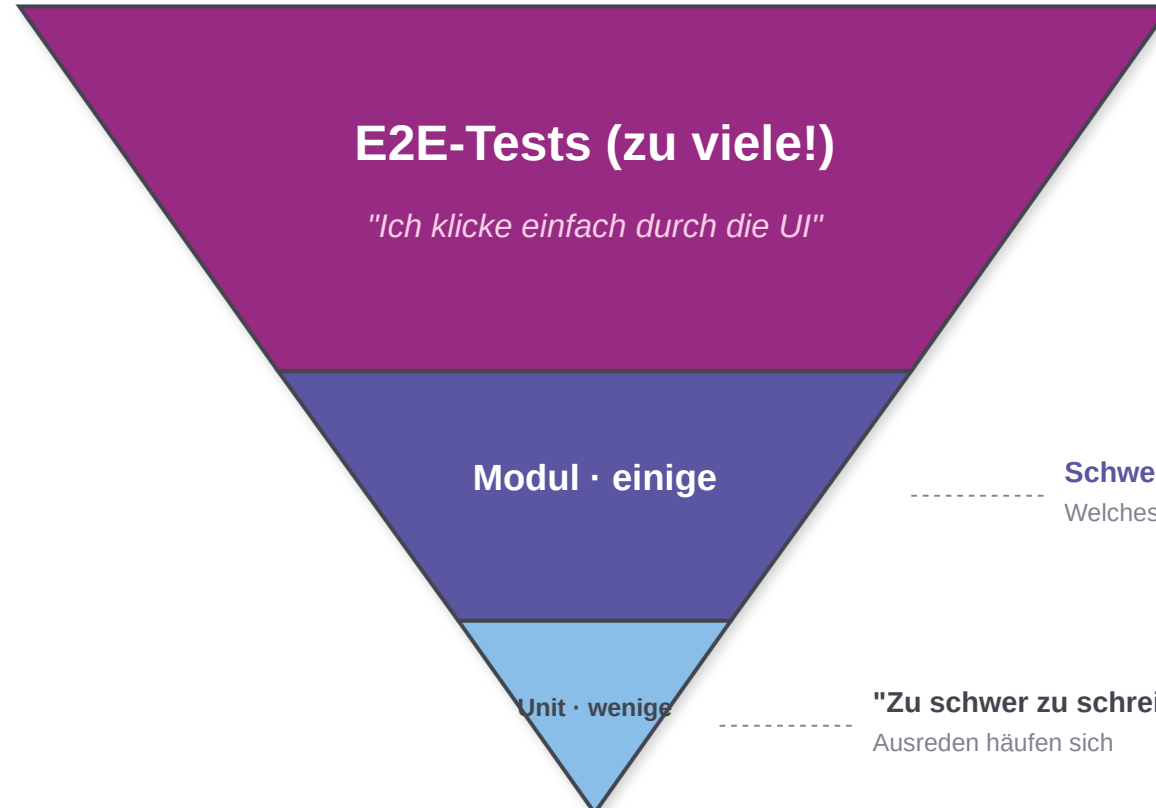
Beispielvergleich:

-  Code schreiben → Tests laufen (0,5s) → Feedback bekommen → Iterieren
-  Code schreiben → Tests laufen (5min) → E-Mails checken → Was habe ich gemacht?

Das Anti-Pattern: Der Testkegel



ANTI-
PATTERN



----- **Langsames Feedback**

Tests werden gemieden

----- **Schwer zu debuggen**

Welches Modul ist kaputt?

----- **"Zu schwer zu schreiben"**

Ausreden häufen sich

Das ist verkehrt herum! Gleiche Farben, umgedrehte Form — jede Ebene wird zur



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Probleme mit dem Testkegel

1. Langsames Feedback:

- Jede Änderung erfordert langsame E2E-Tests
- Entwickler vermeiden es, Tests zu laufen → Bugs häufen sich

2. Schwer zu debuggen:

- E2E-Test schlägt fehl → welches der 10 Module hat es verursacht?
- Unit-Test schlägt fehl → Sie wissen genau, welche Funktion kaputt ist

3. Zerbrechlich:

- UI-Änderungen brechen alle E2E-Tests
- Unit-Tests sind von UI-Änderungen nicht betroffen

4. Teufelskreis:

- Tests sind langsam → Entwickler vermeiden es, sie zu laufen → Mehr Bugs



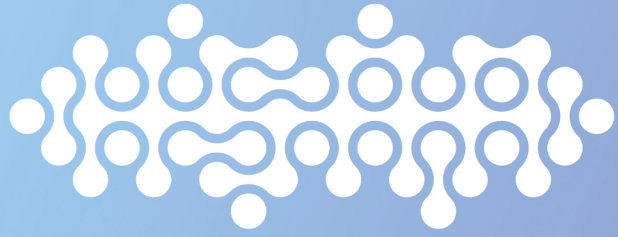
Interaktive Diskussion

Warum glauben Sie, dass Entwickler Testkegel erstellen?

Häufige Gründe:

- "Ich weiß nicht, wie man Unit-Tests schreibt"
- "Test-Setup ist mühsam"
- "Durch die UI zu klicken ist einfach"
- "Unit-Tests fühlen sich wie Mehrarbeit an"

Heutiges Ziel: Unit-Testing so einfach machen, dass Sie es bevorzugen!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Unit-Testing im Detail

3

Vom WARUM zum WIE

Jetzt, da wir wissen, WARUM Unit-Tests wichtig sind, lernen wir, WIE man sie schreibt.

Schlüsselthemen:

1. Was ist ein Unit-Test?
2. Das AAA-Pattern (Arrange-Act-Assert)
3. Warum erschöpfendes Testen unmöglich ist
4. Warum wir pytest brauchen
5. Ihren ersten Unit-Test schreiben
6. Clean Code Prinzipien für Tests

Was ist ein Unit-Test?

Definition:

Ein Unit-Test überprüft, dass eine **einzelne Einheit** (Funktion, Methode, Klasse) **isoliert** korrekt funktioniert.

Schlüsselmerkmale:

-  Testet EINE Funktion oder Methode
-  Keine Abhängigkeiten von externen Systemen (Datenbanken, APIs, UI)
-  Schnell (läuft in Millisekunden)
-  Klar, was getestet wird
-  Unabhängig (kann in beliebiger Reihenfolge laufen)

Beispiel: Testen Sie `find_intersection()` ohne Dash-App zu starten

Beispiel 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"
```

Das ist es! Einfach, fokussiert, schnell.

Das AAA-Pattern: Arrange-Act-Assert

Jeder gute Unit-Test folgt dieser Struktur:

```
def test_example():  
    # ARRANGE: Testdaten und Bedingungen einrichten  
    input_data = prepare_test_data()  
    expected_result = calculate_expected()  
  
    # ACT: Zu testende Funktion aufrufen  
    actual_result = function_under_test(input_data)  
  
    # ASSERT: Überprüfen, ob Ergebnis den Erwartungen entspricht  
    assert actual_result == expected_result
```

Warum dieses Pattern? Klare Struktur macht Tests lesbar, leicht zu sehen, was getestet wird, leicht zu debuggen, wenn Tests fehlschlagen.

AAA-Pattern in der Praxis

```
def test_find_intersection_finds_intersection_for_normal_angle():  
    # ARRANGE: Einfache Straße nach oben erstellen  
    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 # Kamera über der Straße  
  
    # ACT: Schnittpunkt finden  
    x, y, dist = find_intersection(x_road, y_road, angle, camera_x, camera_y)  
  
    # ASSERT: Sollte einen Schnittpunkt finden  
    assert x is not None  
    assert dist > 0  
    assert 0 <= x <= 30 # Innerhalb der Straßengrenzen
```

Test-Frameworks: Warum wir pytest brauchen

Frage: Können wir nicht einfach Tests als normale Python-Funktionen schreiben?

```
# Ohne Test-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

# Test manuell ausführen
manual_test()
```

Probleme mit manuellem Testen

Manuelles Testen hat viele Probleme:

1. ✗ Keine automatische Test-Erkennung (Sie müssen jede Funktion manuell aufrufen)
2. ✗ Keine organisierten Test-Reports (nur Prints)
3. ✗ Keine Test-Isolation (ein Fehler könnte andere beeinflussen)
4. ✗ Keine hilfreichen Fehlermeldungen (Sie müssen alle Checks schreiben)
5. ✗ Keine Test-Fixtures (Setup/Teardown-Code)
6. ✗ Keine parallele Ausführung (langsam!)
7. ✗ Keine Integration mit CI/CD

Enter pytest: Der Industriestandard für Test-Frameworks

Was macht pytest großartig

```
# Automatische Test-Erkennung - findet alle test_*.py Dateien  
$ uv run pytest
```

```
# Mit detaillierter Ausgabe  
$ uv run pytest -v
```

```
# Spezifische Test-Datei ausführen  
$ uv run pytest tests/test_geometry.py
```

```
# Tests ausführen, die einem Muster entsprechen  
$ uv run pytest -k "intersection"
```

```
# Bei erstem Fehler stoppen  
$ uv run pytest -x
```

pytest Features

Was macht pytest zum Industriestandard:

1.  **Einfache Syntax:** Verwenden Sie einfach `assert` (keine speziellen Methoden)
2.  **Auto-Erkennung:** Findet automatisch alle `test_*.py` Dateien
3.  **Reichhaltige Ausgabe:** Zeigt genau, was fehlgeschlagen ist und warum
4.  **Fixtures:** Teilen Sie Setup-Code zwischen Tests
5.  **Parametrisierung:** Führen Sie denselben Test mit verschiedenen Eingaben aus
6.  **Plugins:** Erweitern Sie die Funktionalität (Coverage, Benchmarking, etc.)
7.  **CI/CD-Integration:** Funktioniert nahtlos mit GitHub Actions

pytest Output Beispiel

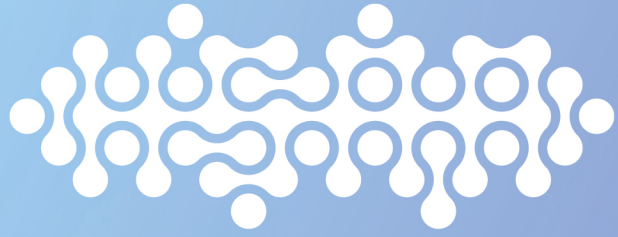
```
$ uv run pytest tests/test_geometry.py -v

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
```

Zeigt genau, welcher Test fehlgeschlagen ist, den Fehler und Pass/Fail-Anzahl



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Praktische Übung

4

Ihren ersten Unit-Test schreiben

Lassen Sie uns gemeinsam einen Test für `find_intersection()` schreiben

Ziel: Den normalen Abwärtswinkel-Fall testen

Schritt 1: Test-Dateistruktur erstellen

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

Schritt 2: Den Test schreiben

Schritt 3: Ihn ausführen und sehen, dass er funktioniert!

Ihr erster 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
```

Ihren ersten Test ausführen

```
$ 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 =====
```

✓ **Ihr erster Unit-Test funktioniert!**

Herzlichen Glückwunsch! Sie haben Ihren ersten pytest Unit-Test geschrieben.



Lassen Sie uns gemeinsam einen weiteren Test schreiben!

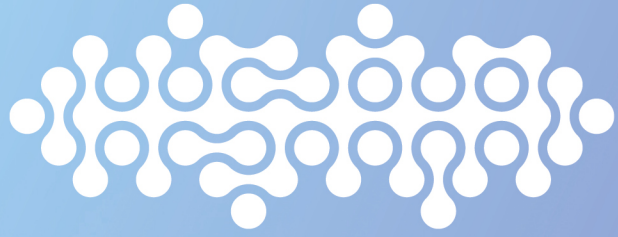
Interaktive Übung: Lassen Sie uns einen Test für `angle = 90°` (vertikaler Winkel) schreiben

Was sollte passieren?

- Die Funktion gibt `None, None, None` zurück (kann keinen Schnittpunkt für vertikalen Strahl finden)

Sie sind dran: Wie würde der Test aussehen?

Hinweis: Verwenden Sie das AAA-Pattern!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Die Grenzen des Testens

5

Die Unmöglichkeit erschöpfenden Testens

Frage: Warum können wir nicht einfach jede mögliche Eingabe testen?

Lassen Sie uns die Mathematik für `find_intersection()` durchrechnen:

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

Parameter:

- `angle_degrees`: float (-180° bis 180°)
- `camera_x`: float
- `camera_y`: float
- `x_road`, `y_road`: Arrays (variable Länge und Werte)

Die Mathematik: Nur einen Parameter testen

Nur `angle_degrees` (ein Float):

- **Mögliche Werte:** ~1.000.000 verschiedene Werte (konservative Schätzung)
- **Zeit pro Test:** ~1 Millisekunde (sehr schnell!)

Gesamtzeit, um nur `angle_degrees` zu testen:

$1.000.000 \text{ Tests} \times 0,001 \text{ Sekunden} = 1.000 \text{ Sekunden} \approx 17 \text{ Minuten}$

Nicht zu schlecht! Aber wir haben 3 Float-Parameter...

Die Mathematik: Alle Kombinationen testen

Alle Kombinationen von 3 Float-Parametern testen:

$1.000.000^3 = 1 \text{ Trillion Tests}$

Bei 1ms pro Test = 32 MILLIARDEN JAHRE 🤖

Alter des Universums: ~14 Milliarden Jahre

Fazit: Erschöpfendes Testen ist buchstäblich unmöglich

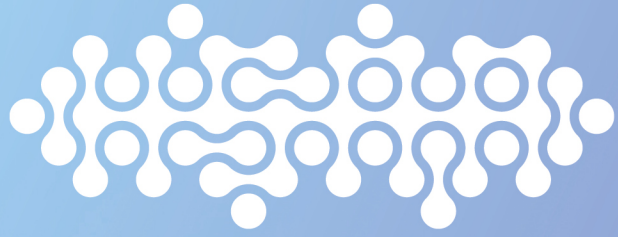
Was machen wir stattdessen?

Da erschöpfendes Testen unmöglich ist, brauchen wir intelligente Teststrategien:

1. **Äquivalenzklassen** - Ähnliche Eingaben gruppieren (Kapitel 03 (Grenzwertanalyse))
2. **Grenzwertanalyse** - Randfälle testen (Kapitel 03 (Grenzwertanalyse))
3. **Risikobasiertes Testen** - Auf kritische Funktionalität fokussieren
4. **Code Coverage** - Sicherstellen, dass alle Code-Pfade getestet sind (Kapitel 03 (TDD und CI))

Heute: Wir lernen die Grundlagen (AAA-Pattern, pytest, Clean Code)

Nächstes Mal: Wir lernen intelligente Teststrategien (Äquivalenzklassen, Grenzen)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Clean Code für Tests

6

Clean Code Prinzipien für Tests

Drei Schlüsselprinzipien aus Robert C. Martins "Clean Code":

1. **Ein Konzept pro Test** - Jeder Test überprüft EIN Verhalten
2. **Ein Assert pro Test** (meistens) - Mehrere OK, wenn gleiches Konzept
3. **Beschreibende Testnamen** - `test_[funktion]_[szenario]_[erwartet]()`

Warum? Wenn ein Test fehlschlägt, wissen Sie sofort, was kaputt ist.

Prinzip 1: Ein Konzept pro Test

✗ Schlecht: Mehrere Konzepte testen

```
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 # Konzept 1: Abwärtswinkel  
  
    x2, y2, dist2 = find_intersection(x_road, y_road, 90.0)  
    assert x2 is None # Konzept 2: Vertikaler Winkel  
  
    x3, y3, dist3 = find_intersection(np.array([]), np.array([]), -10.0)  
    assert x3 is None # Konzept 3: Leere Arrays
```

Problem: Welches Konzept ist fehlgeschlagen?

Ein Konzept pro Test: Gutes Beispiel

✓ Gut: Drei 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
```

Dritter 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
```

- ✓ Testname sagt genau, was fehlgeschlagen ist
- ✓ Jeder Test ist fokussiert und leicht verständlich

Prinzip 2: Ein Assert pro Test

Idealerweise sollte jeder Test EINE Assert-Anweisung haben.

Aber es gibt vernünftige Ausnahmen!

Wann EIN Assert ideal ist:

```
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 # Ein Konzept: Distanz ist positiv
```

Wann mehrere Asserts OK sind

✓ Mehrere Asserts OK, wenn **GLEICHES** Konzept getestet wird:

```
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)

    # Alle überprüfen GLEICHES Konzept: gültige Koordinatengrenzen
    assert x is not None
    assert y is not None
    assert 0 <= x <= 30
    assert 0 <= y <= 6
```

Akzeptabel: Alle Asserts testen Koordinatenvalidität (ein Konzept)

Wann Tests aufteilen

✗ Schlecht: Verschiedene Konzepte

```
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      # Konzept 1
    assert dist > 0   # Konzept 2
```

Problem: Zwei verschiedene Konzepte in einem Test

✓ Gut: 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
```

Prinzip 3: Beschreibende Testnamen

Testnamen sollten beschreiben:

1. **Was** getestet wird
2. **Welches Szenario**
3. **Was das erwartete Ergebnis** ist

Pattern: `test_[funktion]_[szenario]_[erwartetes_verhalten]()`

✗ Schlechte Testnamen:

```
def test_1(): # Was testet das?  
def test_intersection(): # Zu vage  
def test_angle(): # Welcher Winkel? Was ist damit?  
def test_edge_case(): # Welcher Randfall?
```

Beschreibende Testnamen: Gute Beispiele

✓ Gute Testnamen:

```
def test_find_intersection_returns_none_for_empty_arrays():  
    """Klar: Funktion + Szenario + erwartetes Ergebnis"""  
  
def test_find_intersection_finds_intersection_for_downward_angle():  
    """Sie wissen genau, was das testet"""  
  
def test_find_intersection_returns_none_when_ray_misses_road():  
    """Szenario und Ergebnis sind explizit"""  
  
def test_find_intersection_handles_vertical_angle_gracefully():  
    """Klare Randfall-Behandlung"""
```

Warum beschreibende Namen wichtig sind

Test-Ausgabe, wenn ein Fehler auftritt:

```
✗ FAILED test_1
  # Was sagt Ihnen das? Nichts!

✓ FAILED test_find_intersection_returns_none_for_empty_arrays
  # Sofort klar: Behandlung leerer Arrays ist kaputt
```

Sie wissen sofort, was kaputt ist ohne Code zu lesen!



Interaktive Übung

Wie würden Sie einen Test für dieses Szenario benennen?

Testen Sie, dass `find_intersection()` `None` zurückgibt, wenn die Kamera unter der Straße ist und der Strahl nach unten zeigt.

Denken Sie darüber nach:

- Funktion: `find_intersection`
- Szenario: Kamera unter der Straße, Strahl zeigt nach unten
- Erwartet: Gibt `None` zurück (Strahl verfehlt die Straße)

Wie lautet Ihr Testname?

Clean Code Prinzipien: Zusammenfassung

Prinzip	Richtlinie	Warum es wichtig ist
Ein Konzept pro Test	Jeder Test überprüft EIN Verhalten	Wenn Test fehlschlägt, wissen Sie sofort, was kaputt ist
Ein Assert pro Test	Idealerweise eins; mehrere OK wenn gleiches Konzept	Fehler sind eindeutig
Beschreibende Namen	<code>test_[funktion]_[szenario]_[erwartet]()</code>	Test-Ausgabe ist selbstdokumentierend
AAA-Pattern	Arrange-Act-Assert	Tests sind lesbar und konsistent

Diese Prinzipien machen Ihre Tests einfach zu lesen, debuggen und warten.

Zusammenfassung: Was wir gelernt haben

Wichtige Erkenntnisse:

1.  **Code-Qualität $\neq$ Code-Korrektheit** - CI prüft Stil, Tests prüfen Logik
2.  **Testpyramide** - 70% Unit, 20% Modul, 10% E2E
3.  **Testkegel vermeiden** - Unit-Tests sind schneller und leichter zu debuggen
4.  **AAA-Pattern** - Arrange-Act-Assert macht Tests lesbar
5.  **Erschöpfendes Testen ist unmöglich** - intelligente Strategien erforderlich
6.  **pytest ist der Industriestandard** - einfache Syntax, mächtige Features
7.  **Clean Code Prinzipien** - ein Konzept, beschreibende Namen, fokussierte Tests

Was kommt als Nächstes?

Kapitel 03 (Grenzwertanalyse) wird behandeln:

-  Äquivalenzklassen-Partitionierung (intelligente Teststrategien)
-  Grenzwertanalyse (Randfälle finden)
-  LLM-unterstütztes Test-Schreiben (KI zur Testbeschleunigung nutzen)
-  Mensch-KI-Kollaboration beim Testen

Kapitel 03 (TDD und CI) wird behandeln:

-  Test-Driven Development (TDD)
-  Test-Coverage-Metriken
-  pytest zu CI/CD-Pipeline hinzufügen
-  Tests obligatorisch machen

Aber zuerst: Üben Sie, was Sie heute gelernt haben!