



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Chapter 03

BOUNDARY VALUE ANALYSIS &
PYTEST.APPROX()

Today's Learning Objectives

By the end of this lecture, you will be able to:

-  **Identify boundary values** systematically for numeric functions
-  **Use IEEE 754 boundaries** in your tests (`sys.float_info`, `math.inf`, `math.nan`)
-  **Write proper floating-point assertions** with `pytest.approx()`
-  **Apply white-box boundary testing** to find edge cases

Focus: Writing tests that catch real bugs, not just happy-path scenarios!

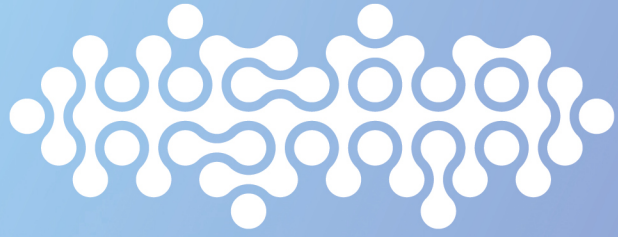
Why Boundaries Matter

Observation: Most bugs hide at the **edges** of equivalence classes, not in the middle.

Common boundary bugs:

- ✗ Off-by-one errors (`<=` vs `<`)
- ✗ Floating-point precision issues
- ✗ Overflow/underflow at numeric limits
- ✗ Division by zero
- ✗ Unhandled special values (NaN, infinity)

Example: Your tests pass for `reciprocal(5.0)`, but fail for `reciprocal(sys.float_info.max)` ✨



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

IEEE 754 & Float Boundaries

1

IEEE 754: The Foundation of Float Testing

What is IEEE 754?

International standard for floating-point arithmetic (all modern languages follow it)

Why this matters for testing:

Understanding IEEE 754 boundaries → Write tests for edge cases at limits of numerical precision

Python floats: 64-bit IEEE 754 doubles (same as C/C++ `double`)

Key insight: There are **specific boundary values** defined by the standard!

Critical Float Boundaries in Python

```
# FINITE boundaries
sys.float_info.max      #  $\approx 1.8e308$  - Largest finite float
sys.float_info.min      #  $\approx 2.2e-308$  - Smallest positive normalized
sys.float_info.epsilon  #  $\approx 2.2e-16$  - Difference between 1.0 and next float
math.ulp(0.0)           #  $\approx 5e-324$  - Smallest positive subnormal

# NON-FINITE special values
math.inf                # Positive infinity
-math.inf               # Negative infinity
math.nan                # Not a Number

# PRECISION tools
math.ulp(x)             # Unit in Last Place (spacing to next float)
math.nextafter(x, y)    # Next representable float toward y
```

Two Tools for Precision Boundary Testing

`math.ulp(x)` - Unit in Last Place

- Returns the **size of the step** to the next representable float from `x`
- Tells you "how much spacing" exists at this magnitude

`math.nextafter(x, y)` - Next After

- Returns the **actual next float** from `x` in the direction of `y`
- Gives you the concrete neighboring value

Key relationship:

```
math.nextafter(x, math.inf) - x == math.ulp(x)  # Moving toward +∞
```

`math.ulp()` and `math.nextafter()` - Examples

```
# Example 1: At 1.0
x = 1.0
step_size = math.ulp(x)           # ≈ 2.22e-16
next_float = math.nextafter(x, 10.0) # Next float toward 10.0

# The relationship holds:
assert next_float - x == step_size  # ✅ True!

# Example 2: Spacing grows with magnitude
math.ulp(1.0)      # ≈ 2.22e-16 (tiny spacing)
math.ulp(1e10)     # ≈ 0.001953 (much larger spacing!)
math.ulp(1e100)    # ≈ 1.94e84 (huge spacing!)

# Example 3: Direction matters for nextafter
math.nextafter(1.0, 10.0)  # 1.0000000000000002 (toward +∞)
math.nextafter(1.0, -10.0) # 0.9999999999999999 (toward -∞)
```

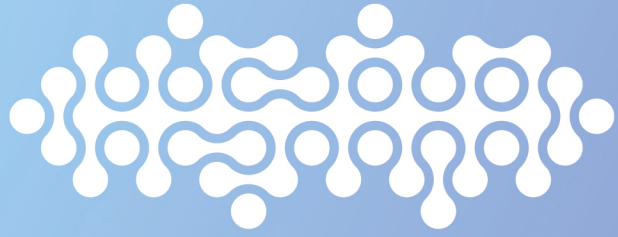
Use case for testing: Create values just **barely** outside valid ranges!



Critical Distinction: `math.inf` vs `sys.float_info.max`

Aspect	<code>math.inf</code>	<code>sys.float_info.max</code>
Type	Non-finite special value	Largest finite float
Value	Positive infinity	$\approx 1.8 \times 10^{308}$
Tests	Infinity handling logic	Finite range limits
Example	<code>1 / math.inf == 0.0</code>	<code>sys.float_info.max * 2 == inf</code>

```
# NOT the same!
math.inf > sys.float_info.max # True
sys.float_info.max * 2        # Overflows to inf!
```



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Boundary Testing Strategy

2

Boundary Testing Strategy: `reciprocal(x) = 1/x`

Equivalence classes → Boundary values

Class	Boundaries to Test
Positive	<code>1e-6</code> , <code>1.0</code> , <code>1e10</code> (near zero, normal, large)
Negative	<code>-1e-6</code> , <code>-1.0</code> , <code>-1e10</code>
Zero	<code>0.0</code> (should raise exception!)
Extreme	<code>sys.float_info.max</code> , <code>sys.float_info.min</code>
Special	<code>math.inf</code> , <code>-math.inf</code> , <code>math.nan</code>

Don't just test the **middle** - test the **edges**!

Example: Testing Boundary - Exactly Zero

```
import pytest

def test_reciprocal_exactly_zero():
    """Boundary: Division by zero should raise exception"""
    with pytest.raises(ZeroDivisionError):
        reciprocal(0.0)
```

Key points:

-  Test the **exact** boundary (not `1e-100`)
-  Verify expected behavior (exception, not crash!)
-  Use descriptive test name

Example: Testing Boundary - Infinity

```
import math

def test_reciprocal_positive_infinity():
    """Boundary:  $1/\infty = 0$  (mathematical limit)"""
    result = reciprocal(math.inf)
    assert result == 0.0

def test_reciprocal_negative_infinity():
    """Boundary:  $1/(-\infty) = -0$ """
    result = reciprocal(-math.inf)
    assert result == 0.0 # Python: 0.0 == -0.0 is True
```

Tests special value handling, not overflow prevention!

Example: Testing Boundary - NaN (Not a Number)

```
import math

def test_reciprocal_nan():
    """Boundary: NaN propagates through operations"""
    result = reciprocal(math.nan)

    # ✗ WRONG: assert result == math.nan
    # NaN == NaN is ALWAYS False!

    # ✓ CORRECT:
    assert math.isnan(result)
```

Critical: `nan == nan` returns `False` (IEEE 754 standard)

Always use `math.isnan()` to check for NaN!

Example: Testing Boundary - Maximum Finite Float

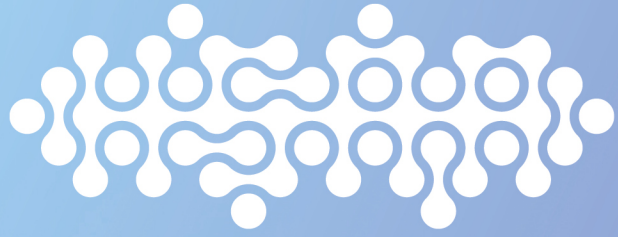
```
import sys

def test_reciprocal_max_finite_float():
    """Boundary: Reciprocal of largest finite → extremely small

    Tests overflow prevention at upper finite boundary.
    Result may underflow to subnormal (denormalized) range.
    """
    result = reciprocal(sys.float_info.max)

    # Check result is positive but tiny
    assert result > 0
    assert result < sys.float_info.min # Underflows to subnormal!
```

Result: $\approx 5.6 \times 10^{-309}$ (subnormal number)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Comparing Floats with `pytest.approx()`

3

The Floating-Point Comparison Problem

```
# ❌ THIS FAILS!
def test_simple_addition():
    result = 0.1 + 0.2
    assert result == 0.3 # AssertionError: 0.30000000000000004 != 0.3

# ❌ THIS IS FRAGILE!
def test_with_manual_tolerance():
    result = 0.1 + 0.2
    assert abs(result - 0.3) < 1e-9 # Works, but not recommended
```

Problem: Floating-point arithmetic is **not exact** due to binary representation

Solution: Use `pytest.approx()` 

pytest.approx() - The Right Way

```
import pytest

# ✓ CORRECT: Use pytest.approx()
result = 0.1 + 0.2
assert result == pytest.approx(0.3)

# ✓ Custom tolerance
assert result == pytest.approx(expected, rel=1e-9, abs=1e-12)
```

Advantages:

- ✓ Clear intent
- ✓ Better error messages
- ✓ Handles edge cases (NaN, infinity)
- ✓ Works with arrays, lists, dicts!

How `pytest.approx()` Works

Default tolerances:

- `rel=1e-6` (relative: 0.0001%)
- `abs=1e-12` (absolute)

Comparison rule: Values equal if **EITHER** tolerance satisfied:

$$|\text{actual} - \text{expected}| \leq \max(\text{rel} \times |\text{expected}|, \text{abs})$$

Why two tolerances?

- **Relative** scales with magnitude (good for large numbers)
- **Absolute** handles near-zero values (where relative breaks down)

`pytest.approx()` - Example with Boundaries

```
import pytest
import sys

def test_reciprocal_near_zero_positive():
    """Boundary: Very small positive → very large result"""
    result = reciprocal(1e-6) # 0.000001
    assert result == pytest.approx(1e6, rel=1e-9) # 1000000

def test_reciprocal_very_large_positive():
    """Boundary: Very large → very small result"""
    result = reciprocal(1e10)
    assert result == pytest.approx(1e-10, rel=1e-9)
```

Notice: We specify `rel=1e-9` for tighter precision than default!

Choosing Tolerances: ULP-Based Precision

For high-precision requirements, use `math.ulp()`:

```
import math
import pytest

def test_reciprocal_ulp_precision():
    """Test with ULP (Unit in Last Place) tolerance"""
    x = 1.0
    result = reciprocal(x)
    expected = 1.0

    # Tolerance: 10 ULPs (10 × spacing between floats)
    tolerance = 10 * math.ulp(expected)

    assert result == pytest.approx(expected, abs=tolerance)
```

ULP: Spacing between consecutive representable floats at a given magnitude

`pytest.approx()` Works with Collections!

```
import numpy as np
import pytest

# ✅ Lists
assert [0.1 + 0.2, 0.2 + 0.4] == pytest.approx([0.3, 0.6])

# ✅ NumPy arrays (most common in this course!)
result = np.array([1.0000001, 2.0000001, 3.0000001])
expected = np.array([1.0, 2.0, 3.0])
assert result == pytest.approx(expected)

# ✅ Dictionaries (compares values)
result_dict = {"mean": 0.1 + 0.2, "std": 0.2 + 0.4}
expected_dict = {"mean": 0.3, "std": 0.6}
assert result_dict == pytest.approx(expected_dict)
```

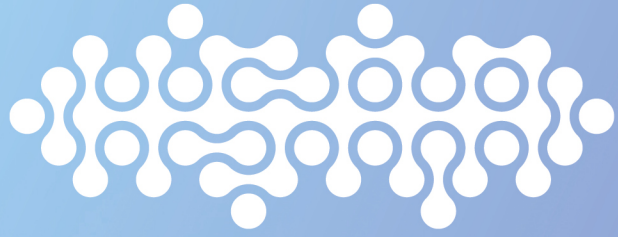
Special Case: Testing with NaN

```
import math
import pytest

# ❌ WRONG: NaN comparisons always fail!
def test_nan_wrong():
    result = compute_invalid() # Returns NaN
    assert result == pytest.approx(math.nan) # FAILS!

# ✅ CORRECT: Use math.isnan()
def test_nan_correct():
    result = compute_invalid()
    assert math.isnan(result)

# ✅ OR: Use nan_ok=True (if comparing collections)
assert [1.0, math.nan] == pytest.approx([1.0, math.nan], nan_ok=True)
```



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Putting It All Together

4

Hands-On Example: Complete Boundary Test Suite

```
import sys, math, pytest

# Define constants for readability
FMAX = sys.float_info.max
FMIN = sys.float_info.min
INF = math.inf
NAN = math.nan

class TestReciprocalBoundaries:
    """Comprehensive boundary tests"""

    def test_exactly_zero(self):
        """Boundary: Division by zero"""
        with pytest.raises(ZeroDivisionError):
            reciprocal(0.0)
```

Hands-On Example: Complete Boundary Test Suite (cont.)

```
def test_positive_infinity(self):  
    """Boundary:  $1/\infty = 0$ """  
    assert reciprocal(INF) == 0.0  
  
def test_negative_infinity(self):  
    """Boundary:  $1/(-\infty) = -0$ """  
    assert reciprocal(-INF) == 0.0  
  
def test_nan(self):  
    """Boundary: NaN propagates"""  
    assert math.isnan(reciprocal(NAN))
```

Hands-On Example: Complete Boundary Test Suite (cont.)

```
def test_max_finite_float(self):
    """Boundary: Largest finite → tiny result"""
    result = reciprocal(FMAX)
    assert result > 0
    assert result < FMIN # Underflows to subnormal

def test_min_normalized_float(self):
    """Boundary: Smallest normalized → huge result"""
    result = reciprocal(FMIN)
    assert result == INF # Overflows!

def test_near_zero_positive(self):
    """Boundary: Near zero → large result"""
    assert reciprocal(1e-6) == pytest.approx(1e6, rel=1e-9)
```

Common Pitfalls to Avoid

✗ DON'T:

- Use `==` for float comparisons
- Test only "happy path" values (middle of ranges)
- Ignore special values (`inf`, `nan`)
- Forget about overflow/underflow

✓ DO:

- Use `pytest.approx()` for all float comparisons
- Test boundaries systematically
- Handle special values explicitly
- Document why each boundary matters (docstrings!)

Summary: Your Testing Toolkit

Boundary Value Analysis:

1. Identify equivalence classes
2. **Focus on boundaries** (not middle values!)
3. Use IEEE 754 constants (`sys.float_info`, `math.inf`, `math.nan`)
4. Test special values explicitly

`pytest.approx()`:

1. **Always** use for float comparisons
2. Customize tolerances when needed (`rel`, `abs`)
3. Use `math.ulp()` for high-precision requirements
4. Works with arrays, lists, dicts!

Key Takeaways

- ✓ **Bugs hide at boundaries** - Test edges, not middles
 - ✓ **Know your float boundaries** - `sys.float_info.max`, `math.inf`, `math.nan`
 - ✓ **Use `pytest.approx()`** - The correct way to compare floats
 - ✓ **Test systematically** - Cover all boundary cases
 - ✓ **Write descriptive tests** - Future you will thank you!
- Next:** Apply these techniques to your Road Profile Viewer project! 🚀



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Questions?

GREAT TESTS CATCH BUGS AT
THE BOUNDARIES 