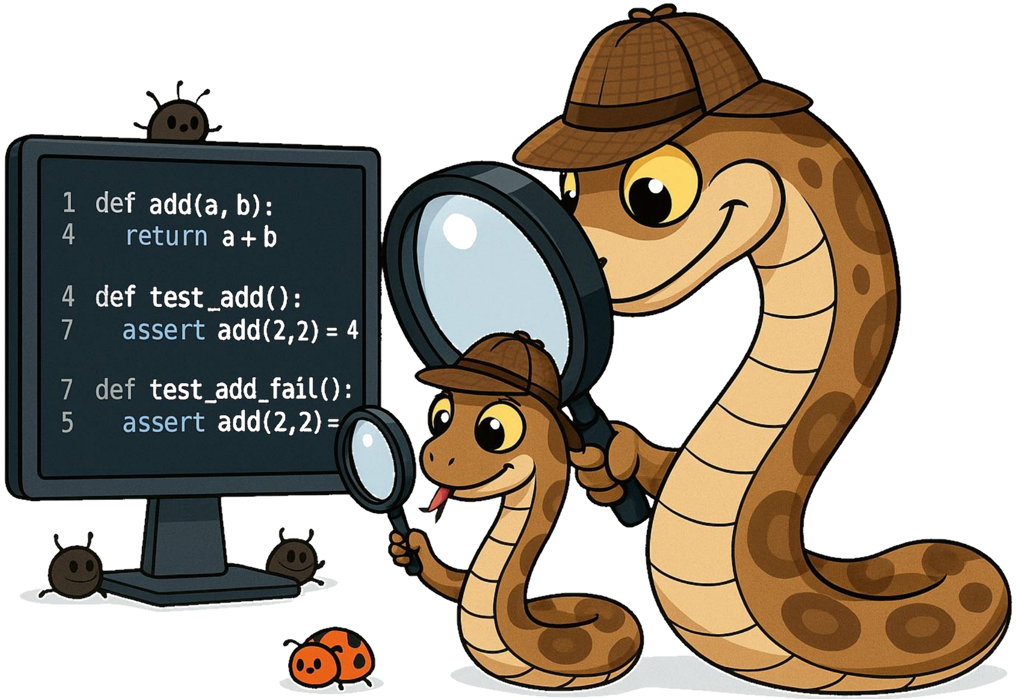
Testing the Tests

Assess and Improve Your Python Testing Code



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

def test_add():
  assert add(2,2) = 4

def test_add_fail():
  assert add(2,2) =
```

Stefan Baerisch, stefan@stbaer.com, 2025-06-24

About Stefan

- Software Developer, Product Owner, Freelancer
- I've been using Python since 2005 for a wide range of tasks – from data wrangling and backend services to test automation.
- Location: I'm based in Munich, another city that, like Prague, is famous for its excellent beer
- Get in touch: **stefan@stbaer.com**

What We'll Cover Today

- What Makes a "Good" Test?
  - Exploring the concepts of **effectiveness**, **efficiency**, and **coverage**
- An Introduction to **Mutation Testing**
  - How to "*test your tests*" to find hidden weaknesses.

Good Tests?

Coverage?

Better tests!

Mutation Testing

Last Words

The 5 E's of Test Quality

- The *easy* E's
  - (Some) tests **exist**.
  - The tests are **executed** regularly .
  - There are **enough** tests (either in total or in terms of coverage)
- The *essential* E's
  - **Efficiency**: Do they run quickly and provide fast feedback?
  - **Effectiveness**: Do the tests actually find bugs?

Test Efficiency

An efficient test an be created, executed and maintained with minimal effort

- **Automation is necessary for efficiency!** Automated tests are repeatable and more efficient. Manual testing is expensive and prone to error.
- Only **Fast Tests run often**: The more often a test runs, the earlier it finds errors
- **Tests need to be maintainable.** Otherwise, they get left behind

Implementation Efficiency & Test Quality Pyramid

- A few End-to-End Tests:
  - Complete user workflows from start to finish.
  - Slowest tests, often involving UI automation.
  - (more) Expensive to create and maintain.
- Some Integration Tests:
  - Interaction between components, Contracts between services
  - May involve real databases or APIs.
- Many Unit Tests:
  - Single functions or methods in isolation.
  - Extremely fast, simple to implement when system is designed with testing in mind.

Effectiveness - Function Test Characteristics

An effective test suite locates bugs

- **Isolation**: A failure in one test should never cause another to fail.
- **Single Responsibility**: test function verifies one specific behaviour.
- **Meaningful Assertions and Expected Values**
- **Coverage?**: You can only find bugs in the code you actually test.

Good Tests?

Coverage?

Better tests!

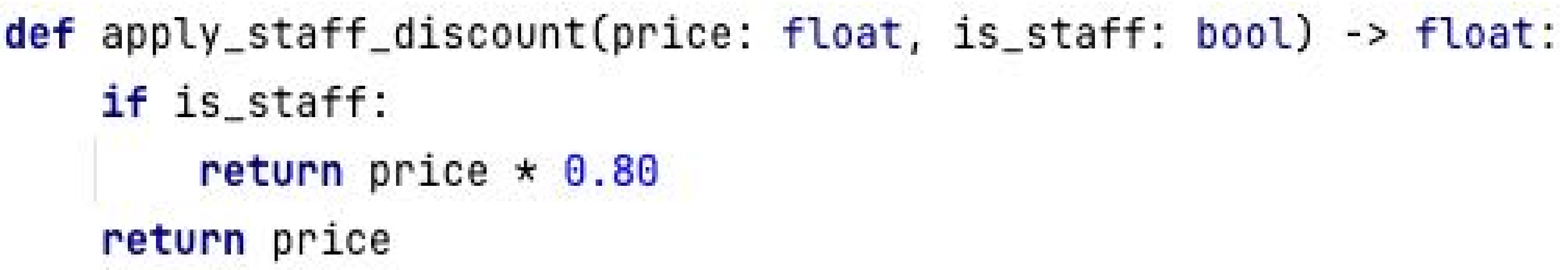
Mutation Testing

Last Words

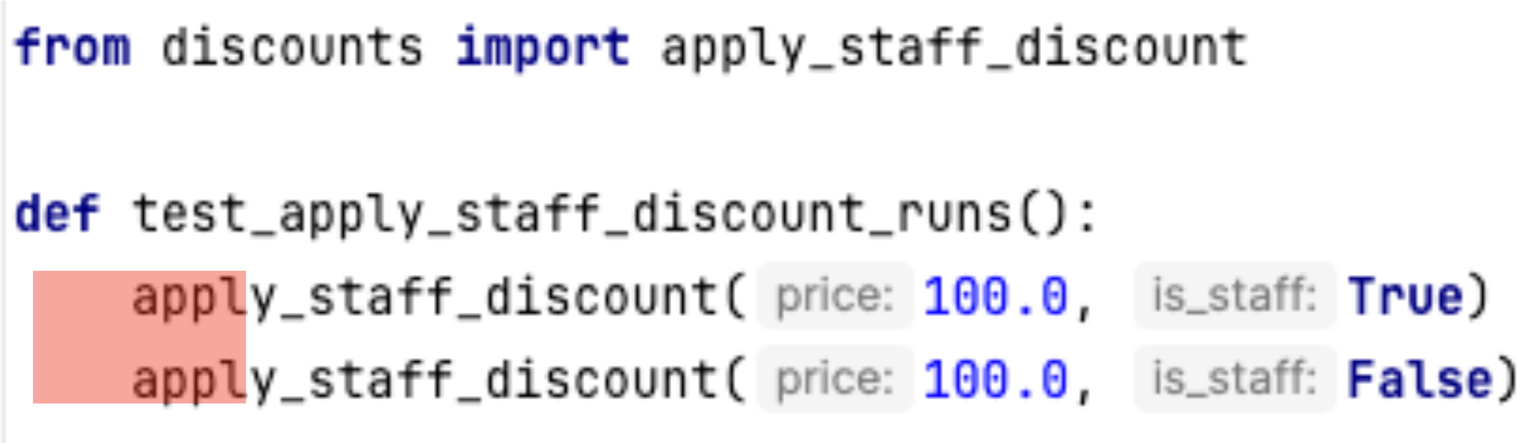
What is Test Coverage?

- **What it is:** A measurement of which lines, branches, and input combinations of your source code are executed by your test suite.
  - Mostly used on the unit test level
- **What it's good for**: Getting an idea of the portion of the code that is tested
- **What it's not good for:** Telling you that your tests are effective

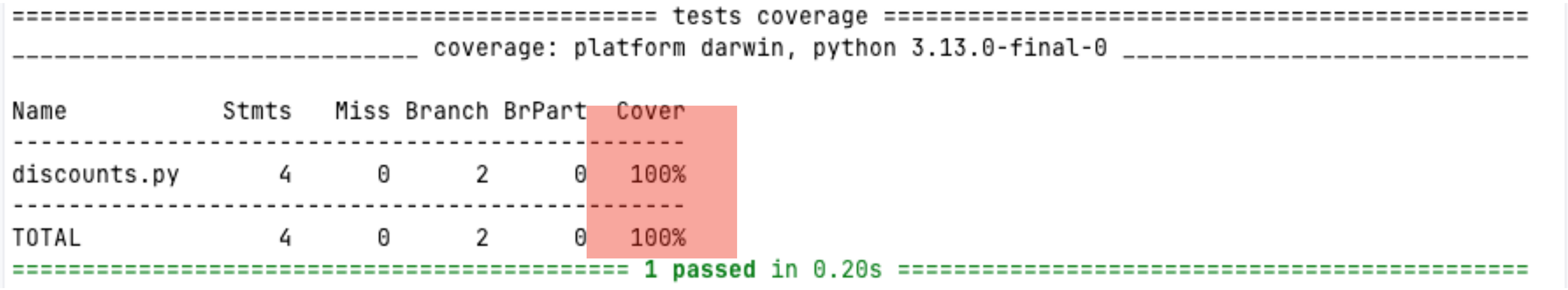
Coverage Problem 1 : Misleading Percentages



```
def apply_staff_discount(price: float, is_staff: bool) -> float:
    if is_staff:
        return price * 0.80
    return price
```



```
from discounts import apply_staff_discount


def test_apply_staff_discount_runs():
    apply_staff_discount( price: 100.0, is_staff: True)
    apply_staff_discount( price: 100.0, is_staff: False)
```

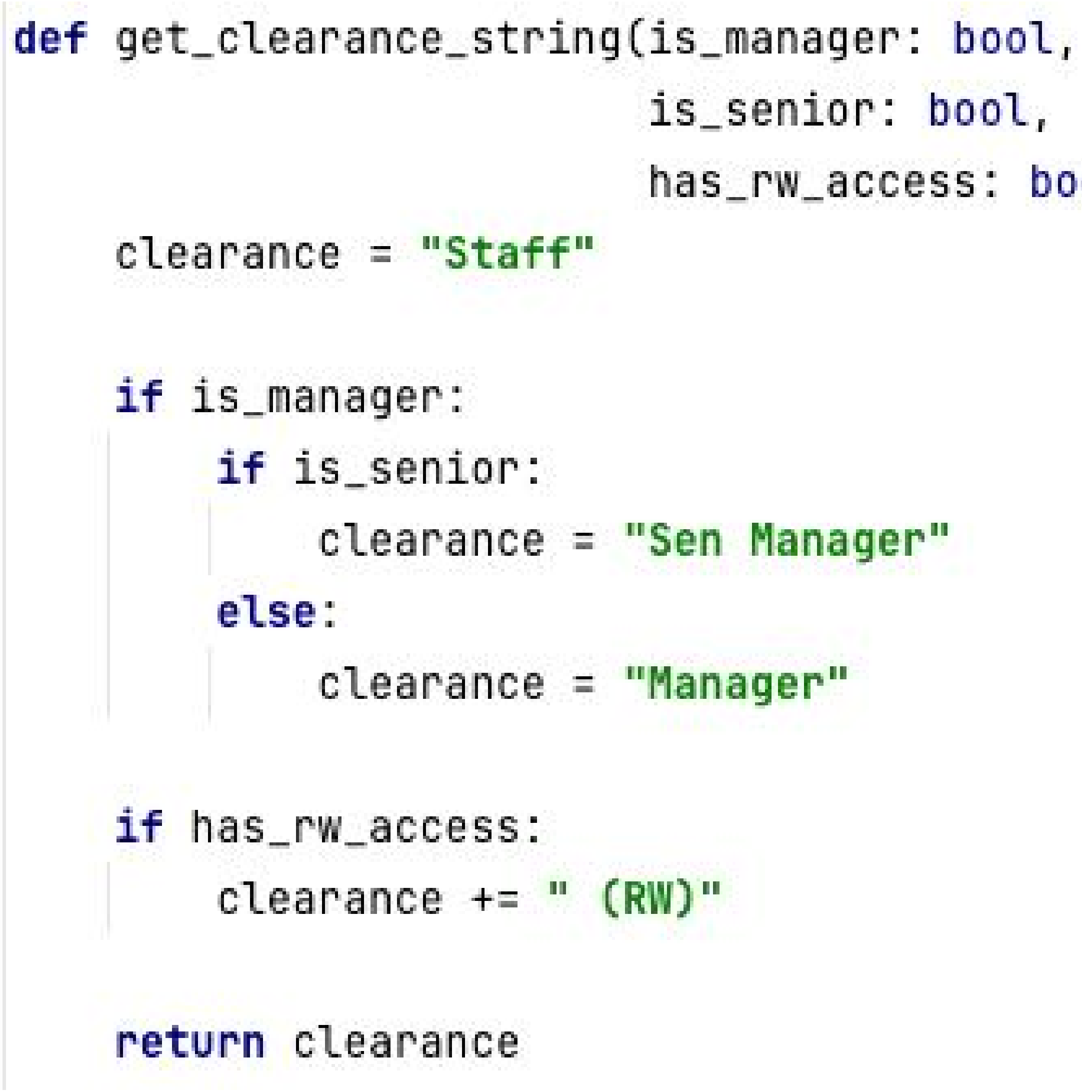


```
============================================== tests coverage ===============================================
_____________________________ coverage: platform darwin, python 3.13.0-final-0 ______________________________

Name              Stmts   Miss Branch BrPart  Cover
-----------------------------------------------------
discounts.py          4      0      2      0   100%
-----------------------------------------------------
TOTAL                 4      0      2      0   100%
============================================ 1 passed in 0.20s =============================================
```

Coverage Problem 2: Misleading Percentages

- **Goodhart's Law**, expanded by Marilyn Strathern: ***"When a measure becomes a target, it ceases to be a good measure"***
- If you judge developers by lines of code, you will get lines of code. Not working software.
- If you judge by coverage, you will get coverage, not effective tests

Coverage Types

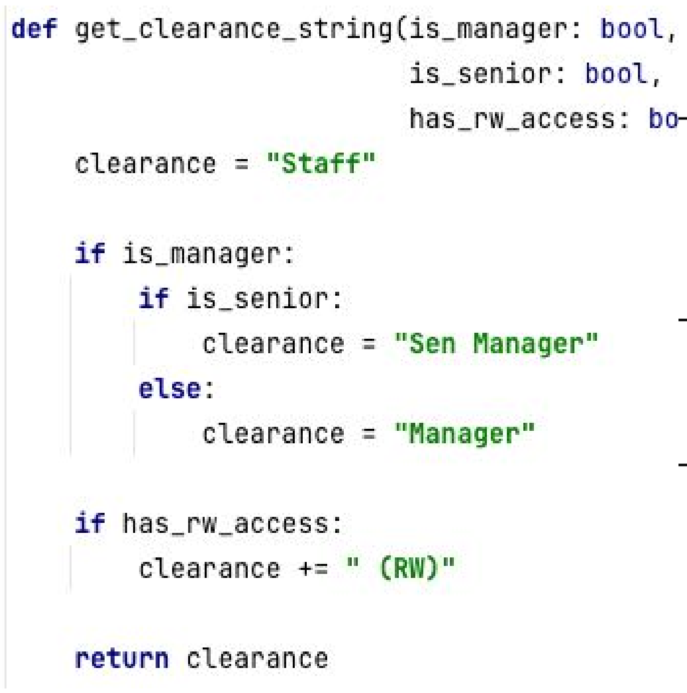


```
def get_clearance_string(is_manager: bool,
                         is_senior: bool,
                         has_rw_access: bo
    clearance = "Staff"

    if is_manager:
        if is_senior:
            clearance = "Sen Manager"
        else:
            clearance = "Manager"

    if has_rw_access:
        clearance += " (RW)"

    return clearance
```

- **Line Coverage**: Has every executable line of code been run at least once?"
- **Branch Coverage**: For every if/while statement, have both paths been executed in tests?
- (Simple) **Condition Coverage**: For a complex decision like if `A and B`:, has each individual condition (A, B) been evaluated to both True and False?
- **Path Coverage**: Has every single possible route through a function been executed?

Coverage Types Example

- How many tests do we need to achieve coverage?



```
def get_clearance_string(is_manager: bool,
                         is_senior: bool,
                         has_rw_access: bo
    clearance = "Staff"

    if is_manager:
        if is_senior:
            clearance = "Sen Manager"
        else:
            clearance = "Manager"

    if has_rw_access:
        clearance += " (RW)"

    return clearance
```

Coverage Criteria	Parameter Value			Expected Result
	is_manager	is_senior	has_rw_access	
Statement Coverage	TRUE	TRUE	TRUE	Sen Manager (RW)
	TRUE	FALSE	FALSE	Manager
Branch Coverage	TRUE	TRUE	TRUE	Sen Manager (RW)
	TRUE	FALSE	FALSE	Manager
	FALSE	FALSE	FALSE	Staff
Condition Coverage	TRUE	TRUE	TRUE	Sen Manager (RW)
	FALSE	FALSE	FALSE	Staff
Path Coverage	FALSE	FALSE	FALSE	Staff
	FALSE	FALSE	TRUE	Staff (RW)
	TRUE	FALSE	FALSE	Manager
	TRUE	FALSE	TRUE	Manager (RW)
	TRUE	TRUE	FALSE	Sen Manager
	TRUE	TRUE	TRUE	Sen Manager (RW)

Good Tests?

Coverage?

Better tests!

Mutation Testing

Last Words

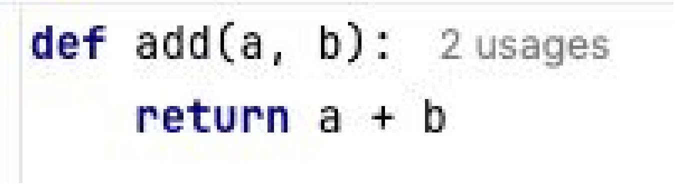
Beyond Basic Coverage - Getting Effective Tests (1)

- **Coverage is necessary, not sufficient**: Missing Coverage means no effective Tests
- **Use Path Coverage as a Thinking Aid**
- **How to improve our tests?**
- Approach 1: Get tests for as **many relevant usage scenarios and test values** as possible
  - This will also raise condition / path coverage
- Approach 2: **Test the tests themselves**

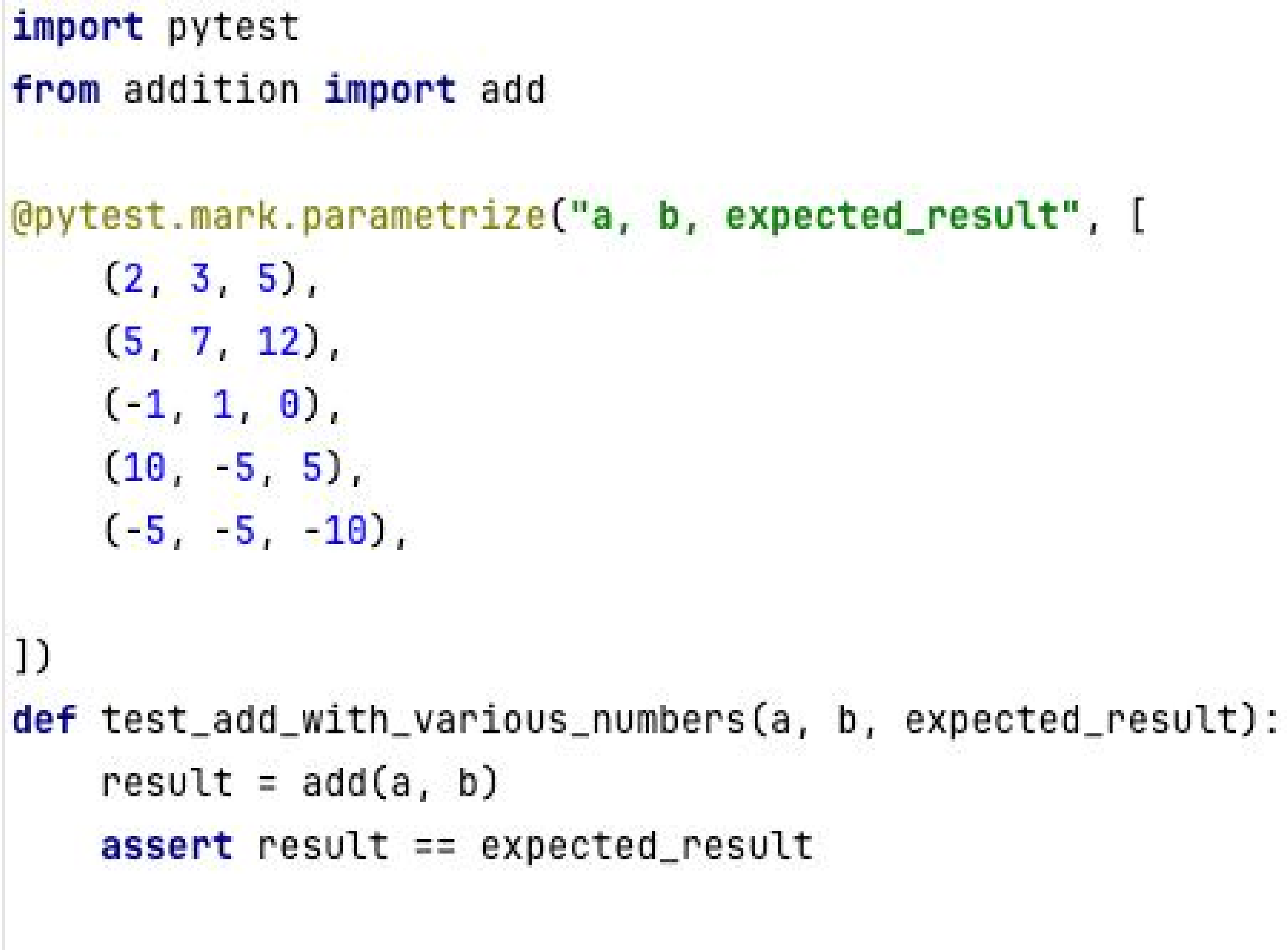
Beyond Basic Coverage - Getting Effective Tests (2)

	Manual	Automated
Get more Testing Inputs Tested	Table-Driven Testing / Test Parametrisation Behaviour Driven Testing (BDD)	Property-Based Testing
Check if Tests find Bugs		Mutation Testing

Better Test Values: Table-Driven Testing

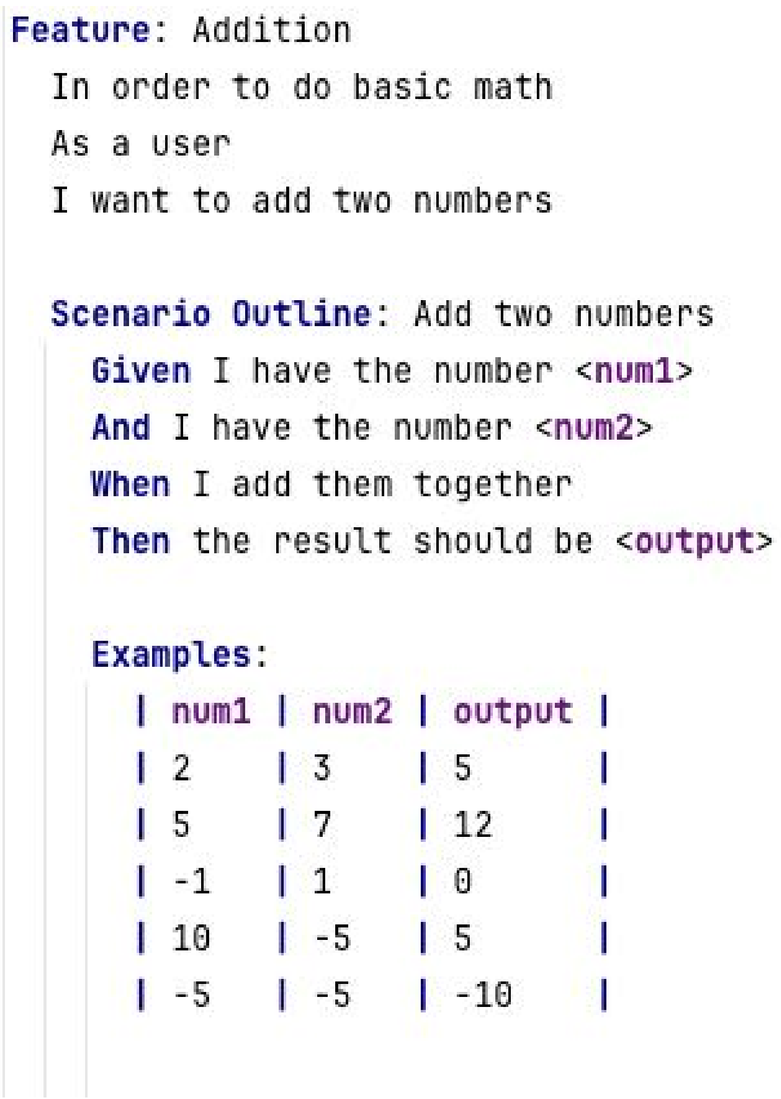


```
def add(a, b):  2 usages
    return a + b
```

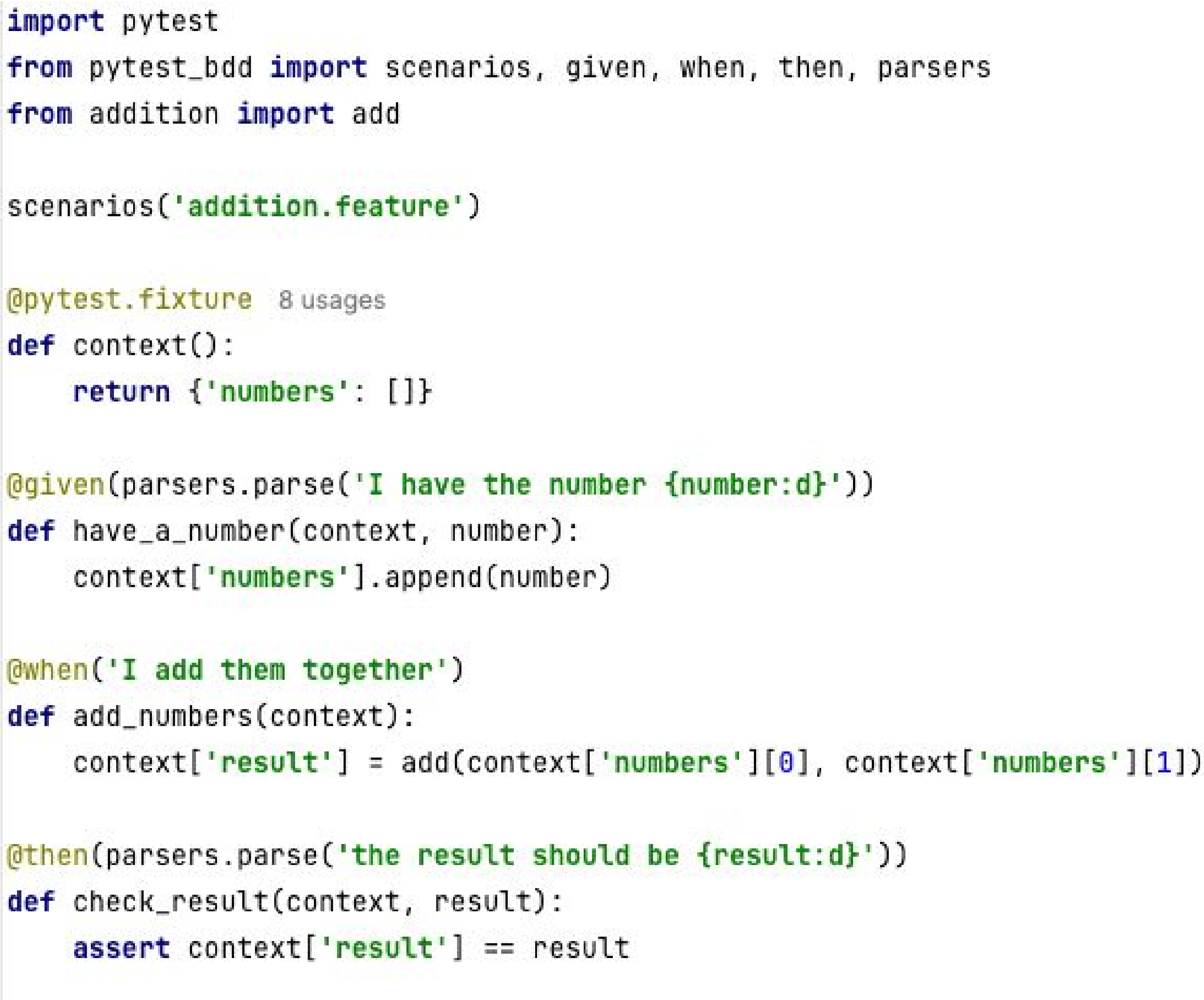


```
import pytest
from addition import add

@pytest.mark.parametrize("a, b, expected_result", [
    (2, 3, 5),
    (5, 7, 12),
    (-1, 1, 0),
    (10, -5, 5),
    (-5, -5, -10),

])
def test_add_with_various_numbers(a, b, expected_result):
    result = add(a, b)
    assert result == expected_result
```

Better Test Values: Behaviour-Driven Development

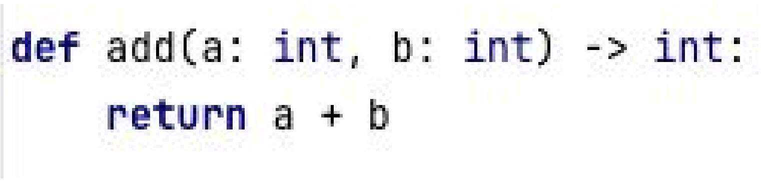


```
Feature: Addition
  In order to do basic math
  As a user
  I want to add two numbers

  Scenario Outline: Add two numbers
    Given I have the number <num1>
    And I have the number <num2>
    When I add them together
    Then the result should be <output>

    Examples:
      | num1 | num2 | output |
      | 2    | 3    | 5      |
      | 5    | 7    | 12     |
      | -1   | 1    | 0      |
      | 10   | -5   | 5      |
      | -5   | -5   | -10    |
```

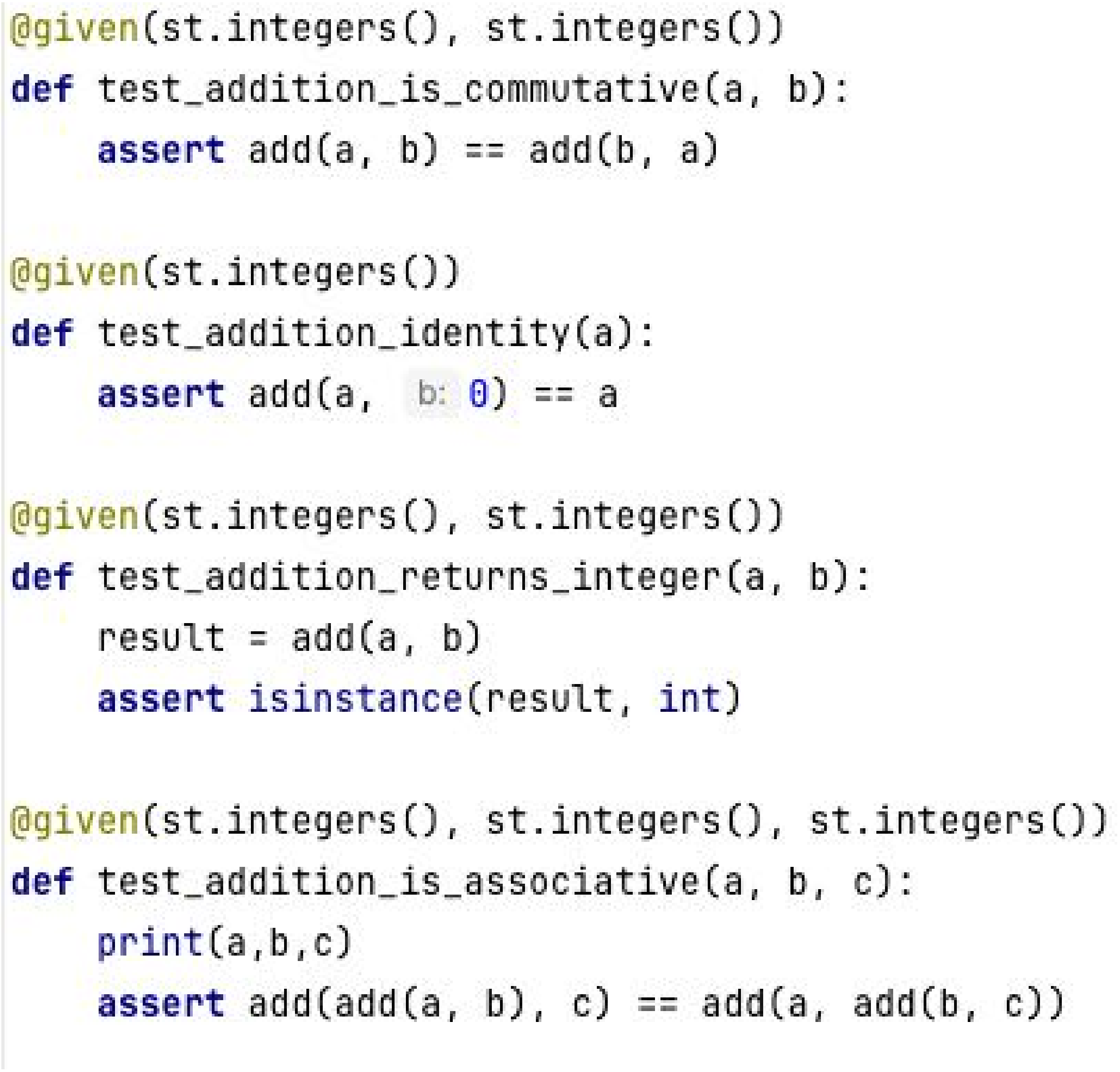


```python
import pytest
from pytest_bdd import scenarios, given, when, then, parsers
from addition import add

scenarios('addition.feature')

@pytest.fixture  8 usages
def context():
    return {'numbers': []}

@given(parsers.parse('I have the number {number:d}'))
def have_a_number(context, number):
    context['numbers'].append(number)

@when('I add them together')
def add_numbers(context):
    context['result'] = add(context['numbers'][0], context['numbers'][1])

@then(parsers.parse('the result should be {result:d}'))
def check_result(context, result):
    assert context['result'] == result
```

Better Test Values: Property-Based Testing



```python
def add(a: int, b: int) -> int:
    return a + b
```



```python
@given(st.integers(), st.integers())
def test_addition_is_commutative(a, b):
    assert add(a, b) == add(b, a)

@given(st.integers())
def test_addition_identity(a):
    assert add(a,  b: 0) == a

@given(st.integers(), st.integers())
def test_addition_returns_integer(a, b):
    result = add(a, b)
    assert isinstance(result, int)

@given(st.integers(), st.integers(), st.integers())
def test_addition_is_associative(a, b, c):
    print(a,b,c)
    assert add(add(a, b), c) == add(a, add(b, c))
```

Good Tests?

Coverage?

Better tests!

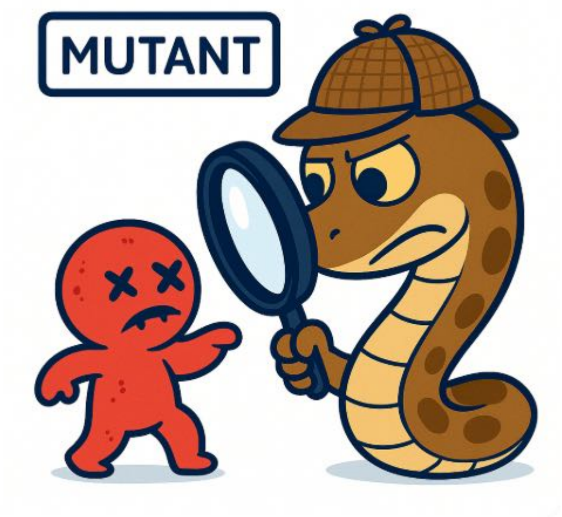
Mutation Testing

Last Words

The Idea Behind Mutation Testing

Mutation Testing **mutates (changes) your code**

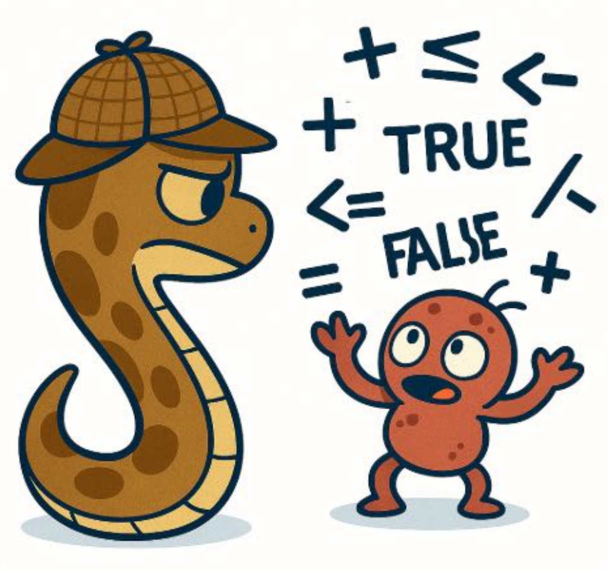
It **introduces small mistakes**, then **runs the tests** against the new, mutated code.



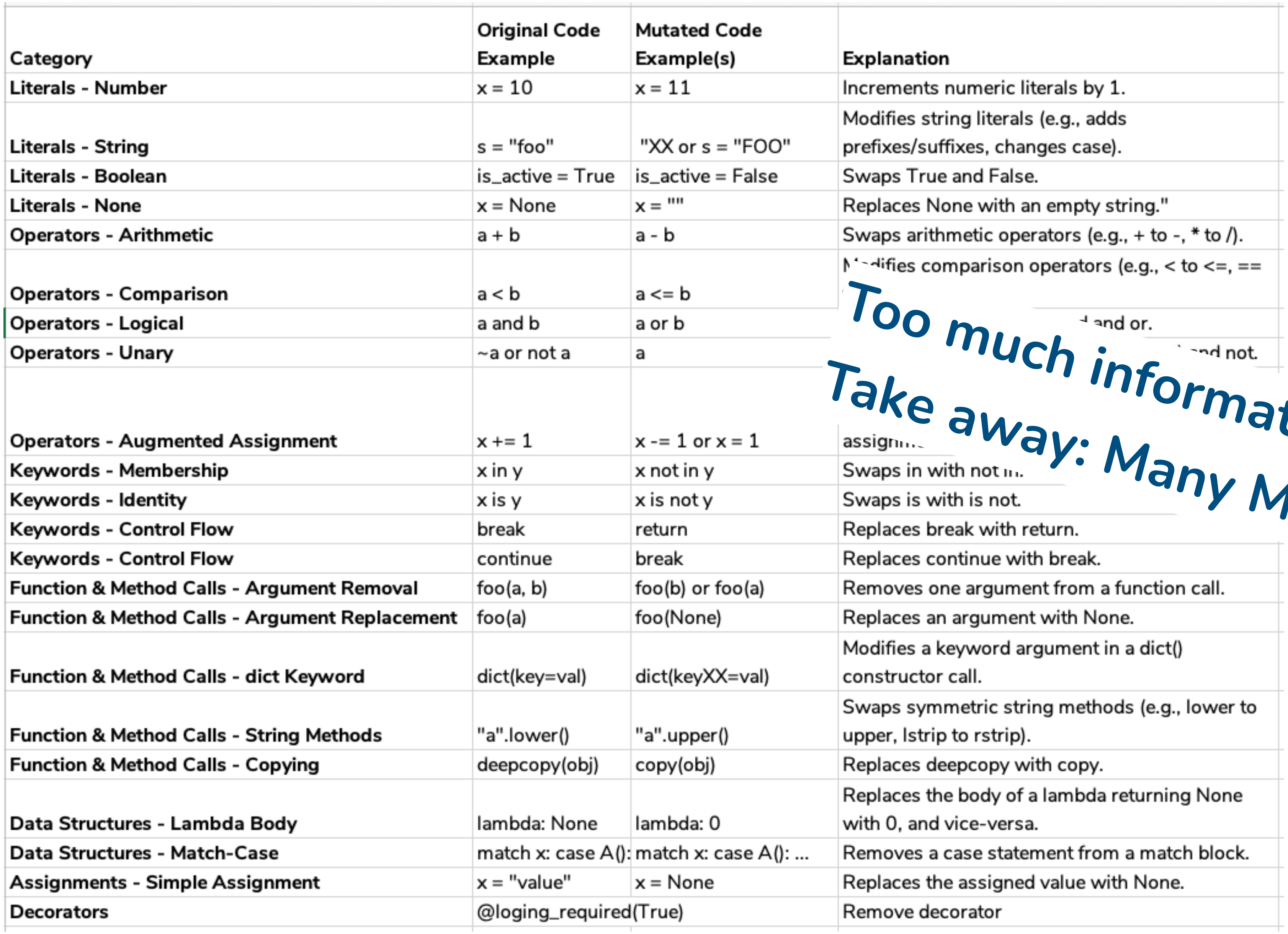
Types of Mutations

- **Basic Operators:**
  - *Arithmetic* Operator Replacement: + → -, * → /
  - *Relational* Operator Replacement: > → >=, == → !=
  - *Logical* Operator Replacement: and → or
  - *Constant* Replacement: True → False, 0 → 1, "hello" → ""

- **Advanced Operators:**
  - *Statement Deletion*: Remove an entire line of code.
  - *Decorator Removal*: Remove an @decorator.
  - *Keyword Replacement*: break → continue
  - …



mutmut Mutation Overview

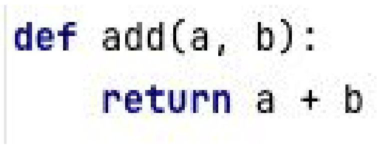


Category	Original Code Example	Mutated Code Example(s)	Explanation
Literals - Number	x = 10	x = 11	Increments numeric literals by 1.
Literals - String	s = "foo"	"XX or s = "FOO"	Modifies string literals (e.g., adds prefixes/suffixes, changes case).
Literals - Boolean	is_active = True	is_active = False	Swaps True and False.
Literals - None	x = None	x = ""	Replaces None with an empty string."
Operators - Arithmetic	a + b	a - b	Swaps arithmetic operators (e.g., + to -, * to /).
Operators - Comparison	a < b	a <= b	Modifies comparison operators (e.g., < to <=, ==
Operators - Logical	a and b	a or b	and or.
Operators - Unary	~a or not a	a	and not.
Operators - Augmented Assignment	x += 1	x -= 1 or x = 1	assign
Keywords - Membership	x in y	x not in y	Swaps in with not in.
Keywords - Identity	x is y	x is not y	Swaps is with is not.
Keywords - Control Flow	break	return	Replaces break with return.
Keywords - Control Flow	continue	break	Replaces continue with break.
Function & Method Calls - Argument Removal	foo(a, b)	foo(b) or foo(a)	Removes one argument from a function call.
Function & Method Calls - Argument Replacement	foo(a)	foo(None)	Replaces an argument with None.
Function & Method Calls - dict Keyword	dict(key=val)	dict(keyXX=val)	Modifies a keyword argument in a dict() constructor call.
Function & Method Calls - String Methods	"a".lower()	"a".upper()	Swaps symmetric string methods (e.g., lower to upper, lstrip to rstrip).
Function & Method Calls - Copying	deepcopy(obj)	copy(obj)	Replaces deepcopy with copy.
Data Structures - Lambda Body	lambda: None	lambda: 0	Replaces the body of a lambda returning None with 0, and vice-versa.
Data Structures - Match-Case	match x: case A():	match x: case A(): ...	Removes a case statement from a match block.
Assignments - Simple Assignment	x = "value"	x = None	Replaces the assigned value with None.
Decorators	@loging_required(True)		Remove decorator

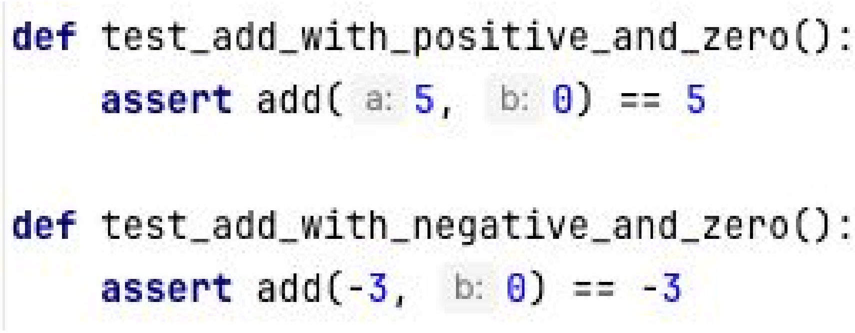
Too much information 😉

Take away: Many Mutations

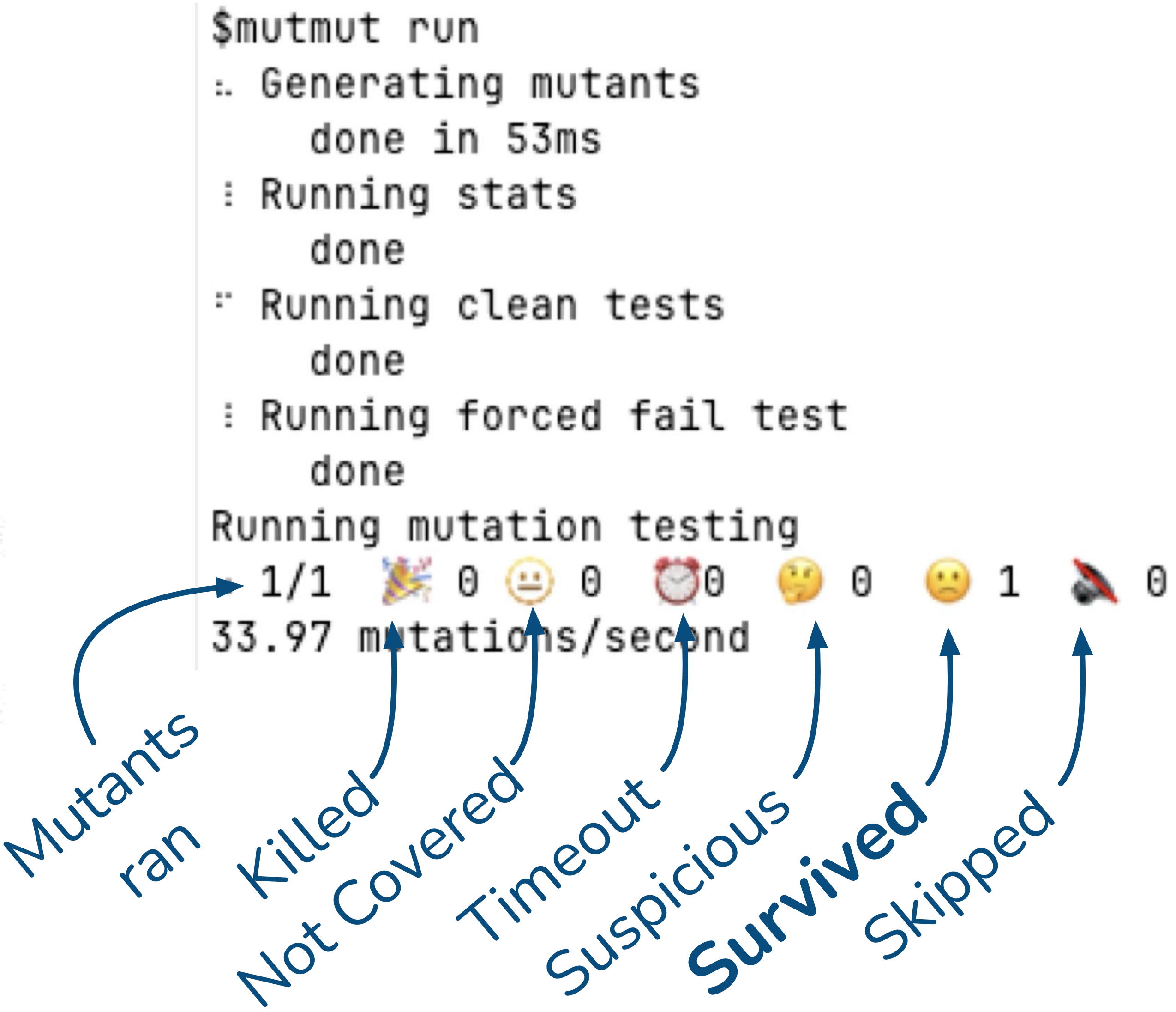
Mutation Testing Example - Survival



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



```
def test_add_with_positive_and_zero():
    assert add( a: 5,  b: 0) == 5

def test_add_with_negative_and_zero():
    assert add(-3,  b: 0) == -3
```

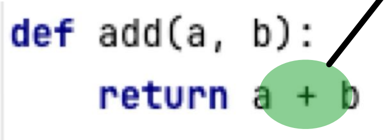


```
$mutmut run
⠦ Generating mutants
    done in 53ms
⠇ Running stats
    done
⠏ Running clean tests
    done
⠇ Running forced fail test
    done
Running mutation testing
⠸ 1/1  🎉 0  😐 0  ⏰0  🤔 0  🙁 1  🔇 0
33.97 mutations/second
```

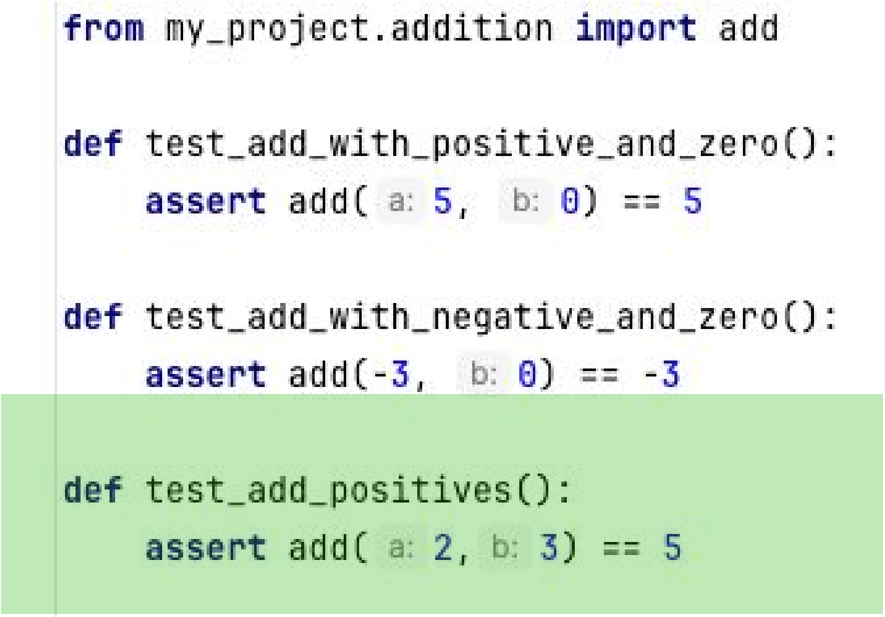
Mutation Testing Example - Killed

Mutate to -

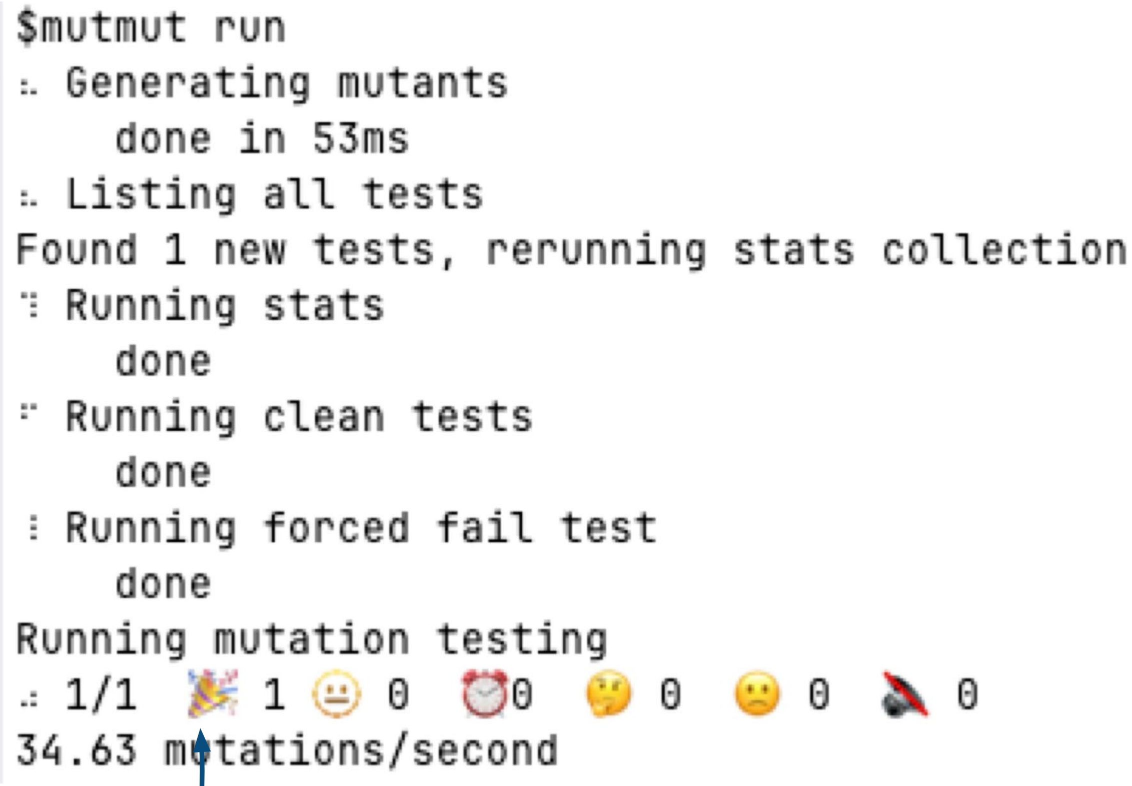
(a minus b)



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



```
from my_project.addition import add

def test_add_with_positive_and_zero():
    assert add( a: 5,  b: 0) == 5

def test_add_with_negative_and_zero():
    assert add(-3,  b: 0) == -3

def test_add_positives():
    assert add( a: 2, b: 3) == 5
```



```
$mutmut run
⠦ Generating mutants
    done in 53ms
⠦ Listing all tests
Found 1 new tests, rerunning stats collection
⠹ Running stats
    done
⠙ Running clean tests
    done
⠸ Running forced fail test
    done
Running mutation testing
⠴ 1/1  🎉 1 😐 0  ⏰0  🤔 0  🙁 0  🔇 0
34.63 mutations/second
```

Killed

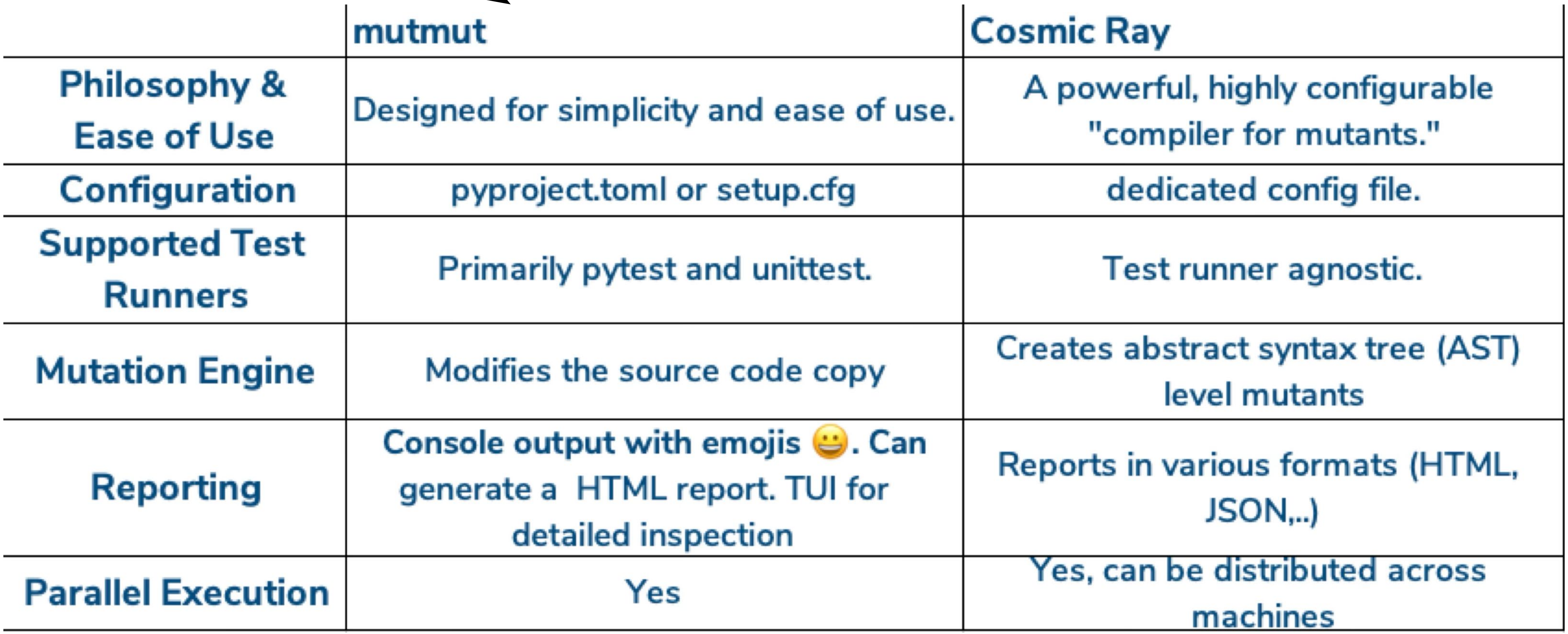
The Mutation Testing Cycle

- **Configure & Run**: Set up your mutation testing tool and run it on your codebase.
- **Analyze Survivors**: The tool reports which mutants survived.
- **Improve Tests**: For each meaningful survivor, write a new test that "kills" it.
- **Repeat**: Re-run the mutation tool

Mutation Testing Library Overview

Start here

Go here if curious or a missing something



	mutmut	Cosmic Ray
Philosophy & Ease of Use	Designed for simplicity and ease of use.	A powerful, highly configurable "compiler for mutants."
Configuration	pyproject.toml or setup.cfg	dedicated config file.
Supported Test Runners	Primarily pytest and unittest.	Test runner agnostic.
Mutation Engine	Modifies the source code copy	Creates abstract syntax tree (AST) level mutants
Reporting	Console output with emojis 😀. Can generate a HTML report. TUI for detailed inspection	Reports in various formats (HTML, JSON,..)
Parallel Execution	Yes	Yes, can be distributed across machines

Limitations of Mutation Testing

- **The Equivalent Mutant Problem:**
  - An "equivalent mutant" change the outcom.
  - Toy example: **x = y; return x*x;**
  - is mutated to **x = - y return x*x**
  - These mutants can never be killed,.
- **Computational Cost**: Running the test suite for every mutant can be very slow.
- **Unproductive Mutants**: Some mutants are technically killable but not useful to kill. For example, text of a log messages

Mutation Testing Performance Considerations

- **Don't Run on Every Commit**
- **Target Critical Code**: Focus your efforts on the most critical, complex, or high-risk modules first.
- **Use Coverage Data:** Tools like mutmut can use coverage data to only run the tests that actually execute the mutated line of code.
- **Incremental / Diff-Based Analysis**: The most advanced strategy for CI is to only mutate the lines of code that have changed in a specific pull request.

Good Tests?

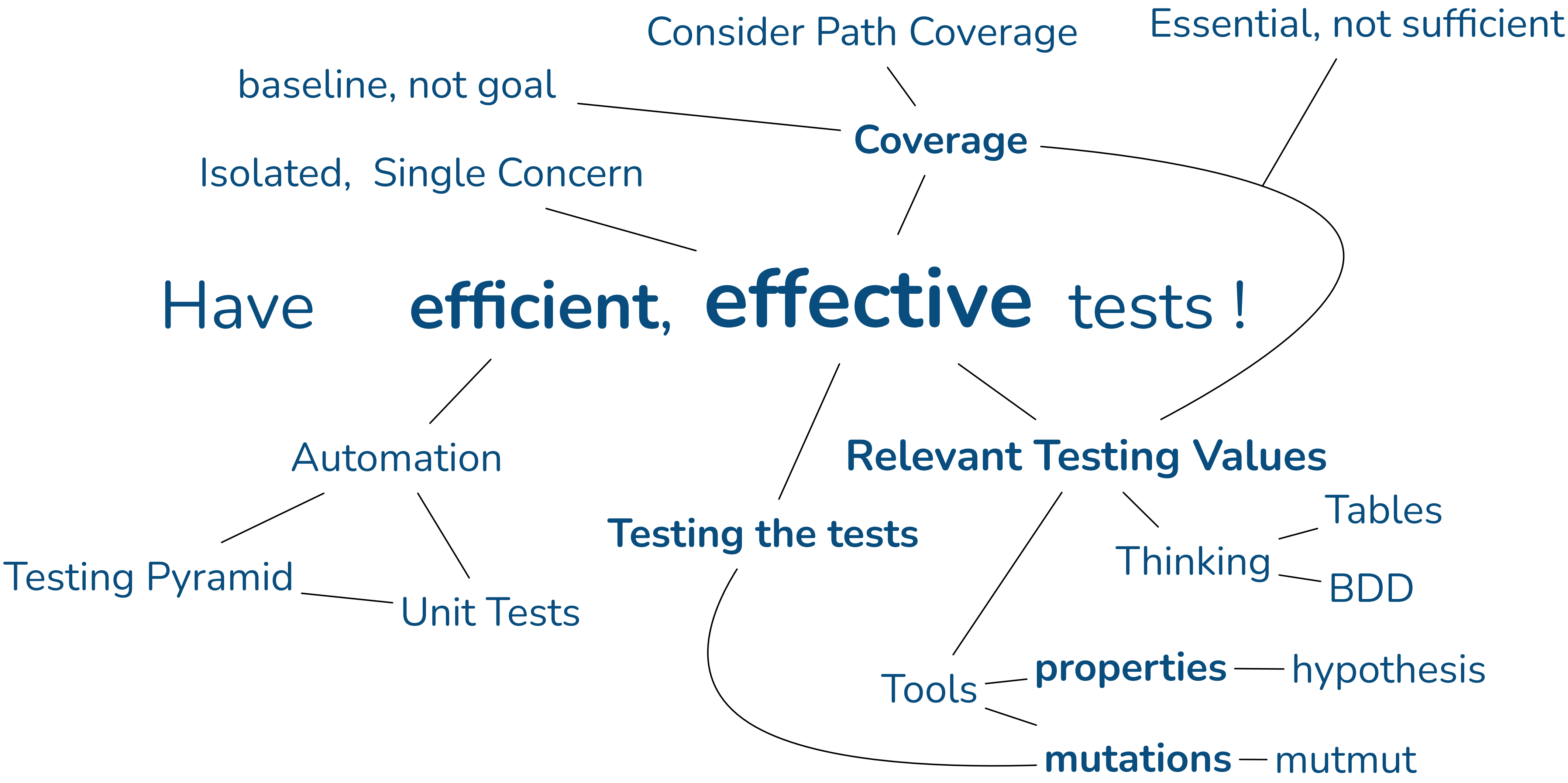
Coverage?

Better tests!

Mutation Testing

Last Words

Conclusion



Consider Path Coverage

Essential, not sufficient

baseline, not goal

Coverage

Isolated, Single Concern

Have **efficient, effective** tests !

Automation

Relevant Testing Values

Testing the tests

Tables

Thinking

BDD

Testing Pyramid

Unit Tests

Tools

properties — hypothesis

mutations — mutmut

Call to Action

- Try **Mutation Testing (mutmut)**
- Try **property-based testing (hypothesis)**
- Write a **testing table**
- **Check the path coverage** of your favourite complex function

Thank you!

Stefan Baerisch, stefan@stbaer.com, 2025-06-24