# Testing

*Writing reliable code through automated testing*

# What is testing?

# What is testing?

> Software testing is the act of checking whether software satisfies expectations.

– Wikipedia

CC-BY 4.0

# Why?

# Why?

## Update code without fear

- Detect programming errors when they are introduced
- Reduce time for debugging
- Write cleaner code
- Make sure bugs don’t return

# Lecture Overview

- What to test
- Writing tests
- Running tests

# What to test

- Functioning
- Performance
- Usability (incl accessibility)

# Functional tests

- Ensure code runs
- Ensure correct results
  - Even for error cases
- Ensure interfaces match (think a US plug in a German socket)

CC-BY 4.0

# Performance testing

- Avoid performance degradation over time
- Useful for catching algorithmic regressions
- Measure time for one execution (if short, many)
- Challenge: Find a test that runs fast but catches real problems
- Set realistic thresholds that won't fail due to system load

# Usability testing

- Use/reuse your own modules
- Watch users work with your code
  - From asking a colleague to eye tracking, …

 | CC-BY 4.0

# What not to test

- Third-party libraries - they usually are covered upstream
- Trivial one-liners – you don’t need 100% coverage[1] for the sake of coverage

1. https://en.wikipedia.org/wiki/Code_coverage

# Levels of testing

- Unit tests
- Integration tests
- System tests

 | CC-BY 4.0

# Unit tests: Small, focused tests

- Isolated and independent of each other
- Usually test just one function or method
- Short and easy to understand
- Quick and cheap (can run often)

# Unit tests are used for

- Testing pure functions with no side effects
- Validating logic
- Testing edge cases and error conditions

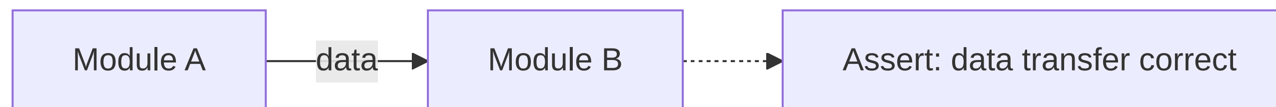
# Simple unit test example

```
1 def add(a, b):
2     """Add two numbers together."""
3     return a + b
```

```
1 def test_add():
2     # Arrange
3     a = 1
4     b = 2
5 
6     # Act
7     c = add(a, b)
8 
9     # Assert
10    assert c == 3 # ensure expression is true, raise AssertionError otherwi
```

# Integration tests

- Ensure that different modules work well together
- Verify interfaces between components
- Test data flow between modules
- Usually more expensive than unit tests



# Integration tests are used for

- Testing database interactions
- Testing file I/O operations
- Testing API integrations

*Do not test with real data/systems*

# System tests

- Verify the behavior of the full application end-to-end
  - Without checking intermediate states
- Most expensive but closest to real usage
- e.g. a small setup of a climate model

*Your production is **not** a system test!*

# System tests are used for

- Testing complete scientific workflows
- Validating climate model runs (e.g., ICON buildbot tests)
- Regression testing of output files

 | CC-BY 4.0

# When should you write a test

# When to write a test

- Before writing a function (test-driven development)
- After writing a function
- When finding a bug (regression testing)
- Before modifying (untested) code

# Test driven development

- Define expected behavior first
- Write minimal code to pass
- Take expected behavior one step further and repeat cycle

 | CC-BY 4.0

# Tests written after coding a function

- Verify it works as intended
- Document expected behavior
- Ensure behavior will be preserved in the future

CC-BY 4.0

# Regression testing

- Ensure the bug never returns
- Add test that fails with bug, passes with fix

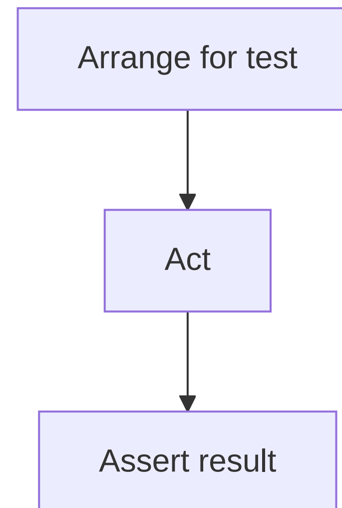
# Before modifying code

- Ensure you understood the function
- Ensure changes don’t break existing functionality
- Refactor with confidence

# Writing tests

CC-BY 4.0

# The AAA pattern




The entire process should run fully automatically.

# Example integration test

Testing that a writer and reader work together correctly:

```
import pytest
from io_module import save_results, load_results

def test_save_load_roundtrip(tmp_path):
    data = [1.0, 2.5, 3.14]                   # Arrange: data
    path = tmp_path / "results.txt"           # Arrange: temp file

    save_results(path, data)                  # Act: write
    result = load_results(path)               # Act: read back

    assert result == data         # Assert: data survived the round-trip
```

(definition of `io_module`)

# Inputs for functions to be tested

- Standard cases
- Corner cases / Special cases
- Extreme cases
- Error provoking arguments

CC-BY 4.0

# Execution

- Ensure you don’t accidentally remove your production data.
- Often expensive functions are replaced with mock functions that return pre-calculated results.

 | CC-BY 4.0

# Result Validation

- Compare with reference result
  - Be careful with exact equality for floats / images / …
- Check properties of the result
  - Conservation laws
  - Mathematical relationships
- Prefer testing properties of functions over exact values

# Discussion and Hands-On

```
1  import math
2
3  def is_prime(n):
4      # check if n is divisible by any number in range 2...√(n) (rounded up)
5      for i in range(2, int(math.ceil(math.sqrt(n)))):
6          if n % i == 0:
7              return False
8      return True
```

- What are good test cases?
- Write tests for the function, and fix any bugs you encounter.

# Running tests

# When to run tests

- When you have written a bit of code
- On every git commit (in CI, see next lectures)
- On demand in pull requests
- Nightly
- …

# How to run tests

- python: pytest
- C++: googletest
- Build-system: CMake / CTest

# pytest

Simple and powerful testing framework for Python

- Automatic test discovery
- Parametrized tests for multiple inputs
- Fixtures for setup/teardown
- Test for exceptions
- Coverage report

# `pytest` looks for tests

- Files: `test_*.py` or `*_test.py`
- Functions: `test_*`

 | CC-BY 4.0

# `pytest` Fixtures

Fixtures provide reusable setup (and teardown) for tests

```python
import pytest

@pytest.fixture
def sample_data(tmp_path):
    # Arrange: create a temporary CSV file
    data_file = tmp_path / "data.csv"
    data_file.write_text("1,2,3\n4,5,6\n")
    return data_file

def test_row_count(sample_data):
    rows = sample_data.read_text().splitlines()
    assert len(rows) == 2
```

- `tmp_path` is a built-in pytest fixture, cleaned up after the test
- Fixtures are shared across tests without duplicating code

 | CC-BY 4.0

# Hands-On: `pytest`

- Integrate your tests for the prime numbers with `pytest`
- Run them (you may need to install it)
- Add further test(s) to the test suite

CC-BY 4.0

# Summary

# Lessons learned

- Testing makes your and the life of the users easier!
- Test variety of environments (think of your colleagues)
- Automate tests
- Test often

# Writing tests

- Every feature needs a test
- Test all possible execution paths (code coverage)
- Also test proper error handling (invalid input)
- Turn bugs into tests (regression testing)
- Try to keep tests lightweight and simple

# Further reading

- Python Testing with Pytest (Brian Okken) for learning how to test in Python
- Working Effectively with Legacy Code (Michael C. Feathers) if you have inherited a pile of code
- Clean Code: A Handbook of Agile Software Craftsmanship (Robert C. Martin) for the full development method
- Test-Driven Development by Example (Kent Beck) hard-core variant for writing good code

# Annex

# IO Module for the integration test

```
1  # io_module.py
2  def save_results(path, data):
3      with open(path, "w") as f:
4          f.write("\n".join(str(x) for x in data))
5
6  def load_results(path):
7      with open(path) as f:
8          return [float(line) for line in f]
```