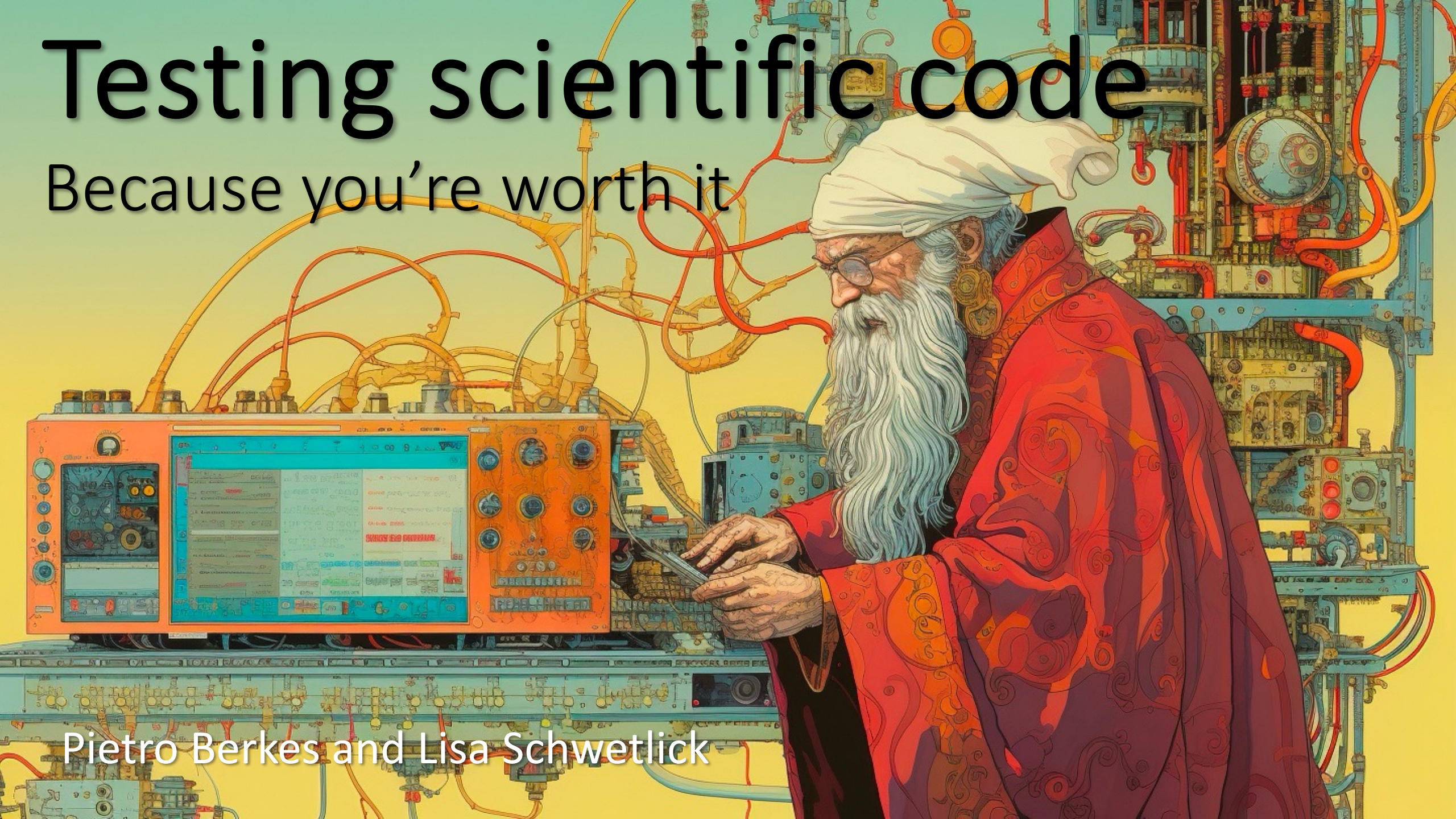
Testing scientific code

Because you're worth it



Pietro Berkes and Lisa Schwetlick

You, as the Master of Research

You start a new project and identify a number of possible leads.

You **quickly develop a prototype** of the most promising ones; once a prototype is finished, you can **confidently decide** whether it is is a dead end, or worth pursuing.

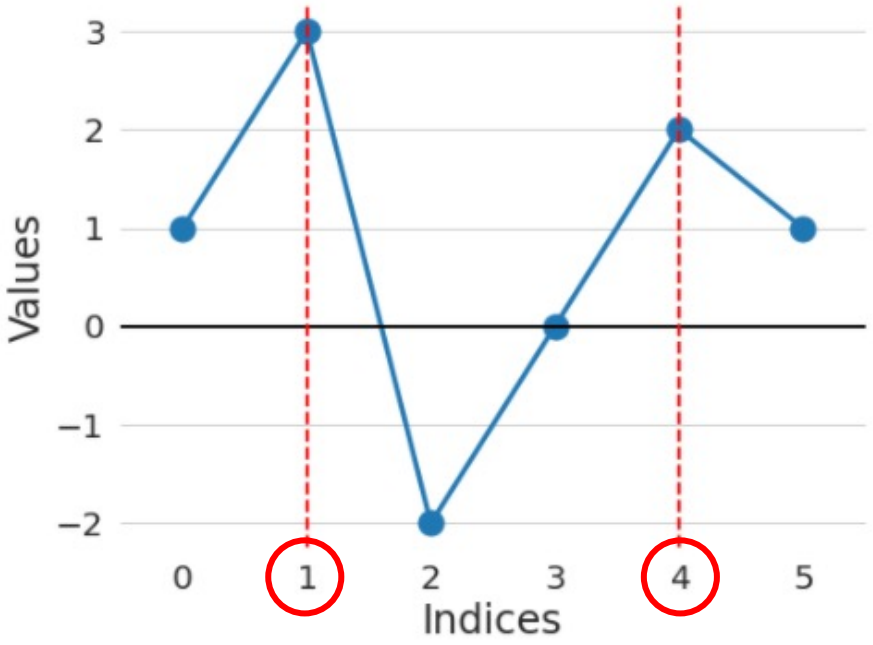
Once you find an idea on which it is worth spending energy, you take the prototype and **easily re-organize and optimize it** so that it scales up to the full size of your problem.

As expected, the scaled-up experiment delivers good results, and your next paper is under way.

March 2024, CC BY-SA 4.0

Warm-up project

- Go to the directory called `hands_on/local_maxima`
- In the file called `local_maxima.py`, write a function `find_maxima` that finds the indices of local maxima in a list of numbers



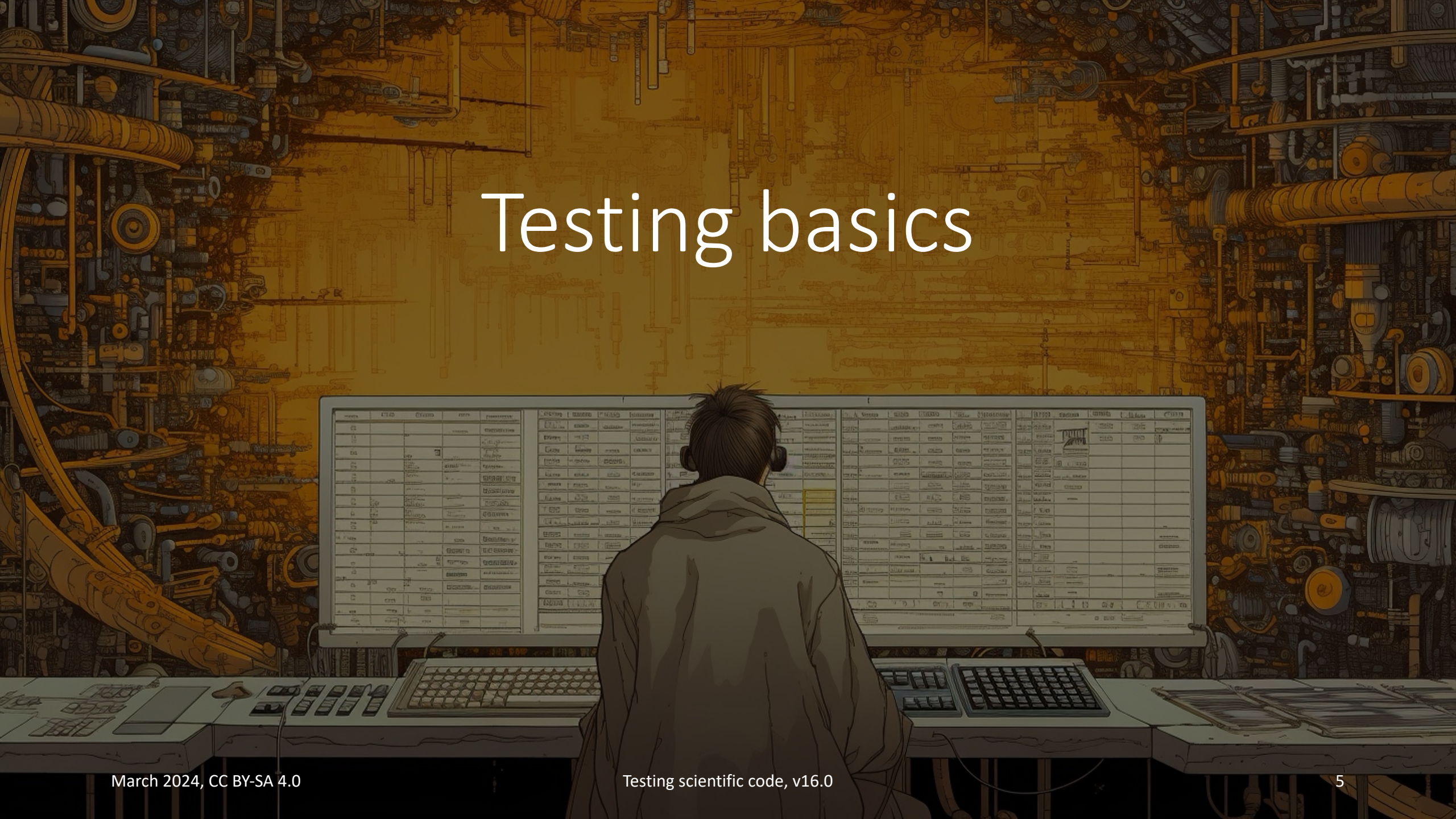
For example,
`find_maxima([1, 3, -2, 0, 2, 1])`
should return
`[1, 4]`

Warm-up project

- Write a function `find_maxima` that finds the indices of local maxima in a list of numbers

- Check your solution with these inputs:
  - Input: [1, 3, -2, 0, 2, 1] Expected result: [1, 4]
  - Input: [4, 2, 1, 3, 1, 5] Expected result: [0, 3, 5]
  - Input: [] Expected result: []
  - Input: [1, 2, 2, 1] Expected result: [1] (or [2], or [1, 2])
  - Input: [1, 2, 2, 3, 1] Expected result: [3]

Testing basics



A test is just another function

- Imagine we wrote this new function, and we wanted to test it

```
def times_3(x):
    """Multiply x by 3.

    Parameters
    ----------
    x : The item to multiply by 3.
    """
    return x * 3
```

Testing frameworks

- The collection of tests written to test a package is called a “test suite”
- Execution of a test suite is automated: external software runs the tests and provides reports and statistics
- Main testing frameworks for python:
  - unittest: in the standard library
  - **pytest: what is most commonly used**

```
============================================= test session starts ==============================================
platform darwin -- Python 3.11.3, pytest-7.3.1, pluggy-1.0.0
collected 2 items

test_first.py::test_times_3_integer PASSED                                                              [ 50%]
test_first.py::test_times_3_string PASSED                                                               [100%]

============================================== 2 passed in 0.00s ===============================================
```

Hands-on!

- Go to `hands_on/first`

1. Discover all tests in all subdirectories
   ```
   pytest -v
   ```
2. Execute all tests in one module
   ```
   pytest -v test_first.py
   ```
3. Execute one single test
   ```
   pytest -v test_first.py::test_times_3_string
   ```

Test suites in Python with pytest

- Writing tests with pytest is simple:
  - Tests are collected in files called `test_abc.py` , which usually contains tests for the functions defined in a corresponding module `abc`
  - Each test is a function called `test_jkl_feature`, and usually it tests feature `feature` of a function called `jkl`
  - Each test tests **one feature** in your code, and checks that it behaves correctly using “assertions”. An exception is raised if it does not work as expected.

Basic structure of test

- A good test is divided in three parts:
  - **Given**: Put your system in the right state for testing
    - Create data, initialize parameters, define constants…
  - **When**: Execute the feature that you are testing
    - Typically, one or two lines of code
  - **Then**: Compare outcomes with the expected ones
    - Define the expected result of the test
    - Set of *assertions* that check that the new state of your system matches your expectations

Assertions

- `assert` statements check that some condition is met, and raise an exception otherwise
- Check that statement is true/false:

```
assert 'Hi'.islower()      => fail
assert not 'Hi'.islower()  => pass
```

- Check that two objects are equal:

```
assert 2 + 1 == 3              => pass
assert [2] + [1] == [2, 1] => pass
assert 'a' + 'b' != 'ab'   => fail
```

- `assert` can be used to compare all sorts of objects, and pytest will take care of producing an appropriate error message

What a good test looks like

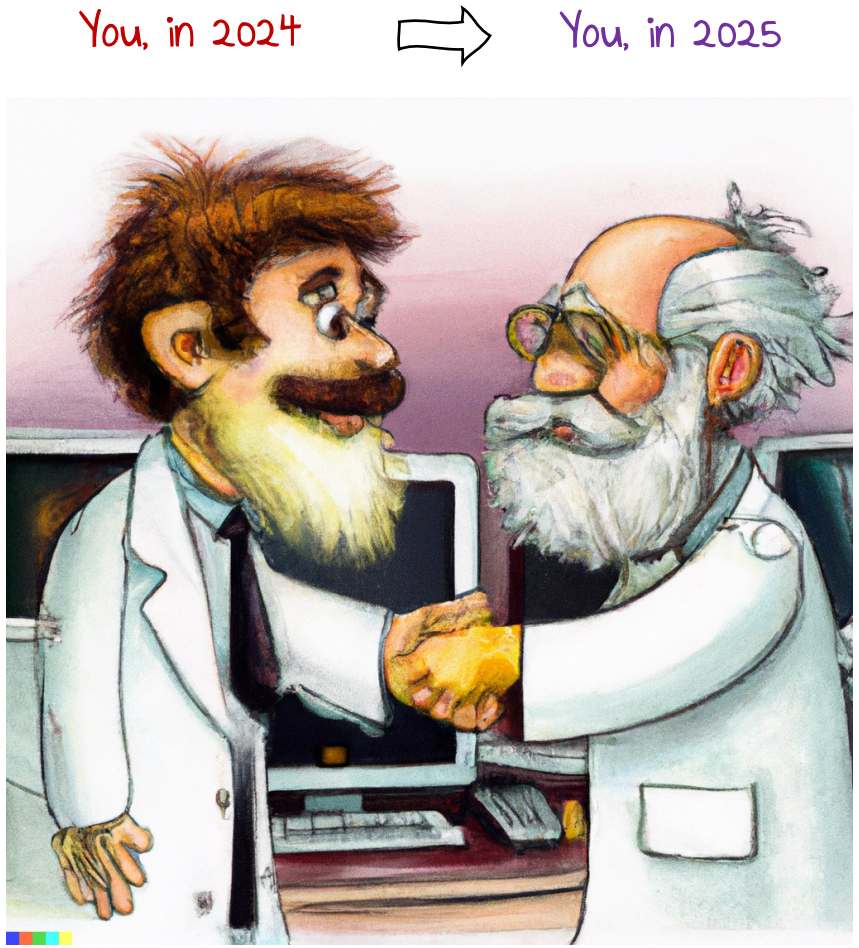
- What does a good test look like? What should I test?
- Good:
  - Short and quick to execute
  - Easy to read
  - Tests *one* thing
- Bad:
  - Relies on data files
  - Messes with “real-life” files, servers, databases

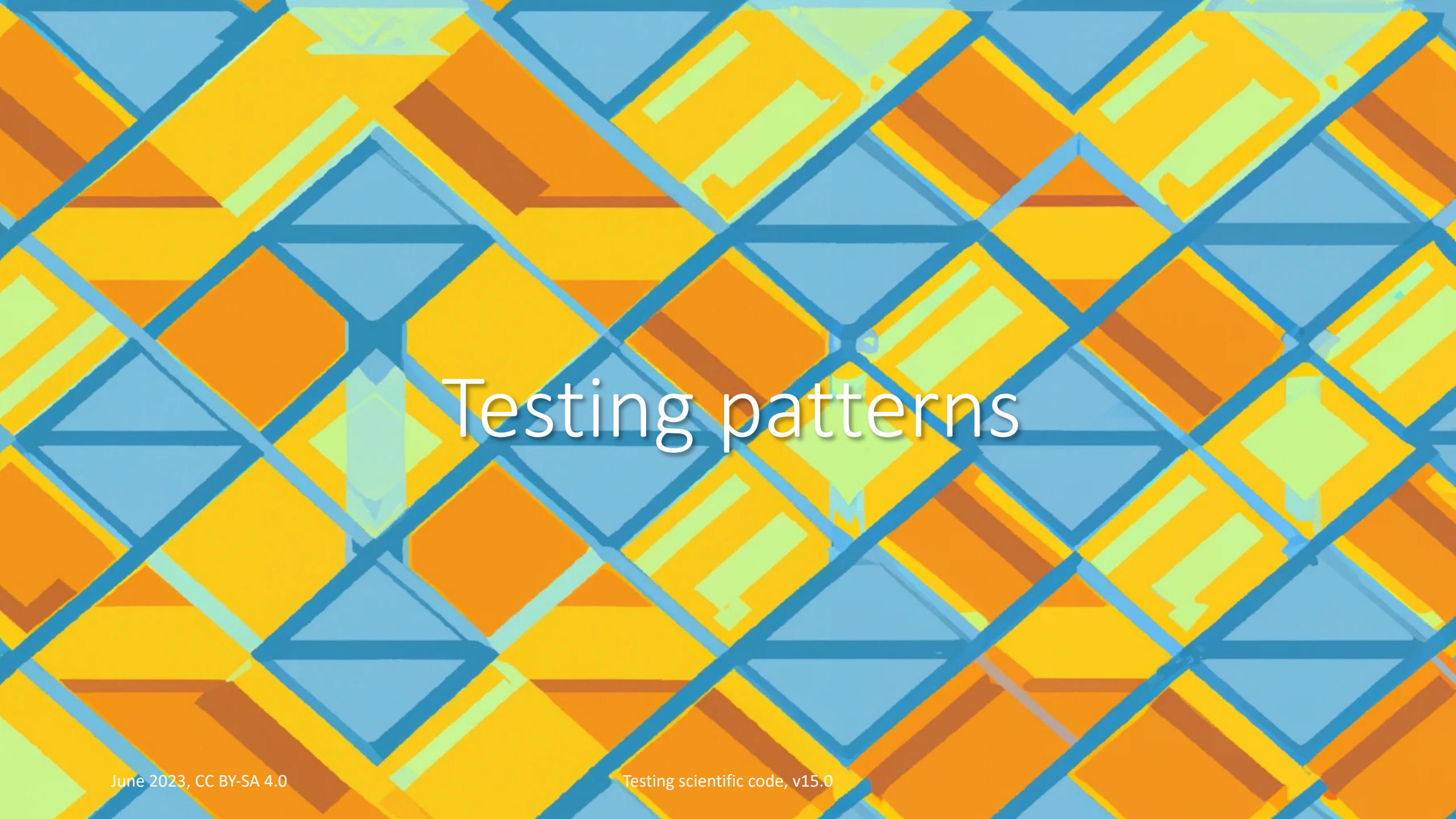
Write a working version of `find_maxima`, with testing

- **Read carefully the description of Issue #2 on GitHub**
- Submit a Pull Request for Issue #2
  - Fork the repository (if you haven't already)
  - Create a new branch on the fork called, e.g., `fix-2`
  - Solve the issue with one or more commits
  - Push the branch to your GitHub fork
  - On GitHub, go to "Pull Requests" and open a pull request against branch `main` of the official ASPP repository
  - In the PR description write "Fixes #2" somewhere, this is going to create an automatic link to the issue, and close the issue if the PR is merged

Testing is good for your self-esteem

- Immediately: Always be confident that your results are correct, whether your approach works of not
- In the future: **save your future self some trouble!**
- If you are left thinking “it’s cool but I cannot test *my* code because XYZ”, talk to us and we’ll show you how to do it ;-)




Testing patterns

Floating point equality

- Real numbers are represented approximately as “floating point” numbers. When developing numerical code, we have to allow for approximation errors.
- Check that two numbers are approximately equal:

```
from math import isclose
def test_floating_point_math():
    assert isclose(1.1 + 2.2, 3.3)                  => pass
```

- `abs_tol` controls the absolute tolerance:

```
assert isclose(1.121, 1.2, abs_tol=0.1)         => pass
assert isclose(1.121, 1.2, abs_tol=0.01)        => fail
```

- `rel_tol` controls the relative tolerance:

```
assert isclose(120.1, 121.4, rel_tol=0.1)       => pass
assert isclose(120.4, 121.4, rel_tol=0.01)      => fail
```

Testing with numpy arrays

```
def test_numpy_equality():
    x = np.array([1, 1])
    y = np.array([2, 2])
    z = np.array([3, 3])
    assert x + y == z
```

```
_____________________________________ test_numpy_equality _____________________________________

    def test_numpy_equality():
        x = numpy.array([1, 1])
        y = numpy.array([2, 2])
        z = numpy.array([3, 3])
>       assert x + y == z
E       ValueError: The truth value of an array with more than one element is ambiguous.
Use a.any() or a.all()

code.py:47: ValueError
```

Testing with numpy arrays

- The module `np.testing` defines helper functions:

```
assert_equal(x, y)
assert_allclose(x, y, rtol=1e-07, atol=0)
```

- If you need to check more complex conditions:
  - `np.all(x)`: returns True if all elements of x are true
    `np.any(x)`: returns True is any of the elements of x is true
  - combine with `logical_and, logical_or, logical_not`:

```
# test that all elements of x are between 0 and 1
assert all(logical_and(x > 0.0, x < 1.0))
```

Watch out for `nans`!

- In general, `nan` is not equal to itself (IEEE standard)

```
In [2]: np.nan == np.nan
Out[2]: False
```

- `assert_equal` and `assert_allclose` consider `nans` equal by default

```
def test_allclose_with_nan():
    x = np.array([1.1, np.nan])
    y = np.array([2.2, np.nan])
    z = np.array([3.3, np.nan])
    assert_allclose(x + y, z)

test_numpy_equality.py::test_allclose_with_nan      PASSED
```

Common for-loop pattern for testing multiple cases

- Often these cases are collected in a single test:

```
def test_lower():
    # Given
    # Each test case is a tuple of (input, expected_result)
    test_cases = [('HeLlO wOrld', 'hello world'),
                  ('hi', 'hi'),
                  ('123 ([?', '123 ([?'),
                  ('', '')]

    for string, expected in test_cases:
        # When
        output = string.lower()
        # Then
        assert output == expected
```

The for-loop pattern can be improved

- It is repetitive to write the for-loop pattern
- If one of the cases break, it can be complicated to figure out which one

- pytest has many helpers for simplifying common testing cases!
- One of them is the `parametrize` decorator, that simplifies running the same test with multiple cases

Simple example

```
def test_for_loop_simple():
    cases = [1, 2, 3]
    for a in cases:
        assert a > 0
```

`test_for_loop_simple`
runs once and loops over
3 test cases

Simple example, with the `parametrize` decorator

Name of the variable that varies

List of values for the variable

```
@pytest.mark.parametrize('a', [1, 2, 3])
def test_parametrize_simple(a):
    assert a > 0
```

The test must take an argument with the same name

`test_parametrize_simple`
runs 3 times
with a=1, a=2, and a=3

Simple example, with the `parametrize` decorator

Name of the variable that varies

List of values for the variable

```python
@pytest.mark.parametrize('a', [1, 2, 3])
def test_parametrize_simple(a):
    assert a > 0
```

The test must take an argument with the same name

```
============================================= test session starts ==============================================
platform darwin -- Python 3.11.3, pytest-7.3.1, pluggy-1.0.0 -- /Users/pietro.berkes/miniconda3/envs/aspp/bin/python
cachedir: .pytest_cache
rootdir: /Users/pietro.berkes/o/ASPP/testing_project/demos
plugins: anyio-3.5.0
collected 3 items

test_parametrize.py::test_parametrize_simple[1] PASSED                                               [ 33%]
test_parametrize.py::test_parametrize_simple[2] PASSED                                               [ 66%]
test_parametrize.py::test_parametrize_simple[3] PASSED                                               [100%]

============================================== 3 passed in 0.00s ===============================================
```

pytest automatically creates one separate test for each test case

Example with multiple values

- This is a more typical case with several input values and the expected result of the test

```
def test_for_loop_multiple():
    cases = [
       (1, 'hi', 'hi’),
       (2, 'no', 'nono’)
    ]
    for a, b, expected in cases:
        result = b * a
        assert result == expected
```

`test_for_loop_multiple`
runs once and loops over
2 test cases

Same example, with the `parametrize` decorator

Name of all the variables, separated by commas in one string

List of tuples with the values for each varialbe, one for each test case

```
@pytest.mark.parametrize('a, b, expected', [(1, 'hi', 'hi'), (2, 'no', 'nono')])
def test_parametrize_multiple(a, b, expected):
    result = b * a
    assert result == expected
```

The test must take arguments with the same names as in the string

`test_parametrize_multiple` runs 2 times with

1) `a=1  b='hi' expected='hi'`

and

2) `a=2 b='no', expected='nono'`

Same example, with the `parametrize` decorator

Name of all the variables, separated by commas in one string

List of tuples with the values for each varialbe, one for each test case

```python
@pytest.mark.parametrize('a, b, expected', [(1, 'hi', 'hi'), (2, 'no', 'nono')])
def test_parametrize_multiple(a, b, expected):
    result = b * a
    assert result == expected
```

The test must take arguments with the same names as in the string

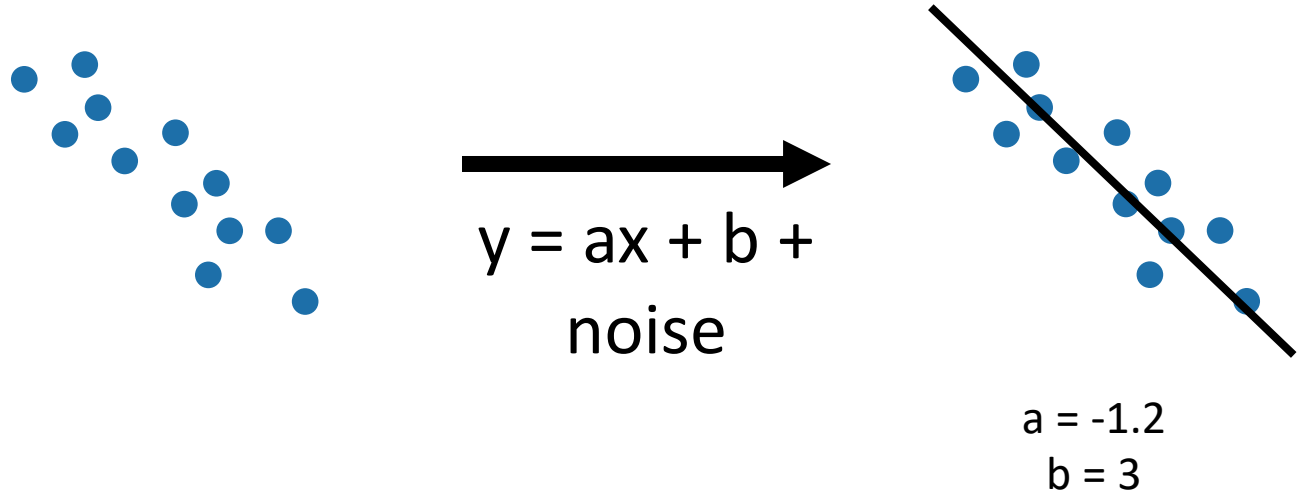
```
↳ pytest -v test_parametrize.py::test_parametrize_multiple
============================================= test session starts ==============================================
platform darwin -- Python 3.11.3, pytest-7.3.1, pluggy-1.0.0 -- /Users/pietro.berkes/miniconda3/envs/aspp/bin/python
cachedir: .pytest_cache
rootdir: /Users/pietro.berkes/o/ASPP/testing_project/demos
plugins: anyio-3.5.0
collected 2 items

test_parametrize.py::test_parametrize_multiple[1-hi-hi] PASSED                                        [ 50%]
test_parametrize.py::test_parametrize_multiple[2-no-nono] PASSED                                      [100%]

============================================== 2 passed in 0.01s ===============================================
```

pytest automatically creates one separate test for each test case

Strategies for testing learning algorithms

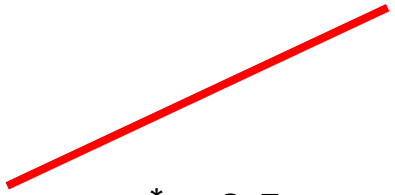
- Learning algorithms can get stuck in local maxima, the solution for general cases might not be known (e.g., unsupervised learning)
- Turn your validation cases into tests
- Stability tests:
  - Start from final solution; verify that the algorithm stays there
  - Start from solution and add a small amount of noise to the parameters; verify that the algorithm converges back to the solution
- Parameter Recovery: Generate synthetic data from the model with known parameters, then test that the code can learn the parameters back

Learning algorithms fit the parameters of a model to observed data



Generate synthetic data from the model to test the learning algorithm by recovering the parameters

1) Fix initial parameters

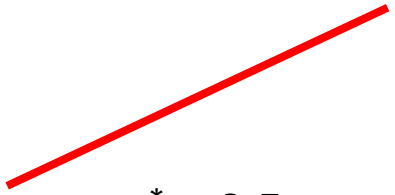


$a^* = 0.5$

$b^* = -1.3$

Generate synthetic data from the model to test the learning algorithm by recovering the parameters

1) Fix initial parameters

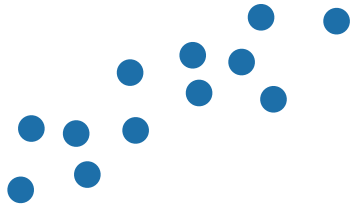


$a^* = 0.5$

$b^* = -1.3$

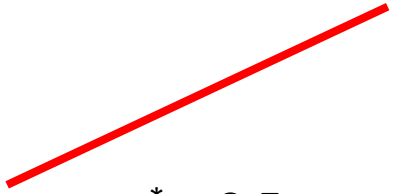
$y = a^* x + b^*$ + noise

2) Generate synthetic data



Generate synthetic data from the model to test the learning algorithm by recovering the parameters

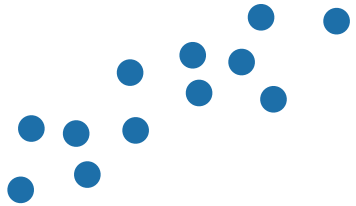
1) Fix initial parameters



$a^* = 0.5$
$b^* = -1.3$

$y = a^* x + b^*$
+ noise

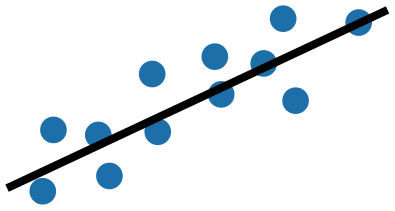
2) Generate synthetic data



3) Run the algorithm

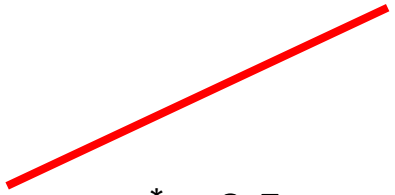
$y = ax + b$
+ noise

$a = 0.5098$
$b = -1.287$



Generate synthetic data from the model to test the learning algorithm by recovering the parameters

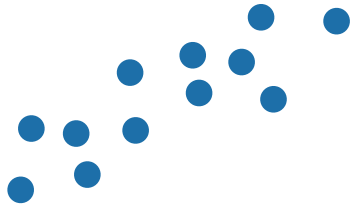
1) Fix initial parameters



$a^* = 0.5$
$b^* = -1.3$

$y = a^* x + b^*$
+ noise

2) Generate synthetic data

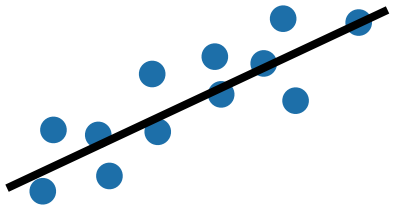


3) Run the algorithm

$y = ax + b$
+ noise

4) Compare

$a = 0.5098$
$b = -1.287$



Randomness in Testing

- Using randomness in testing can be useful
  - To check that the code is stable and works correctly in many different cases
  - To find corner cases or numerical problems

```
def test_logistic_fit_randomized():
    random_state = np.random.RandomState(SEED)
    for _ in range(100):
        x0 = random_state.uniform(0.0001, 0.9999)
        r = round(random_state.uniform(0.001, 3.999), 3)

        xs = iterate_f(it=17, x0=x0, r=r)
        recovered_r = fit_r(xs)

        assert_allclose(r, recovered_r, atol=1e-3)
```

Random Seeds and Reproducibility

- When running tests that involve radomness and some test doesn't pass it is vital to be able to reproduce that test exactly!
- Computers produce pseudo-random numbers: setting a seed resets the basis for the random number generator
- This is essential for reproducibility
- At a minimum, you should manually set the seed for each of your random tests

```
SEED = 42
random_state = np.random.RandomState(SEED)
random_state.rand()
```

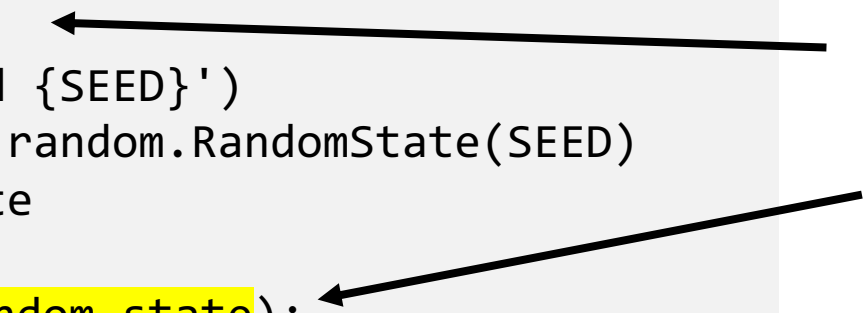
A Pytest Solution

- Non-scientific coding uses random testing more rarely, so there is no helper tools for that in pytest
- However, in scientific coding it is quite common
- What do we want?
  - For each (random) test there should be a seed
  - For each run of the test, the seed should be different
  - That seed should be printed with the test result
  - It needs to be possible to explicitely run the test again with that seed!

Fixtures (minimal solution)

- Fixtures are functions that are run before the tests are executed

```python
import numpy as np
import pytest

# set the random seed for once here
SEED = np.random.randint(0, 2**31)

@pytest.fixture
def random_state():
    print(f'Using seed {SEED}')
    random_state = np.random.RandomState(SEED)
    return random_state

def test_something(random_state):
    random_state.rand()
```



If an input argument of a test matches the name of a fixture, then the fixture is called and the return value assigned to the argument.

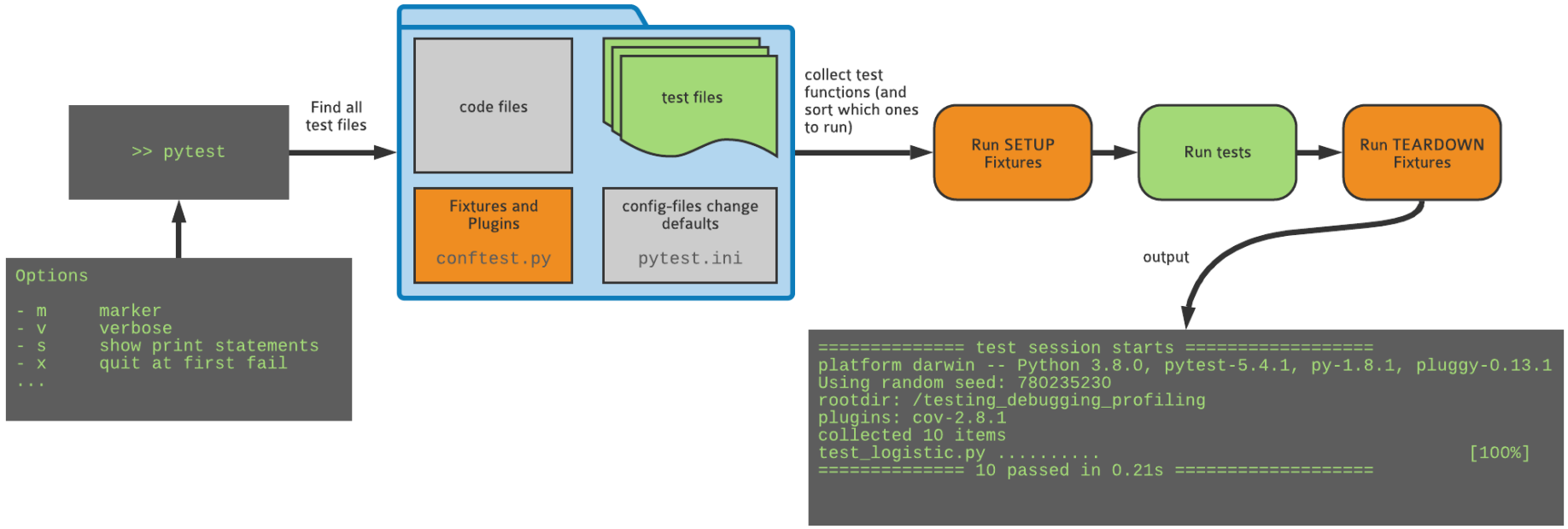
pytest handles that automatically as part of running the test suite

Hands On!

a) Write a randomized test that checks that `fit_r` can recover `r` for any random value of `x0` and `r`

b) Add a fixture at the top of your test file, that lets you print the seed to the console.

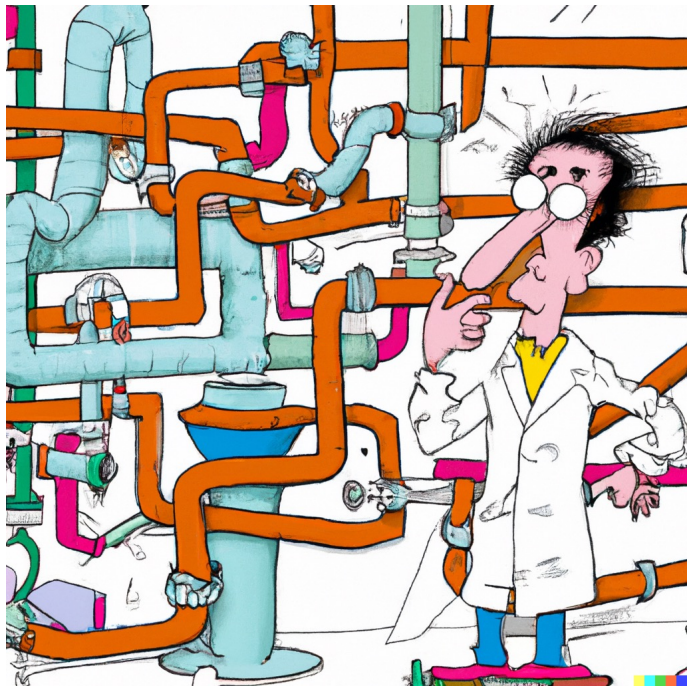
What happens when you run pytest



Fixtures (real solution)

- `conftest.py` is a special `pytest` config file (don't import it!)
- `conftest.py` can be used to define custom behavior or plugins. Fixtures can also be defined here, so that they can be used by all tests.

- See the file `hands_on/randomness/_conftest.py` in the repo you forked. If you move it to the main folder and rename it, the functions defined there select a seed for each test and allow you to pass a seed on the command line using `--seed 123`



Decorating “special” tests

- **`@xfail`**: Expected failure, outputs an “x” (or “X”) in the report

```
@pytest.mark.xfail
def test_something():
    ...
```

- **`@skip`**: Skip test, useful e.g. when the feature doesn’t exist yet

```
@pytest.mark.skip(reason=“functionality not yet
implemented”)
def test_something():
    ...
```

- **`@skipif`**: Skip the test if a condition is met, useful for tests that only works on a specific platform, or for a specific version of Python

```
@pytest.mark.skipif(sys.version_info < (3, 10),
                    reason="requires python3.10 or higher")
def test_something():
    ...
```

Marking tests with custom markers

- If you have lots of tests, you can categorize them with your own markers
  - although for custom mark names you need to register the marks “pytest.ini”
  - https://docs.pytest.org/en/7.1.x/example/markers.html#registering-markers
- Example:
  - Smoke tests check for really basic features: run these frequently
  - Other tests may be many or too slow to run every time and test for more edge cases

```
@pytest.mark.smoke
def test_something_basic():
    ...
```

```
> pytest -m smoke
> pytest -m "smoke and not slow"
```

Writing temporary files: `tmp_path`

- To test functions that write to disk without leaving around the files when the test is finished, use the `tmp_path` fixture
- The value of `tmp_path` is a `pathlib.Path` object
- The directory is created at the start of the test, and removed at the end

```python
def test_create_file(tmp_path):
    d = tmp_path / "sub"
    d.mkdir()
    p = d / "hello.txt"
    content = "some random text"
    p.write_text(content)
    assert p.read_text() == content
    assert len(list(tmp_path.iterdir())) == 1
```

All you need to do is add an argument with this exact name