Testing scientific code, Part II

Because you're worth it

Lisa Schwetlick and Pietro Berkes

Testing patterns

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
  - Exercise *one* thing
- Bad:
  - Relies on data files
  - Messes with "real-life" files, servers, databases

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

Test simple but general cases

- Start with simple, general case
  - Take a realistic scenario for your code, try to reduce it to a simple example
- Tests for ‘lower’ method of strings

```
def test_lower():
    # Given
    string = 'HeLlO wOrld'
    expected = 'hello world'

    # When
    output = string.lower()

    # Then
    assert output == expected
```

Test special cases and boundary conditions

- Code often breaks in corner cases: empty lists, None, NaN, 0.0, lists with repeated elements, non-existing file, …
- This often involves making design decision: respond to corner case with special behavior, or raise meaningful exception?

```
def test_lower_empty_string():
    # Given
    string = ''
    expected = ''

    # When
    output = string.lower()

    # Then
    assert output == expected
```

- Other good corner cases for string.lower():
  - 'do-nothing case': `string = 'hi'`
  - symbols: `string = '123 (!'`

Common testing pattern

- 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
```

Parametrize

- Sometimes you want to run the same test multiple times with different values
- Option 1: for loop in your test
- Option 2: parametrize

```
@pytest.mark.parametrize("a", [1,2,3,4])
def test_addition_increases(a):
    assert 5+a>a
```

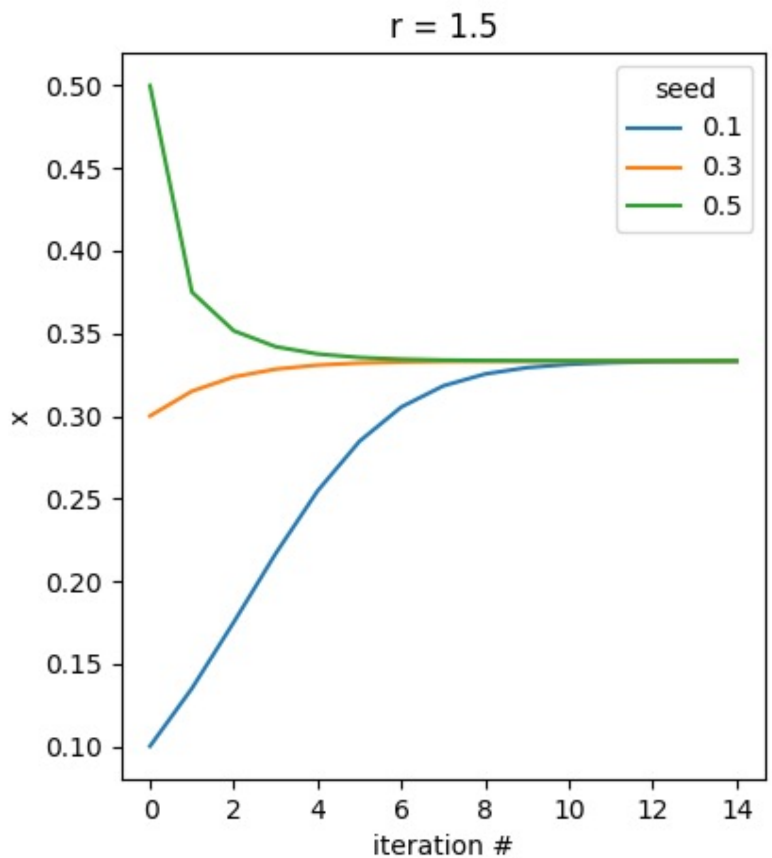
Parametrize

- … is also useful when you want to test different cases and their outcomes!

```
@pytest.mark.parametrize("string, expected",
                          [('HeLlO wOrld', 'hello world’),
                           ('hi', 'hi’),
                            ('','')])
def test_lower(string, expected):
        # When
        output = string.lower()
        # Then
        assert output == expected
```

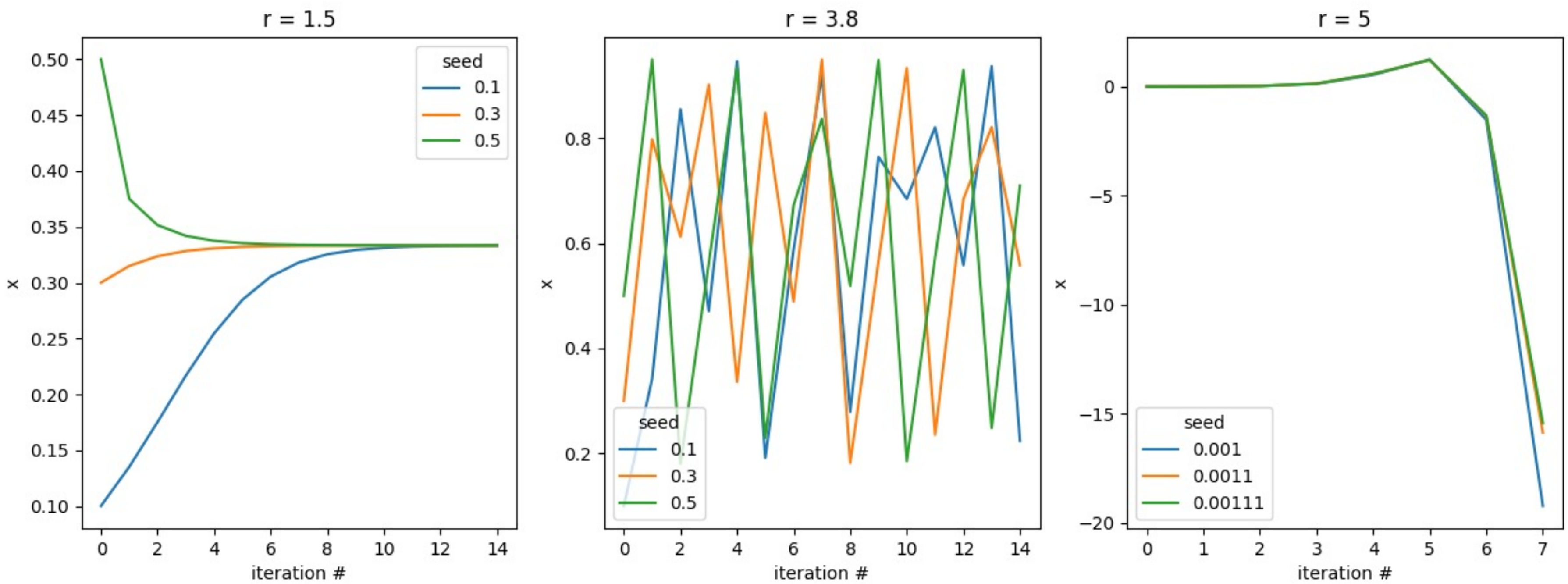
Excursion: Logistic Map

- Sometimes used as a simple model for population growth



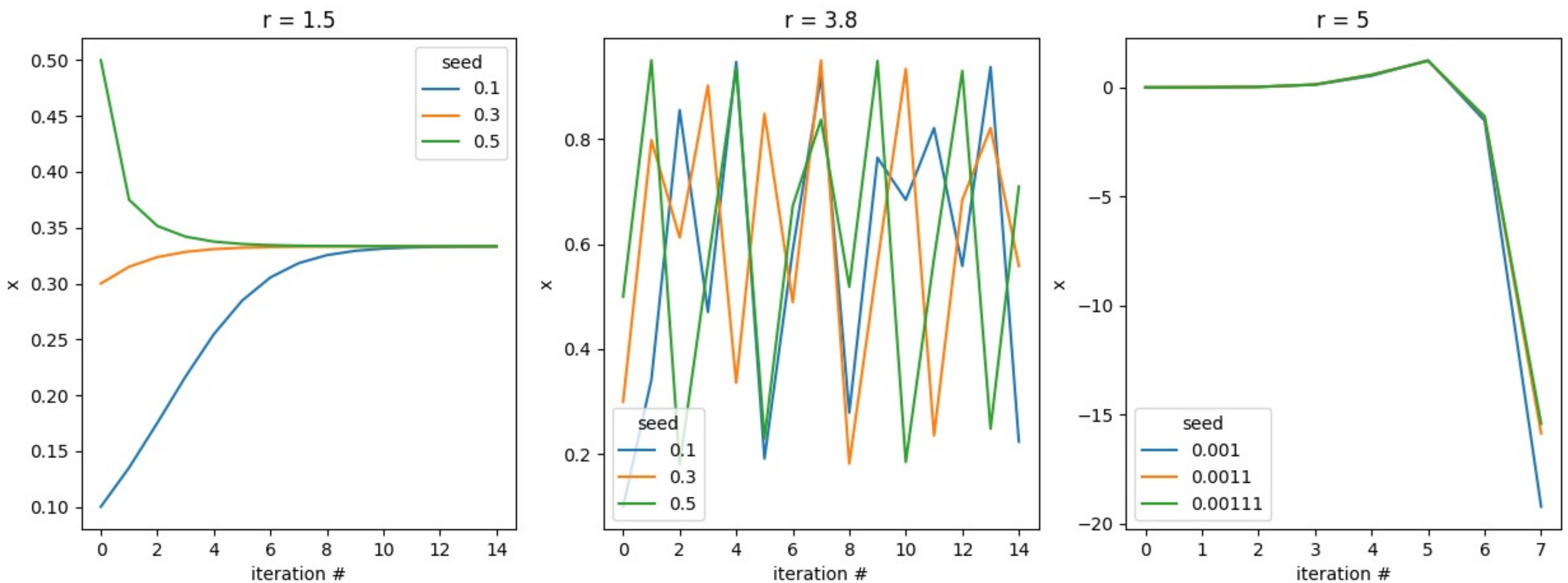
Excursion: Logistic Map

- x_0 should be between 0 and 1
- $f(x) = r * x * (1 - x)$
- Iterated function: `f(x`$_0$`)=x`$_1$ **->** `f(x`$_1$`)=x`$_2$ **->** `f(x`$_2$`)=x`$_3$



Excursion: Logistic Map

- Looking at these plots, what could you test?



Hands-on!

First fork the repo https://github.com/ASPP/2022-bilbao-testing-project on GitHub and clone your own copy!

a) Implement the logistic map f(x)=$r*x*(1-x)$. Use `@parametrize` to test the function for the following cases:

- `x=0.1, r=2.2 => f(x, r)=0.198`
- `x=0.2, r=3.4 => f(x, r)=0.544`
- `x=0.75, r=1.7 => f(x, r)=0.31875`

b) Implement the function `iterate_f` that runs `f` for `it` iterations, each time passing the result back into f. Use `@parametrize` to test the function for the following cases:

- `x=0.1, r=2.2, it=1`
  `=> iterate_f(it, x, r)=[0.198]`
- `x=0.2, r=3.4, it=4`
  `=> f(x, r)=[0.544, 0.843418, 0.449019, 0.841163]`
- `x=0.75, r=1.7, it=2`
  `=> f(x, r)=[0.31875, 0.369152]]`

c) Use the `plot_trajectory` function from the `plot_logfun` module to look at the trajectories generated by your code. Try with values `r<3`, `r>4`, and `3<r<4` to get an intuition for how the function behaves differently with different parameters.

Marking tests (xfail)

- Aside from parametrize, there are some other built in markers
- Sometimes you have a test that fails, but for good reason or you just want to deal with it later…
- Expected failure (xfail)
- Outputs an "x" (or "X") in place of the "."

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

Marking tests (skip)

- It is also possible to skip tests
- Useful when the feature doesn't exist yet or the test is very slow

```
@pytest.mark.skip(reason="functionality not yet implemented")
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 failure: 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():
        ...
```

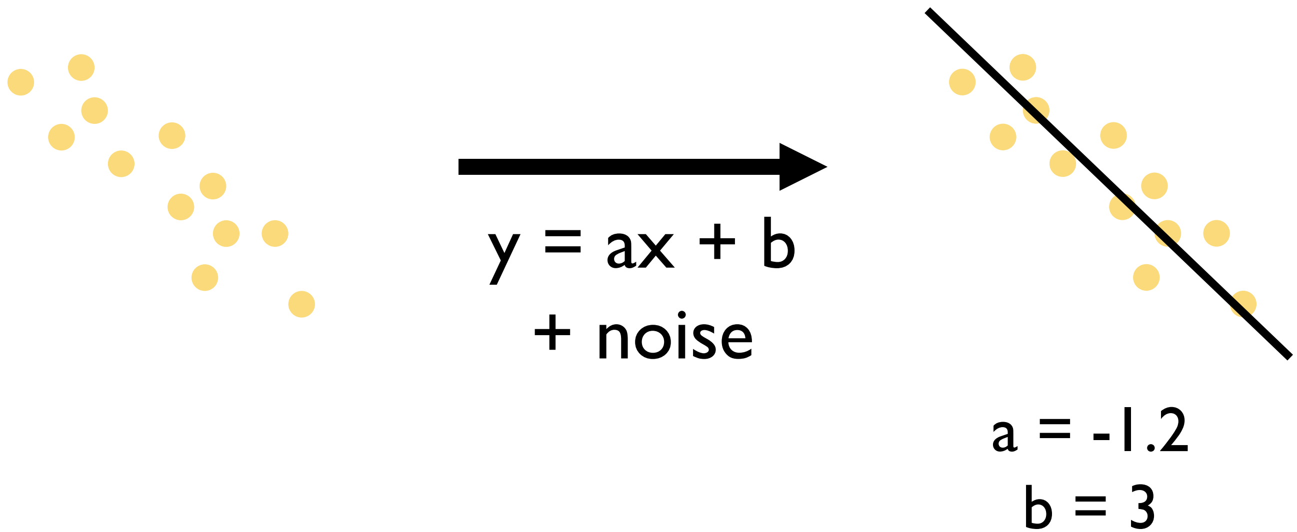
```
> pytest -m smoke
> pytest -m "smoke and not slow"
```

Strategies for testing scientific code

Strategies for testing learning algorithms

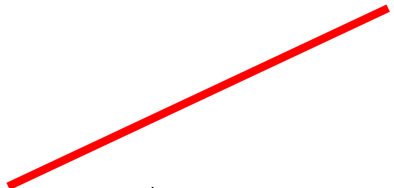
- Learning algorithms can get stuck in local maxima, the solution for general cases might not be known (e.g., unsupervised learning)
- Turn your validation cases into tests
- Stability tests:
  - Start from final solution; verify that the algorithm stays there
  - Start from solution and add a small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

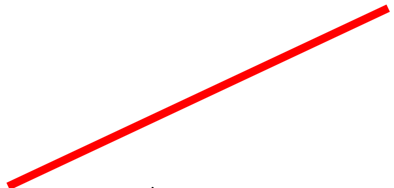
1) Fix initial parameters



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

Generate synthetic data from the model to test the learning algorithm

1) Fix initial parameters



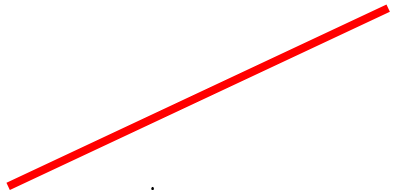
$a^* = 0.5$
$b^* = -1.3$

$y = a^* x + b^*$
$+ \text{noise}$

2) Generate synthetic data

Generate synthetic data from the model to test the learning algorithm

1) Fix initial parameters



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

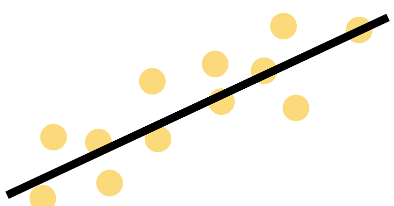
$y = a^* x + b^*$
$+ \text{noise}$

2) Generate synthetic data

3) Run the algorithm

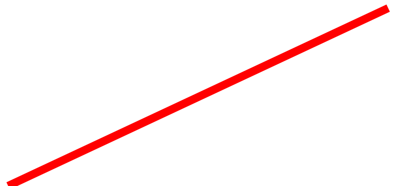
$y = ax + b$
$+ \text{noise}$

$a = 0.5098$
$b = -1.287$



Generate synthetic data from the model to test the learning algorithm

1) Fix initial parameters



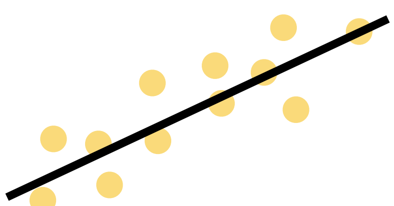
$a^* = 0.5$
$b^* = -1.3$

$y = a^* x + b^*$
$+ \text{noise}$

2) Generate synthetic data

4) Compare

$a = 0.5098$
$b = -1.287$



$y = ax + b$
$+ \text{noise}$

3) Run the algorithm

Other common cases

- Test general routines with specific ones
  - Example: test `polynomial_expansion(data, degree)` with `quadratic_expansion(data)`
- Test optimized routines with brute-force approaches
  - Example: test function computing analytical derivative with numerical derivative

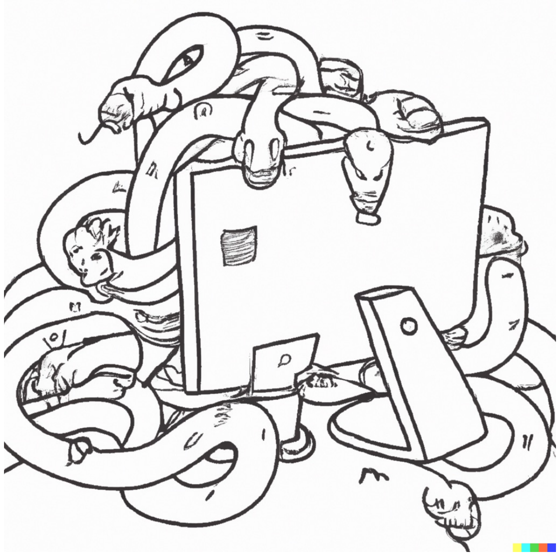
Example: eigenvector decomposition

- Consider the function `values, vectors = eigen(matrix)`
- Test with simple but general cases:
  - use full matrices for which you know the exact solution (from a table or computed by hand)
- Test general routine with specific ones:
  - use the analytical solution for 2x2 matrices
- Generate data from the model:
  - generate random eigenvalues, random eigenvector; construct the matrix; then check that the function returns the correct eigenvalues and -vectors
- Test with boundary cases:
  - test with diagonal matrix: is the algorithm stable?
  - test with a singular matrix: is the algorithm robust? Does it raise appropriate error when it fails?

Randomness in Testing

- Using randomness in testing can be useful
  - For confirming generalizability and stability
  - For finding corner cases or numerical problems
  - Using Random/Sampled input data to test whether the result is as expected

```
def test_something():
  for _ in range(10):
    r = np.random.rand()
    assert my_random_function(r)
```



Random Seeds and Reproducibility

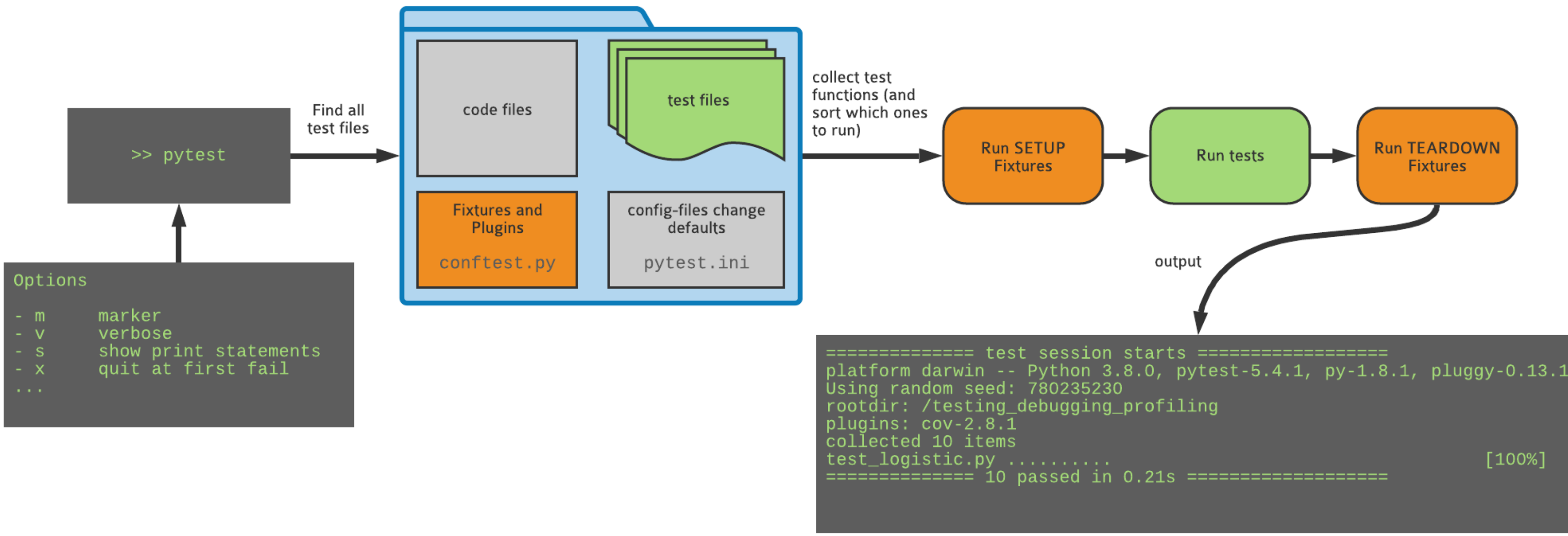
- When running tests that involve radomness and some test doesn't pass it is vital to be able to reproduce that test exactly!
- Computers produce pseudo-random numbers: setting a seed resets the basis for the random number generator
- This is essential for reproducibility
- At a minimum, you should manually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

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

A Pytest Solution

- This is not so prominent in the docs, because non-scientific coding uses random testing more rarely
- In scientific coding, when you deal with randomness it is very relevant
- 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!

Pytest



>> pytest
Options
- m marker
- v verbose
- s show print statements
- x quit at first fail
...
Find all
test files
code files
test files
Fixtures and
Plugins
conftest.py
config-files change
defaults
pytest.ini
collect test
functions (and
sort which ones
to run)
Run SETUP
Fixtures
Run tests
Run TEARDOWN
Fixtures
output
============== test session starts ==================
platform darwin -- Python 3.8.0, pytest-5.4.1, py-1.8.1, pluggy-0.13.1
Using random seed: 780235230
rootdir: /testing_debugging_profiling
plugins: cov-2.8.1
collected 10 items
test_logistic.py .... [100%]
============== 10 passed in 0.21s ===================

Fixtures (minimal solution)

- Fixtures are functions that are run before the tests are executed
- They are defined in a file called `conftest.py`, in the same directory as the tests

```
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()
```

Fixtures (real solution)

- `conftest.py` is a magical file! (don't import it!)
- Some test suites require specific or custom fixtures and plugins. They can be defined in `conftest.py`
- See the file in the repo you forked. The functions defined there select a seed for each test and allow you to pass a seed on the commandline using `--seed 123`

Hands On!

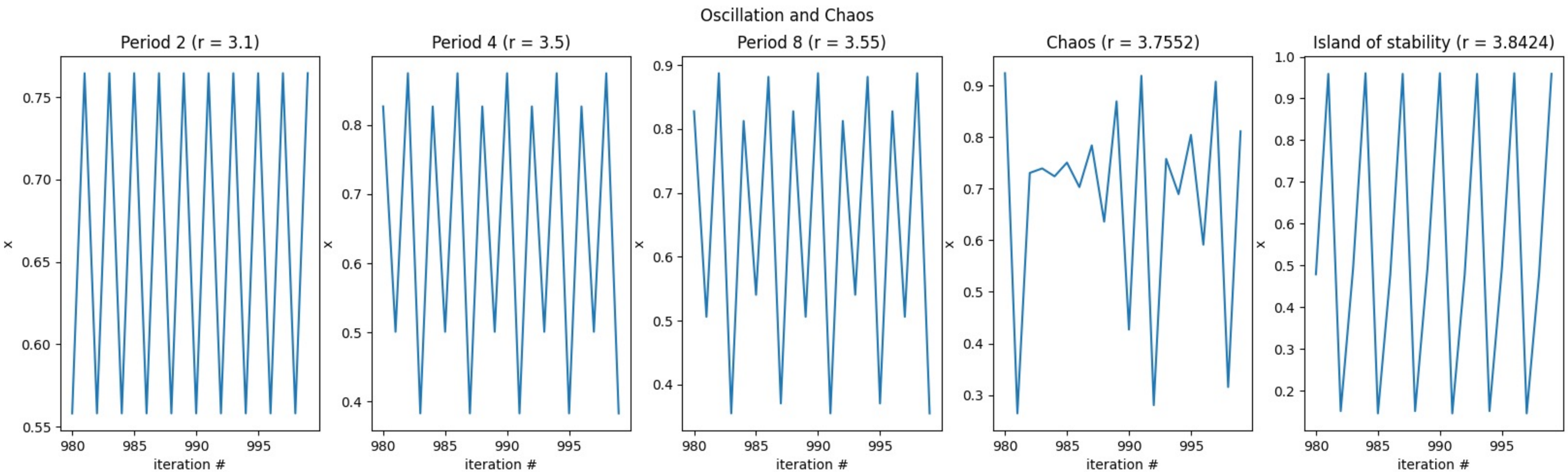
a) Write a randomized test that checks that, for `r=1.5`, any random starting points converge to the attractor `f(x, r) = 1/3`.

b) Add a conftest.py file to set a random seed before each run and make the failure reproducible

c) Check that the console output of pytest now includes the seed!

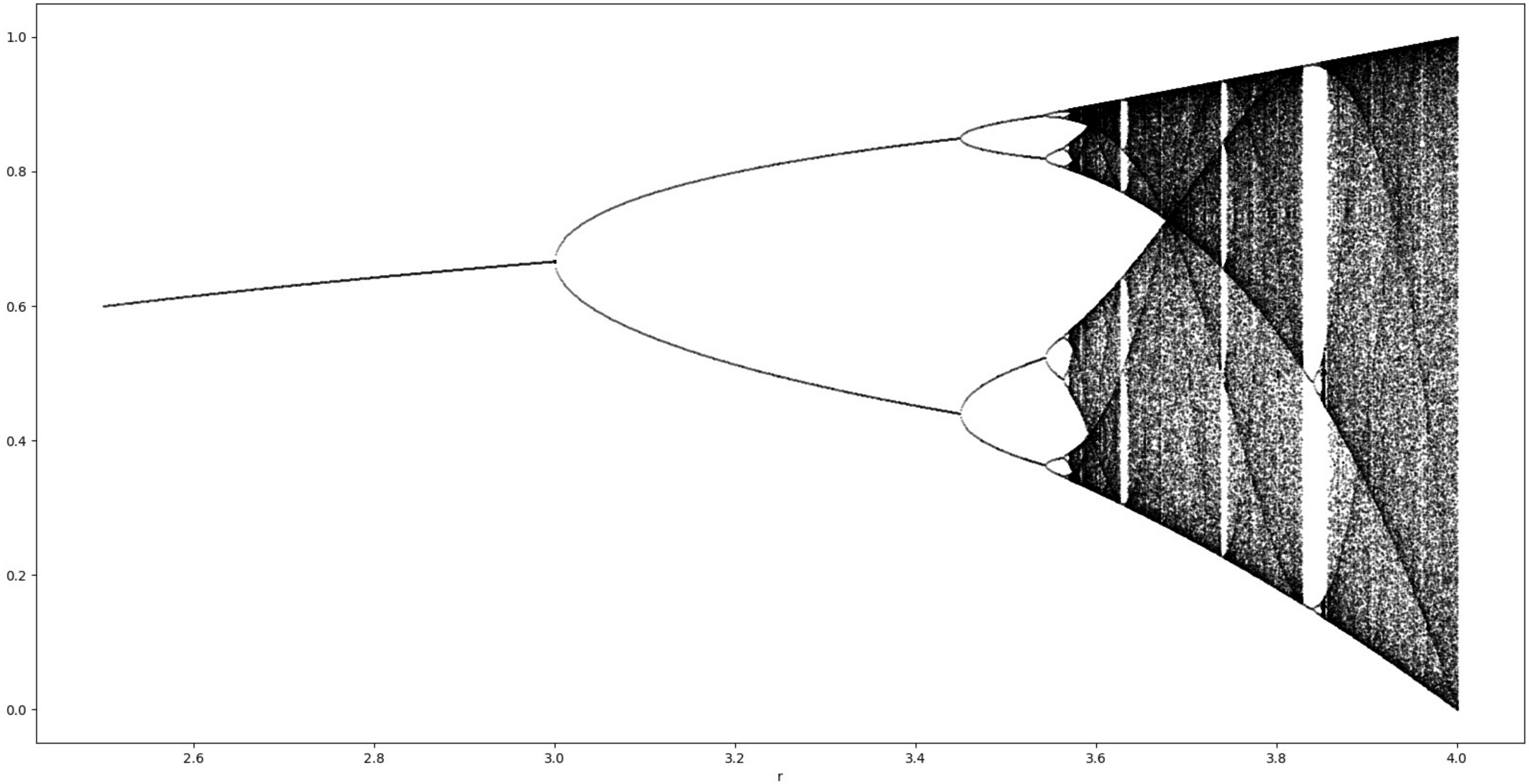
```
└─$ pytest
================================ test session starts ================================
platform darwin -- Python 3.8.0, pytest-5.4.1, py-1.8.1, pluggy-0.13.1
Using random seed: 892358865
```

Excursion: Logistic Equation

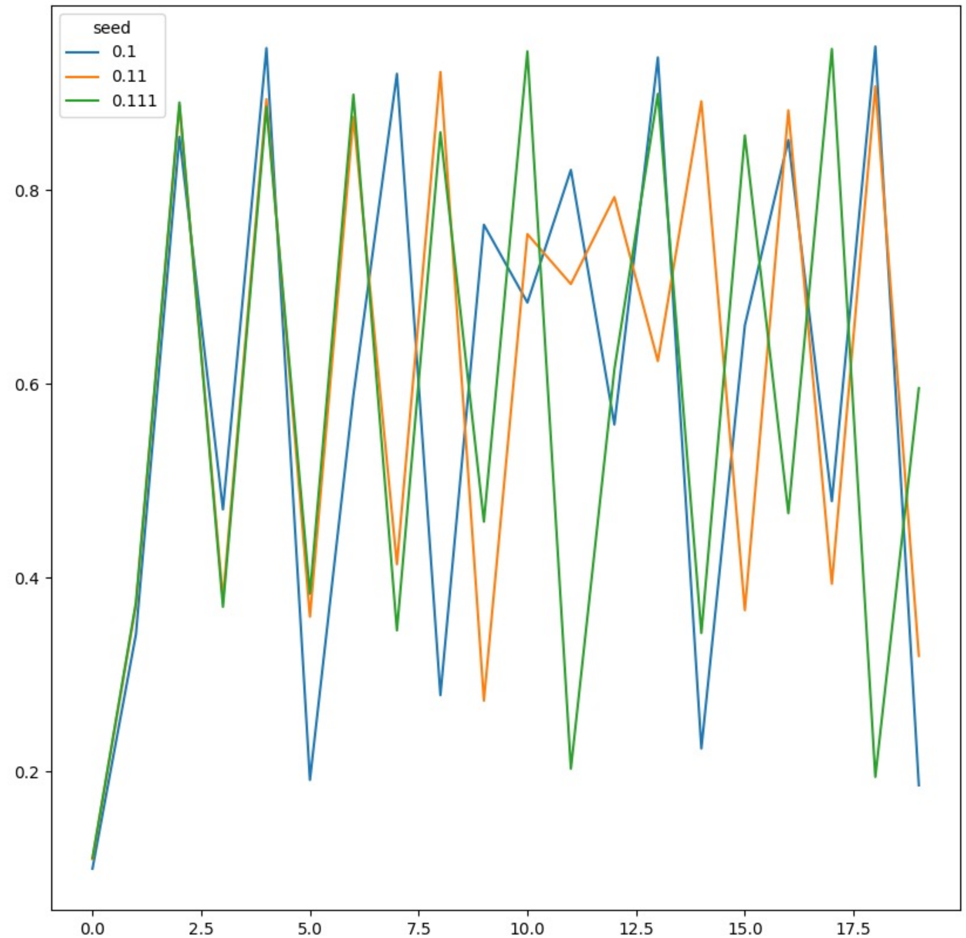


- Between r=3 and r=4 the logistic map has a range of behaviors
- Periodic vs. chaotic

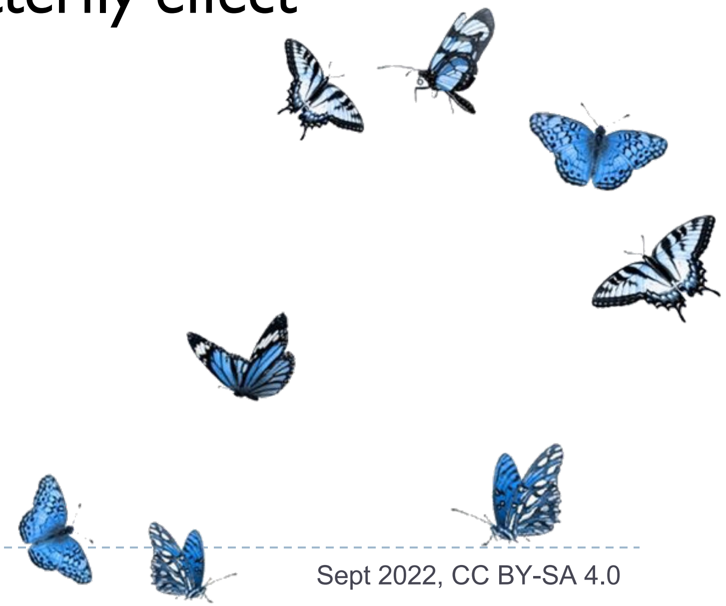
Excursion: Logistic Equation



Excursion: Logistic Equation



- Sensitive Dependence on Initial Conditions (SDIC)
- Even seeds that are very close, quickly find completely different itineraries
- Butterfly effect



Hands on!

Some r values for 3 < r < 4 have some interesting properties: a chaotic trajectory neither diverges nor converges.

a) Use the `plot_bifurcation` function from the `plot_logfun` module using your implementation of `f` and `iterate_f` to look at the bifurcation diagram. The script generates an output image, `bifurcation_diagram.png`

b) Write a test that checks for chaotic behavior when `r=3.8`. Run the logistic map for 100000 iterations and verify the conditions for chaotic behavior:

1) The function is deterministic: *this does not need to be tested in this case*
2) Orbits must be bounded: check that all values are between 0 and 1
3) Orbits must be aperiodic: check that the last 1000 values are all different
4) Sensitive dependence on initial conditions: *this is the bonus exercise (in readme)*

The test should check conditions 2) and 3)!

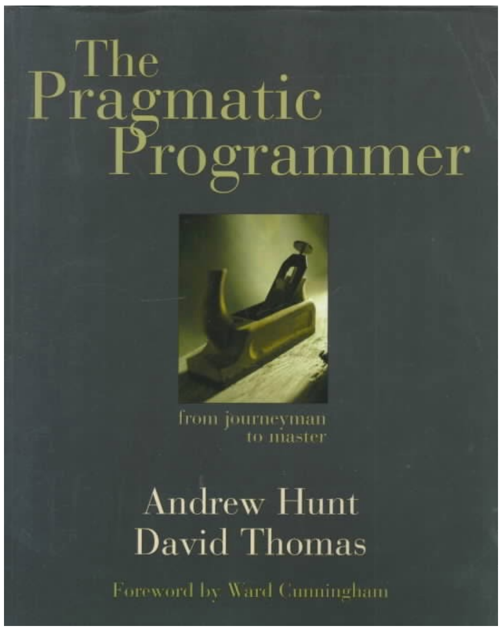
Testing is good for your self-esteem

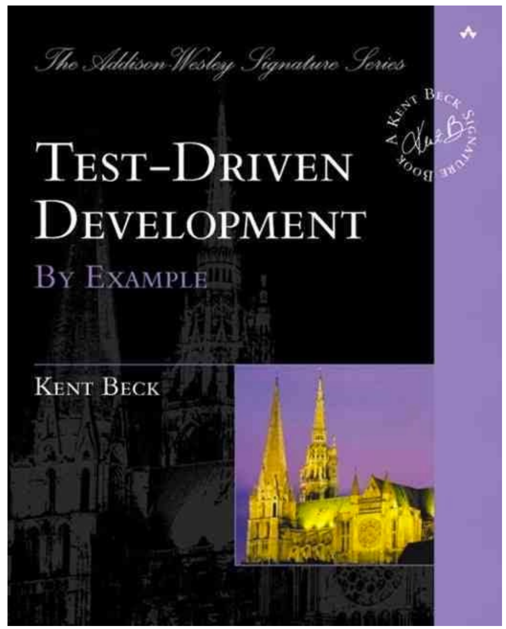
- Immediately: Always be confident that your results are correct, whether your approach works of not
- In the future: save your future self some trouble!
- If you are left thinking "it's cool but I cannot test *my* code because XYZ", talk to us during the week and we'll show you how to do it ;-)

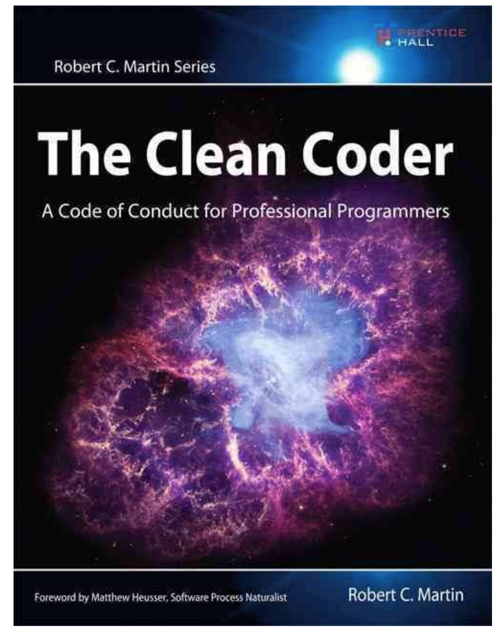
Final thoughts

- Good programming practices, with testing in the front line, make us confident about our results, and efficient at navigating our research projects
- The agile programming cycle gives you intermediate goals to build upon

Recommended reading







Thank you!