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
```

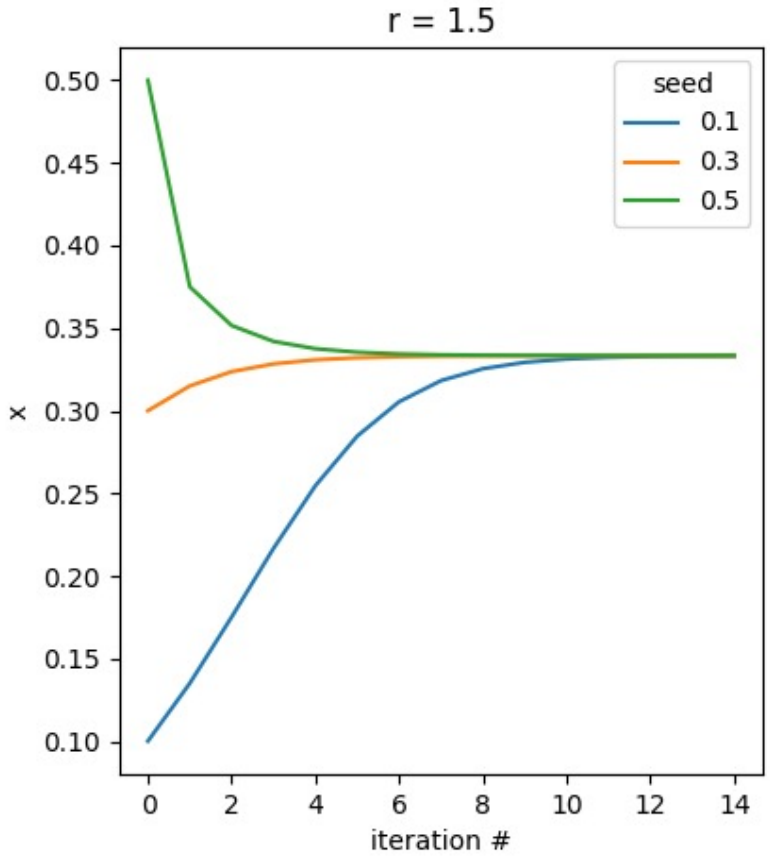
Parametrize

- … 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
```

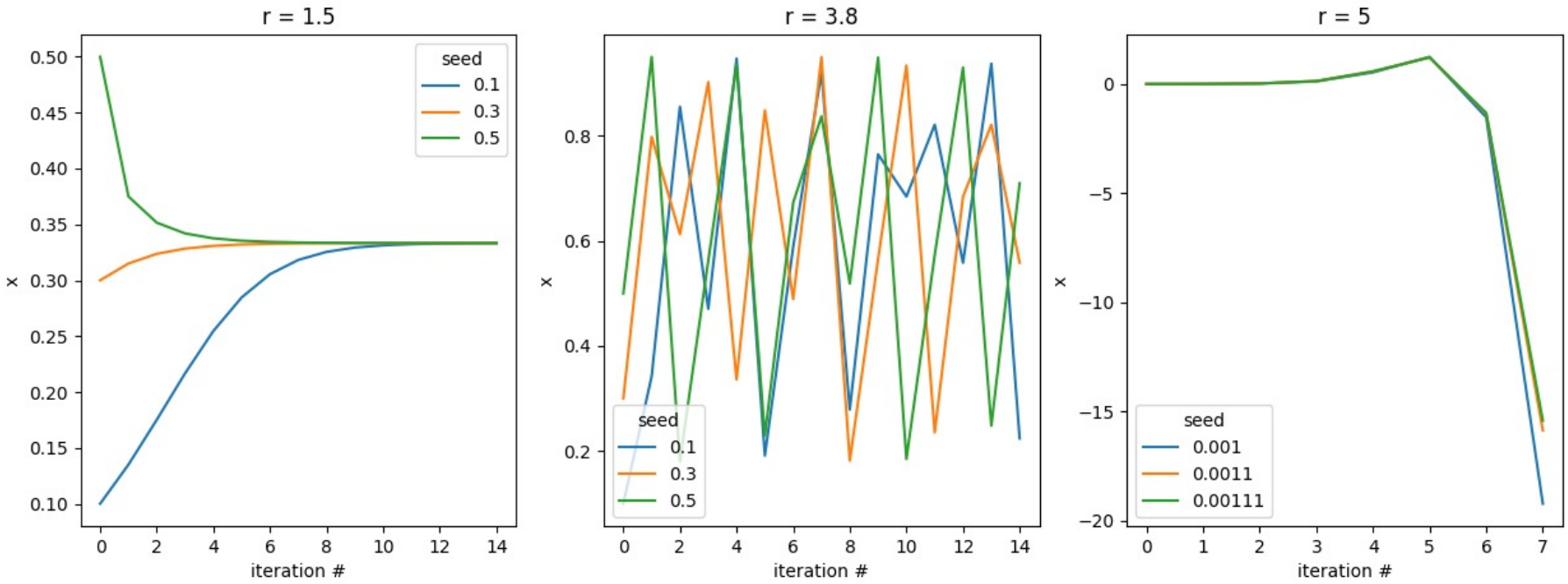
Excursion: Logistic Map

- Sometimes used as a simple model for population growth



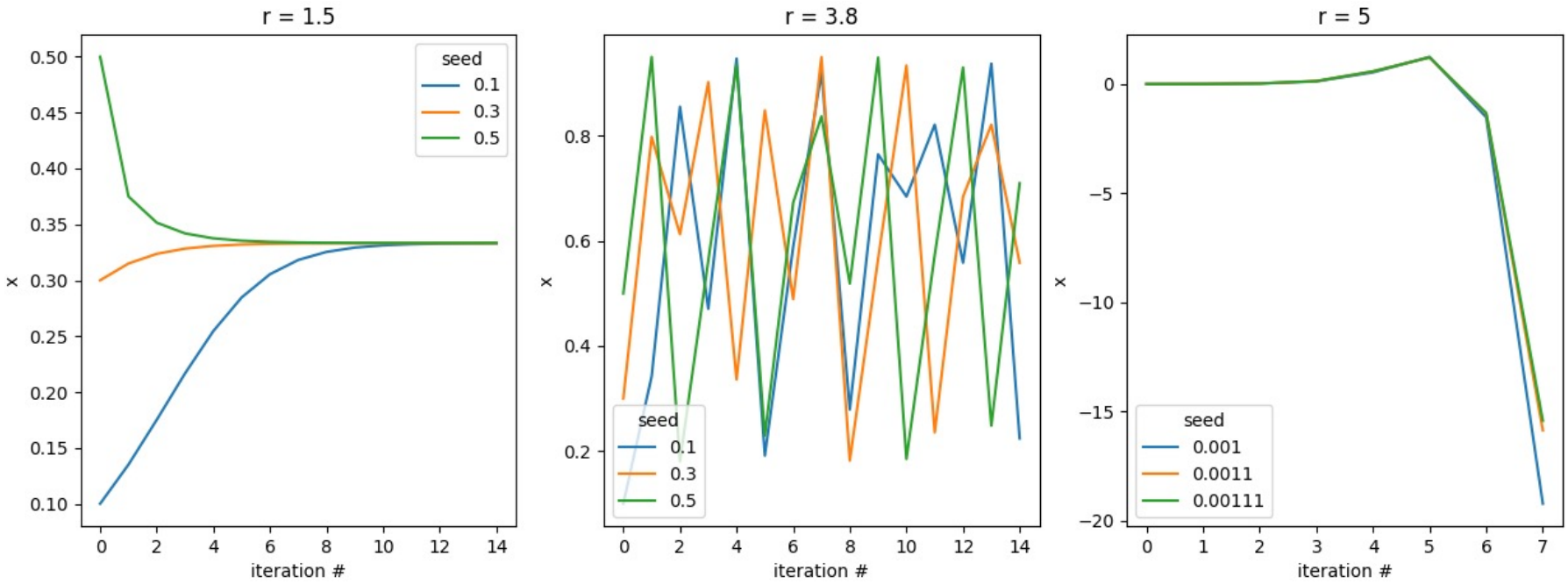
Excursion: Logistic Map

- 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/2021-bordeaux-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
- 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"
```

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 data from the model with known parameters
  - E.g., linear regression: generate data as $y = a*x + b + noise$ for random a, b, and x, then test that the algorithm is able to recover a and b

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

Numerical fuzzing

- Use deterministic test cases when possible
- In most numerical algorithm, this will cover only over-simplified situations; in some, it is impossible
- Fuzz testing: generate random input
  - Outside scientific programming it is mostly used to stress-test error handling, memory leaks, safety
  - For numerical algorithm, it is often used to make sure one covers general, realistic cases
  - The input may be random, but you still need to know what to expect
  - Make failures reproducible by saving or printing the random seed

Numerical fuzzing example

```
def test_mean_deterministic():
    x = numpy.array([-2.0, 2.0, 6.0])
    expected = 2.0
    assert isclose(numpy.mean(x), expected)


def test_mean_fuzzing():
    rand_state = numpy.random.RandomState(1333)

    N, D = 100000, 5
    # Goal means: [0.1 ,  0.45,  0.8 ,  1.15,  1.5]
    expected = numpy.linspace(0.1, 1.5, D)

    # Generate random, D-dimensional data with the desired mean
    x = rand_state.randn(N, D) + expected
    means = numpy.mean(x, axis=0)
    numpy.testing.assert_allclose(means, expected, rtol=1e-2)
```

Hands On!

Check the convergence of an attractor using fuzzing

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

b) Use `pytest.mark` to mark the tests from the previous exercise with one mark (they relate to the correct implementation of the logistic function) and the test from this exercise with another (relates to the behavior of the logistic function). Try executing first the first set of tests and then the second set of tests separately.

Random Seeds and Reproducibility

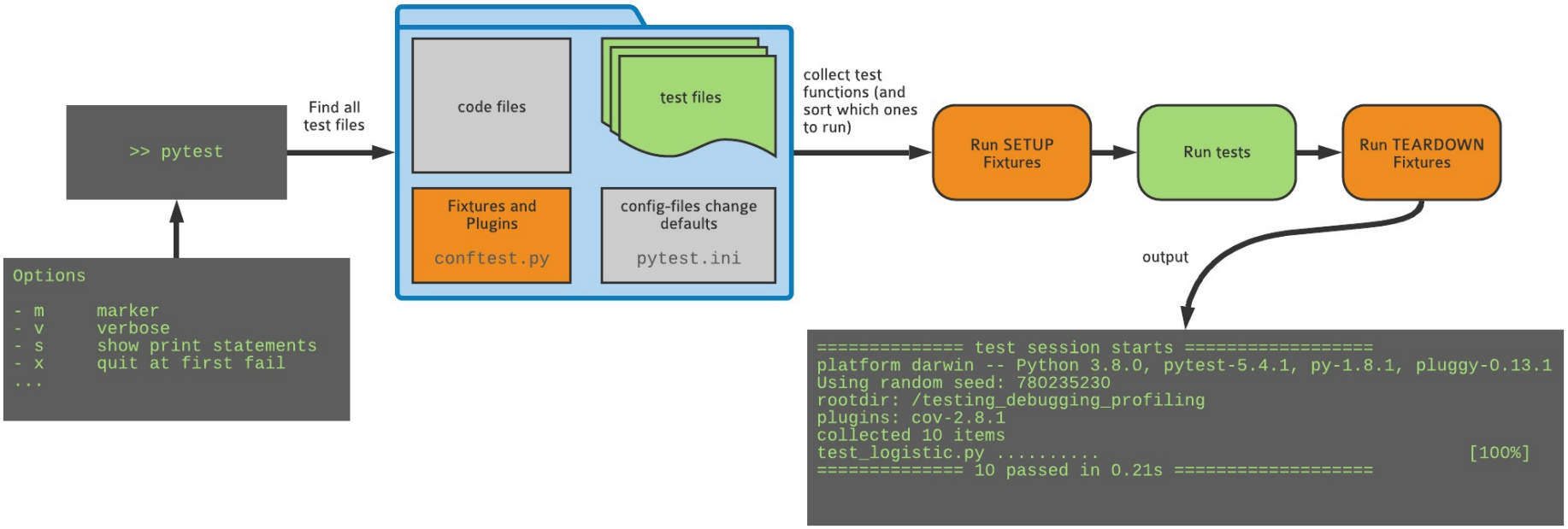
- When running fuzzy tests and some test doesn't pass it is vital to be able to reproduce that test exactly!
- Computers produce pseudo-random numbers: setting a seed resets the basis for the random number generator
- This is essential for reproducibility
- At a minimum, you should manually set the seed for your fuzzy 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 fuzzy testing more rarely
- In scientific coding, when you deal with randomness it is very relevant
- What do we want?
  - For each (fuzzy) 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



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)

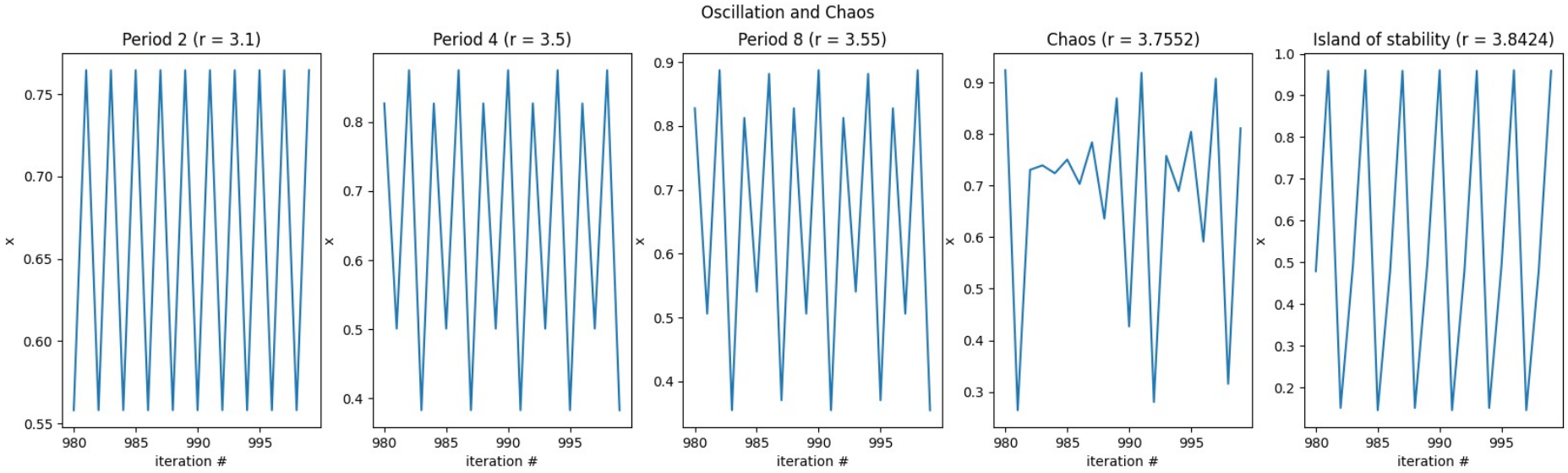
- `conftest.py` is a magical file! (don't import it!)
- Some test suites require specific or custom fixtures and plugins. They can be defined in `conftest.py`
- See the file in the repo you forked. The functions defined there select a seed for each test and allow you to pass a seed on the commandline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

Hands On!

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

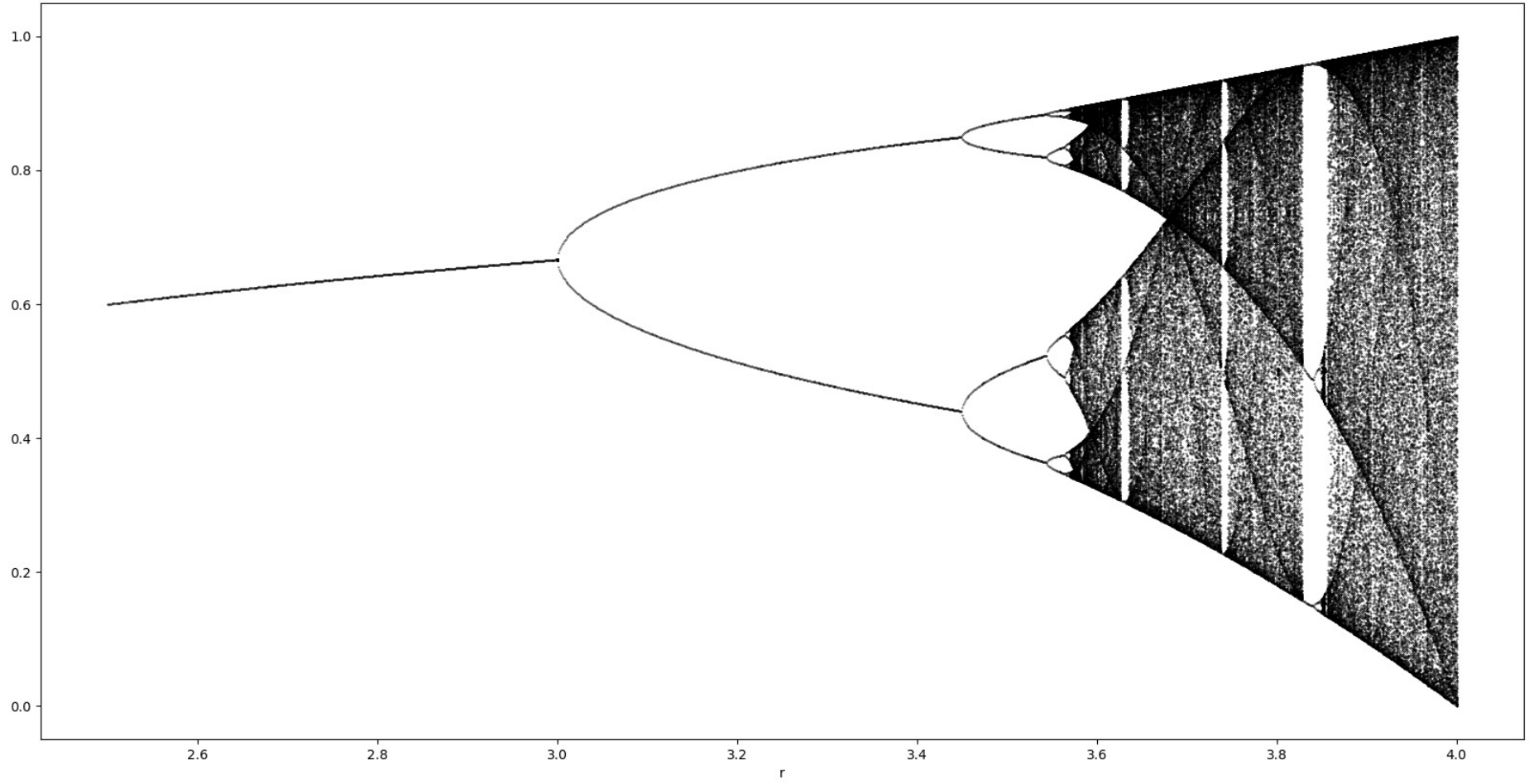
b) Check that the console output of pytest now includes the seed!

Excursion: Logistic Equation

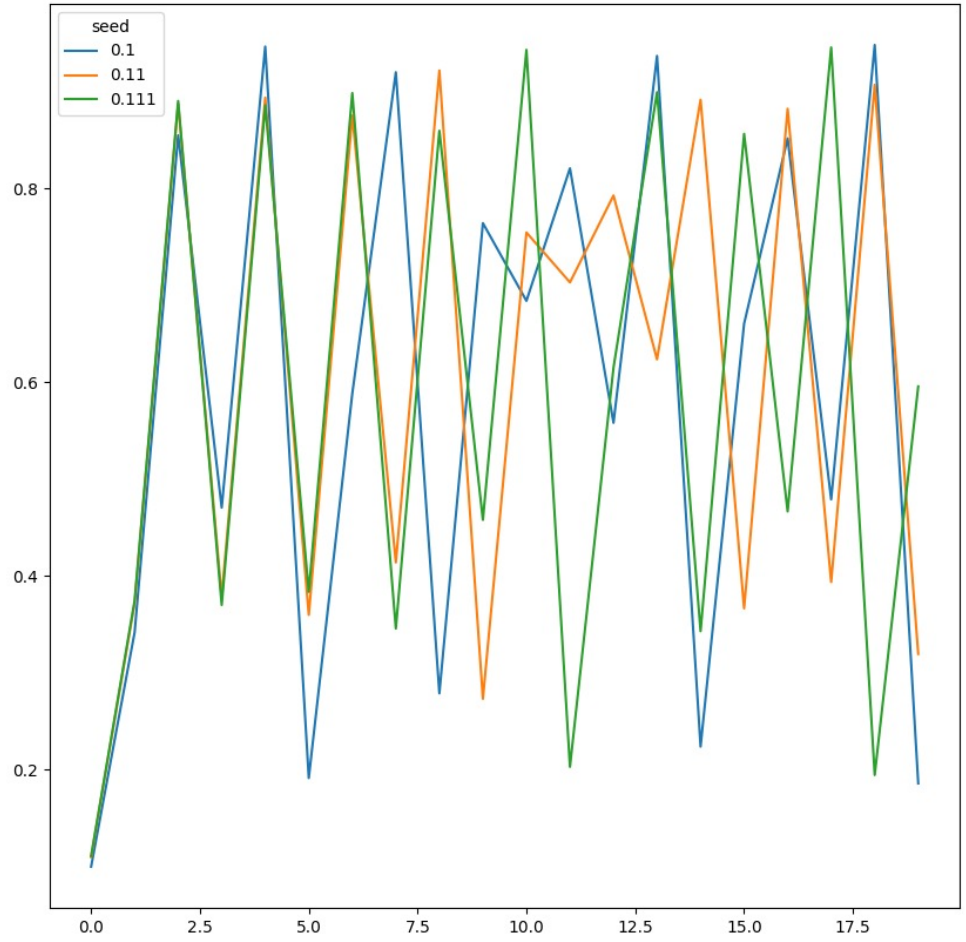


- Between r=3 and r=4 the logistic map has a range of behaviors
- Periodic vs. chaotic

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)!

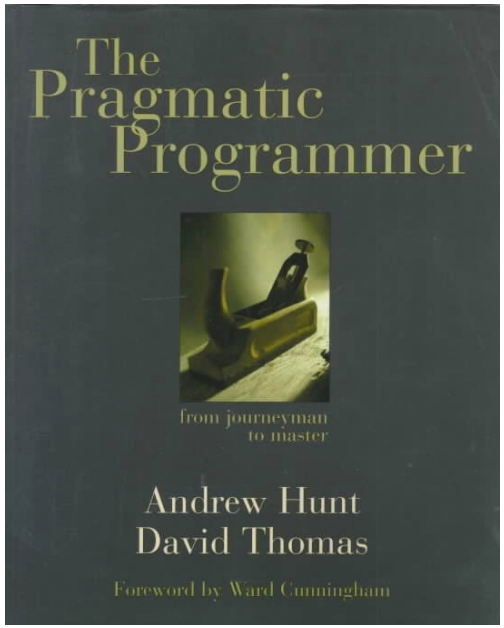
Testing is good for your self-esteem

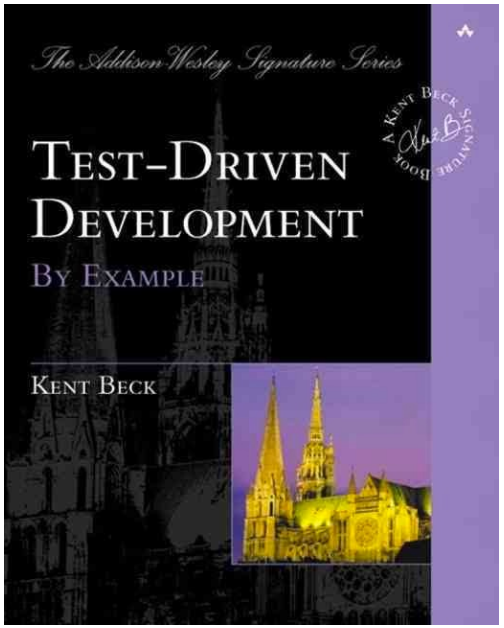
- Immediately: Always be confident that your results are correct, whether your approach works of not
- In the future: save your future self some trouble!
- If you are left thinking "it's cool but I cannot test *my* code because XYZ", talk to me during the week and I'll show you how to do it ;-)

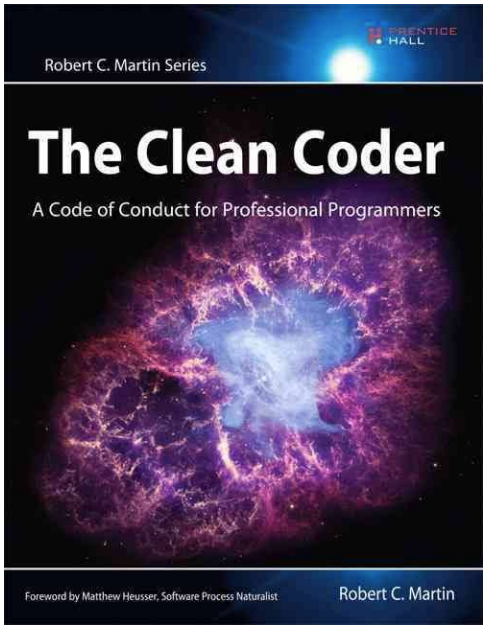
Final thoughts

- 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!