Testing scientific code

Because you’re worth it

Lisa Schwetlick and Pamela Hathway

Introduction to testing project

Excursion: Logistic Map

- Simple, discrete model for population growth

$$f(x) = \overbrace{r * x}^{\text{reproduction}} * \underbrace{(1 - x)}_{\text{starvation}}$$

growth rate, 0…4

current population size, as fraction of maximum possible size, 0…1

Excursion: Logistic Map

- Simple, discrete model for population growth

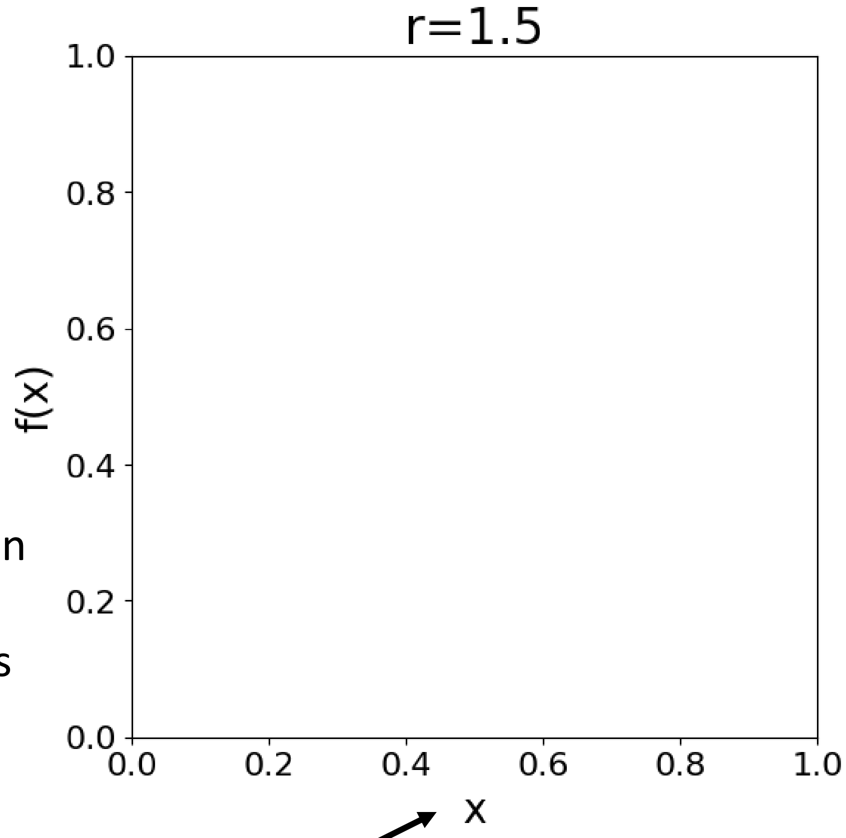
$$f(x) = \overbrace{r * x}^{\text{reproduction}} * (1 - x)$$

starvation

growth rate, 0…4

current population size, as fraction of maximum possible size, 0…1

Next population size (e.g. # bunnies in 2026)



Current population size (e.g. # bunnies in 2025)

Excursion: Logistic Map

- Simple, discrete model for population growth

reproduction

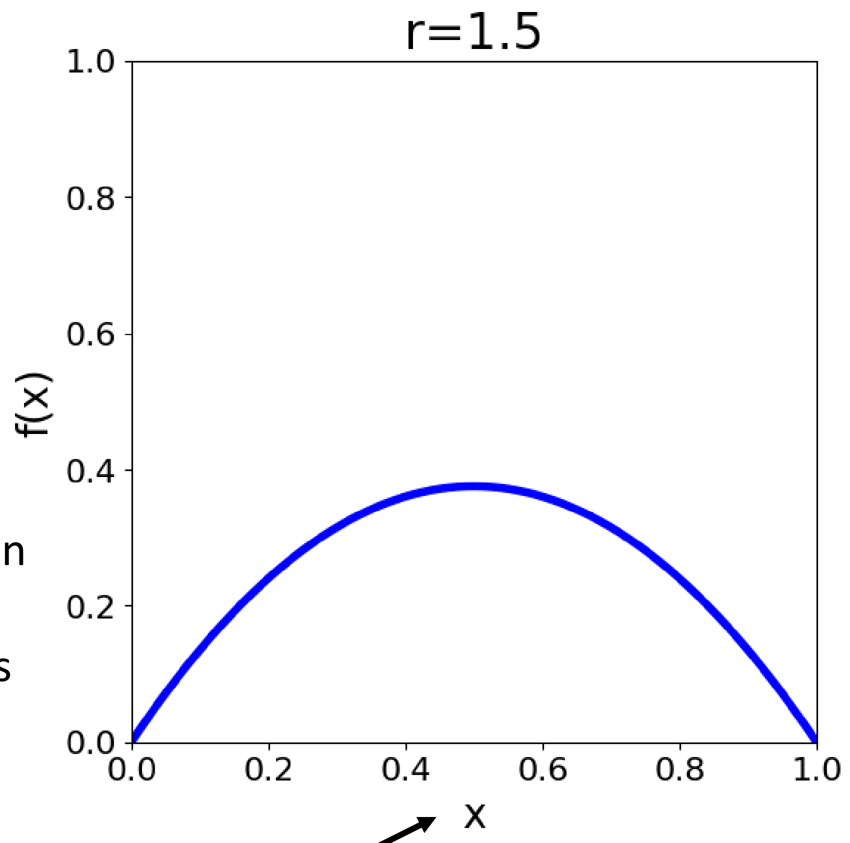
$$f(x) = \underbrace{r * x}_{} * (1 - x)$$

growth rate, 0…4

starvation

current population size, as fraction of maximum possible size, 0…1

Next population size (e.g. # bunnies in 2026)



Current population size (e.g. # bunnies in 2025)

Excursion: Logistic Map

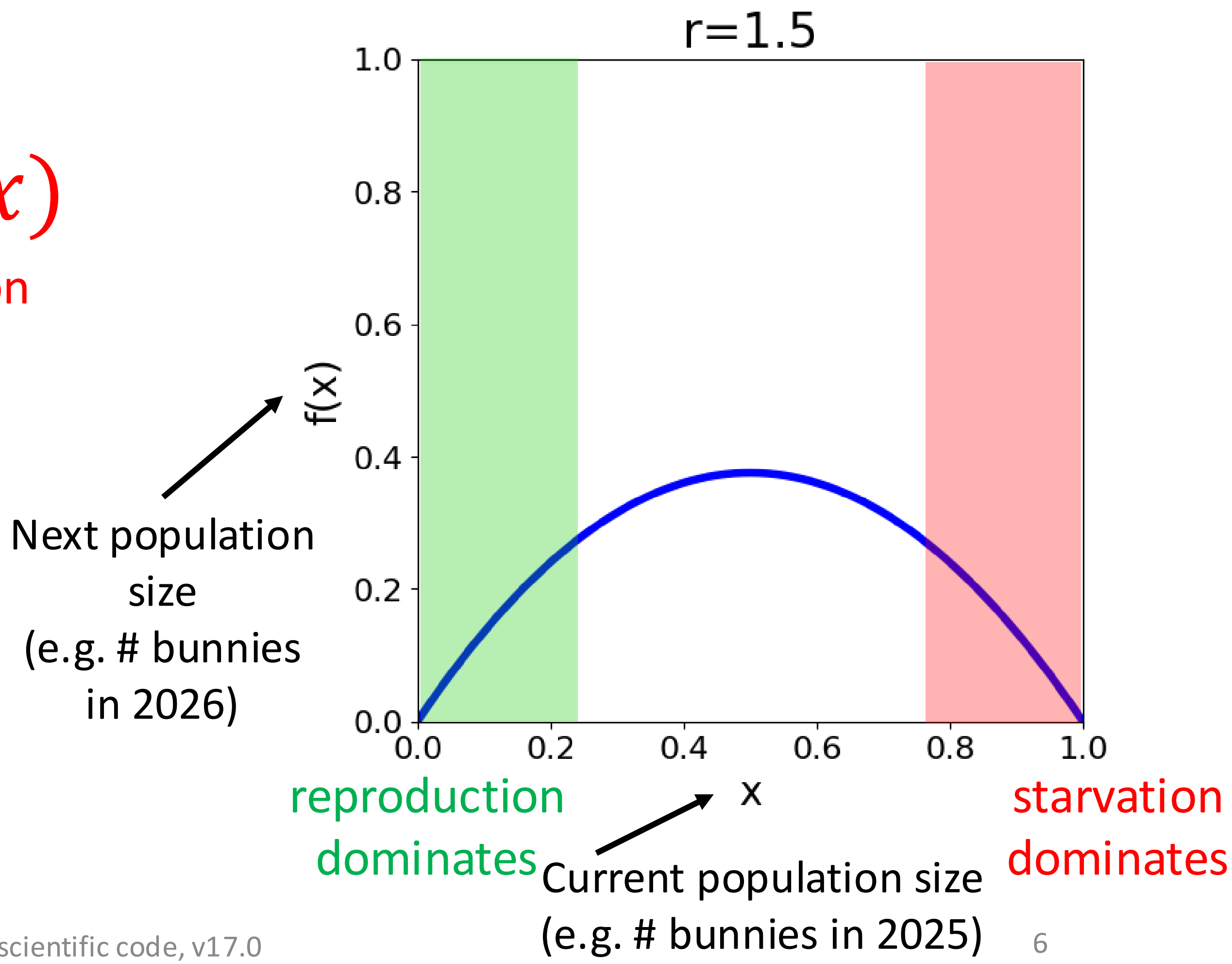
- Simple, discrete model for population growth

$$f(x) = \overbrace{r * x}^{\text{reproduction}} * (1 - x)$$

starvation

growth rate, 0…4

current population size, as fraction of maximum possible size, 0…1



Excursion: Logistic Map

- Simple, discrete model for population growth

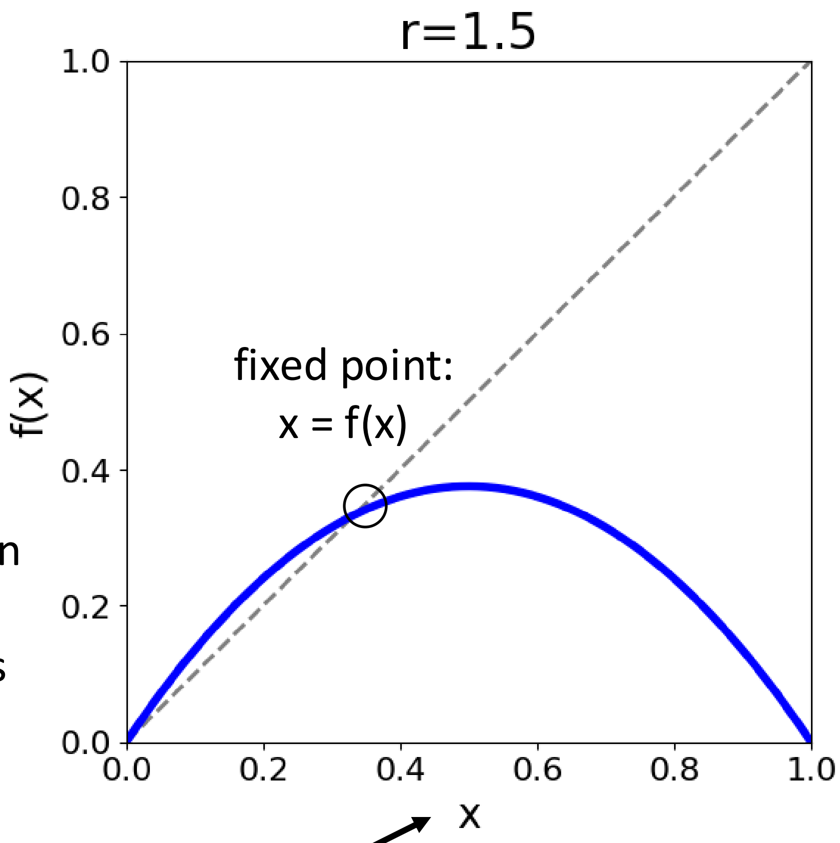
$$f(x) = \overbrace{r * x}^{\text{reproduction}} * (1 - x)$$

starvation

growth rate, 0…4

current population size, as fraction of maximum possible size, 0…1

Next population size (e.g. # bunnies in 2026)

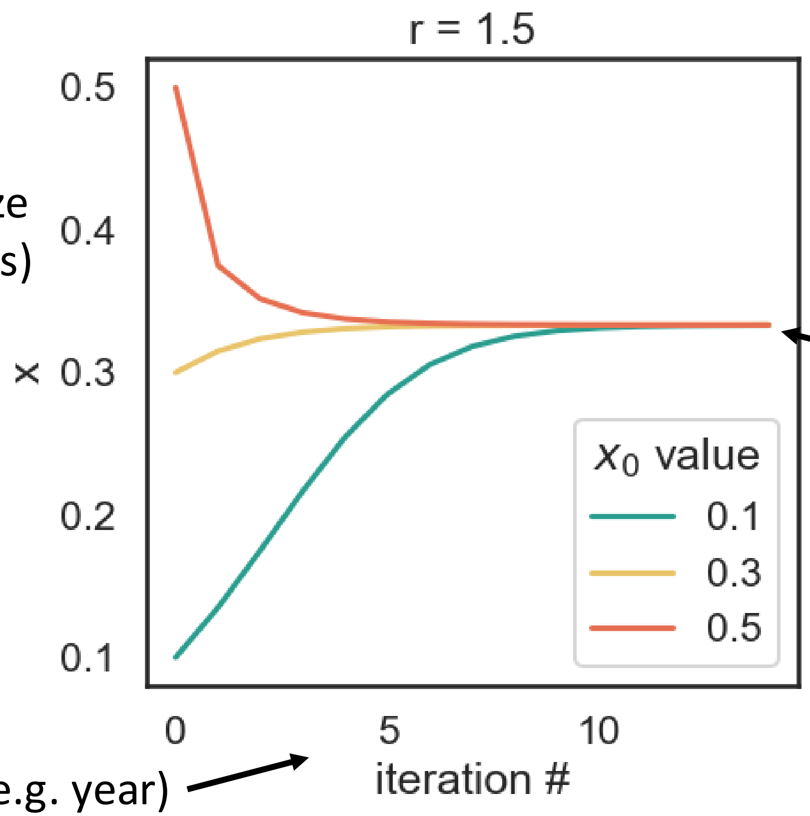


Current population size (e.g. # bunnies in 2025)

Excursion: Logistic Map

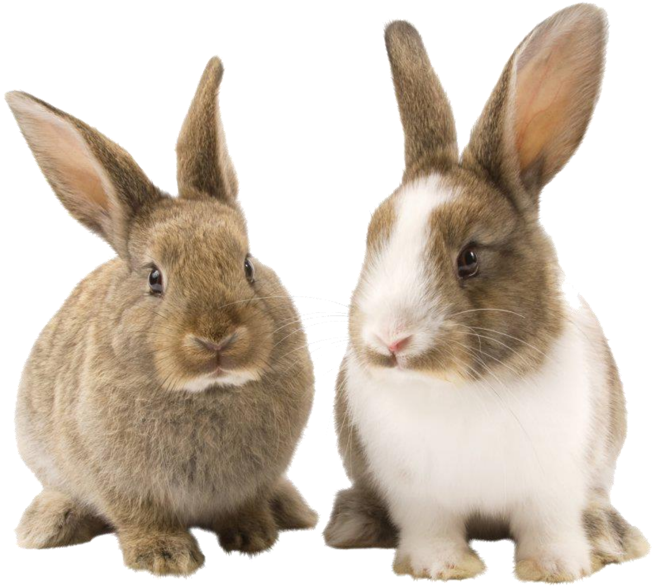
- $\mathbf{x}_0$: initial population size
- Iterated function: $\mathbf{f}(\mathbf{x}_0)=\mathbf{x}_1$ -> $\mathbf{f}(\mathbf{x}_1)=\mathbf{x}_2$ -> $\mathbf{f}(\mathbf{x}_2)=\mathbf{x}_3$

Population size (e.g. # bunnies)



converges to fixed point

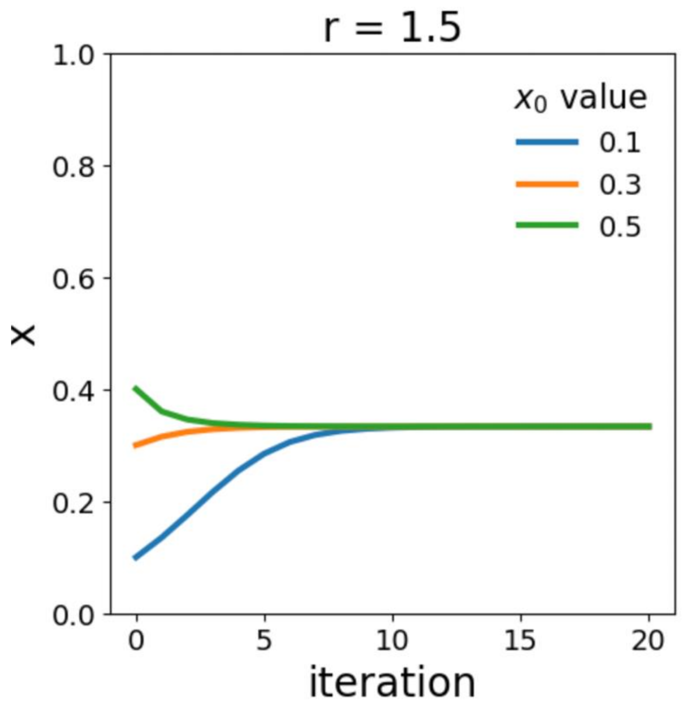
Time (e.g. year)



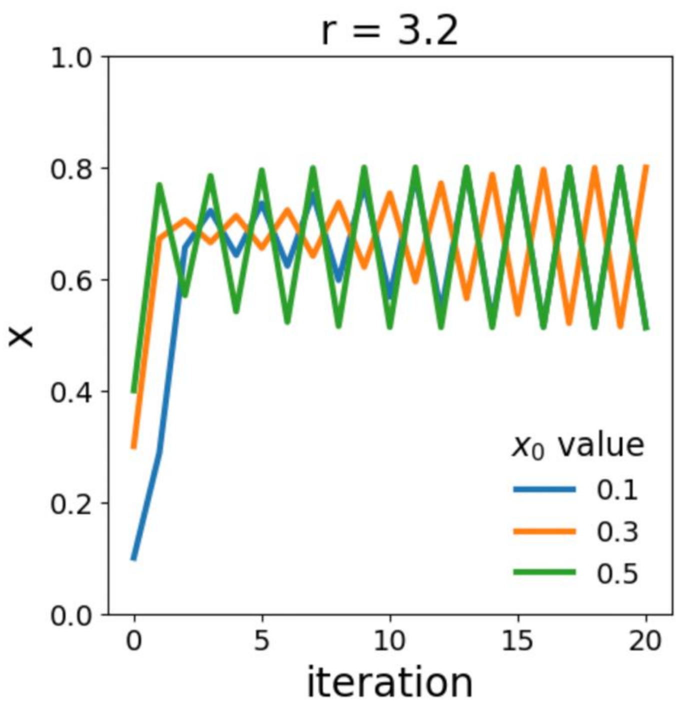
Excursion: Logistic Map

- Different **growth rates** lead to a variety of population dynamics

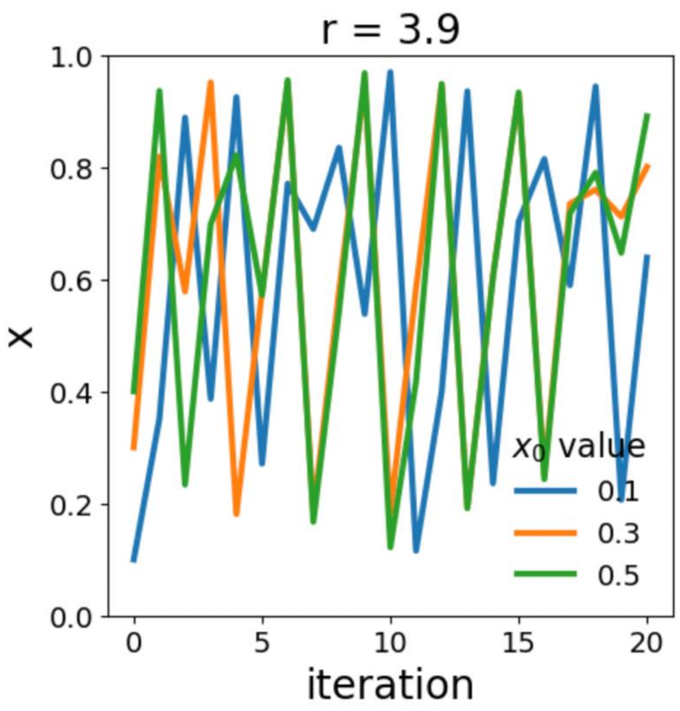
Convergence to fix point



Convergence to oscillations



Chaos



Testing patterns

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

Basic structure of test

- **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 the simplest example

- Example: Tests for ‘lower’ method of strings

```python
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: handle corner case with special behavior, or raise a meaningful exception?

```python
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 (!'`

Similar testcases

```python
def test_lower_hello_world():
    # Given
    string = "HeLlO wOrld"
    expected = "hello world"
    # When
    output = string.lower()
    # Then
    assert output == expected
```

```python
def test_lower_hi():
    # Given
    string = "hi"
    expected = "hi"
    # When
    output = string.lower()
    # Then
    assert output == expected
```

```python
def test_lower_numbers_symbols():
    # Given
    string = "123 ([?"
    expected = "123 ([?"
    # When
    output = string.lower()
    # Then
    assert output == expected
```

```python
def test_lower_empty_string():
    # Given
    string = ""
    expected = ""
    # When
    output = string.lower()
    # Then
    assert output == expected
```

Common for-loop pattern for testing multiple cases

- Often these cases are collected in a single test:

```python
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
```

Discuss!

- Take a look at the logistic map $f(x) = r * x * (1 - x)$
- or, in Python

```python
def f(x, r):
    """ Compute the logistic map for a given value of x and r. """
    return r * x * (1 - x)
```

- What should we test?
    - Generic cases
    - Corner cases

Hands-on 1!

- In the `testing_project` folder, open the file `logistic.py` and implement the logistic function, `f(x, r)`

- In `test_logistic.py` we already added a reference test for these corner cases:
  - `x=0, r=1.1 => f(x, r)=0`
  - `x=1, r=3.7 => f(x, r)=0`

- Add a new test for these generic cases using the for-loop pattern:
  - `x=0.1, r=2.2 => f(x, r)=0.198`
  - `x=0.2, r=3.4 => f(x, r)=0.544`
  - `x=0.5, r=2   => f(x, r)=0.5`

The for-loop pattern can be improved

- It is repetitive to write the for-loop pattern
- If **one of the cases breaks**, 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

```python
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

```
@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

Hands-on 2!

- Rewrite the test with the generic cases for the logistic map using `parametrize`

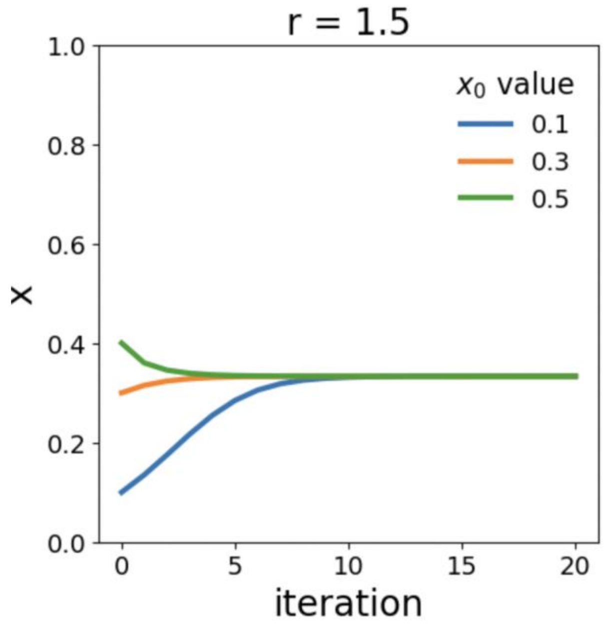
- Reference example for the corner cases test:

```
import pytest

@pytest.mark.parametrize('x, r, expected', [
        (0, 1.1, 0),
        (1, 3.7, 0),
    ]
)
def test_f_special_x_values(x, r, expected):
    result = f(x, r)
    assert_allclose(result, expected)
```

Hands-on 3! Simulate a population over time

1. Implement a function `iterate_f` that runs `f` for `it` iterations. Write tests for the following cases:
   - `x=0.1, r=2.2, it=1`
     `=> iterate_f(it, x, r)=[0.1, 0.198]`
   - `x=0.2, r=3.4, it=4`
     `=> iterate_f(it, x, r)=[0.2, 0.544, 0.843418, 0.449019, 0.841163]`
   - `x=0.5, r=2, it=3`
     `=> iterate_f(it, x, r)=[0.5, 0.5, 0.5]`

2. (Bonus) Import the `plot_trajectory` function from the `plot_logistic` module and use it to visualize the trajectories generated by your code.
   Try with values `r<3`, and `3<r<4` to get an intuition for how the function behaves differently with different parameters.

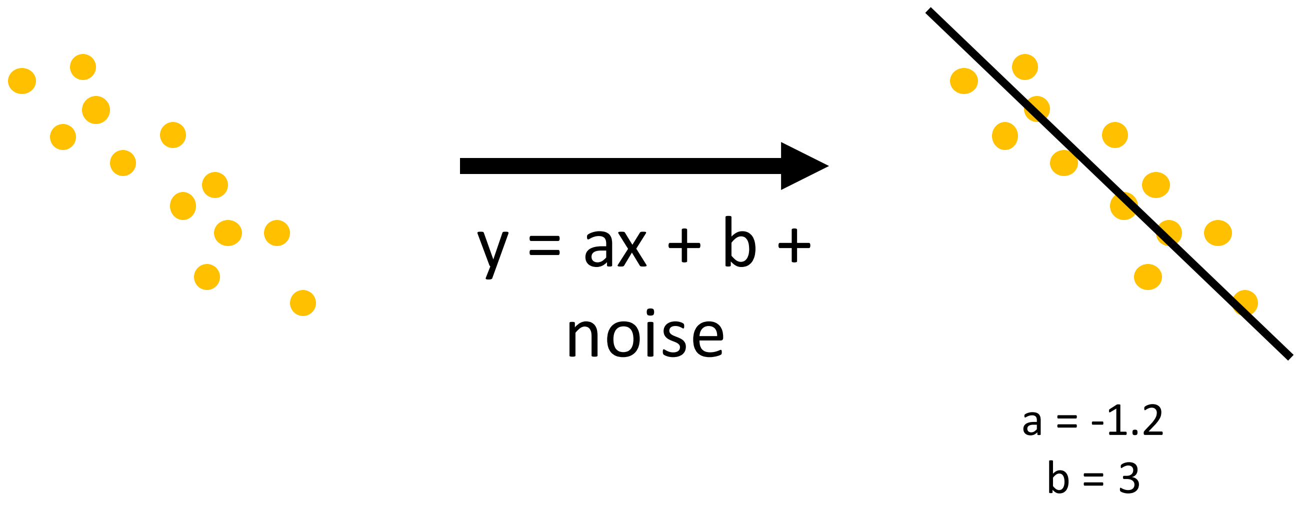


Strategies for testing scientific code

Strategies for testing learning algorithms

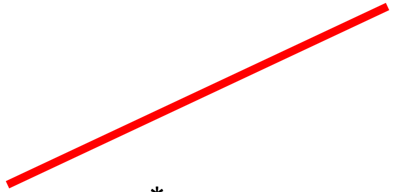
- **Learning algorithms** can get stuck in local maxima, the solution for general cases might not be known (e.g., unsupervised learning)
- Turn your validation cases into tests
- Stability tests:
  - Start from **final solution**; verify that the algorithm stays there
  - Start from solution and **add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

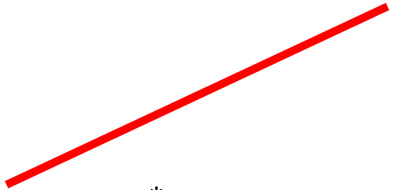


$a^* = 0.5$

$b^* = -1.3$

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

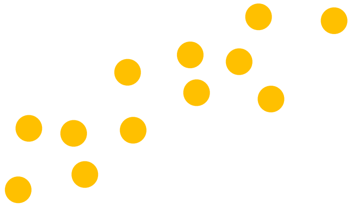
1) Fix initial parameters



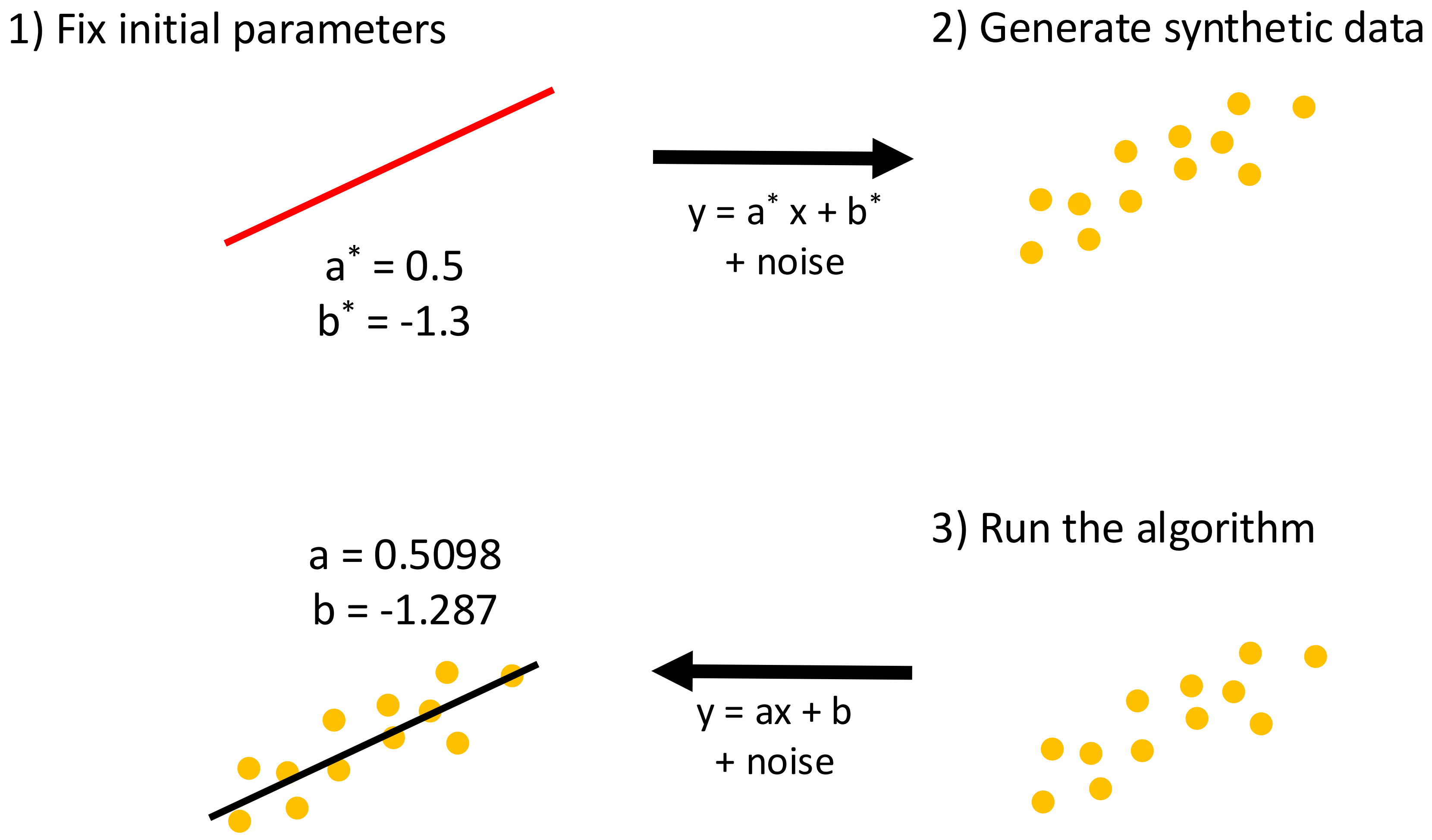
$a^* = 0.5$
$b^* = -1.3$

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

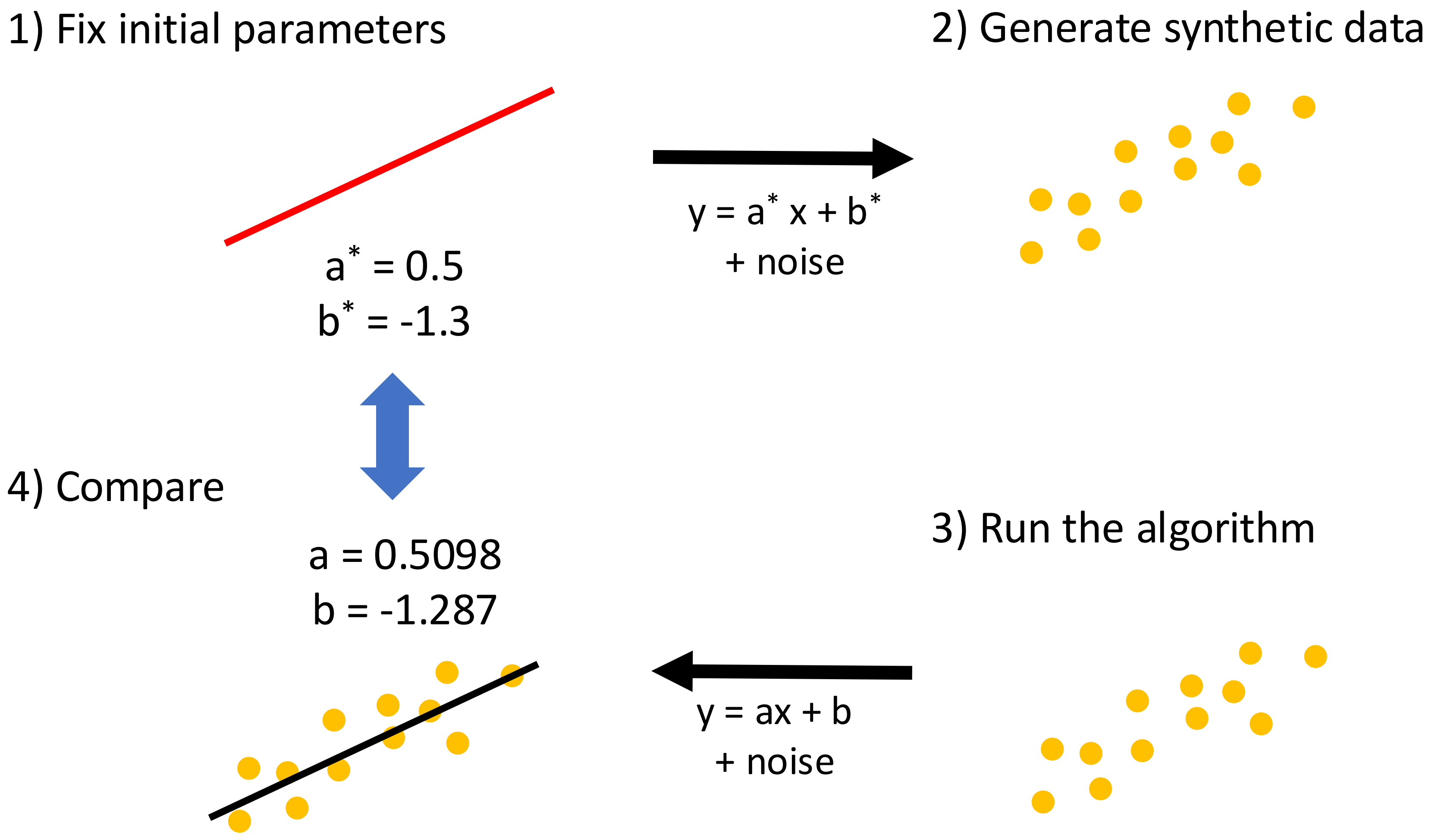
2) Generate synthetic data



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



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



Hands-on 4! Recover the population growth, r

- In the module `logistic_fit`, we implemented a function `fit_r` that, given a population trajectory, finds the value of `r` that generated it

- For example:

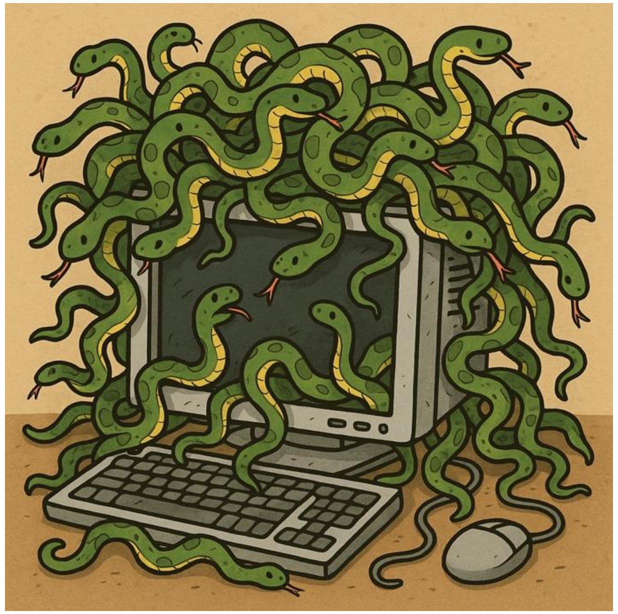
```
In [1]: from logistic import iterate_f
In [2]: from logistic_fit import fit_r
In [3]: xs = iterate_f(it=23, x0=0.3, r=3.421)

In [4]: fit_r(xs)
Out[4]: 3.4210000000000003
```

Hands-on 4!

- Write a test for the function `fit_r` using the parameters recovery method in a new `test_logistic_fit.py` test file.
- The test should
  1. Set a initial value for `x0` and `r`
  2. Use `iterate_f` to generate a population trajectory
  3. Pass the population trajectory to `fit_r` and collect the result parameters
  4. Check that the fitted `r` is close enough to the original `r`

```
In [1]: from logistic import iterate_f
In [2]: from logistic_fit import fit_r
In [3]: xs = iterate_f(it=23, x0=0.3, r=3.421)

In [4]: fit_r(xs)
Out[4]: 3.4210000000000003
```

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()
```

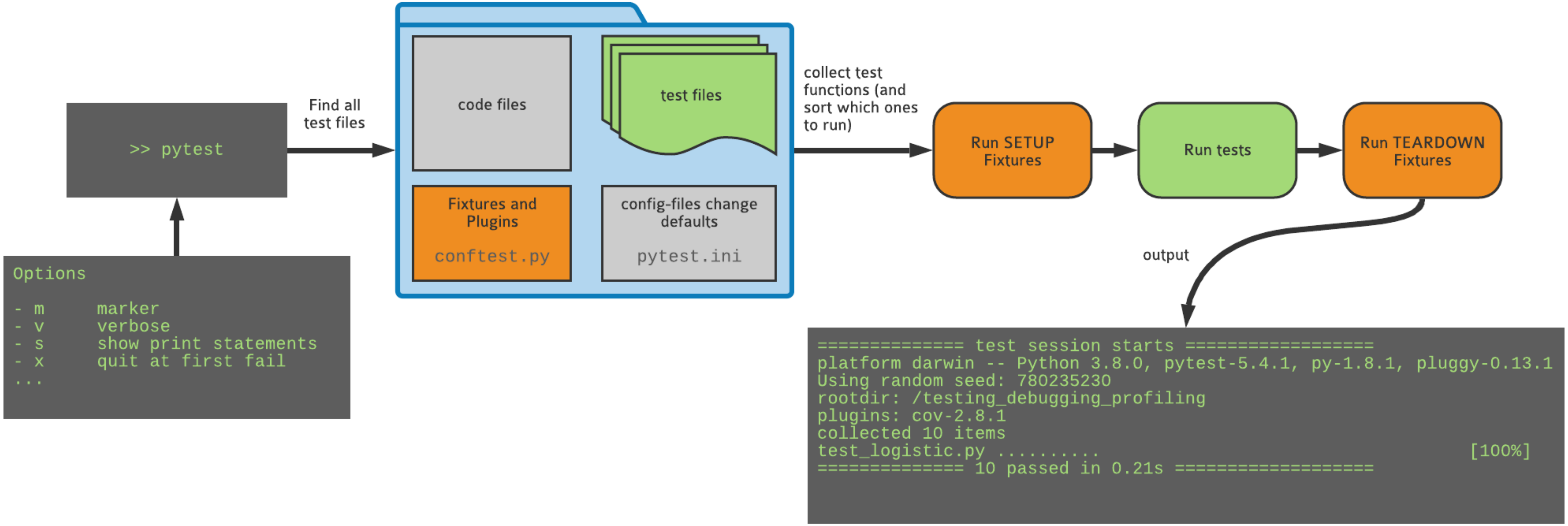
Hands On 5!

a) In `test_logistic_fit.py` write a randomized test that checks that `fit_r` can recover `r` for any random value of `x0` and `r`
   - Write a for loop of 100 iterations, in each iteration create a random `x0` and `r`
   - Test that fit_r(xs) == r , where xs = iterate_f(…)

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?**
  1. For each (random) test there should be a seed
  2. For each run of the test, the seed should be different
  3. That seed should be printed with the test result
  4. It needs to be possible to explicitely run the test again with that seed!

What happens when you run pytest?



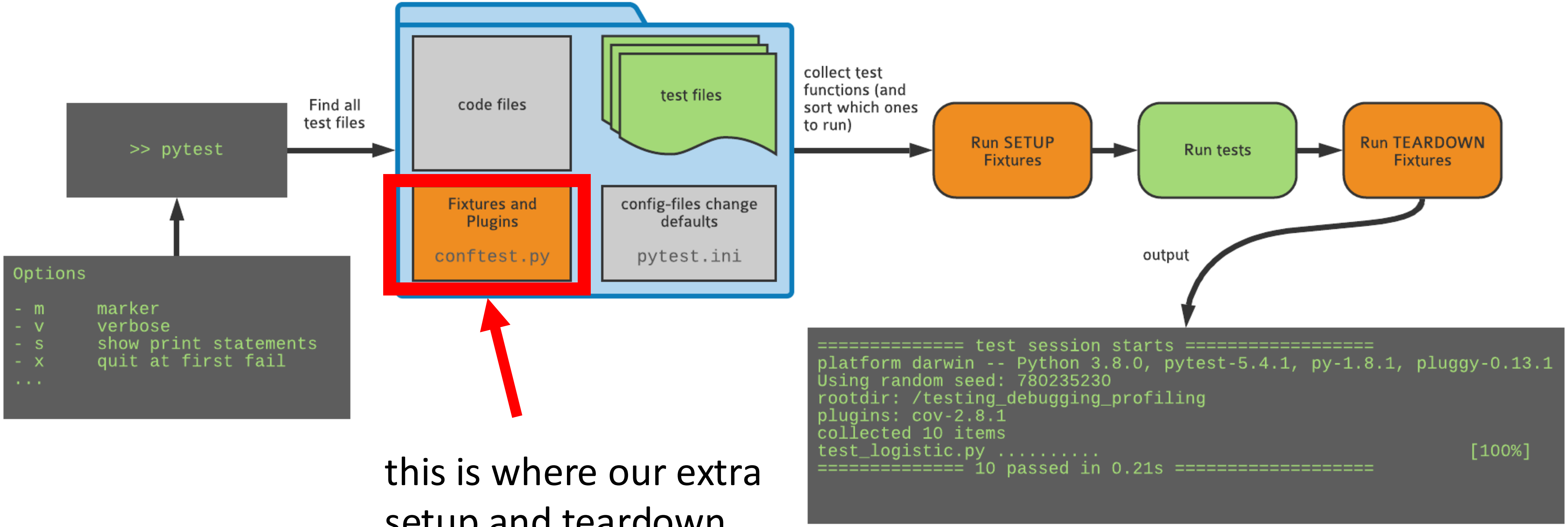
Stop! Fixtures???

- A fixture is a reusable setup tool: mostly used for integration tests/pipeline-level tests
- It’s a function that prepares some data, objects, or state that your test needs, so you don’t have to repeat the same setup code in every test.

e.g.
- Generate some synthetic data
- Create temporary files
- Create some synthetic environment
  - like connection to microscope test environment or a server

e.g.
- delete temporary files
- Shut down some synthetic environment

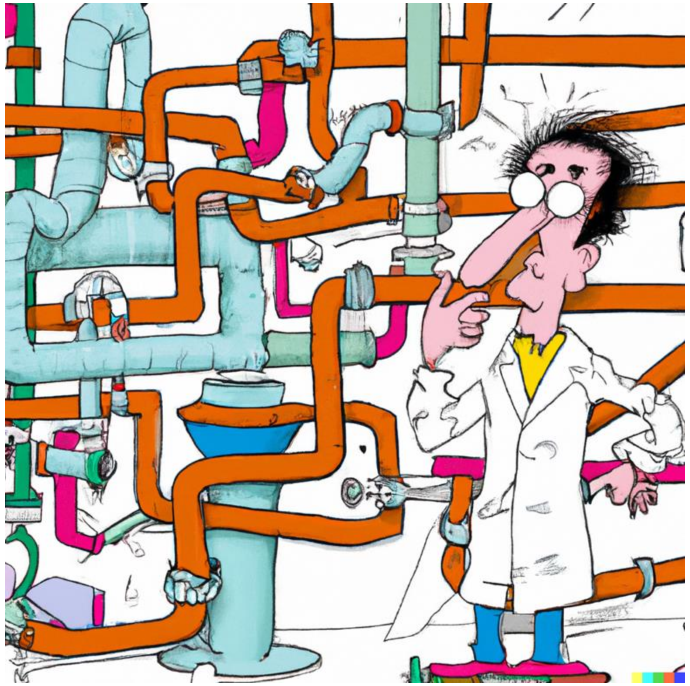
What happens when you run pytest



this is where our extra setup and teardown code can live

Conftest.py

- `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.



```
└─$ pytest
================================================================== test session starts ============
platform darwin -- Python 3.10.6, pytest-8.2.2, pluggy-1.5.0
Using random seed: 740070729
rootdir: /Users/lisa/Documents/Projects/ASPP/2025-plovdiv-testing-debugging/testing_project
configfile: pytest.ini
collected 14 items

test_logistic.py 🌟🌟🌟🌟                                                                 [ 28%]
test_logistic_fit.py 🌟🌟                                                                [ 42%]
test_logistic_parametrize.py 🌟🌟🌟🌟🌟🌟🌟🌟                                              [100%]

===============================
🌟✨ PYTEST PARTY REPORT ✨🌟
===============================

🍾 14 TESTS PASSED 🍾

=====================
🌟✨ TOTAL: 14 ✨🌟
=====================

🎆🎇✨ PYTEST PARTY OVER ✨🎇🎆
```

Setting up randomness

- **What do we want?**
    1. For each (random) test there should be a seed
    2. For each run of the test, the seed should be different
    3. That seed should be printed with the test result
    4. It needs to be possible to explicitely run the test again with that seed!

- **A setup fixture, called before any tests are run:**
    1. Creates a random state for the tests to use
    2. Picks a random SEED by default
    3. Prints that SEED with the test results
    4. Allows us to input a SEED so we can reproduce a speific run

Hands On 5!

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

b) Add the `conftest.py` file the root directory of the project (hint: it is hiding in the demos folder!). It sets a random seed before each run and makes it possible to reproduce failures in random tests

c) `conftest.py` defines a new `random_state` fixture. Modify your test to take `random_state` (the name of the fixture) as an argument. You can then use it like a regular variable:

```
def test_random_convergence_decorator(random_state)
```

d) Check that the console output of pytest now includes the seed and that you can pass a seed too using `--seed 123123`!

```
└─$ 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
```

Other commonly used helpers in pytest

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”
```

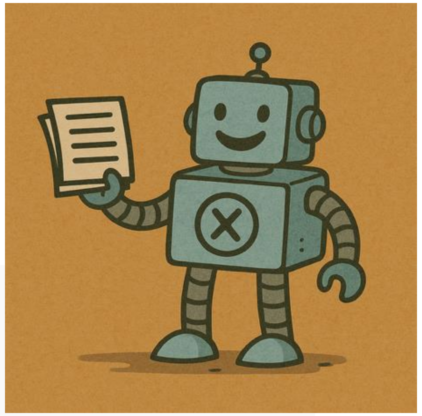
Writing temporary files: `tmp_path`

- 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

```
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

Hi, ChatGPT, write some tests for **logistic.py** !



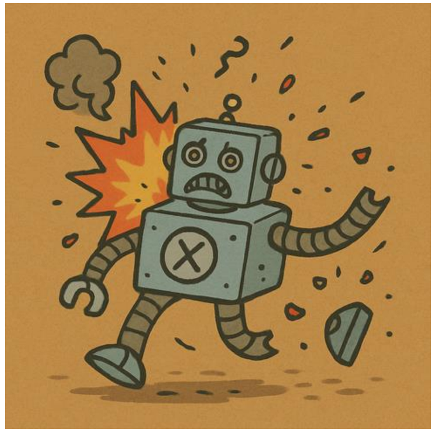
```python
def test_iterate_f_matches_manual_iterations():
    it = 5
    x0 = 0.2
    r = 3.0
    xs = iterate_f(it, x0, r)

    # Compute manually
    manual = [x0]
    x = x0
    for _ in range(it):
        x = f(x, r)
        manual.append(x)

    assert np.allclose(xs, manual)
```

```python
def test_iterate_f_length():
    it = 10
    xs = iterate_f(it, 0.1, 3.5)
    # Should have it+1 values (including x0)
    assert len(xs) == it + 1
```

```python
def test_iterate_f_first_value_is_x0():
    x0 = 0.123
    xs = iterate_f(5, x0, 3.2)
    # First value should be exactly the initial condition
    assert xs[0] == pytest.approx(x0)
```

… aside for generating very satisfying dots in pytest, are these tests useful?

If you *must*....

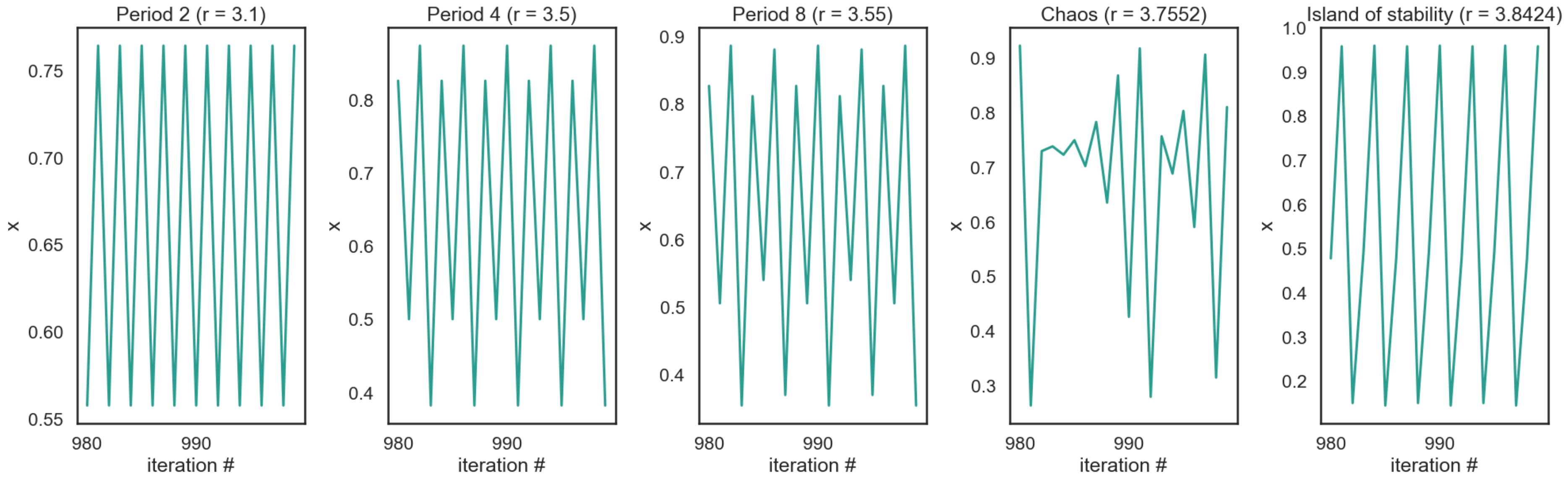


... at least ask yourself:

- Does this test just **re-implement** my function?
- Does this test use my function in a **sane** way/context?
- Does this test check something that can actually **plausibly go wrong**?
  - e.g. the length of a list in a function that just iterates and appends would be quite strange to go wrong.
- Does this test check something that is **critical if it fails**?
  - e.g. checking function does not return NaN -> maybe that's not the worst case
- Does this test use code that I don't **understand**?
  - Or does it use python code that I would never normally use and/or won't understand when I check back in a few months.

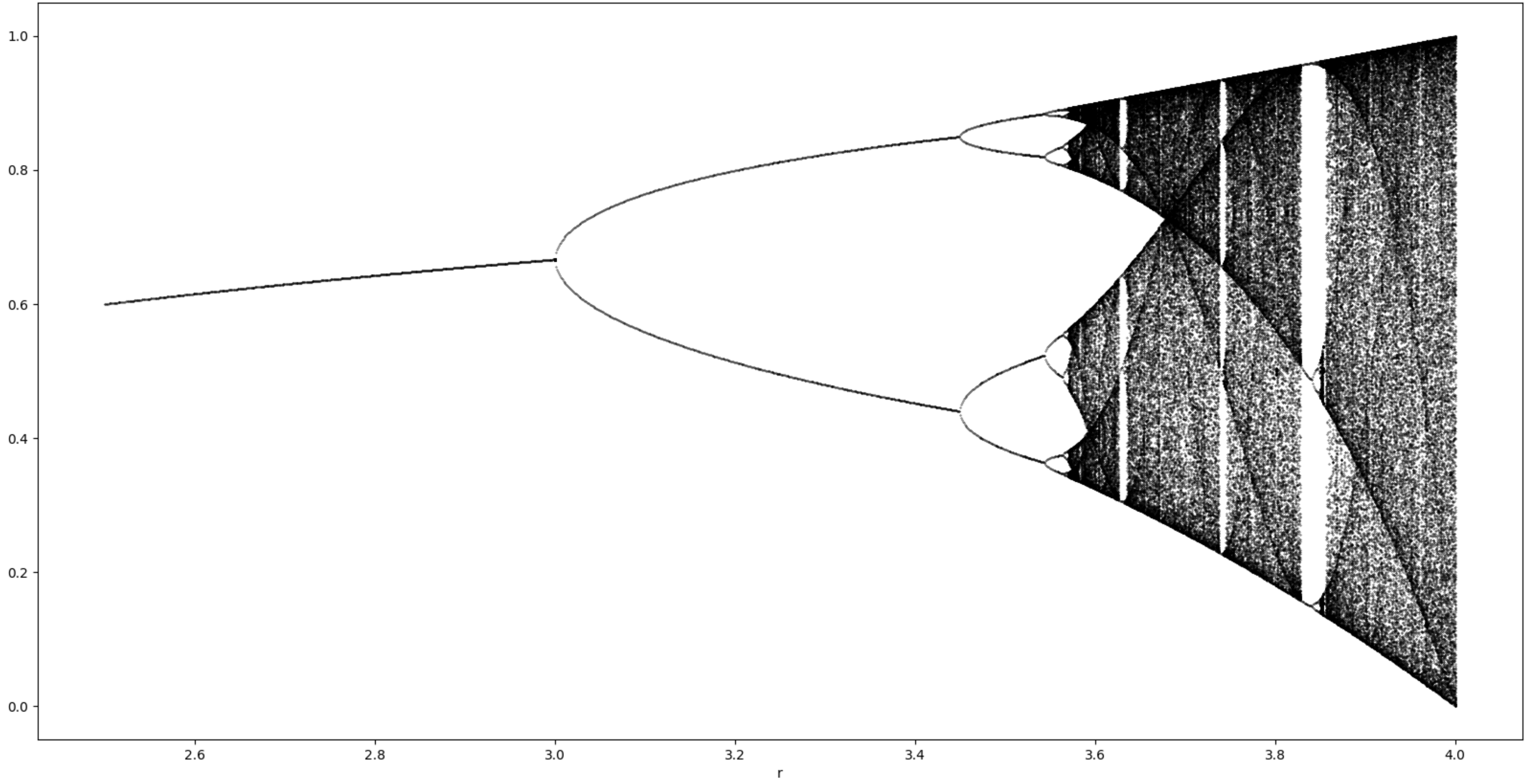
Final exercise

Excursion: Logistic map

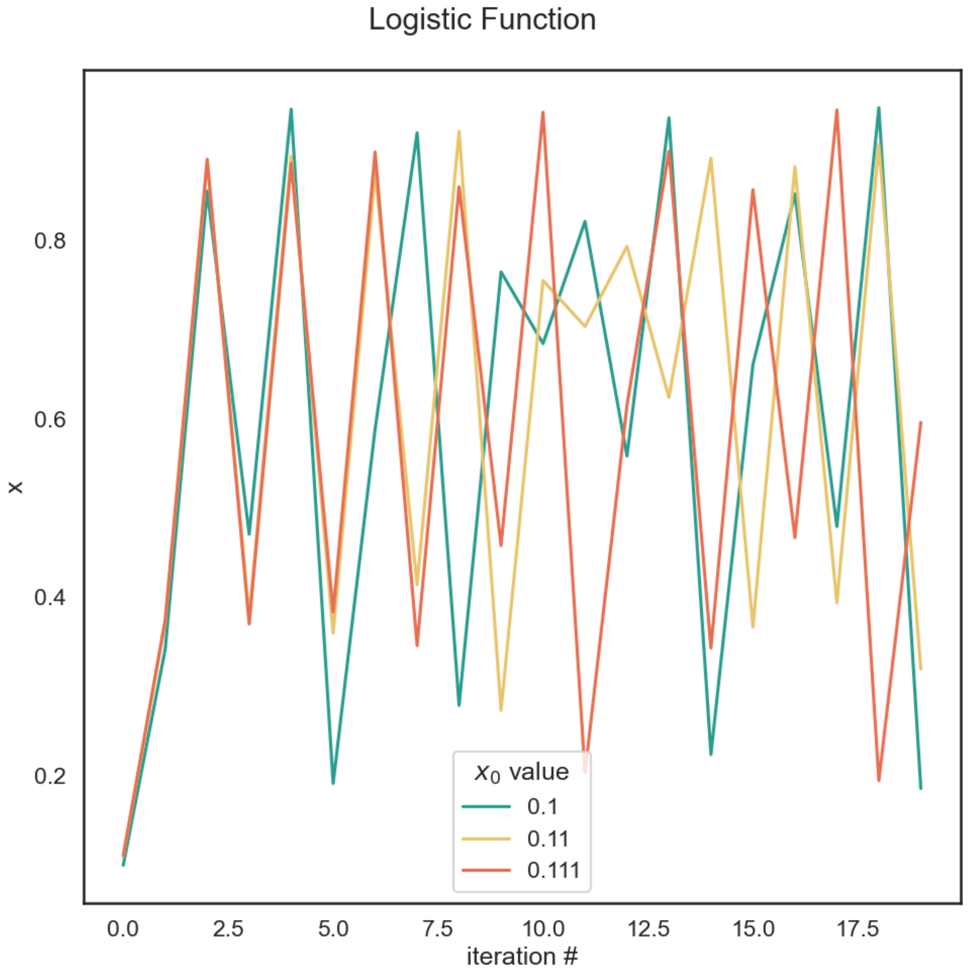


- Between r=3 and r=4 the logistic map has a range of behaviors
- Periodic vs. chaotic

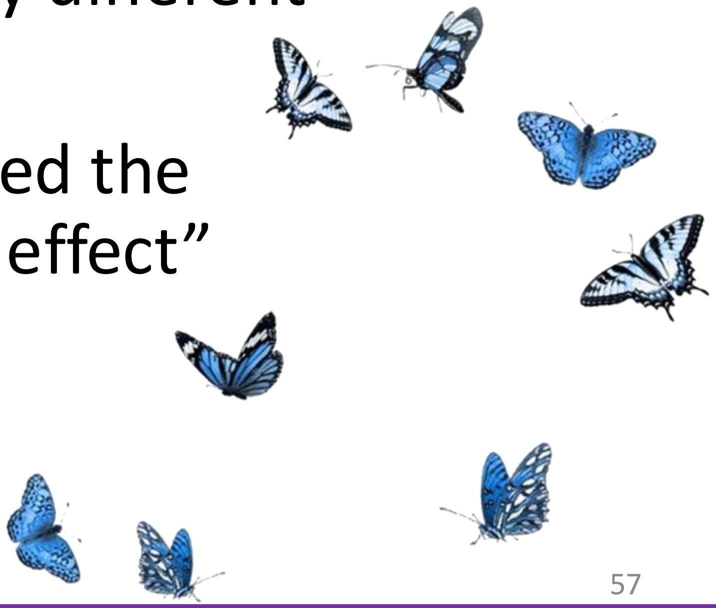
Excursion: Logistic map



Excursion: Logistic map and chaos



- Sensitive Dependence on Initial Conditions (SDIC)
- Even starting points that are very close quickly diverge to completely different itineraries
- This is called the "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` to look at the bifurcation diagram. The function 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 100’000 iterations and verify the conditions for chaotic behavior:

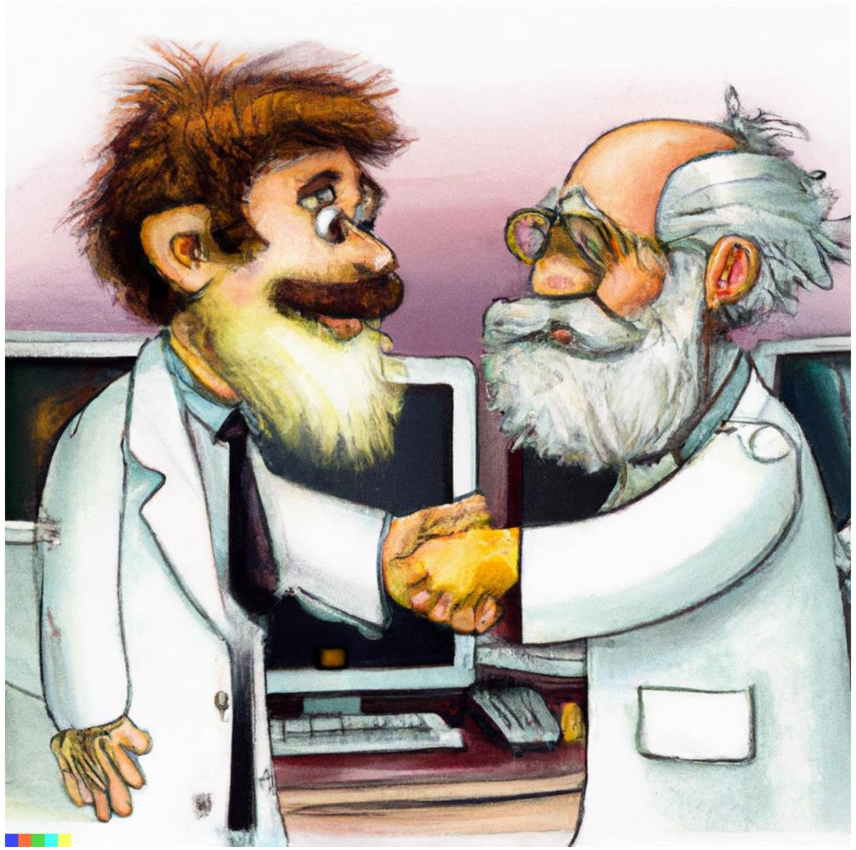
1) The function is deterministic: *this does not need to be tested in this case*
2) Orbits must be bounded: check that all values are between 0 and 1
3) Orbits must be aperiodic: check that the last 1000 values are all different
4) Sensitive dependence on initial conditions: *this is the bonus exercise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

You, in 2025 ⇨ You, in 2026



Up next:

Debugging