Testing scientific code

Because you’re worth it

Lisa Schwetlick and Pamela Hathway

Introduction to testing project

Excursion: Logistic Map

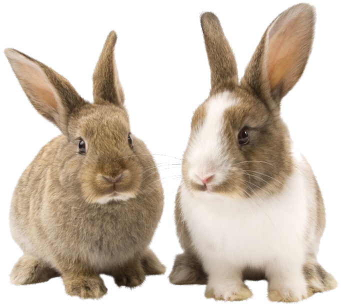
- Simple, discrete model for population growth

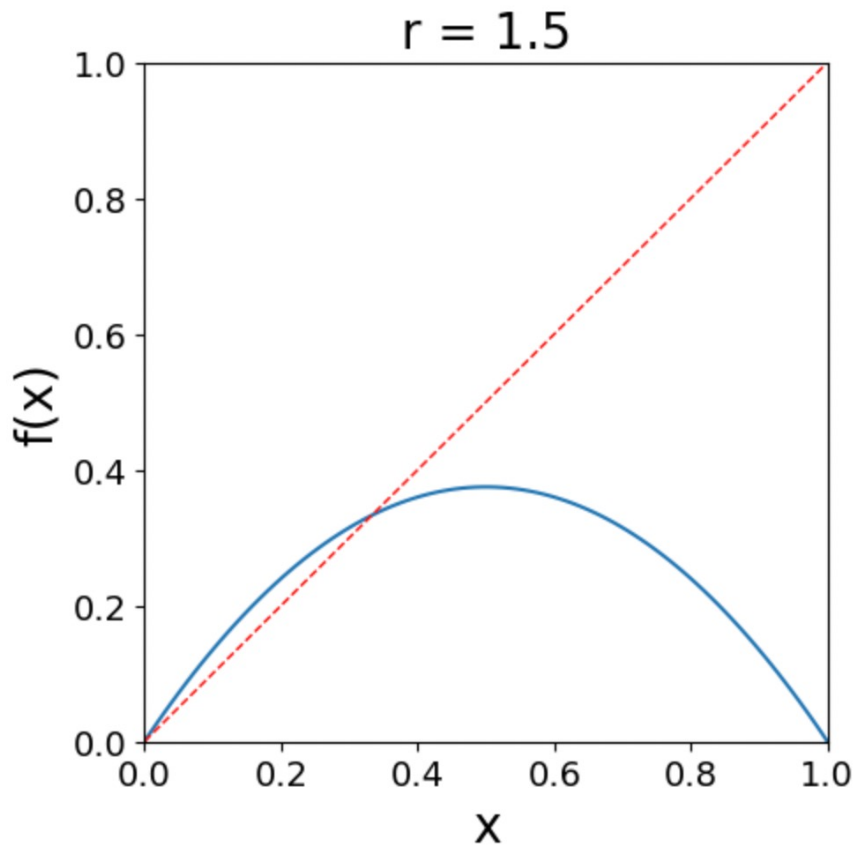
growth rate, 0…4

$$f(x) = r * x * (1 - x)$$

reproduction　　starvation

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





Excursion: Logistic Map

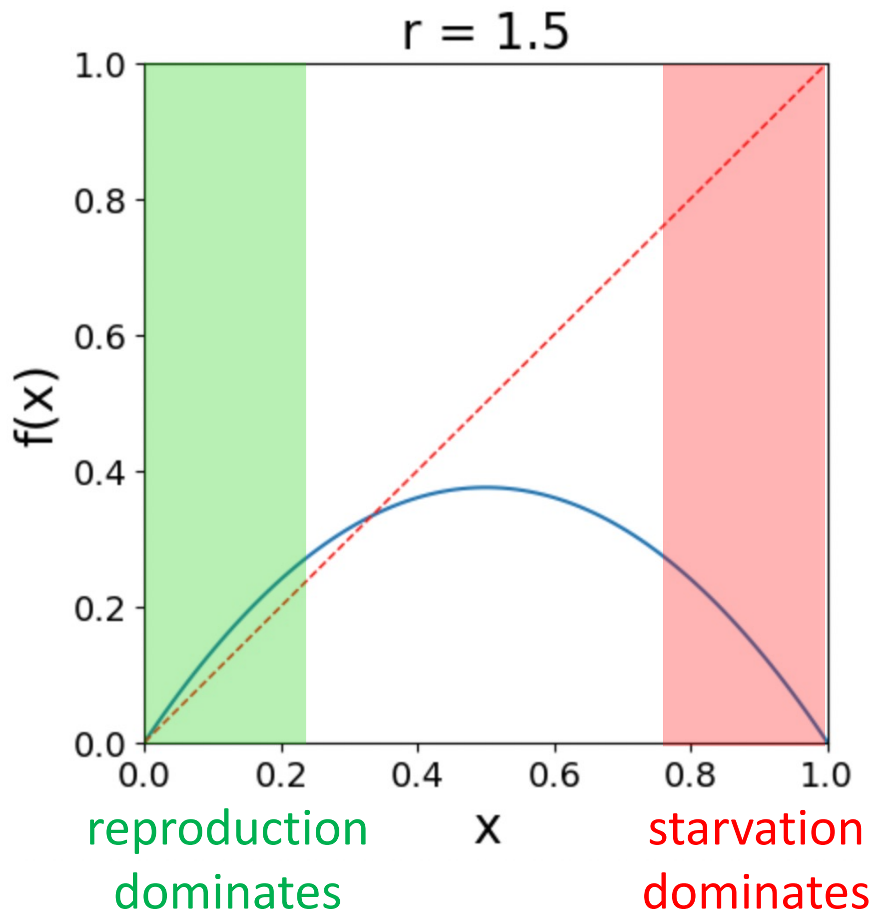
- Simple, discrete model for population growth

growth rate, 0…4

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

reproduction starvation

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



Excursion: Logistic Map

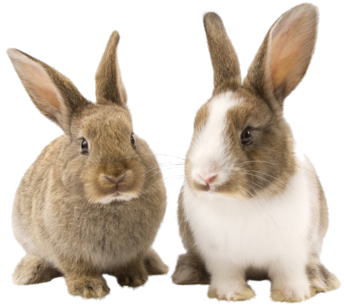
- Simple, discrete model for population growth

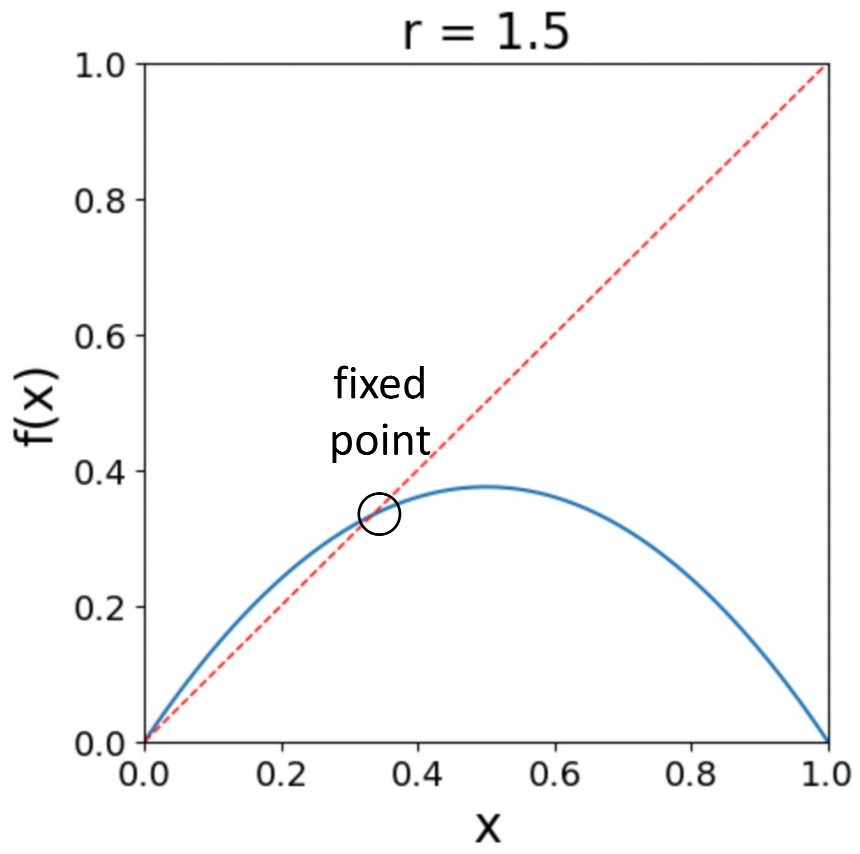
growth rate, 0…4

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

reproduction   starvation

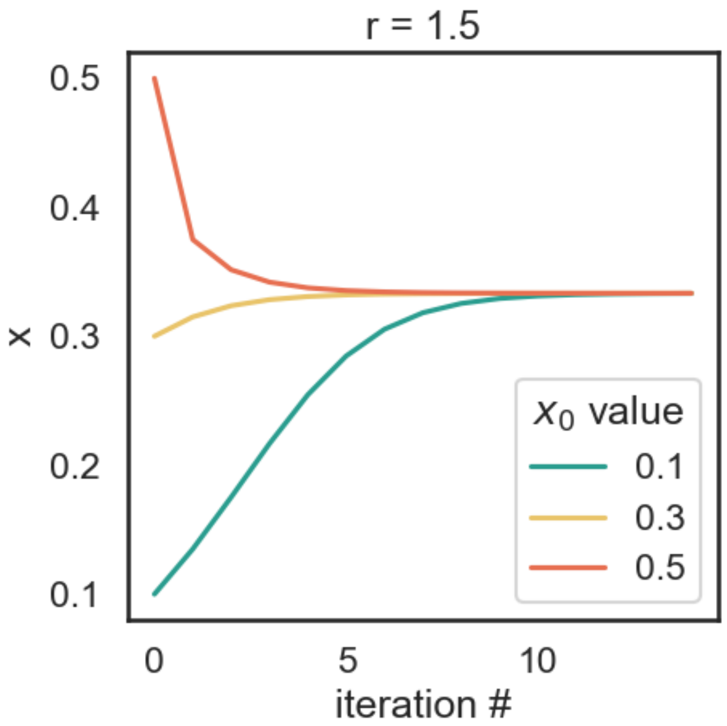
current population size, as
fraction of maximum
possible size, 0…1





Excursion: Logistic Map

- `x0` : initial population size
- Iterated function: `f(x0)=x1` **->** `f(x1)=x2` **->** `f(x2)=x3`

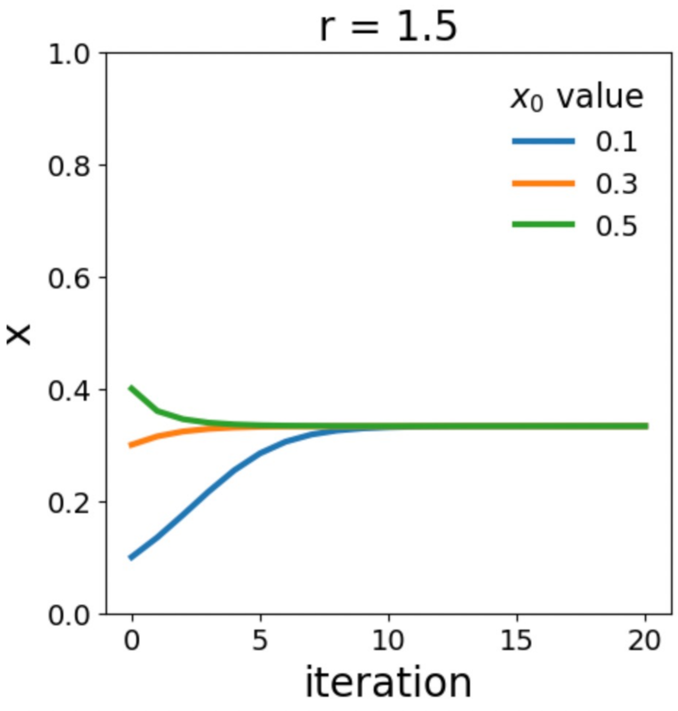


converges to fixed point

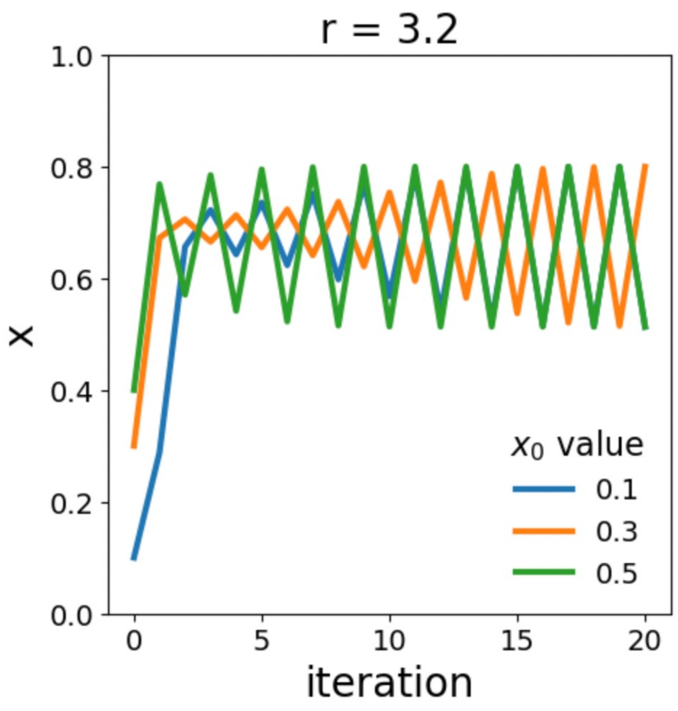
Excursion: Logistic Map

- Different growth rates lead to a variety of population dynamics

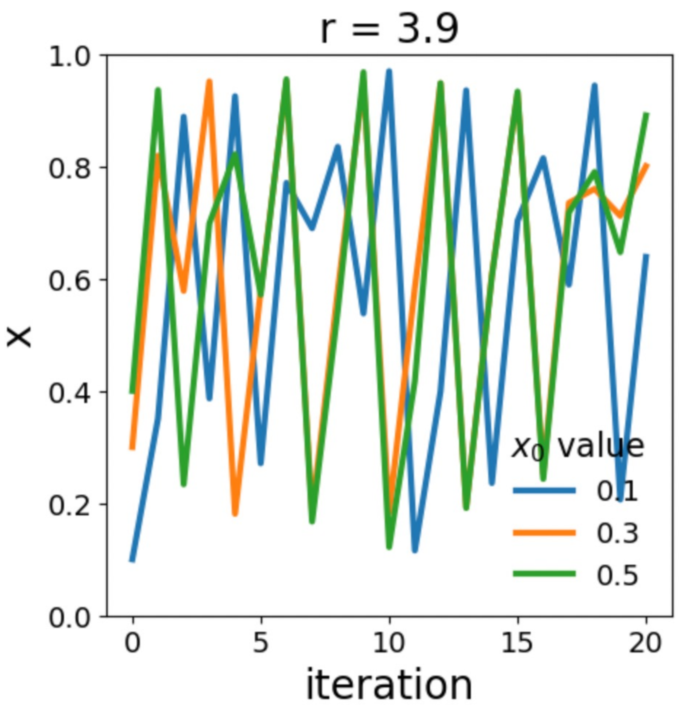
Convergence
to fix point



Convergence
to oscillations



Chaos



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
  - Tests *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 the simplest example

- 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: 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 (!'`

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
```

Hands-on!

- Take a look at the logistic map $\mathrm{f}(x) = r * x * (1 - x)$
- or, in 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! for

- 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 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

```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!

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

- Reference example for the corner cases test:

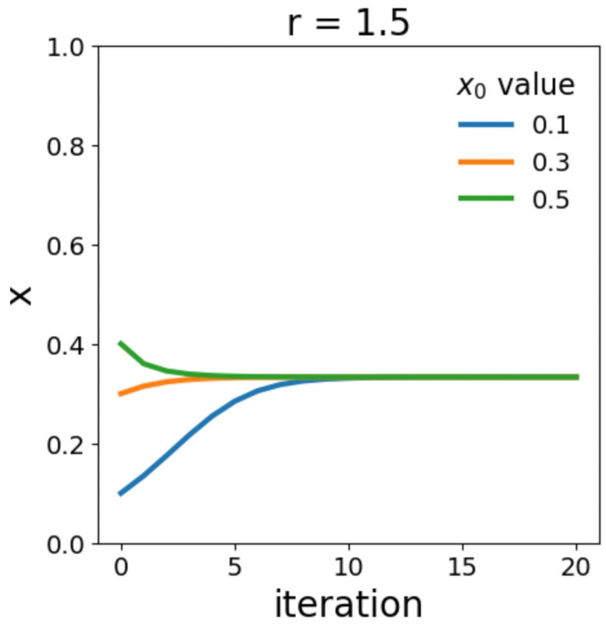
```
import pytest

@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! 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.

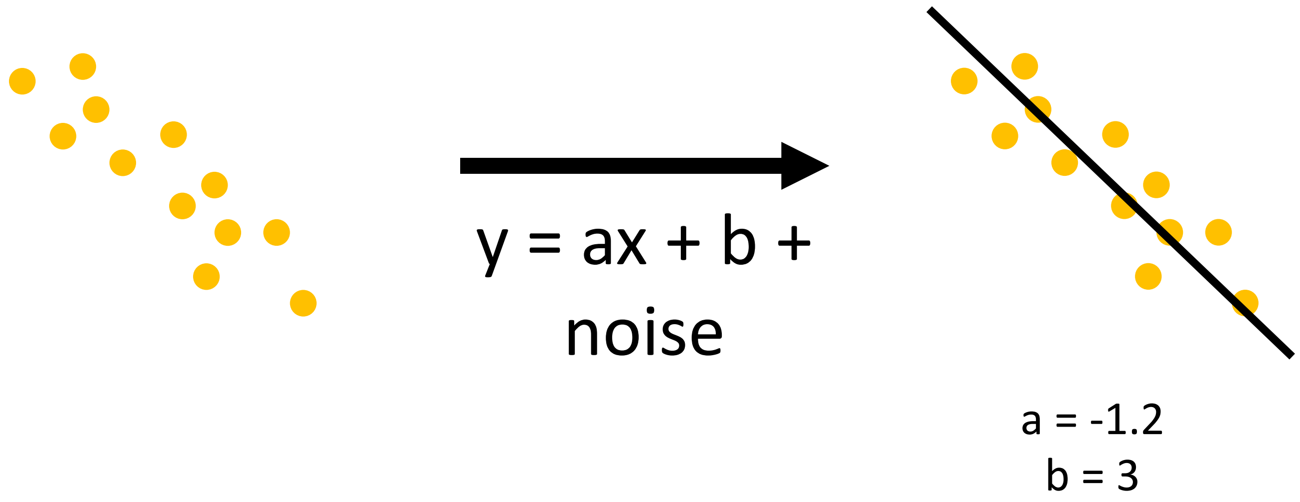


Strategies for testing scientific code

Strategies for testing learning algorithms

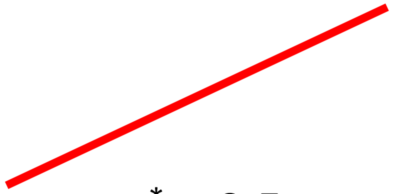
- Learning algorithms can get stuck in local maxima, the solution for general cases might not be known (e.g., unsupervised learning)
- Turn your validation cases into tests
- Stability tests:
    - Start from final solution; verify that the algorithm stays there
    - Start from solution and add a small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

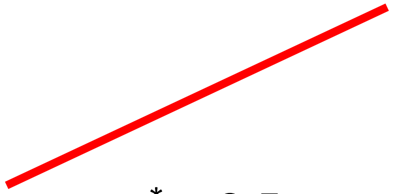
1) Fix initial parameters



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

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

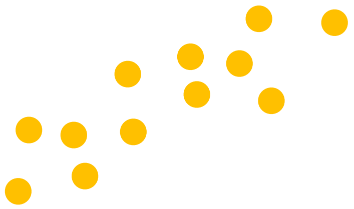
1) Fix initial parameters



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

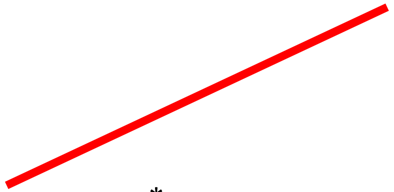
$y = a^* x + b^*$
$+ \text{noise}$

2) Generate synthetic data



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

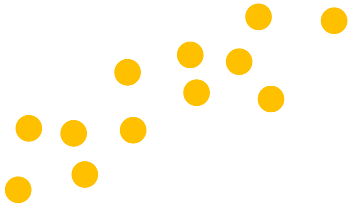
1) Fix initial parameters



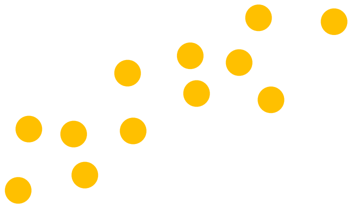
$a^* = 0.5$
$b^* = -1.3$

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

2) Generate synthetic data

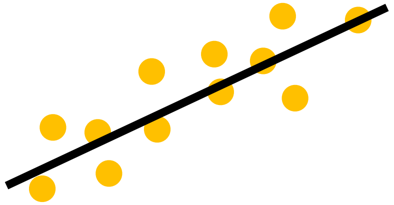


3) Run the algorithm

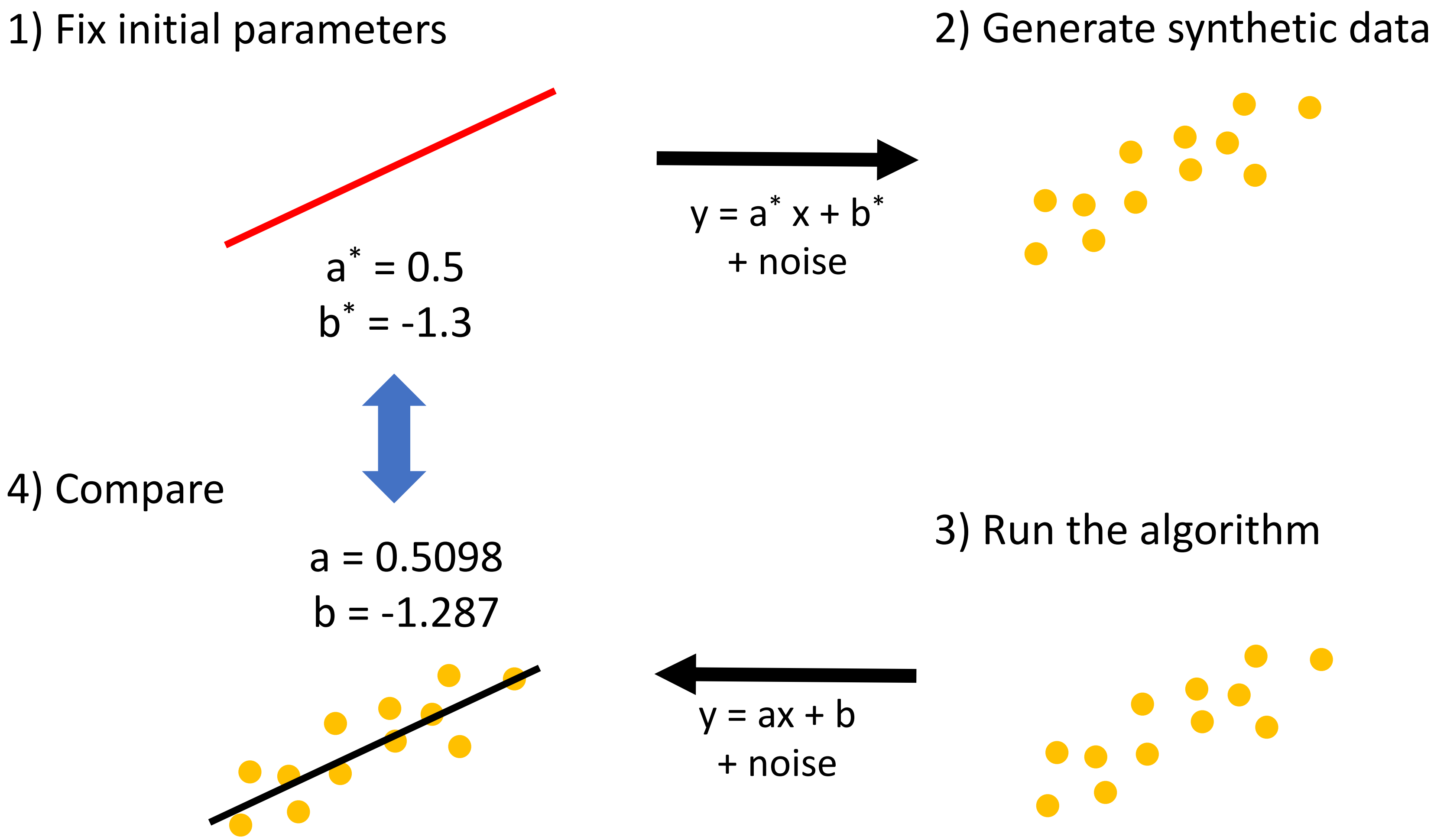


$y = ax + b$
$+$ noise

$a = 0.5098$
$b = -1.287$



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



Hands-on! 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!

- Write a test for the function `fit_r` using the parameters recovery method
- 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
```

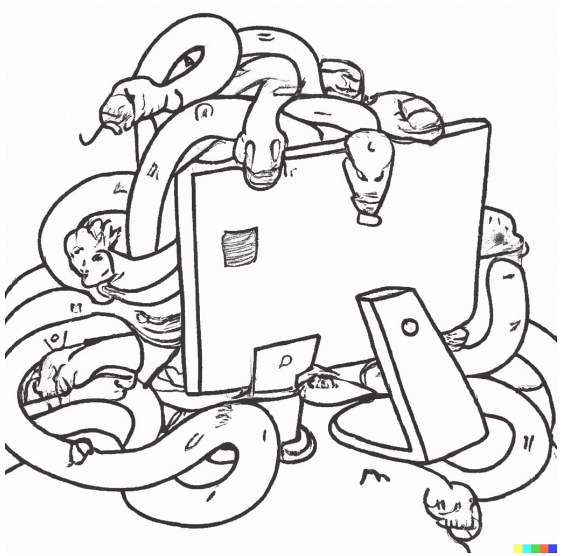
Randomness in Testing

- 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()
```

Hands On!

a) Write a randomized test that checks that for `r=1.5`, for any random starting point `x0`, the logistic equation converges to `1/3`
   - Write a for loop of 100 iterations, in each iteration create a random `x0`
   - For each value of `x0`, test that after many iterations in `iterate_f` the value of x is equal to 1/3

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

```
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.

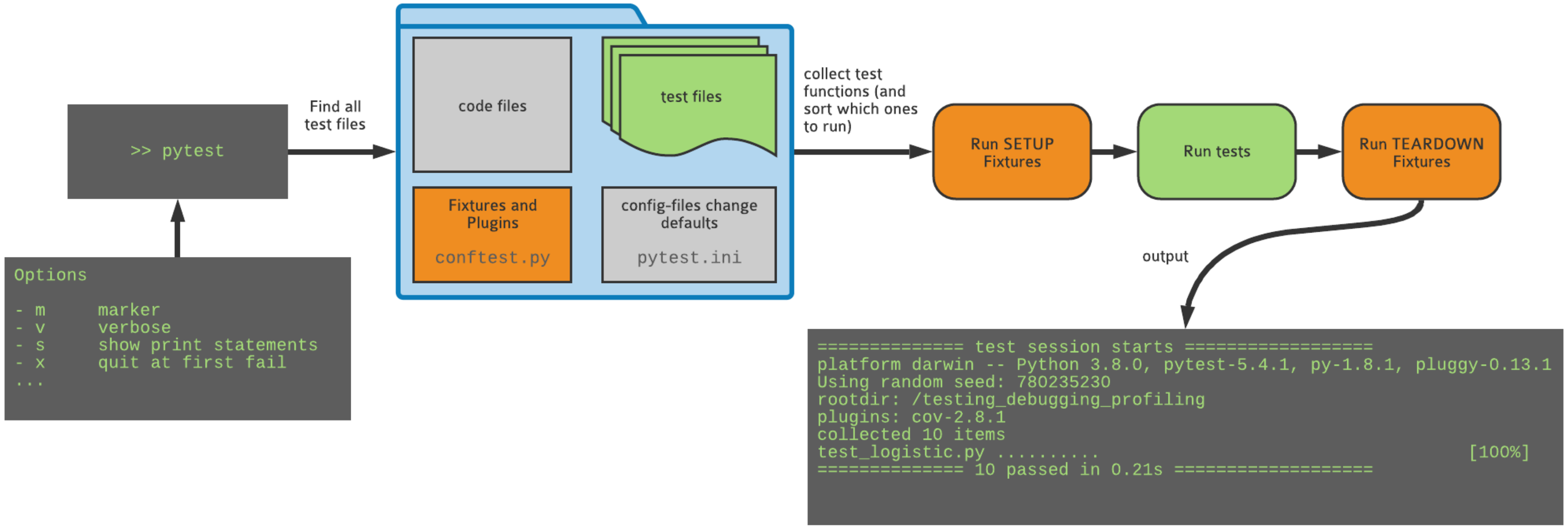
pytest handles that automatically as part of running the test suite

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.

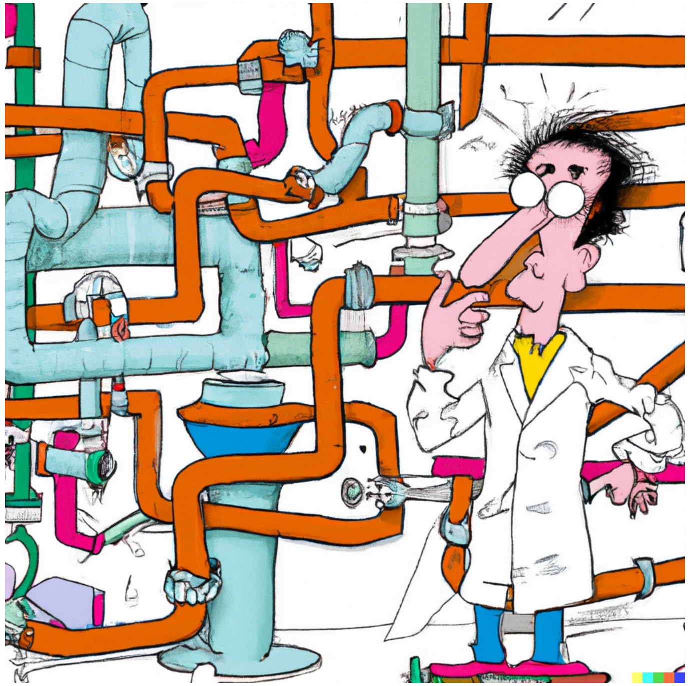
What happens when you run pytest



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 `demos/conftest_example.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`



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.
c) 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
d) `conftest.py` defines a new random_state fixture, modify your test accordingly
e) Check that the console output of pytest now includes the seed!

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

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

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

Writing temporary files: `tmp_path`

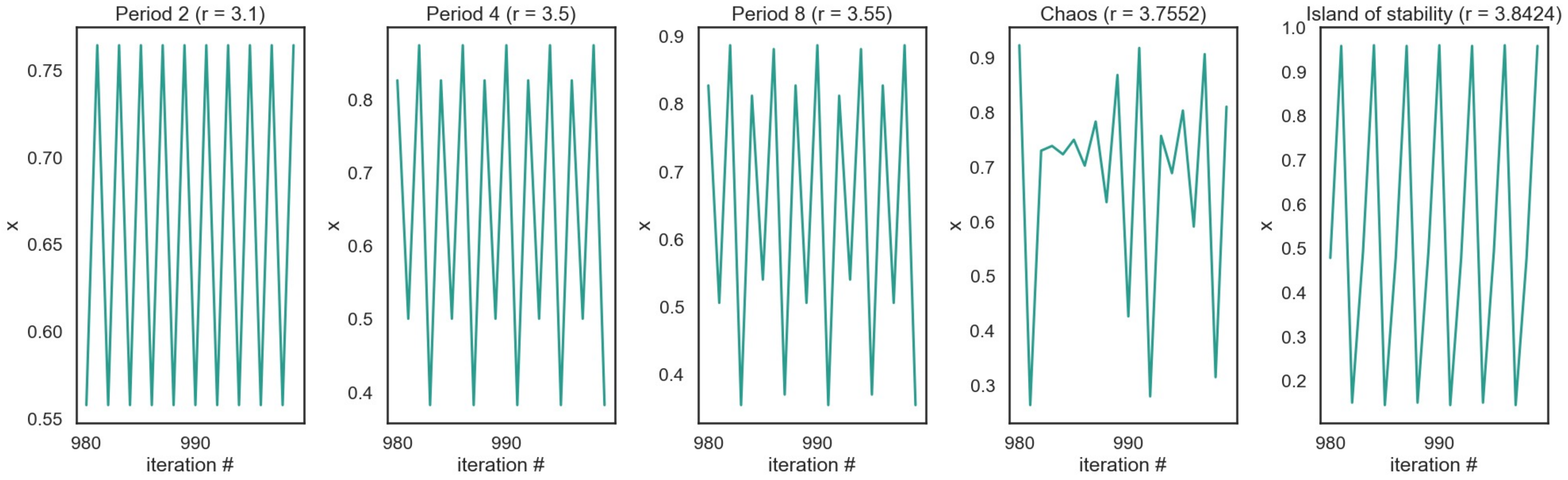
- To test functions that write to disk without leaving around the files when the test is finished, use the `tmp_path` fixture
- The value of `tmp_path` is a `pathlib.Path` object
- The directory is created at the start of the test, and removed at the end

```
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

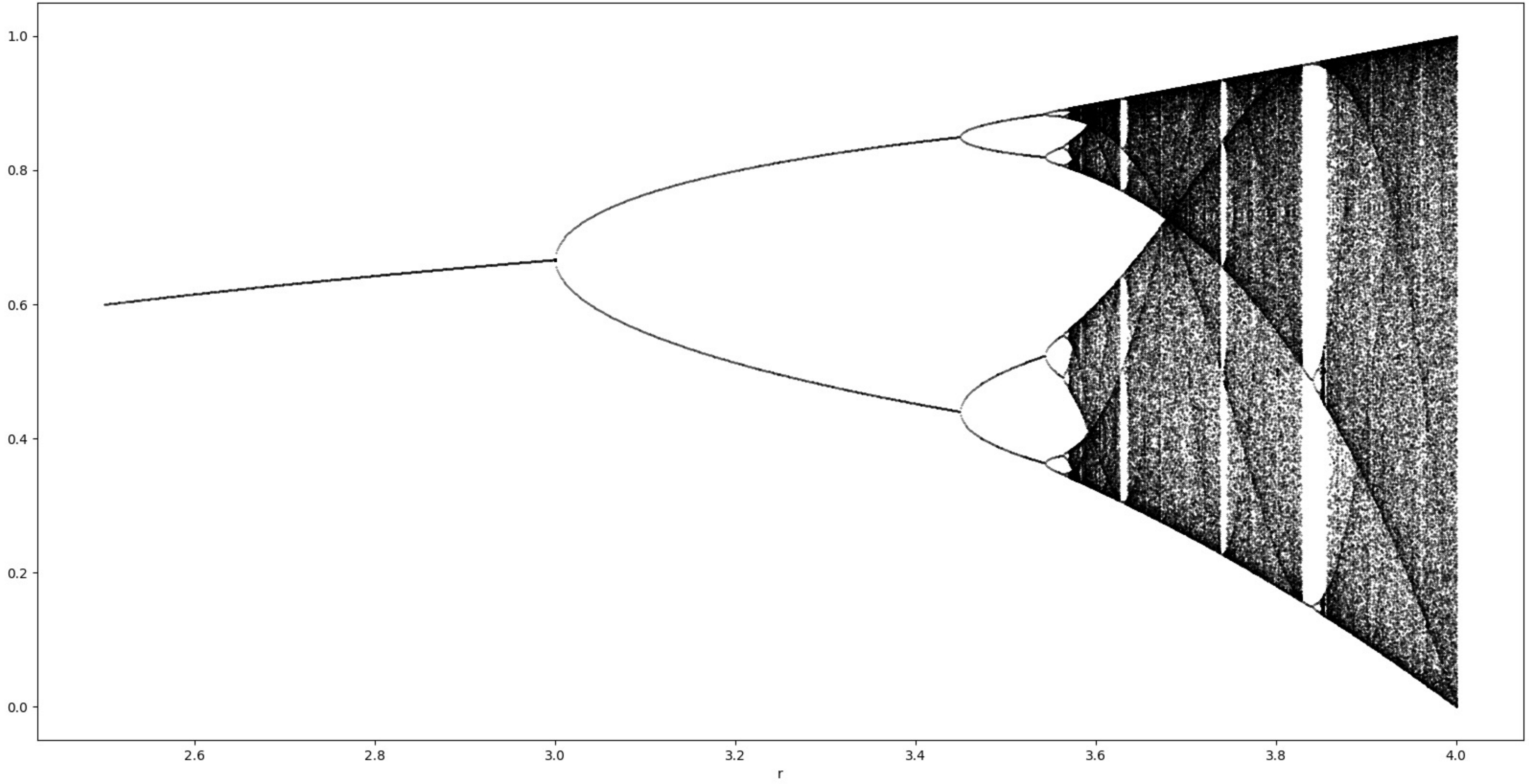
Final exercise

Excursion: Logistic map

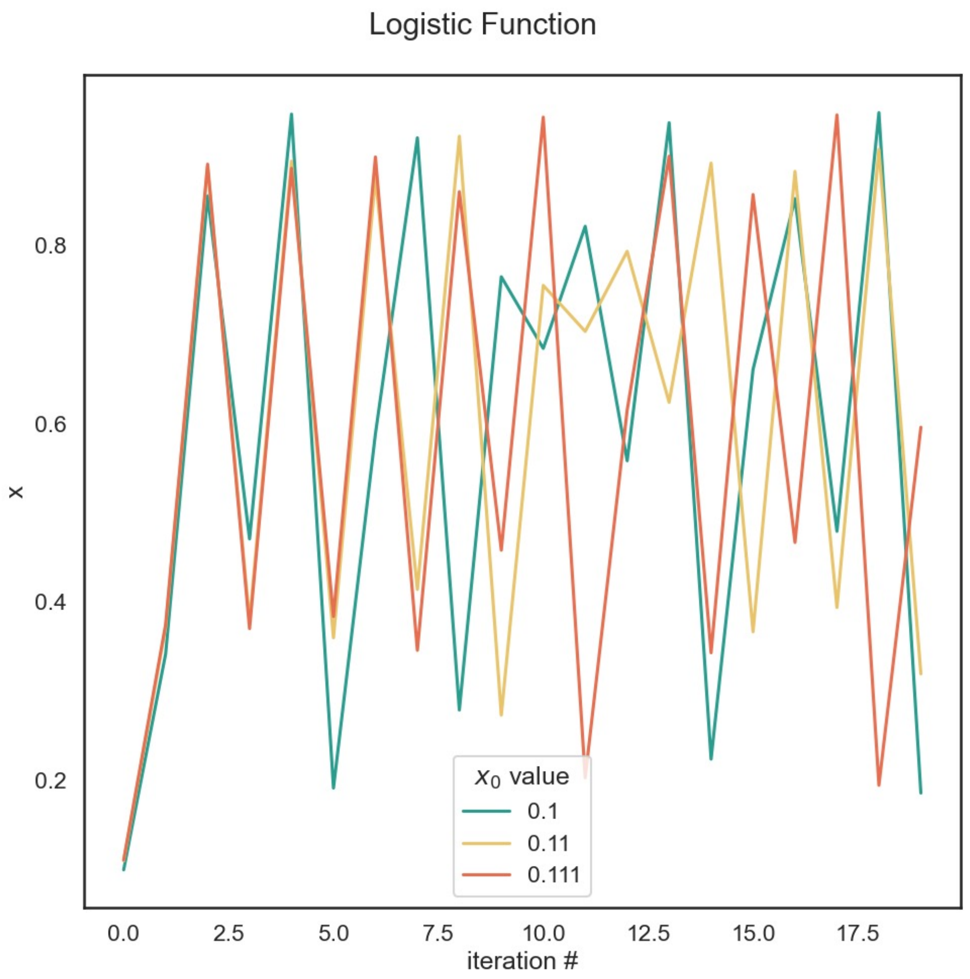


- Between r=3 and r=4 the logistic map has a range of behaviors
- Periodic vs. chaotic

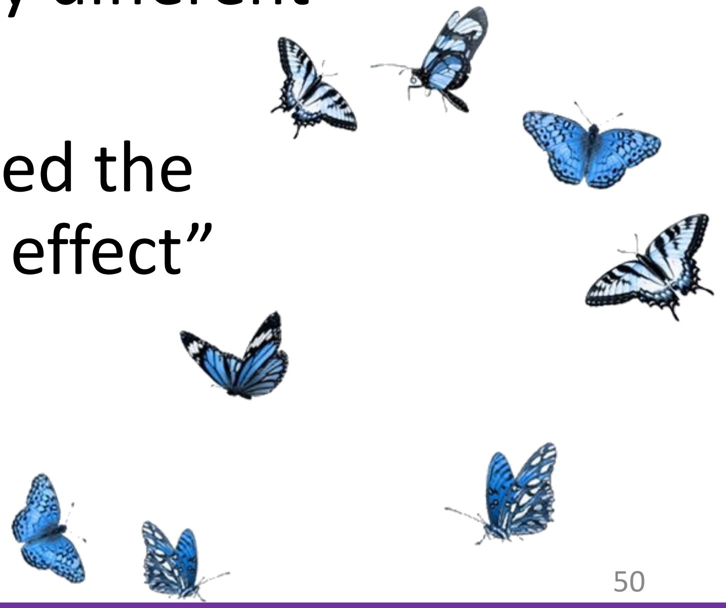
Excursion: Logistic map



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 2023 → You, in 2024

Up next:

Debugging