Software Testing
Advanced Features

Agenda

1. Pytest
2. Mocking
3. Pytest+Mocking Exercise
4. Tox basics*
5. Locust basics*

Pytest

Third-party testing frameworks

- Unittest is a fine and mature testing framework, however…
- There are many other frameworks which accomplish similar or different goals
- Some of them are tailored for certain test types

- One of the most popular test frameworks out there is **pytest**

What makes pytest unique?

Pytest's main features are:

- Detailed info on failing assert statements
- Auto-discovery of test modules and functions
- Modular fixtures (we will get to those later in this module)
- Running unittest (including trial) and nose test suites out of the box
- Plugins

Installing pytest

To install pytest, run `pip install -U pytest`

Example 01-pytest/test_example-01.py

```python
def greeting(name):
    return f"Hello, {name}"


def test_greeting():
    assert greeting("Kate") == "Hello, Kate"
```

```
$ pytest 01-pytest/test_example-01.py
```

```
============================= test session starts ==============================
platform linux-- Python 3.7.2, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/jakub/SDA/software-testing-advanced-features
collected 1 item

01-pytest/test_example-01.py .                                           [100%]

============================== 1 passed in 0.01s ===============================
```

Writing the first test

Notice a few small things regarding the first test:

1. We did not need to import pytest
2. We have used a simple, built-in `assert` statement
3. Test function had `_test` suffix
4. Test file had `_test` suffix
5. We have ran tests using `pytest <path>`, where with unittest we used `python -m unittest <path>`

Running tests with pytest

Notice a few small things regarding the first test:

1. We did not need to import pytest
2. Test function had `test_` prefix
3. Test file had `test_` prefix
4. We have ran tests using `pytest <path>`, where with unittest we used `python -m unittest <path>`

Basic pytest discovery rules

1. File has to be prefixed or suffixed with `test`
2. Test functions have to be prefixed with `test_`
3. Path leading to the file does not have to be changed
4. The discovery search can be narrowed down to a certain path
5. * Run `pytest --collect-only` to see the discovery in action

Exercise 1

1. Run tests in exercise-01 directory only
2. How many tests were discovered?
3. Were all the tests discovered? Why not?
4. Fix the exercise files so that the missing test is also started by pytest

Example 01-pytest/test_example-02.py

```python
def test_greeen():
    assert 1 > 0


def test_identity_fail():
    assert False is True


def test_exception_fail():
    assert 1 / 0
```

```
$ pytest 01-pytest/test_example-02.py
```

```
01-pytest/test_example-02.py .FF                                                    [100%]
================================================== FAILURES ==================================================
_____________________________________________ test_identity_fail _____________________________________________

    def test_identity_fail():
>       assert False is True
E       assert False is True

01-pytest\test_example-02.py:5: AssertionError
_____________________________________________ test_exception_fail _____________________________________________

    def test_exception_fail():
>       assert 1 / 0
E       ZeroDivisionError: division by zero

01-pytest\test_example-02.py:8: ZeroDivisionError
============================================== 2 failed, 1 passed in 0.04s ===============================================
```

Exercise 2

1. Fix the failing assert, so that `test_identity_fail` passes
2. Run pytest again

Handling expected exception

1. `test_exception_fail` is still failing
2. We could use `try… except` to catch the exception here
3. But it is safer and more canonical to use `pytest.raises()` context manager

Example 01-pytest/test_example-03.py

```python
import pytest


def test_expected_exception():
    with pytest.raises(ZeroDivisionError):
        assert 1 / 0
```

```
$ pytest 01-pytest/test_example-03.py
```

```
==================================== test session starts =====================================
platform linux -- Python 3.7.2, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/jakub/SDA/software-testing-advanced-features
collected 1 item

01-pytest/test_example-03.py .                                                         [100%]

===================================== 1 passed in 0.01s ======================================
```

Exercise 3

1. Fix the failing assert tests in `01-pytest/test_exercise-03.py` using `pytest.raises()`
2. Run pytest

Test parametrization

- Sometimes you need to test a function multiple times, but with different parameters
- Instead of creating a separate test for each of the cases…
- You can use `@pytest.mark.parametrize()` decorator instead
- Each of the parameter sets will be treated as a separate test

Example 01-pytest/test_example-04.py

```python
import pytest


def is_palindrome(word):
    if len(word) == 0:
        return False
    return word == word[::-1]


@pytest.mark.parametrize(
    "word,expected_result",
    [
        ("", False),
        ("A", True),
        ("kayak", True),
        ("Kayak", False),  # the function is case-sensitive
        ("noon", True),
        ("afternoon", False),
    ],
)
def test_palindrome_check(word, expected_result):
    assert is_palindrome(word) is expected_result
```

```
$ pytest 01-pytest/test_example-04.py
```

```
================ test session starts ================
platform linux -- Python 3.7.2, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
Rootdir: /home/jakub/SDA/software-testing-advanced-features
collected 6 items

01-pytest/test_example-04.py ......            [100%]

================= 6 passed in 0.03s =================
```

Multiple parametrizations

- We can use more than one `pytest.mark.parametrize()` decorator per test
- In that case, pytest will test all the combinations of provided parameters

Example 01-pytest/test_example-05.py

```python
import pytest


def shorten(text, max_length):
    if len(text) <= max_length:
        return text
    return text[: max_length - 3] + "..."


@pytest.mark.parametrize(
    "text",
    ["A simple statement.", "This is a longer sentence.", "An exclamation! How rude!"],
)
@pytest.mark.parametrize("max_length", [7, 5, 20, 700])
def test_shortened_text(text, max_length):
    short_text = shorten(text, max_length)
    assert len(short_text) > 0
    if len(text) > max_length:
        assert len(short_text) == max_length
```

```
$ pytest 01-pytest/test_example-05.py
```

```
================================ test session starts ================================
platform linux -- Python 3.7.2, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/jakub/SDA/software-testing-advanced-features
collected 12 items

01-pytest/test_example-05.py ............                                     [100%]

================================= 12 passed in 0.30s ================================
```

Exercise 4

1. Use `pytest.mark.parametrize()` to test functions provided in `01-pytest/test_exercise-04.py`

Running select tests

- You can narrow down the discovery to tests with a certain name
- To do that, you have to provide a pattern when calling pytest
- All tests with a name matching the pattern will be ran
- Other tests will be ignored
- `pytest -k <pattern>`

Example 01-pytest/test_example-06.py

```python
def test_int_addition():
    assert 2 + 3 == 5

def test_int_multiplication():
    assert 2 * 2 == 4

def test_float_multiplication():
    assert 2.0 * 1.0 == 2.0
```

```
$ pytest -k int -v 01-pytest/test_example-06.py
```

```
============================= test session starts ==============================
platform linux-- Python 3.7.2, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/jakub/SDA/software-testing-advanced-features
collected 3 items / 1 deselected / 2 selected

01-pytest/test_example-06.py::test_int_addition PASSED                [ 50%]
01-pytest/test_example-06.py::test_int_multiplication PASSED          [100%]
====================== 2 passed, 1 deselected in 0.00s =======================
```

```
$ pytest -k multiplication -v 01-pytest/test_example-06.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.7.2, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/jakub/SDA/software-testing-advanced-features
collected 3 items / 1 deselected / 2 selected

01-pytest/test_example-06.py::test_int_multiplication PASSED          [ 50%]
01-pytest/test_example-06.py::test_float_multiplication PASSED        [100%]

====================== 2 passed, 1 deselected in 0.02s =======================
```

Test markers

- Pytest provides options to mark tests to be skipped or known failing cases
- These markers should not be overused, since they may obscure real problems within the tested code
- To skip a test use `@pytest.mark.skip(reason=„”)`
- To mark a known failing test use `@pytest.mark.xfail`

Example 01-pytest/test_example-07.py

```python
import pytest


@pytest.mark.skip(reason="function deprecated")
def test_foo():
    assert foo() == 1


def get_from_db():
    return NotImplemented


@pytest.mark.xfail
def test_get_from_db():
    # Represents future expected behaviour
    result = get_from_db()
    assert isinstance(result, list)
```

```
$ pytest 01-pytest/test_example-07.py
```

```
============================ test session starts ============================
platform linux -- Python 3.7.2, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/jakub/SDA/software-testing-advanced-features
collected 2 items

01-pytest/test_example-07.py sx                                      [100%]

======================= 1 skipped, 1 xfailed in 0.03s =======================
```

Test fixtures

- Pytest supports setup/teardown functionality using fixtures
- Fixtures have defined scope (function, class, module, session)
- Fixtures may return/yield a value to be passed to the test function
- To create a fixture use `@pytest.mark.fixture(scope=<scope>)`
- To use a fixture in test function, use `def test_function(<fixture_name>):`
- Module or session level fixtures can be defined in `conftest.py`

Example 01-pytest/test_example-08.py

```python
import pytest


@pytest.fixture()  # scope="function" is default
def populated_dict():
    return {i + 1: chr(i + 97) for i in range(26)}


def test_replacement(populated_dict):
    populated_dict[1] = "z"
    assert populated_dict[1] == "z"


def test_alphabet(populated_dict):
    # Fixture runs for every function
    assert populated_dict[1] == "a"
```

```
$ pytest 01-pytest/test_example-08.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.7.2, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/jakub/SDA/software-testing-advanced-features
collected 2 items

01-pytest/test_example-08.py ..                                          [100%]

============================== 2 passed in 0.03s ===============================
```

Example 01-pytest/test_example-09.py

```python
import tempfile
import pytest


@pytest.fixture()  # scope="function" is default
def tmp_file():
    f = tempfile.TemporaryFile("w+t")
    yield f
    if not f.closed:
        f.close()


def test_write(tmp_file):
    text = "Hello, World"
    written_bytes = tmp_file.write(text)
    assert written_bytes == len(text)
```

```
$ pytest 01-pytest/test_example-09.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.7.2, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/jakub/SDA/software-testing-advanced-features
collected 1 item

01-pytest/test_example-09.py .                                           [100%]

============================== 1 passed in 0.08s ===============================
```

Exercise 5

1. We want to use temporary file more than once:
   - Create `conftest.py` (should work anywhere, but its best created at the project root)
   - Move the `tmp_file` fixture there, changing its scope to `session`
2. Add another test, which:
   - Grabs current position using `.tell()`
   - Writes something to file, capturing `bytes_written`
   - Grabs final position using `.tell()` and asserts the difference between the two is equal `bytes_written`

Mocking

What is mocking?

- Mocking lets you substitute executable parts of your code with Mock objects
- This helps check whether they were properly called or simulate their response, without actually having to execute them
- This is especially useful when testing slow functions, like filesystem or database calls
- Python standard library offers mocking capabilities via `unittest.mock`

Mock object

- `Unittest.mock` provides `Mock` object
- This object can substitute any other object, since it generates its attributes on the fly
- It allows you to :
  - Check whether it was called
  - Check what parameters were used to call it
  - Mock its response

Example 02-mock/test_example-01.py

```python
from unittest.mock import Mock

# Let's mock 'json' library
json = Mock()


def test_json_loads_called():
    assert isinstance(json, Mock)
    json.loads.assert_not_called()
    json_str = '{"example": 1}'
    json.loads(json_str)
    json.loads.assert_called_once_with(json_str)
```

```
$ pytest 02-mock/test_example-01.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.7.2, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/jakub/SDA/software-testing-advanced-features
collected 1 items

02-mock/test_example-01.py .                                             [100%]

============================== 1 passed in 0.10s ===============================
```

Where is mocking useful?

`Mock` is often used for parts of the code that are:

- Too difficult to call (require a specific setup)
- Calls to external processes
- Slow calls
- Operations on the system

Specifying return value

`Mock` is often used for parts of the code that are:

- Too difficult to call (require a specific setup)
- Calls to external processes
- Slow calls
- Operations on the system

Example 02-mock/test_example-02.py

```python
from db import DB
from unittest.mock import Mock


def test_db_connection_real():
    db = DB()  # db simulates slow database connection
    assert len(db.list_tables()) == 3
    assert db.list_tables() == ("users", "products", "orders")


def test_db_connection_mock():
    db = Mock()
    db.list_tables.return_value = ("users", "products", "orders")
    assert len(db.list_tables()) == 3
    assert db.list_tables() == ("users", "products", "orders")
```

```
$ pytest -k connection_real 02-mock/test_example-02.py
```

```
(…)
collected 2 items / 1 deselected / 1 selected

02-mock/test_example-02.py .                                          [100%]
====================== 1 passed, 1 deselected in 4.10s ===================
```

```
$ pytest -k connection_mock 02-mock/test_example-02.py
```

```
(…)
collected 2 items / 1 deselected / 1 selected

02-mock/test_example-02.py .                                          [100%]
====================== 1 passed, 1 deselected in 0.02 s ==================
```

Exercise 1

1. Write a fixture returning a mocked database object:
   1. `count()` with any parameters should return 1
   2. `list_tables()` should return `(„users”)`
   3. `get()` should return a dictionary with `username`, `first_name`, `last_name` and `email` keys
2. Use the fixture and test each of those functions; Don't forget to check parameters passed to them.

Side effects

- Sometimes calling a function ends with an `Exception` being raised
- We can simulate that, using `side_effect`
- `side_effect` accepts functions, iterables or exceptions, but we will Focus solely on the Exception part
- The Exception will be raised by the mock when calling a function with a defined `side_effect`
- This behavior is often used when we want to test if our code handles exceptions as designed

Example 02-mock/test_example-03.py

```python
import pytest
from unittest.mock import Mock


def test_exceptions():
    m = Mock()
    m.get.return_value = "ok"
    m.get.side_effect = IndexError

    m.get.assert_not_called()
    with pytest.raises(IndexError):
        m.get(3)
    m.get.assert_called_once_with(3)
```

```
$ pytest 02-mock/test_example-03.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.7.2, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/jakub/SDA/software-testing-advanced-features
collected 1 items

02-mock/test_example-03.py .                                             [100%]

============================== 1 passed in 0.10s ===============================
```

Exercise 2

When `DB.get()` is called:

- With non-existent `table_name` it raises a `KeyError`
- With non-existent `index` it raises an `IndexError`

Use `Mock()` with `side_effect` to make the tests in `02-mock/test_exercise-02.py` run faster.

Patch

- To mock an object without accessing the internals, like we did with the DB class, we can use the `patch()` function
- `patch()` can be used either as a decorator, or as a context manager to mock any object
- The import path to the mocked object is passed as an argument to the `patch()` function
- Syntax: `unittest.mock.patch(<import_path_to_object>)`

Example 02-mock/test_example-04.py

```python
import pytest
import datetime
from unittest.mock import patch


@pytest.fixture
def override_time():
    """Sets current date to 1999-12-31T23:59.00"""
    date = datetime.datetime(1999, 12, 31, 23, 59, 0)
    with patch("datetime.datetime") as patched_datetime:
        patched_datetime.now.return_value = date
        yield


def test_current_date(override_time):
    now = datetime.datetime.now()
    assert now.year == 1999
    assert now.month == 12
    assert now.day == 31


def test_current_date_no_override():
    assert datetime.datetime.now().year > 1999
```

```
$ pytest 02-mock/test_example-04.py
```

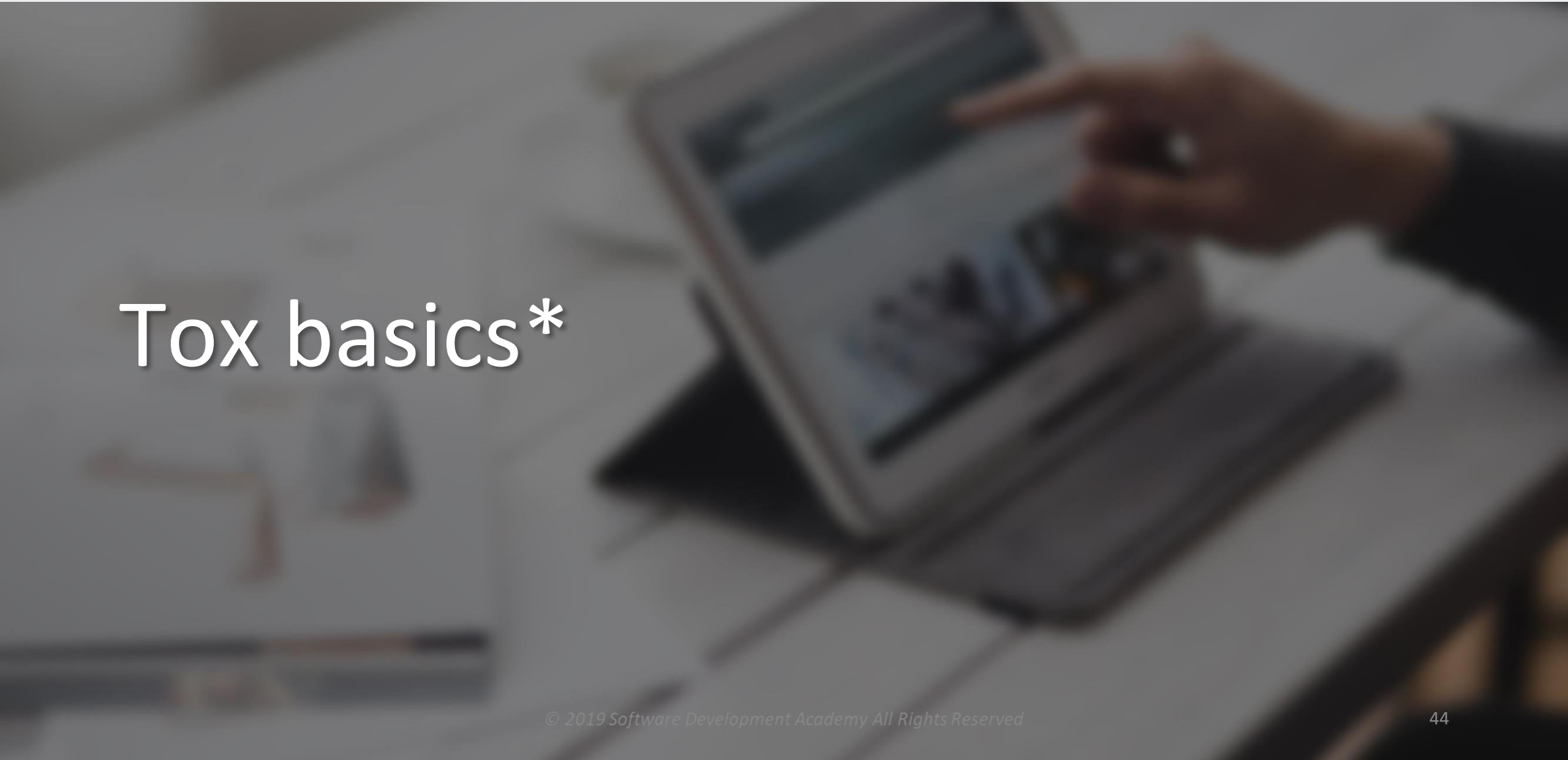
```
(…)
02-mock/test_example-04.py ..                                        [100%]

============================== 2 passed in 0.04s ===============================
```

Exercise 3

You are given two libraries (plant_shop and db) and a file which tests them (test_exercise-03.py).

1. Run the tests. The DB is not mocked here, so they take a while to complete
2. Mock the DB object and the functions used in existing tests required to make the them run faster

   Hint: It is easier to create separate fixtures than to mock everything in one of them
3. Cover the code with test – test every functon at least once

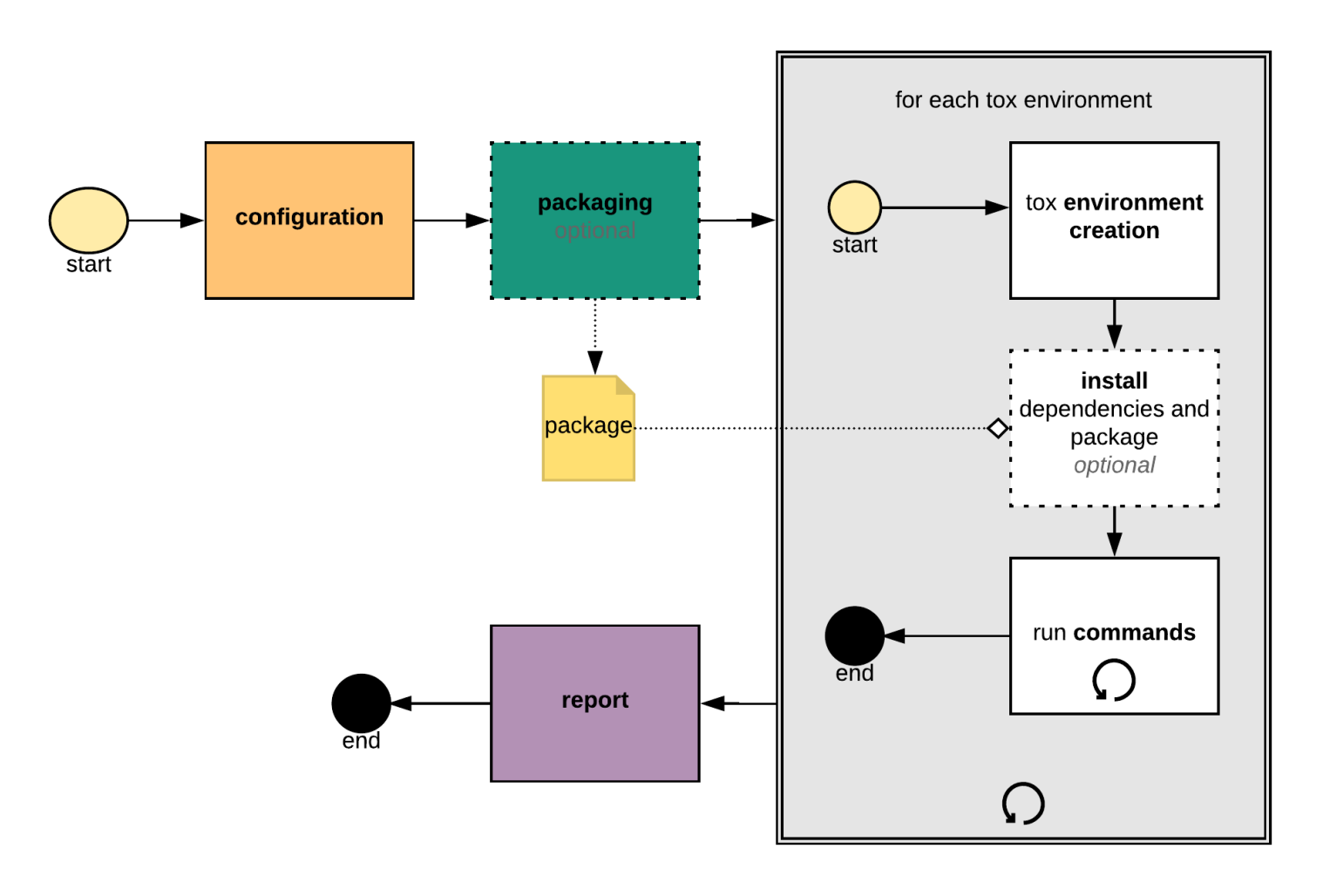


Tox basics*

What is tox?

- Tox is a generic virtualenv management package
- It lets you build and test your code on many versions of Python simultaneously
- To harness its capabilities, your system should have more than one version of Python installed
- If you want to install another version of Python, head to https://www.python.org/downloads/release
- `pip install tox`

Tox workflow



Source: https://tox.readthedocs.io/en/latest/

Tox configuration

- Just after starting the tox workflow, it reads the configuration
- The configuration is stored in `tox.ini` file

Example 03-tox - definitions

tox.ini

```
[tox]
# I'm using py27 and py37 since I have Python 2.7 and 3.7 installed.
# If you have picked another version, tox supports the following by default:
# py, py2, py27, py3, py34, py35, py36, py37, py38, jython, pypy, pypy2,
# pypy27, pypy3, pypy35
envlist=py27,py37
# Allows us to skip installation phase. We want this, since we don't want to
# define setup.py
skipsdist=True

[testenv]
# Install testing framework, or anything you might need here.
deps = pytest
# and run the tests!
commands = pytest -v
```

test_versions.py

```
# This file will be tested on different python versions using tox

def test_dict_keys_ordered():
    """This test will always pass for Python > 3.7. It passes for current distribution of
    3.6 also. Python < 3.6 will fail here, except for lucky situations where the random
    order will match our order"""
    key_order = ("a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k")
    d = {k: k for k in key_order}
    assert tuple(d.keys()) == key_order
```

```
$ cd 03-tox; tox
```

Example 03-tox -output

```
(…)
py27 run-test-pre: PYTHONHASHSEED='2543335519'
py27 run-test: commands[0] | pytest -v
============================= test session starts ==============================
(…)
test_versions.py::test_dict_keys_ordered FAILED                          [100%]
================================== FAILURES ===================================
_____________________________ test_dict_keys_ordered ______________________________
    def test_dict_keys_ordered():
        """This test will always pass for Python > 3.7. It passes for current distribution of
        3.6 also. Python < 3.6 will fail here, except for lucky situations where the random
        order will match our order"""
        key_order = ("a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k")
        d = {k: k for k in key_order}
>       assert tuple(d.keys()) == key_order
E       AssertionError: assert ('h', 'i', 'j...'d', 'e', ...) == ('a', 'b', 'c'...'e', 'f', ...)
E         At index 0 diff: 'h' != 'a'
E         Full diff:
E         - ('h', 'i', 'j', 'k', 'd', 'e', 'f', 'g', 'a', 'b', 'c')
E         + ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k')

test_versions.py:9: AssertionError
========================== 1 failed in 0.03 seconds ===========================
(…)
py37 run-test-pre: PYTHONHASHSEED='2543335519'
py37 run-test: commands[0] | pytest -v
============================= test session starts ==============================
(…)
test_versions.py::test_dict_keys_ordered PASSED                          [100%]
============================== 1 passed in 0.01s ===============================
_____________________________________ summary ______________________________________
ERROR:   py27: commands failed
  py37: commands succeeded
```

Exercise 1*

1. Add some test cases of your own and run tox to test them

Locust basics*

What is Locust?

- Locust is a scalable load testing framework
- Load testing means testing throughput and response time of web applications
- Locust tests that by creating processes which call webservices as fast as possible and measure response time
- `pip install -U locust`

Locust configuration

- Locust uses a locustfile (customarily called `locustfile.py`) for configuration
- Locustfile tells Locust which endpoints to call and how often to do so
- Locustfile can grow complex to test more complex scenarios, like:
  - Login
  - Add item to cart
  - Checkout
  - Log out

Pre-example configuration

- This next example requires you to also have Flask installed
- `pip install -U flask`
- You are provided with two files: start server first, locust second
- Enter `04-locust` directory

Example 04-locust - definitions

locustfile.py

```
--------------
from locust import HttpLocust, TaskSet, task


class TestEndpoints(TaskSet):
    @task
    def test_homepage(self):
        self.client.get("/")

    @task
    def test_about(self):
        self.client.get("/about")

    @task
    def test_form(self):
        self.client.post("/form", data={"user": "random"})


class MyLocust(HttpLocust):
    task_set = TestEndpoints
```

app.py

```
--------------
import time
from flask import Flask, request

app = Flask(__name__)


@app.route("/")
def homepage():
    return "Hello"


@app.route("/about")
def about():
    time.sleep(0.01)
    return "About"


@app.route("/form", methods=["POST"])
def form():
    print(request.data)
    return "Thanks"


@app.route("/admin")
def admin():
    return "Forbidden", 403
```

Example 04-locust – running the example

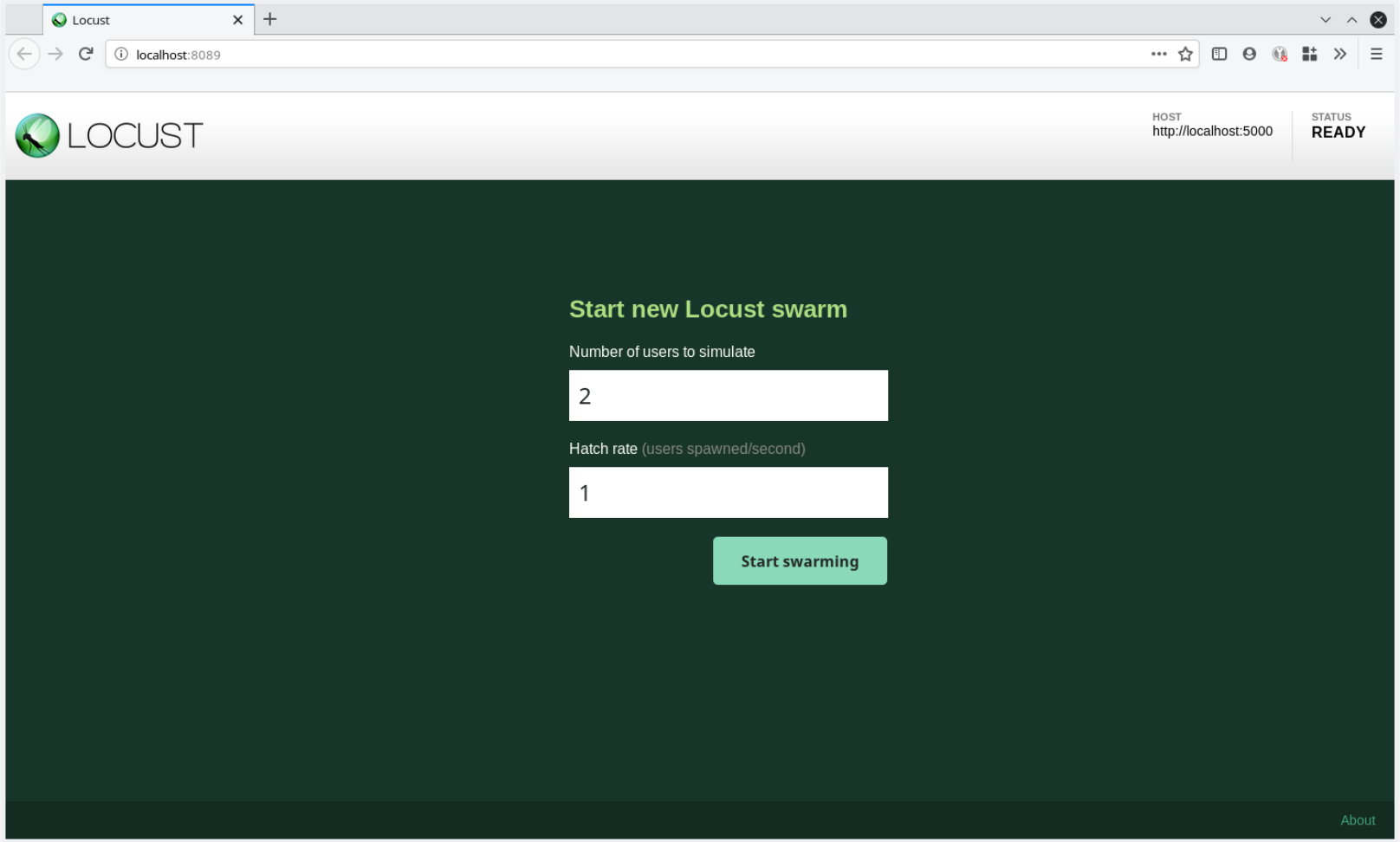
```
$ # To run flask webserver
$ cd 04-locust
$ flask run
```

```
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

(The server is now running. LEAVE THIS TERMINAL OPEN. Ctrl+C to stop the server)
```

```
$ # In a new terminal window
$ cd 04-locust
$ locust --host=„http://127.0.0.1:5000”
```

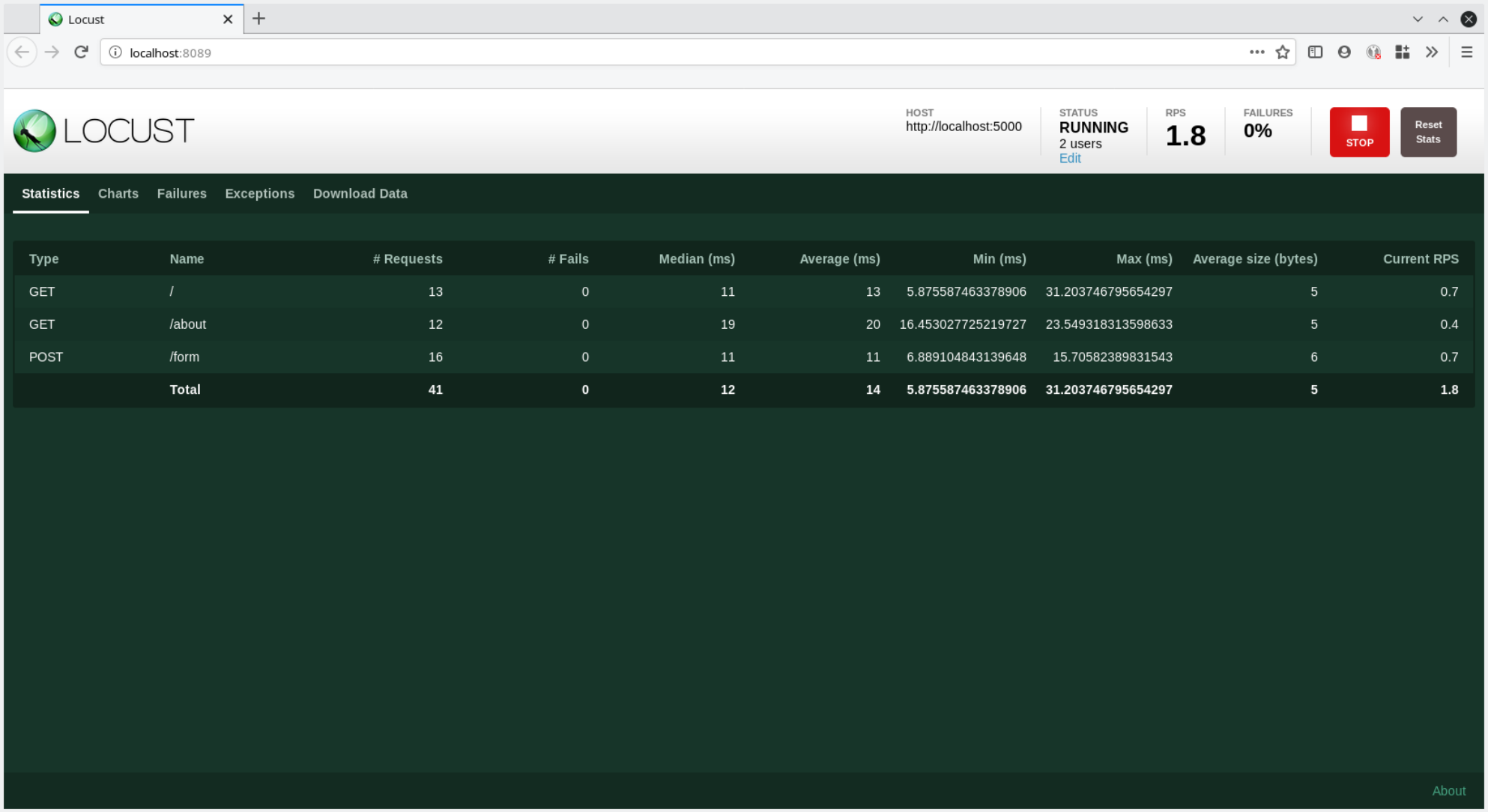
```
[2019-10-28 19:38:48,680] zenbook/INFO/locust.main: Starting web monitor at *:8089
[2019-10-28 19:38:48,681] zenbook/INFO/locust.main: Starting Locust 0.11.0

(The server is now running. LEAVE THIS TERMINAL OPEN. Ctrl+C to stop the server)
```

Example 04-locust – browser setup

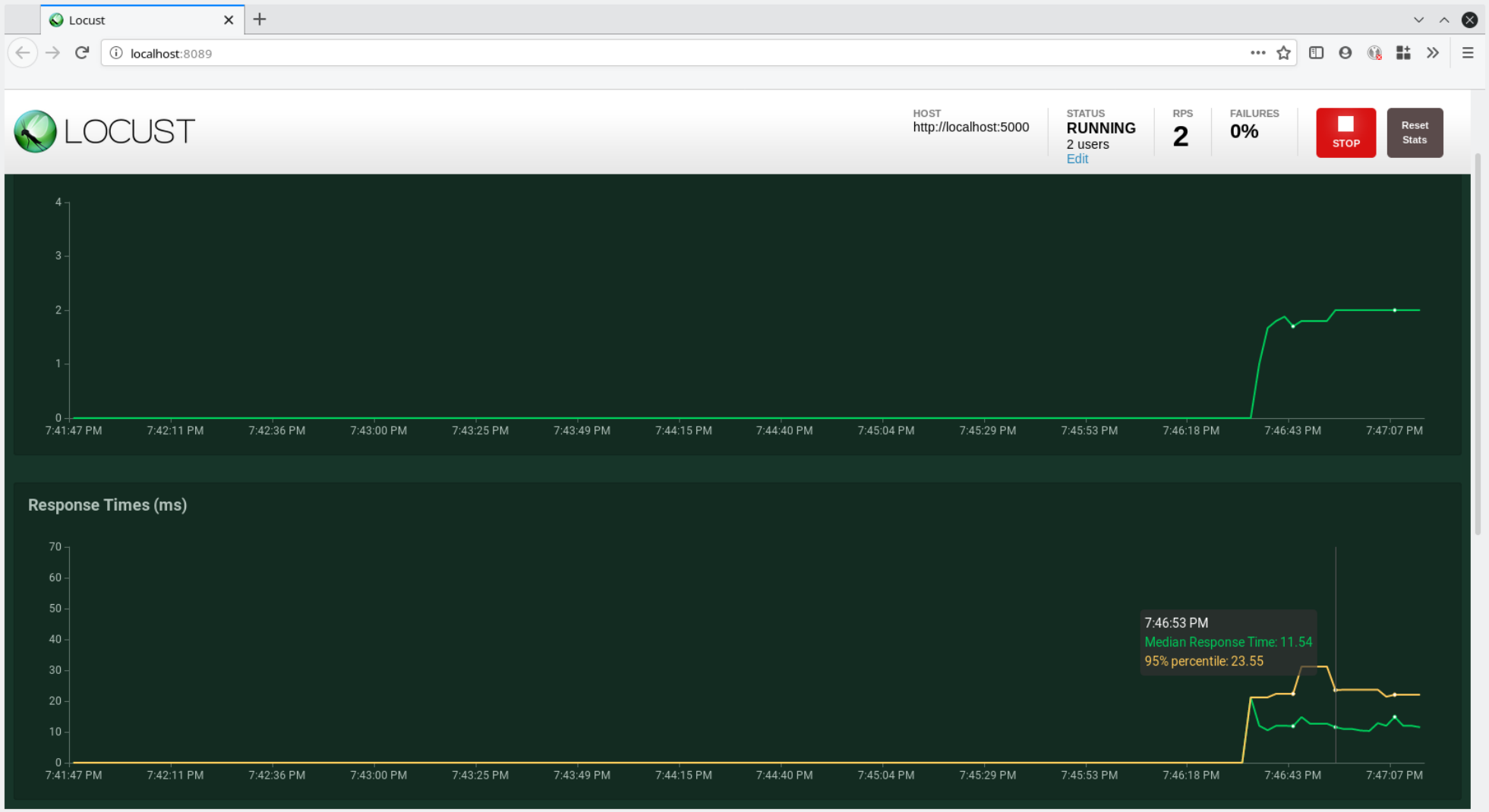


Open http://localhost:8089 in your browser, input a numer of users and hatch rate, hit start!

Example 04-locust – watch results



Example 04-locust – watch results



Example 04-locust – watch results

When you are done, hit stop button. You can export the results on the last tab.

Exercise 1*

You may have noticed that the admin endpoint in `app.py` does not have a corresponding taks in `locustfile.py`.

1. Add the task
2. Run locust again with the new configuration