kiwipycon

Waihōpai Invercargill

Friday 15th to Sunday 17th of September 2023

https://kiwipycon.nz/

pytest is awesome

Menno-Finlay-Smits

Email: hello@menno.io
GitHub: mjs
Mastodon: @menn0@mastodon.nzoss.nz

I like interruptions

What’s pytest?

- A testing framework for writing and running tests
- Easy to use
- Low on ceremony
- Lots of helpful magic
- It’s fun!
- Arguably the defacto testing framework for Python
- Formerly “py.test”

Minimal Example

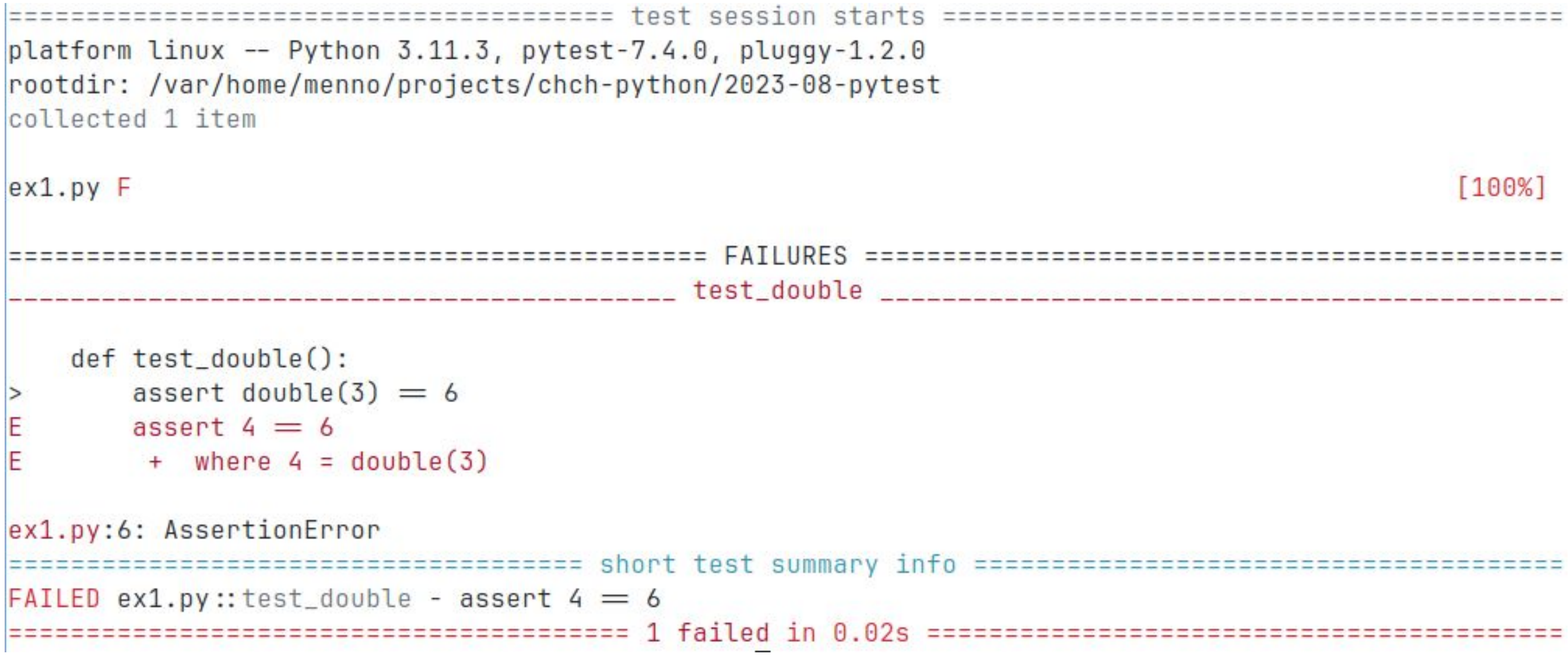
Implementation:

```
def double(x):
    return x + 1
```

Test (usually in a separate file):

```
from some.module import double

def test_double():
    assert double(3) == 6
```

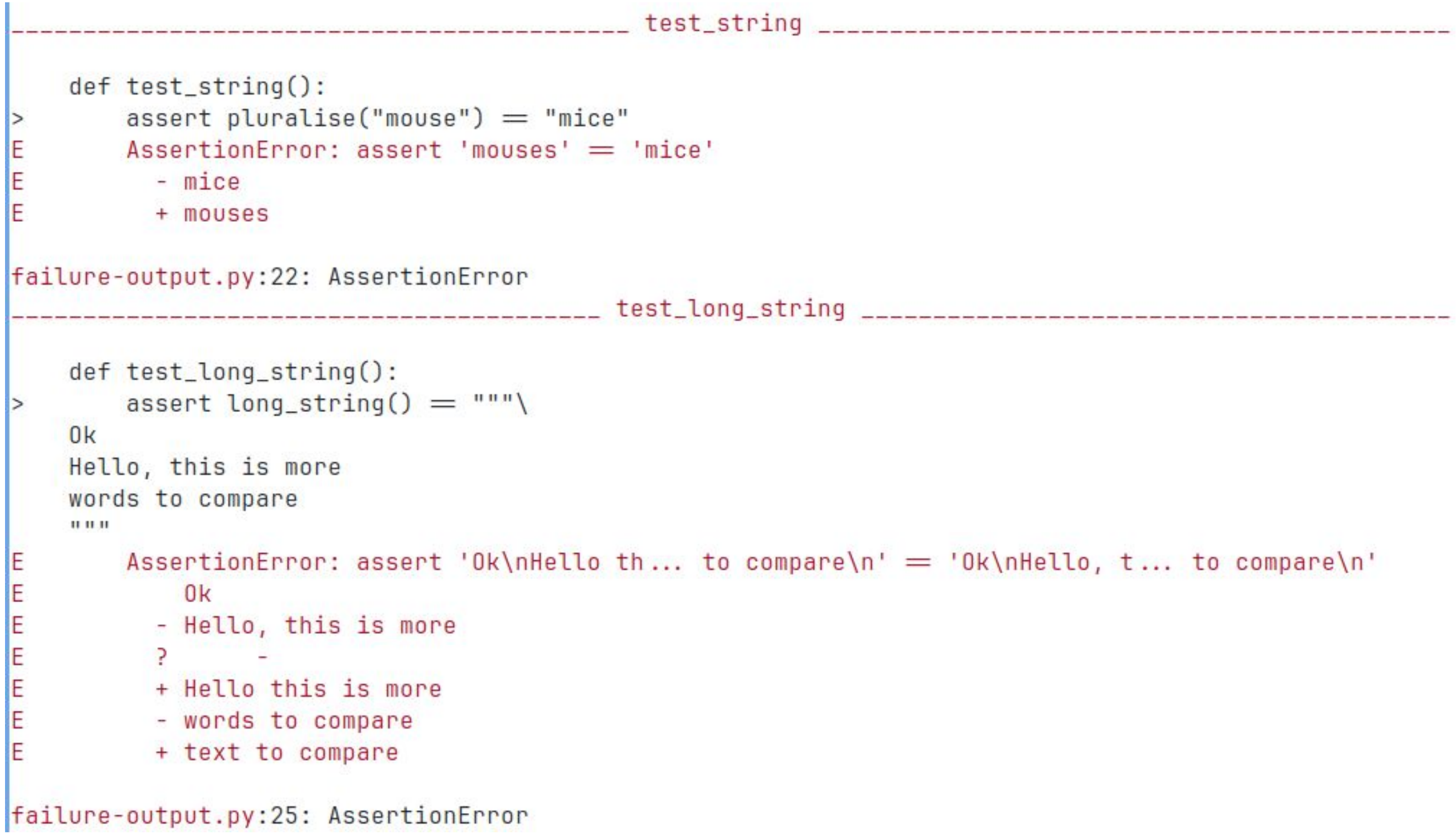
Example Output



```
======================================= test session starts ========================================
platform linux -- Python 3.11.3, pytest-7.4.0, pluggy-1.2.0
rootdir: /var/home/menno/projects/chch-python/2023-08-pytest
collected 1 item

ex1.py F                                                                                      [100%]

============================================= FAILURES =============================================
___________________________________________ test_double ____________________________________________

    def test_double():
>       assert double(3) == 6
E       assert 4 == 6
E        +  where 4 = double(3)

ex1.py:6: AssertionError
===================================== short test summary info ======================================
FAILED ex1.py::test_double - assert 4 == 6
======================================== 1 failed in 0.02s =========================================
```

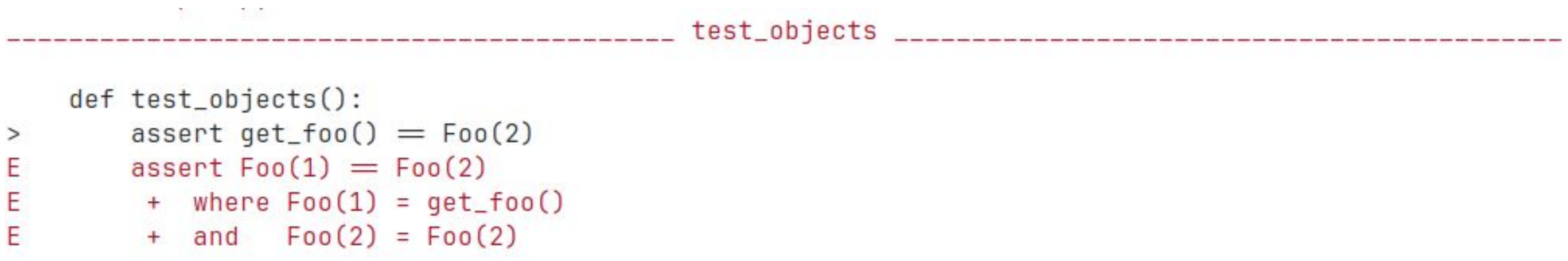
Rich Failure Output

Just use assert



```
__________________________________________ test_string ___________________________________________

    def test_string():
>       assert pluralise("mouse") == "mice"
E       AssertionError: assert 'mouses' == 'mice'
E         - mice
E         + mouses

failure-output.py:22: AssertionError
_________________________________________ test_long_string _________________________________________

    def test_long_string():
>       assert long_string() == """\
    Ok
    Hello, this is more
    words to compare
    """
E       AssertionError: assert 'Ok\nHello th... to compare\n' == 'Ok\nHello, t... to compare\n'
E           Ok
E         - Hello, this is more
E         ?      -
E         + Hello this is more
E         - words to compare
E         + text to compare

failure-output.py:25: AssertionError
```

```
____________________________________ test_dict _____________________________________

    def test_dict():
>       assert some_dict() == {"1": "one", 2: "two"}
E       AssertionError: assert {1: 'one', 2:...', 3: 'three'} == {'1': 'one', 2: 'two'}
E         Omitting 1 identical items, use -vv to show
E         Left contains 2 more items:
E         {1: 'one', 3: 'three'}
E         Right contains 1 more item:
E         {'1': 'one'}
E         Use -v to get more diff

failure-output.py:32: AssertionError
____________________________________ test_list _____________________________________

    def test_list():
>       assert some_list() == [1, 3, 4]
E       assert [1, 2, 3] == [1, 3, 4]
E         At index 1 diff: 2 != 3
E         Use -v to get more diff
```



```
___________________________________ test_objects ___________________________________

    def test_objects():
>       assert get_foo() == Foo(2)
E       assert Foo(1) == Foo(2)
E        +  where Foo(1) = get_foo()
E        +  and   Foo(2) = Foo(2)
```

- Takes advantage of __repr__ and __str__ methods if implemented
- Otherwise the output isn’t as helpful

Running Tests

Test Discovery

- Just running `pytest` will cause pytest to go looking for tests
- Running `pytest some/dir` will start from that location
- Recursive search through directories
- `test_*.py` and `*_test.py` files
  - e.g. `test_thing.py`

Inside a test file…

- Runs functions named `test_*`
  - `e.g. def test_foo()`
- Also `test_*` methods on classes named `Test*`
  - `e.g. TestFoo.test_hello()`
- Will also run doctests, unittest and nosetests style tests

Selecting Tests to Run

- `-k <regex>` - Run only tests with a name matching a regex
- `-m <mark>` - Run only tests matching a mark (decorator)
- `-x` - Stop after first test failure (fail fast)
- `--lf` - Run only the tests that failed during the last test run
- `--ff` - Run all tests but run the ones that failed previously first
- `--nf` - Run all tests, but run tests in the new files first

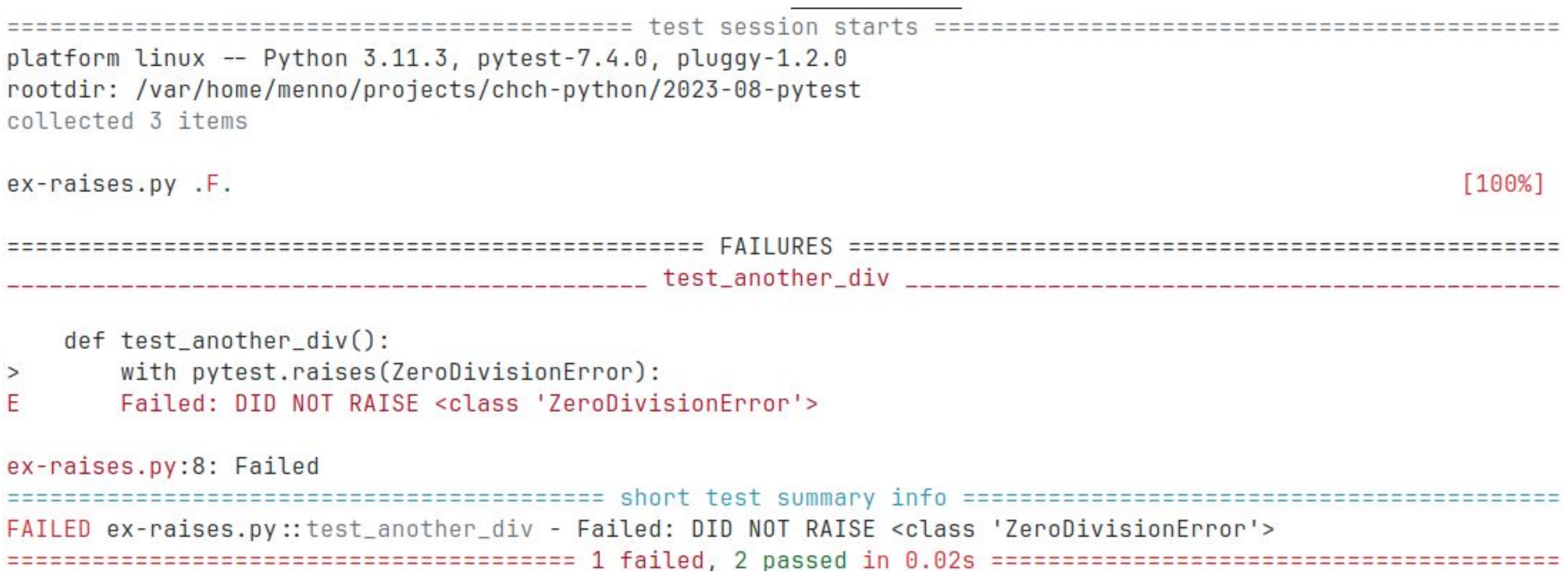
Many of these can be used together

Testing Exceptions

pytest.raises

```
import pytest

def test_zero_div():
    with pytest.raises(ZeroDivisionError):
        1 / 0

def test_another_div():
    with pytest.raises(ZeroDivisionError):
        4 / 2

def test_err_message():
    with pytest.raises(ValueError, match= "foo.+"):
        raise ValueError("foo bar")
```

Exception Testing Output



```
=========================================== test session starts ===========================================
platform linux -- Python 3.11.3, pytest-7.4.0, pluggy-1.2.0
rootdir: /var/home/menno/projects/chch-python/2023-08-pytest
collected 3 items

ex-raises.py .F.                                                                                    [100%]

================================================ FAILURES =================================================
_____________________________________________ test_another_div ____________________________________________

    def test_another_div():
>       with pytest.raises(ZeroDivisionError):
E       Failed: DID NOT RAISE <class 'ZeroDivisionError'>

ex-raises.py:8: Failed
========================================= short test summary info =========================================
FAILED ex-raises.py::test_another_div - Failed: DID NOT RAISE <class 'ZeroDivisionError'>
======================================== 1 failed, 2 passed in 0.02s ========================================
```

Test Setup and Teardown

“Test Fixtures”

Flexible, modular test setup and teardown

Pros

- Only the exact setup need for test is used
- Test setup dependencies are explicit
- Concise
- Controlled scope

Cons

- A little too “magic”?

Text Fixtures Example

```
import pytest

@pytest.fixture
def a_list():
   return [1, 2]

def test_can_append_3(a_list):
   assert len(a_list) == 2
   a_list.append(3)
   assert 3 in a_list

def test_can_append_99(a_list):
   assert len(a_list) == 2
   a_list.append(99)
   assert 99 in a_list
```

More Fixtures

```python
import os
import sqlite3
import pytest

@pytest.fixture
def db():
    DB_NAME = "__test.db"
    con = sqlite3.connect( DB_NAME)
    con.execute("CREATE TABLE person(name, age)" )
    con.execute("INSERT INTO person VALUES ('Sam', 25)" )
    yield con
    con.close()
    os.unlink(DB_NAME)
```

Continued…

```
def test_can_insert(db):
    db.execute("INSERT INTO person VALUES ('Sofia', 27)")
    assert len(db.execute("SELECT * FROM person").fetchall()) == 2


def test_fixture_resets(db):
    assert len(db.execute("SELECT * FROM person").fetchall()) == 1


def test_something_else():
    ...
```

Comparison to unittest

```
import os, sqlite3, unittest

class TestDB(unittest.TestCase):
    def setUp(self):
        self.con = sqlite3.connect(DB_NAME)
        self.con.execute("CREATE TABLE person(name, age)" )
        self.con.execute("INSERT INTO person VALUES ('Sam', 25)" )

    def tearDown(self):
        con.close()
        os.unlink(DB_NAME)

    def test_can_insert(db):
        db.execute("INSERT INTO person VALUES ('Sofia', 27)" )
        assert len(db.execute("SELECT * FROM person").fetchall()) == 2
```

More on Fixtures

- Tests can request more than one fixture
- Fixtures can request other fixtures!
- Fixtures can be scoped
- Fixtures can be shared
- Fixtures can be automatically applied (autouse)

Useful Built-In Fixtures

- **tmp_path** - Creates a temporary directory that will be automatically cleaned up
- **caplog** - Capture logs emitted by the logging package
- **capfd** - Capture stdout and stderr
- **monkeypatch** - See the next section

(Monkey)patching

Monkeypatching

- Temporarily changing dependencies of code being tested
- Replace with a fake/mock/stub object
- Controls test environment
- Helps avoid calls to external dependencies
- Don’t overdo it!

Etymology: *guerrilla patch -> gorilla patch -> monkey patch*

Patching Example - Implementation

```python
import requests

URL = "https://some.api/users"

def call_api():
    r = requests.get(URL)
    return r.json()

def get_user_ids():
    return [u.id for u in call_api()["users"]]
```

Patching Example - Test

```
import requests
import app

class MockResponse:
   @staticmethod
   def json():
       return {"users": [{"id": 1}, {"id": 2}]}

def test_get_user_ids(monkeypatch):
   def mock_get(url):
       assert url == "https://some.api/users"
       return MockResponse()
   monkeypatch.setattr(requests,"get", mock_get)

   assert app.get_user_ids() == [1, 2]
```

More on monkeypatching

Can modify:

- attributes of modules and objects
- dict items (including deletion)
- environment variables (including deletion)
- working directory

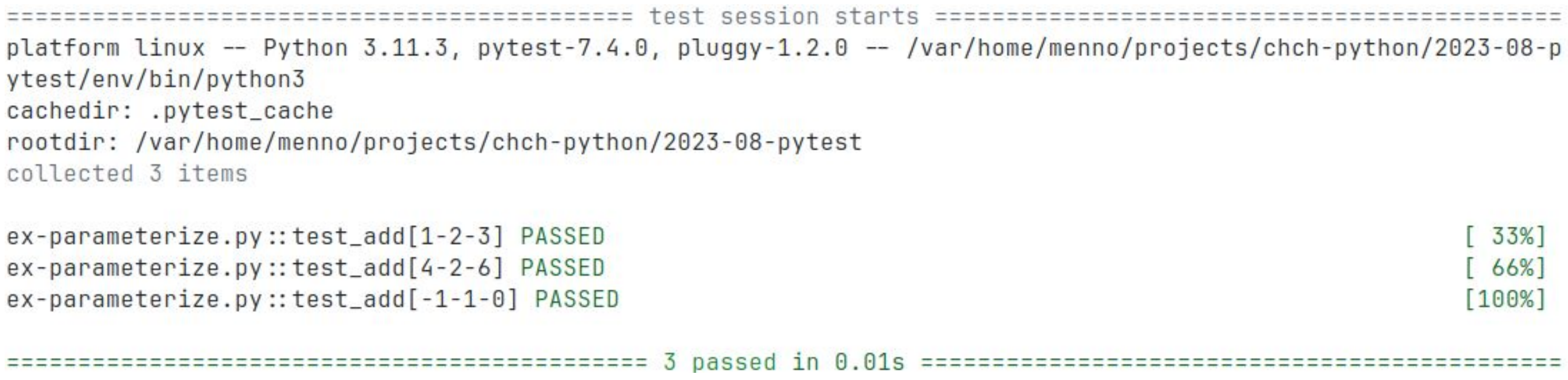
Parameterising Tests

Running Tests With Multiple Sets of Inputs

```
import pytest

def add(a, b):
    return a + b

@pytest.mark.parametrize("a,b,want", [(1, 2, 3), (4, 2, 6), (-1, 1, 0)])
def test_add(a, b, want):
    assert add(a, b) == want
```

Multiple Tests are Generated

with verbose output (`-v` flag)



```
======================================== test session starts ========================================
platform linux -- Python 3.11.3, pytest-7.4.0, pluggy-1.2.0 -- /var/home/menno/projects/chch-python/2023-08-p
ytest/env/bin/python3
cachedir: .pytest_cache
rootdir: /var/home/menno/projects/chch-python/2023-08-pytest
collected 3 items

ex-parameterize.py::test_add[1-2-3] PASSED                                                    [ 33%]
ex-parameterize.py::test_add[4-2-6] PASSED                                                    [ 66%]
ex-parameterize.py::test_add[-1-1-0] PASSED                                                   [100%]

============================================ 3 passed in 0.01s ============================================
```

Pros and Cons

Pros

- Easy to add new cases
- Easier to identify the failing cases
- Separate tests means tests continue after a failure

Cons

- Hard to read when many parameters
- Good taste is required when there’s many test cases

Cleaner Way of Defining Cases

```
add_cases = [
    (1, 2, 3),
    (4, 2, 6),
    (-1, 1, 0),
]

@pytest.mark.parametrize("a,b,want", add_cases)
def test_add(a, b, want):
    assert add(a, b) == want
```

Parameterizing All Tests on a Class

```
@pytest.mark.parametrize("n,expected", [(1, 2), (3, 4)])
class TestClass:
    def test_simple_case(self, n, expected):
        assert n + 1 == expected

    def test_weird_simple_case(self, n, expected):
        assert (n * 1) + 1 == expected
```

Moar!

More pytest functionality which for real world projects:

- Conditionally or unconditionally skipping tests
    - These are tracked and highlighted separately from passing and failing tests
- Handling of tests which are known to fail (xfail)
    - Perhaps only under certain conditions
    - Will fail if the test doesn’t fail in the expected way

Questions?