pytest tips and tricks

for a better testsuite

Florian Bruhin

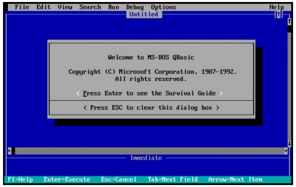
PyConDE & PyData Berlin

April 22nd, 2024

About me

Florian Bruhin <florian@bruhin.software>, The-Compiler, https://bruhin.software

2006	Started programming (QBasic, bash)
2009 – 2013	Apprenticeship in electrical engineering
2011	Started using Python
2013	Started developing qutebrowser, writing tests
2015	Switched to pytest, ended up as a maintainer, started giving courses in companies / at conferences
2016 – 2019	BSc. in Computer Science (OST Rapperswil) Eastern Switzerland University of Applied Sciences
2020	Small one-man company: Bruhin Software


File Edit View Search Run Debug Options Help
Untitled
Immediate
F1=Help Enter=Execute Esc=Cancel Tab=Next Field Arrow=Next Item

Welcome to MS-DOS QBasic

Copyright (C) Microsoft Corporation, 1987-1992.
All rights reserved.

< Press Enter to see the Survival Guide >

< Press ESC to clear this dialog box >

BRUNNER
dedication + innovation

2–3 days/week in autumn (OST): Teaching Python to first-semester BSc. students
rest: open-source and training/consulting (Bruhin Software): Python, pytest, Qt

About me

Florian Bruhin <florian@bruhin.software>, The-Compiler, https://bruhin.software

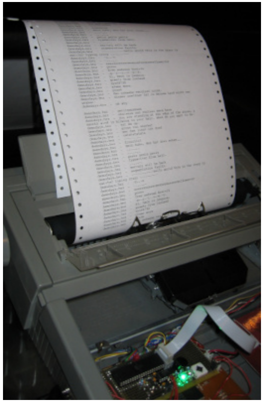
2006	Started programming (QBasic, bash)
2009 – 2013	Apprenticeship in electrical engineering
2011	Started using Python
2013	Started developing qutebrowser, writing tests
2015	Switched to pytest, ended up as a maintainer, started giving courses in companies / at conferences
2016 – 2019	BSc. in Computer Science (OST Rapperswil) Eastern Switzerland University of Applied Sciences
2020	Small one-man company: Bruhin Software

2–3 days/week in autumn (OST): Teaching Python to first-semester BSc. students
rest: open-source and training/consulting (Bruhin Software): Python, pytest, Qt

About me

Florian Bruhin <florian@bruhin.software>, The-Compiler, https://bruhin.software

2006	Started programming (QBasic, bash)
2009 – 2013	Apprenticeship in electrical engineering
2011	Started using Python
2013	Started developing qutebrowser, writing tests
2015	Switched to pytest, ended up as a maintainer, started giving courses in companies / at conferences
2016 – 2019	BSc. in Computer Science (OST Rapperswil) Eastern Switzerland University of Applied Sciences
2020	Small one-man company: Bruhin Software



2–3 days/week in autumn (OST): Teaching Python to first-semester BSc. students
rest: open-source and training/consulting (Bruhin Software): Python, pytest, Qt

About me

Florian Bruhin <florian@bruhin.software>, The-Compiler, https://bruhin.software

2006	Started programming (QBasic, bash)
2009 – 2013	Apprenticeship in electrical engineering
2011	Started using Python
2013	Started developing qutebrowser, writing tests
2015	Switched to pytest, ended up as a maintainer, started giving courses in companies / at conferences
2016 – 2019	BSc. in Computer Science (OST Rapperswil) Eastern Switzerland University of Applied Sciences
2020	Small one-man company: Bruhin Software

2–3 days/week in autumn (OST): Teaching Python to first-semester BSc. students
rest: open-source and training/consulting (Bruhin Software): Python, pytest, Qt

About me

Florian Bruhin <florian@bruhin.software>, The-Compiler, https://bruhin.software

2006	Started programming (QBasic, bash)
2009 – 2013	Apprenticeship in electrical engineering
2011	Started using Python
2013	Started developing qutebrowser, writing tests
2015	Switched to pytest, ended up as a maintainer, started giving courses in companies / at conferences
2016 – 2019	BSc. in Computer Science (OST Rapperswil) Eastern Switzerland University of Applied Sciences
2020	Small one-man company: Bruhin Software

2–3 days/week in autumn (OST): Teaching Python to first-semester BSc. students
rest: open-source and training/consulting (Bruhin Software): Python, pytest, Qt

Course content

- **About pytest**: Popularity, history overview
- **Parametrization**: Running tests against sets of input/output data
- **Fixtures**: Providing test data, setting up objects, modularity
- **Built-in fixtures**: Temporary files, patching, capturing output, test information
- **Fixtures advanced**: Caching, cleanup/teardown, implicit fixtures, parametrizing
- **Mocking**: Dealing with dependencies which are in our way, monkeypatch and unittest.mock, mocking libraries and helpers, alternatives
- **Plugin tour**: Coverage, distributed testing, output improvements, alternative test syntax, testing C libraries, plugin overview
- **Property-based testing**: Using *hypothesis* to generate test data
- **Writing plugins**: Extending pytest via custom hooks, domain-specific languages

pytest

Setup

You...

- **Set up** pytest?
- Used **virtualenv**?
- Cloned the repo with example code?
- Know what an RPN calculator is?

Setup overview

- We'll use Python 3.8 or newer, with pytest 8.1 ($\geq$ 7.0 is okay).
  - Use `python3 --version` or `py -3 --version` (Windows) to check your version.
- You can use whatever editor/IDE you'd like – if you don't use one yet, PyCharm (Community Edition) or VS Code are good choices.
- However, we'll first start exploring pytest on the command line, in order to see how it works "under the hood" and explore various commandline arguments.
- Download example code for exercises: `https://github.com/The-Compiler/pytest-tips-and-tricks`

Setup with PyCharm

- Open `code/` folder as PyCharm project
- Install requirements when prompted
- Open PyCharm terminal at the bottom
- You should be able to run `pytest --version`

Virtual environments: Isolation of package installs

Virtual environments:

- Provide isolated environments for Python package installs
- Isolate different app/package-install configurations
- Are built into Python since 3.4 (but a separate `virtualenv` tool also exists)

With a virtual environment, we can avoid running `sudo pip install ...`, which can mess up your system (on Linux/macOS).

Chris Warrick (`chriswarrick.com`):
"Python Virtual Environments in Five Minutes"

Using virtual environments

Installing and creating

Install venv:

(Debian-based Linux distributions only, shipped with Python elsewhere)

```
apt install python3-venv
```

Create a local environment (once, can be reused):

```
py -m venv venv
```

```
python3 -m venv .venv
```

(or `virtualenv` instead of `venv`)

This will create a local Python installation in a `venv` or `.venv` folder.
Any dependencies installed with its `pip` will only be available in this environment/folder.

Using virtual environments

Running commands and activating

Run commands "inside" the environment:

```
venv\Scripts\pip
venv\Scripts\python
venv\Scripts\pytest
```

```
.venv/bin/pip
.venv/bin/python
.venv/bin/pytest
```

Alternatively, **activate** the environment:

(changes `PATH` temporarily, so that `pip`, `python`, `pytest` etc. use the binaries from the virtualenv)

```
venv\Scripts\activate.bat
```

```
Set-ExecutionPolicy Unrestricted -Scope Process
venv\Scripts\Activate
```

```
source .venv/bin/activate
```

Installing pytest

Install pytest and other dependencies within the activated environment:

```
pip install -r code/requirements.txt
```

(or just `pip install pytest`, which covers most but not all of the training)

Now let's see if it works:

```
pytest --version
```

The basics

You…

- Used **pytest.approx**?
- Know what 0.1 + 0.1 + 0.1 equals (according to ~~Python~~ computers)?
- Used **pytest.raises**?
- …with `match=`?
- Think pytest is around five years old? Ten? Fifteen? Twenty?

Some quick history

late 2002	PyPy (alternative Python implementation) was born
mid 2004	New `utest` test framework in PyPy, plain assertions
June 2004	`std` library (“complementary stdlib”): `std.utest`
Sep./Oct. 2004	`std` renamed to `py`, test framework is now `py.test`

Some quick history

late 2002	PyPy (alternative Python implementation) was born
mid 2004	New `utest` test framework in PyPy, plain assertions
June 2004	`std` library ("complementary stdlib"): `std.utest`
Sep./Oct. 2004	`std` renamed to `py`, test framework is now `py.test`
August 2009	py 1.0.0: plugins, fixtures (funcargs), etc.
November 2010	pytest 2.0.0, released independently from `py`.

Some quick history

late 2002	PyPy (alternative Python implementation) was born
mid 2004	New `utest` test framework in PyPy, plain assertions
June 2004	`std` library (“complementary stdlib”): `std.utest`
Sep./Oct. 2004	`std` renamed to `py`, test framework is now `py.test`
August 2009	py 1.0.0: plugins, fixtures (funcargs), etc.
November 2010	pytest 2.0.0, released independently from `py`.
August 2016	pytest 3.0.0, new `pytest` entry point (instead of `py.test`)
October 2022	pytest 7.2.0, dropping `py` dependency, vendoring only remaining part (`py.path`)

Asserting expected exceptions

— rpncalc/utils.py —

```python
def calc(a, b, op):
    ...
    elif op == "/":
        return a / b
    raise ValueError("Invalid operator")
```

— basic/test_raises.py —

```python
def test_zero_division():
    with pytest.raises(ZeroDivisionError):
        calc(3, 0, "/")
```

Demo:

- In `basic/test_raises.py`, write another test with `pytest.raises`, to ensure that `ValueError` is raised when calling `calc` with an invalid operator
- Adjust test so that no exception or a different exception is raised. Rerun.
- Pass a regex pattern to the `match` argument to check the exception message:
  `with pytest.raises(ValueError, match=r"..."):`

Exception infos and checking for warnings

You can also access the exception value manually and assert on it:

```python
def test_invalid_operator:
    with pytest.raises(ValueError) as excinfo :
        calc(1, 2, "@")
    assert str(excinfo.value) == "Invalid operator"
```

Exception infos and checking for warnings

You can also access the exception value manually and assert on it:

```python
def test_invalid_operator:
    with pytest.raises(ValueError) as excinfo :
        calc(1, 2, "@")
    assert str(excinfo.value) == "Invalid operator"
```

There is `pytest.warns` as well, to check for Python warnings (e.g. `DeprecationWarning`):

```python
def test_warning():
    with pytest.warns(UserWarning, match=...):
        warnings.warn(...)
```

Comparing floating point numbers

```
def test_add():
    res = calc(0.2, 0.1, "+")
    assert res == 0.3
```

Comparing floating point numbers

```
def test_add():
    res = calc(0.2, 0.1, "+")
    assert res == 0.3

E   assert 0.30000000000000004 == 0.3
```

- Floating point numbers have limited precision, thus comparisons via == are tricky
- This is a problem in almost every language: 0.30000000000000004.com (yes, really!)

Comparing floating point numbers

```python
def test_add():
    res = calc(0.2, 0.1, "+")
    assert res == 0.3
```

basic/test_approx.py

```python
def test_add():
    res = calc(0.2, 0.1, "+")
    assert res == pytest.approx(0.3)
```

```
E    assert 0.30000000000000004 == 0.3
```

- Floating point numbers have limited precision, thus comparisons via == are tricky
- This is a problem in almost every language: 0.30000000000000004.com (yes, really!)

Comparing floating point numbers

```python
def test_add():
    res = calc(0.2, 0.1, "+")
    assert res == 0.3
```

```
E       assert 0.30000000000000004 == 0.3
```

basic/test_approx.py

```python
def test_add():
    res = calc(0.2, 0.1, "+")
    assert res == pytest.approx(0.3)
```

- Floating point numbers have limited precision, thus comparisons via == are tricky
- This is a problem in almost every language: `0.30000000000000004.com` (yes, really!)
- `pytest.approx` instead of Python's `math.isclose` gives you nicer output
- Can override tolerance (rel/abs), e.g. 20 ± 2°: `assert temperature == pytest.approx(20, abs=2)`

Supports various data types:

- Sequences of numbers (e.g. lists or tuples)
- Dictionary values
- `numpy` arrays

Marks

You...

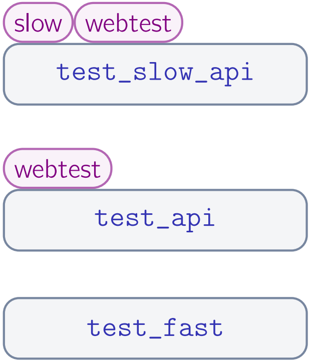
- Used **pytest.mark.skip**?
- Used **pytest.mark.xfail**?
- Used the **parametrize** mark?
- Used **pytest.param**?
- Customized **test ids**?
- Used **indirect parametrization**?
- Used the **pytestmark** global variable?

pytest.mark: Custom marking

Mark functions or classes:

— marking/test_marking.py

```python
@pytest.mark.slow
@pytest.mark.webtest
def test_slow_api():
    time.sleep(1)

@pytest.mark.webtest
def test_api():
    pass

def test_fast():
    pass
```

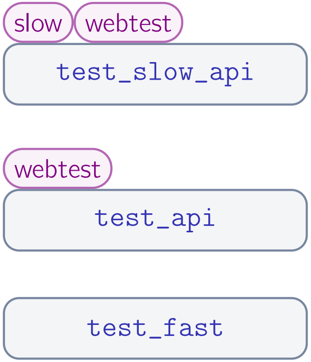


pytest.mark: Custom marking

Mark functions or classes:

— marking/test_marking.py

```
@pytest.mark.slow
@pytest.mark.webtest
def test_slow_api():
    time.sleep(1)

@pytest.mark.webtest
def test_api():
    pass

def test_fast():
    pass
```



On a basic level, marks are *tags* / *labels* for tests.

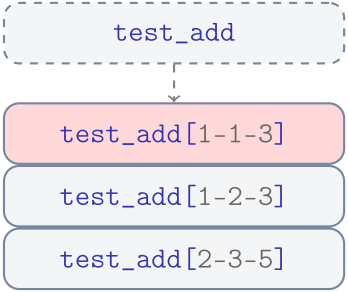
As we'll see later, marks are also used to attach meta-information to a test, used by pytest itself (parametrize, skip, xfail, . . .), by fixtures, or by plugins.

Parametrizing tests

Tests can be parametrized to run them with various values:

— marking/test_parametrization.py —

```python
@pytest.mark.parametrize("a, b, expected", [
    (1, 1, 3),
    (1, 2, 3),
    (2, 3, 5),
])
def test_add(a, b, expected):
    assert calc(a, b, "+") == expected

@pytest.mark.parametrize(
    "op", ["+", "-", "*", "/", "**"])
def test_smoke(op):
    calc(1, 2, op)
```

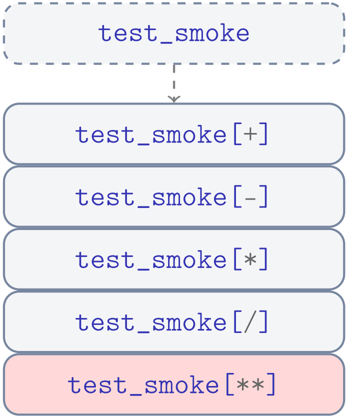


Parametrizing tests

Tests can be parametrized to run them with various values:

— marking/test_parametrization.py —

```python
@pytest.mark.parametrize("a, b, expected", [
    (1, 1, 3),
    (1, 2, 3),
    (2, 3, 5),
])
def test_add(a, b, expected):
    assert calc(a, b, "+") == expected

@pytest.mark.parametrize(
    "op", ["+", "-", "*", "/", "**"])
def test_smoke(op):
    calc(1, 2, op)
```



Parametrizing tests

Tests can be parametrized to run them with various values:

marking/test_parametrization.py

```
@pytest.mark.parametrize("a, b, expected", [
    (1, 1, 3),
    (1, 2, 3),
    (2, 3, 5),
])
def test_add(a, b, expected):
    assert calc(a, b, "+") == expected

@pytest.mark.parametrize(
    "op", ["+", "-", "*", "/", "**"])
def test_smoke(op):
    calc(1, 2, op)
```

Skipping or "xfailing" tests

Skip a test if:

- It cannot run at all on a certain platform
- It cannot run because a dependency is missing

⇒ Test function is not run, result is "skipped" (`s`)

Use `@pytest.mark.skip` (instead of `skipif`) for unconditional skipping.

```python
@pytest.mark.skipif(
    # condition
    sys.platform == "win32",
    # text shown with -v
    reason="Linux only",
)
def test_linux():
    ...
```

Skipping or “xfailing” tests

Skip a test if:

- It cannot run at all on a certain platform
- It cannot run because a dependency is missing

⇒ Test function is not run, result is “skipped” (`s`)

Use `@pytest.mark.skip` (instead of `skipif`) for unconditional skipping.

```python
@pytest.mark.skipif(
    # condition
    sys.platform == "win32",
    # text shown with -v
    reason="Linux only",
)
def test_linux():
    ...
```

“xfail” (“expected to fail”) a test if:

- The implementation is currently lacking
- It fails on a certain platform but should work

⇒ Test function is run, but result is “xfailed” (`x`), instead of failed (`F`). Unexpected pass: XPASS (`X`).

```python
@pytest.mark.xfail(
    # condition optional
    reason="see #1234",
)
def test_new_api():
    ...
```

The strict option for xfail

If a test marked `xfail` ("expected to fail") passes, the result is an `XPASS` (`X`) by default (counts as passed test). This can by changed globally by using:

```
[pytest]
xfail_strict=true
```

or for an individual mark via:

```
@pytest.mark.xfail(reason="...", strict=True)
```

Then, a test which is expected to fail but passes results in a failing test.
This can be useful to let pytest alert you about accidentally fixed bugs!

Note: This won't work with imperative `pytest.xfail(...)`, as test gets skipped.

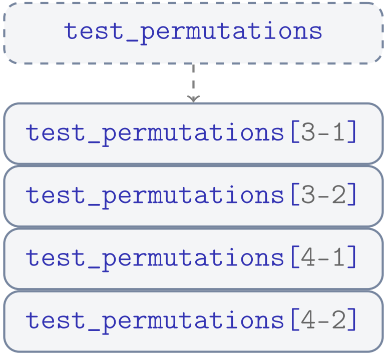
If you're dealing with flaky tests (sometimes passing, sometimes failing), it's better to use a plugin such as `pytest-rerunfailures` instead.

Permutations with parametrize

If we stack `@pytest.mark.parametrize` multiple times, we get all permutations:

marking/test_parametrization.py

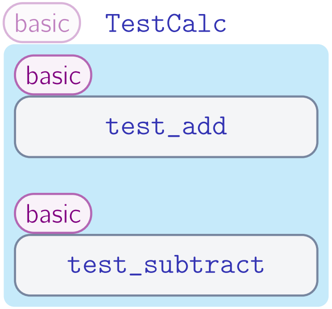
```python
@pytest.mark.parametrize("a", [1, 2])
@pytest.mark.parametrize("b", [3, 4])
def test_permutations(a, b):
    assert calc(a, b, "+") == a + b
```



Marking an entire class

Decorators can be applied to classes as well as functions:

```python
@pytest.mark.basic
class TestCalc:
    def test_add(self):
        assert calc(1, 3, "+") == 4

    def test_subtract(self):
        ...
```



Marking an entire test file

To apply a mark to an entire test file, a special `pytestmark` global variable can be set to a mark, or a list of marks.

```
pytestmark = pytest.mark.skipif(
    sys.platform == "win32",
    reason="Linux only"
)
```

The same technique can be used to skip all tests in a file unconditionally:

```
pytestmark = pytest.mark.skip("Work in progress")
```

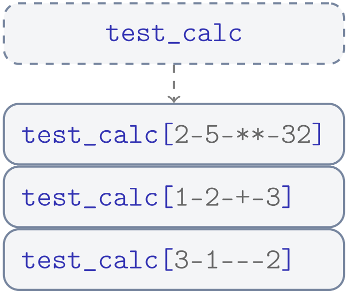
If all tests in a file should be skipped if a library was not found, there's a helper. This is the same as `import pexpect`, but the module is skipped on `ImportError`:

```
pexpect = pytest.importorskip("pexpect")
```

Marking single parameters

If we want to e.g. xfail a single parameter combination in a parametrized test, the mark can be attached to a single value via `pytest.param`:

```python
@pytest.mark.parametrize(
    "a, b, op, expected", [
        pytest.param(2, 5, "**", 32),
        (1, 2, "+", 3),
        (3, 1, "-", 2),
])
def test_calc(a, b, op, expected):
    assert calc(a, b, op) == expected
```

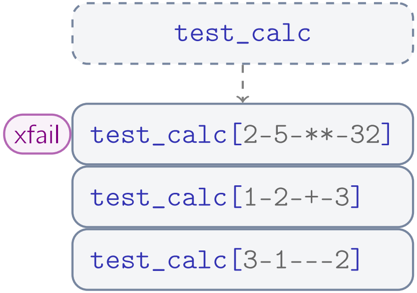


Marking single parameters

If we want to e.g. xfail a single parameter combination in a parametrized test, the mark can be attached to a single value via `pytest.param`:

— marking/test_parametrization_marks.py —

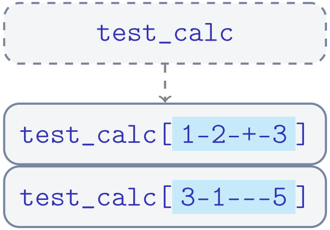
```python
@pytest.mark.parametrize(
    "a, b, op, expected", [
    pytest.param(
        2, 5, "**", 32,
        marks=pytest.mark.xfail(reason=...),
    ),
    (1, 2, "+", 3),
    (3, 1, "-", 5),
])
def test_calc(a, b, op, expected):
    assert calc(a, b, op) == expected
```



Changing test IDs

We can use `pytest.param(..., id="...")` to override the auto-generated test ID:

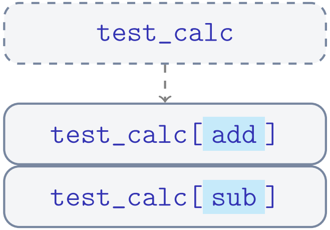
```python
@pytest.mark.parametrize("a, b, op, expected", [
    pytest.param(1, 2, "+", 3),
    pytest.param(3, 1, "-", 5),
])
def test_calc(a, b, op, expected):
    assert calc(a, b, op) == expected
```



Changing test IDs

We can use `pytest.param(..., id="...")` to override the auto-generated test ID:

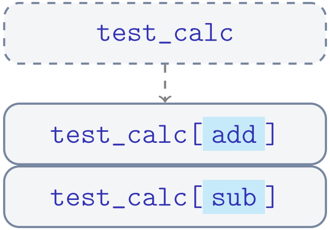
```
@pytest.mark.parametrize("a, b, op, expected", [
    pytest.param(1, 2, "+", 3, id="add"),
    pytest.param(3, 1, "-", 5, id="sub"),
])
def test_calc(a, b, op, expected):
    assert calc(a, b, op) == expected
```



Changing test IDs

We can use `pytest.param(..., id="...")` to override the auto-generated test ID:

```python
@pytest.mark.parametrize("a, b, op, expected", [
    pytest.param(1, 2, "+", 3, id="add"),
    pytest.param(3, 1, "-", 5, id="sub"),
])
def test_calc(a, b, op, expected):
    assert calc(a, b, op) == expected
```



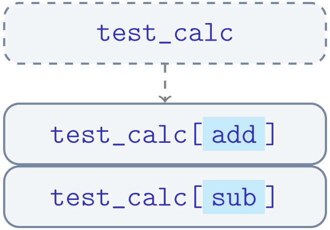
Or pass a list as `ids=` keyword argument to `@pytest.mark.parametrize`:

```python
@pytest.mark.parametrize("a, b, op, expected", [
    (1, 2, "+", 3), (3, 1, "-", 5),
], ids=["add", "sub"])
def test_calc(a, b, op, expected):
    assert calc(a, b, op) == expected
```

Changing test IDs

We can use `pytest.param(..., id="...")` to override the auto-generated test ID:

```python
@pytest.mark.parametrize("a, b, op, expected", [
    pytest.param(1, 2, "+", 3, id="add"),
    pytest.param(3, 1, "-", 5, id="sub"),
])
def test_calc(a, b, op, expected):
    assert calc(a, b, op) == expected
```



Or pass a list as `ids=` keyword argument to `@pytest.mark.parametrize`:

```python
@pytest.mark.parametrize("a, b, op, expected", [
    (1, 2, "+", 3), (3, 1, "-", 5),
], ids=["add", "sub"])
def test_calc(a, b, op, expected):
    assert calc(a, b, op) == expected
```

We can also pass a callable to generate IDs.

e.g. for a list:
`ids=", ".join`

Python dataclasses

Without dataclasses:

```python
class Point:

    def __init__(self, x: int, y: int) -> None:
        self.x = x
        self.y = y

    def __repr__(self) -> str:
        return f"Point(x={self.x!r}, y={self.y!r})"

    def __eq__(self, other: Any) -> bool:
        if not isinstance(other, Point):
            return NotImplemented
        return (self.x, self.y) == (other.x, other.y)
```

With dataclasses:

```python
@dataclass
class Point:
    x: int
    y: int
```

Trick: Using dataclasses with parametrize

```python
@dataclass
class CalcCase:
    name: str
    a: int
    b: int
    result: int
    op: str = "+"
```

Trick: Using dataclasses with parametrize

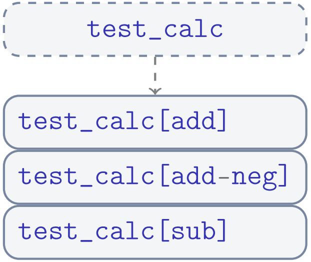
```
@dataclass
class CalcCase:
    name: str
    a: int
    b: int
    result: int
    op: str = "+"
```

```
@pytest.mark.parametrize("tc", [
    CalcCase("add", a=1, b=2, result=3),
    CalcCase("add-neg", a=-2, b=-3, result=-5),
    CalcCase("sub", a=2, b=1, op="-", result=1),
], ids=lambda tc: tc.name)
def test_calc(tc):
    assert calc(tc.a, tc.b, tc.op) == tc.result
```

Trick: Using dataclasses with parametrize

```python
@dataclass
class CalcCase:
    name: str
    a: int
    b: int
    result: int
    op: str = "+"
```

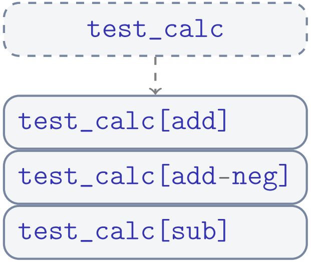
```python
@pytest.mark.parametrize("tc", [
    CalcCase("add", a=1, b=2, result=3),
    CalcCase("add-neg", a=-2, b=-3, result=-5),
    CalcCase("sub", a=2, b=1, op="-", result=1),
], ids=lambda tc: tc.name)
def test_calc(tc):
    assert calc(tc.a, tc.b, tc.op) == tc.result
```



Trick: Using dataclasses with parametrize

```python
@dataclass
class CalcCase:
    name: str
    a: int
    b: int
    result: int
    op: str = "+"
```

```python
@pytest.mark.parametrize("tc", [
    CalcCase("add", a=1, b=2, result=3),
    CalcCase("add-neg", a=-2, b=-3, result=-5),
    CalcCase("sub", a=2, b=1, op="-", result=1),
], ids=lambda tc: tc.name)
def test_calc(tc):
    assert calc(tc.a, tc.b, tc.op) == tc.result
```



- Easy setting of test names – could even write a custom `def __str__(self):` and use `ids=str`
- Thanks to default arguments, we only need to specify values that differ from the default (no `op="+"`)
- Better type safety, better autocompletion
- More readability for complex test cases

Expanding the calculator example

Reverse Polish Notation (RPN)

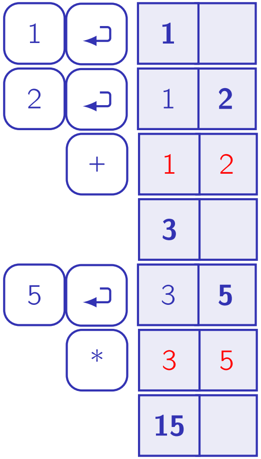
In Python

In `code/`, using `python -m rpncalc.rpn_v1`:

```
> 1
> 2
> p
[1.0, 2.0]
> +
3
> 5
> *
15
> q
```

Demo: Run the calculator,
play around a bit, and try to break it.

Take a first look at its code
(`rpncalc/rpn_v1.py`),
it'll be explained on the next slide.



Reverse Polish Notation (RPN)

rpncalc/rpn_v1.py

```python
from rpncalc.utils import calc

class RPNCalculator:
    def __init__(self):
        self.stack = []

    def run(self):
        while True:
            inp = input("> ")
            if inp == "q":
                return
            elif inp == "p":
                print(self.stack)
            else:
                self.evaluate(inp)

    def evaluate(self, inp):
        if inp.isdigit():
            n = float(inp)
            self.stack.append(n)
        elif inp in "+-*/":
            b = self.stack.pop()
            a = self.stack.pop()
            res = calc(a, b, inp)
            self.stack.append(res)
            print(res)


if __name__ == "__main__":
    rpn = RPNCalculator()
    rpn.run()
```

Fixtures

You…

- Used pytest **fixtures**?
- Used **conftest.py**?
- Used **“yield” in a fixture**?
- Used **tmp_path** or **monkeypatch**?
- Used **scope=...** or **autouse=** for a fixture?
- **Parametrized a fixture**?

Good practices for fixtures

Consider adding type annotations:

```
@pytest.fixture
def rpn() -> RPNCalculator:
    """A RPN calculator with a default config."""
    ...

def test_rpn(rpn: RPNCalculator):
    ...
```

Good practices for fixtures

Consider adding type annotations, write a docstring for your fixtures:

```
@pytest.fixture
def rpn() -> RPNCalculator:
    """A RPN calculator with a default config."""
    ...
```

`--fixtures`	Show all defined fixtures with their docstrings.
`--fixtures-per-test`	Show the fixtures used, grouped by test.

Output:

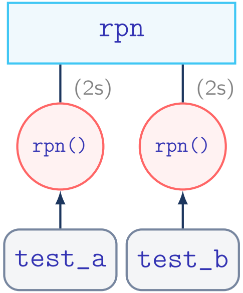
```
------------------ fixtures defined from test_fixture ------------------
rpn -- fixtures/test_fixture.py:7
    A RPN calculator with a default config.
```

Caching fixture results

Fixture functions can declare a caching scope:

```
@pytest.fixture
def rpn() -> RPNCalculator:
    time.sleep(2)
    return RPNCalculator(Config())

def test_a(rpn: RPNCalculator):
    rpn.stack.append(42)
    assert rpn.stack == [42]

def test_b(rpn: RPNCalculator):
    assert not rpn.stack
```

Function scope:



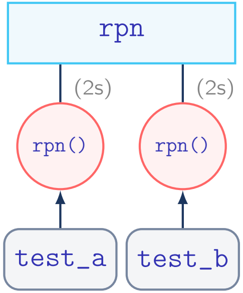
Caching fixture results

Fixture functions can declare a caching scope:

— fixtures/test_fixture_scope.py —

```python
@pytest.fixture(scope="function")
def rpn() -> RPNCalculator:
    time.sleep(2)
    return RPNCalculator(Config())

def test_a(rpn: RPNCalculator):
    rpn.stack.append(42)
    assert rpn.stack == [42]

def test_b(rpn: RPNCalculator):
    assert not rpn.stack
```

Function scope:

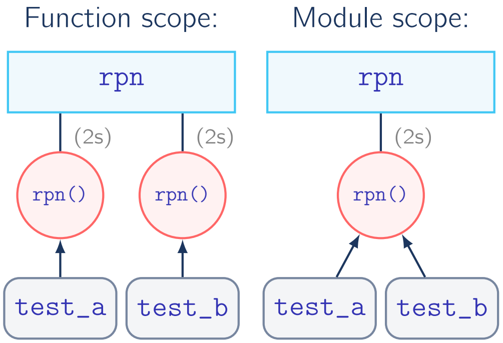


Caching fixture results

Fixture functions can declare a caching scope:

— fixtures/test_fixture_scope.py —

```python
@pytest.fixture(scope="module")
def rpn() -> RPNCalculator:
    time.sleep(2)
    return RPNCalculator(Config())

def test_a(rpn: RPNCalculator):
    rpn.stack.append(42)
    assert rpn.stack == [42]

def test_b(rpn: RPNCalculator):
    assert not rpn.stack
```



Caching fixture results

Fixture functions can declare a caching scope:

— fixtures/test_fixture_scope.py ——

```python
@pytest.fixture(scope="module")
def rpn() -> RPNCalculator:
    time.sleep(2)
    return RPNCalculator(Config())

def test_a(rpn: RPNCalculator):
    rpn.stack.append(42)
    assert rpn.stack == [42]

def test_b(rpn: RPNCalculator):
    assert not rpn.stack
```

Available scopes:

- `"function"` (default)
- `"class"`
- `"module"`
- `"package"`
- `"session"`

Caching fixture results

Fixture functions can declare a caching scope:

— fixtures/test_fixture_scope.py —

```python
@pytest.fixture(scope="module")
def rpn() -> RPNCalculator:
    time.sleep(2)
    return RPNCalculator(Config())

def test_a(rpn: RPNCalculator):
    rpn.stack.append(42)
    assert rpn.stack == [42]

def test_b(rpn: RPNCalculator):
    assert not rpn.stack
```

Available scopes:

- `"function"` (default)
- `"class"`
- `"module"`
- `"package"`
- `"session"`

Can also pass a callable to `scope` to dynamically determine the scope.
It will get called with the fixture name and the `pytest.Config` object.

Caching fixture results

Fixture functions can declare a caching scope:

— fixtures/test_fixture_scope.py —

```python
@pytest.fixture(scope="module")
def rpn() -> RPNCalculator:
    time.sleep(2)
    return RPNCalculator(Config())

def test_a(rpn: RPNCalculator):
    rpn.stack.append(42)
    assert rpn.stack == [42]

def test_b(rpn: RPNCalculator):
    assert not rpn.stack
```

Beware:

+ Faster tests (4s → 2s)

– Less isolation between tests:

```
...::test_a PASSED
...::test_b FAILED
```

```
def test_b(rpn: RPNCalculator):
>     assert not rpn.stack
E     assert not [42]
```

Caching fixture results

Combining scopes

— fixtures/test_fixture_scope_reset.py

```
@pytest.fixture(scope="module")
def rpn_instance() -> RPNCalculator:
    time.sleep(2)
    return RPNCalculator(Config())


@pytest.fixture
def rpn(
    rpn_instance: RPNCalculator,
) -> RPNCalculator:
    rpn_instance.stack.clear()
    return rpn_instance
```

Caching fixture results

Combining scopes

— fixtures/test_fixture_scope_reset.py

```python
@pytest.fixture(scope="module")
def rpn_instance() -> RPNCalculator:
    time.sleep(2)
    return RPNCalculator(Config())


@pytest.fixture
def rpn(
    rpn_instance: RPNCalculator,
) -> RPNCalculator:
    rpn_instance.stack.clear()
    return rpn_instance
```

Caching fixture results

Combining scopes

— fixtures/test_fixture_scope_reset.py

```python
@pytest.fixture(scope="module")
def rpn_instance() -> RPNCalculator:
    time.sleep(2)
    return RPNCalculator(Config())


@pytest.fixture
def rpn(
    rpn_instance: RPNCalculator,
) -> RPNCalculator:
    rpn_instance.stack.clear()
    return rpn_instance
```

```python
def test_a(rpn: RPNCalculator):
    rpn.stack.append(42)
    assert rpn.stack == [42]


def test_b(rpn: RPNCalculator):
    assert not rpn.stack
```

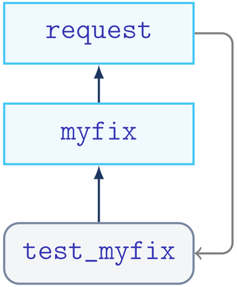
```
...::test_a PASSED
...::test_b PASSED

===== 2 passed in 2.00s =====
```

Introspecting calling site

request

Fixture functions can receive a `request` object with attributes:

```
@pytest.fixture
def myfix(request: pytest.FixtureRequest):
    request.function # test function/method
    request.cls      # class of test
    request.instance # class instance
    request.module   # module of test

    request.fspath   # path object of module
    request.node     # collection node
    request.config   # pytest config object

def test_myfix(myfix):
    ...
```



Introspecting calling site

request: Getting fixtures dynamically

With `request.node.getfixturevalue(...)`, we can get a fixture dynamically:

```
@pytest.fixture
def default_config() -> Config:
    return Config(prompt=">")
```

```
@pytest.fixture
def long_config() -> Config:
    return Config(prompt="rpn>")
```

```
@pytest.mark.parametrize("confname", ["default", "long"])
def test_configs(request: pytest.FixtureRequest, confname: str):
    config = request.node.getfixturevalue(f"{confname}_config")
    assert config.prompt.endswith(">")
```

Alternatives and (much!) more discussion:

- pytest issue 349: Using fixtures in `pytest.mark.parametrize`
- The `pytest-cases` plugin for more sophisticated parametrization

Introspecting calling site

request: Accessing markers

Via `request.node.get_closest_marker(...)` fixture, we can get markers from test nodes:

```python
@pytest.fixture
def config(request: pytest.FixtureRequest) -> Config:
    marker = request.node.get_closest_marker("long_prompt")
    if marker is None:
        return Config(prompt=">")
    return Config(prompt="rpn>")
```

Then a fixture can act differently depending on whether there is a marker:

```python
def test_normal(config: Config):
    assert config.prompt == ">"
```

```python
@pytest.mark.long_prompt
def test_marker(config: Config):
    assert config.prompt == "rpn>"
```

Introspecting calling site

request: Accessing markers

As with e.g. `@pytest.mark.parametrize(...)` or `@pytest.mark.skipif(...)`, you can pass arguments to markers, and access them in your fixture:

```
@pytest.fixture
def server_config(request: pytest.FixtureRequest) -> ServerConfig:
    marker = request.node.get_closest_marker("config_args")
    if marker is None:
        return ServerConfig()
    return ServerConfig(*marker.args , **marker.kwargs)
```

The returned `Mark` object has `args` and `kwargs` attributes with the arguments passed to it. With `@pytest.mark.config_args("production.json", strict=True)`:

```
marker.args    →  ("production.json",)        (single-element tuple)
marker.kwargs  →  {"strict": True}
```

Introspecting calling site

request: Accessing markers

As with e.g. `@pytest.mark.parametrize(...)` or `@pytest.mark.skipif(...)`, you can pass arguments to markers, and access them in your fixture:

```
@pytest.fixture
def server_config(request: pytest.FixtureRequest) -> ServerConfig:
    marker = request.node.get_closest_marker("config_args")
    if marker is None:
        return ServerConfig()
    return ServerConfig(*marker.args, **marker.kwargs)
```

Demo: Adjust the `config` fixture from last slide to use a `rpn_prompt` marker argument as a prompt, then write a test with `@pytest.mark.rpn_prompt("calc>")`.

Debugging failing tests

Arguments for debugging test issues

	`--tb`	Control traceback generation
		`--tb=auto` / `long` / `short` / `line` / `native` / `no`
`-l`	`--showlocals`	Show locals in tracebacks
`--lf`	`--last-failed`	Run last-failed only
`--ff`	`--failed-first`	Run last-failed first
`--nf`	`--new-first`	Run new test files first
`--sw`	`--stepwise`	Look at failures step by step
`-x`	`--maxfail=`*n*	Exit instantly on first / *n*-th failure
	`--pdb`	Drop into Python debugger on failures
	`--trace`	Drop into Python debugger for every test
	`--durations=`*n*	Show the *n* slowest tests/fixtures

See `pytest -h` (`--help`) for many more options.

Tracing fixture setup/teardown

`--setup-show`	Show fixtures as they are set up, used and torn down.
`--setup-only`	Only setup fixtures, do not execute tests.
`--setup-plan`	Show what fixtures/tests would be executed, but don't run.

Output:

```
fixtures/test_fixture.py
    SETUP    F rpn
    fixtures/test_fixture.py::test_empty_stack (fixtures used: rpn).
    TEARDOWN F rpn
```

F: function scope, there is also **C**lass, **M**odule, **P**ackage, and **S**ession

Adding information to an assert

Using a comma after `assert ...`, additional information can be printed:

```
def test_add(rpn: RPNCalculator):
    rpn.evaluate("2")
    rpn.evaluate("3")
    rpn.evaluate("1")
    rpn.evaluate("+")
    assert rpn.stack[-1] == 5, rpn.stack
```

Output:

```
>           assert rpn.stack[-1] == 5, rpn.stack
E       AssertionError: [2.0, 4.0]
E       assert 4.0 == 5
```

Using pdb for debugging

pdb is a command-line debugger for Python.
Trigger it with `--pdb` or `--trace` in pytest, or `breakpoint()` in your code.

At the `(pdb)` prompt, you can use:

`bt` / `w` / `where`	Print the traceback
`l` / `list`	Show the current source code
`h` / `help`	Show help
`p` / `pp`	(Pretty) print a variable
`c` / `continue`	Continue to next breakpoint
`d` / `down`, `u` / `up`	Move up/down the stack
`interact`	Open Python shell

Showing slow test durations

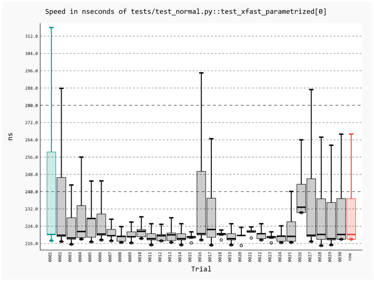
Running e.g. `pytest --durations=20` reveals slow tests:

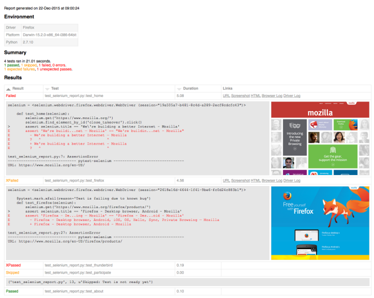
```
2.00s setup    fixtures/test_fixture_scope_reset.py::test_a
2.00s setup    fixtures/test_fixture_scope.py::test_b
2.00s setup    fixtures/test_fixture_scope.py::test_a
1.00s call     marking/test_marking.py::test_slow_api
0.27s call     mocking/test_real.py::test_convert
...

(4 durations < 0.005s hidden.  Use -vv to show these durations.)
```

Some plugins

Plugins, plugins . . .

- **Property-based testing:** hypothesis
- **Customized reporting:** pytest-html, pytest-rich, pytest-instafail, pytest-emoji
- **Repeating tests:** pytest-repeat, pytest-rerunfailures, pytest-benchmark
- **Framework/Language integration:** pytest-twisted, pytest-django, pytest-qt, pytest-asyncio, pytest-cpp
- **Coverage and mock integration:** pytest-cov, pytest-mock
- **Other:** pytest-bdd (behaviour-driven testing), pytest-xdist (distributed testing)
- . . . ≈ 1400 more: `https://docs.pytest.org/en/latest/reference/plugin_list.html`




Report generated on 22-Dec-2015 at 09:00:24

Environment

Driver	Firefox
Platform	Darwin-15.2.0-x86_64-i386-64bit
Python	2.7.10

Summary

4 tests ran in 21.01 seconds.
1 passed, 1 skipped, 1 failed, 0 errors.
1 expected failures, 1 unexpected passes.

Results

Result	Test	Duration	Links
Failed	test_selenium_report.py::test_home	5.08	URL Screenshot HTML Browser Log Driver Log

```
selenium = <selenium.webdriver.firefox.webdriver.WebDriver (session="19a505a7-b481-8c4d-a289-2ecf8cdcfc43")>

    def test_home(selenium):
        selenium.get('https://www.mozilla.org/')
        selenium.find_element_by_id('close_takeover').click()
>       assert selenium.title == 'We\'re building a better Internet - Mozilla'
E       assert 'We're buildi...net - Mozilla' == "We're buildin...net - Mozilla"
E         - We're building a better Internet – Mozilla
E         ?   ^                               ^
E         + We're building a better Internet - Mozilla
E         ?   ^                               ^

test_selenium_report.py:7: AssertionError
------------------------------ pytest-selenium ------------------------------
URL: https://www.mozilla.org/en-US/
```

Result	Test	Duration	Links
XFailed	test_selenium_report.py::test_firefox	4.56	URL Screenshot HTML Browser Log Driver Log

```
selenium = <selenium.webdriver.firefox.webdriver.WebDriver (session="2618e16d-4644-1f41-9be6-fc0d24c883b1")>

    @pytest.mark.xfail(reason='Test is failing due to known bug')
    def test_firefox(selenium):
        selenium.get('https://www.mozilla.org/firefox/products/')
>       assert selenium.title == 'Firefox - Desktop browser, Android - Mozilla'
E       assert 'Firefox - De...ing - Mozilla' == 'Firefox - Des...oid - Mozilla'
E         - Firefox - Desktop browser, Android, iOS, OS, Hello, Sync, Private Browsing – Mozilla
E         + Firefox - Desktop browser, Android - Mozilla

test_selenium_report.py:27: AssertionError
------------------------------ pytest-selenium ------------------------------
URL: https://www.mozilla.org/en-US/firefox/products/
```

Result	Test	Duration	Links
XPassed	test_selenium_report.py::test_thunderbird	0.19	
Skipped	test_selenium_report.py::test_participate	0.00	

```
('test_selenium_report.py', 13, u'Skipped: Test is not ready yet')
```

Result	Test	Duration	Links
Passed	test_selenium_report.py::test_about	0.10	

Reporting hooks

```python
def pytest_report_header(
    config: pytest.Config,
    start_path: pathlib.Path,
) -> str | list[str]:
    """Return a string or list of strings to be displayed
    as header info for terminal reporting."""

def pytest_terminal_summary(
    terminalreporter,   # currently undocumented...
    exitstatus: pytest.ExitCode,
    config: pytest.Config,
) -> None:
   """Add a section to terminal summary reporting."""
```

Adding to header and summary

hooks/reporting/conftest.py

```python
def pytest_report_header() -> list[str]:
    return ["extrainfo: line 1"]


def pytest_terminal_summary(terminalreporter) -> None:
    if terminalreporter.verbosity >= 1:
        terminalreporter.section("my special section")
        terminalreporter.line("report something here")
```

Adding to header and summary

```
$ pytest
======================= test session starts ========================
platform linux -- Python ..., pytest-..., pluggy-...
extrainfo: line 1
...
======================= no tests ran in 0.00s ======================
```

Adding to header and summary

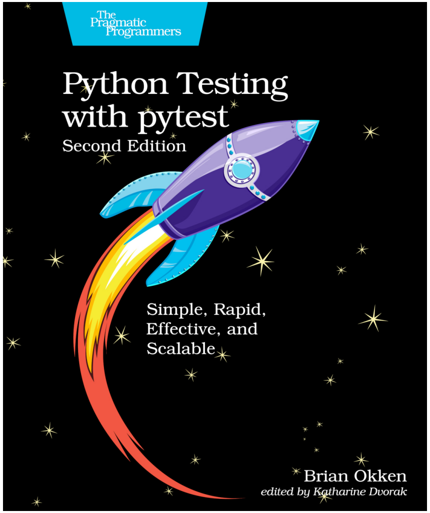
```
$ pytest
======================== test session starts =========================
platform linux -- Python ..., pytest-..., pluggy-...
extrainfo: line 1
...
======================= no tests ran in 0.00s ========================
```

```
$ pytest -v
======================== test session starts =========================
platform linux -- Python ..., pytest-..., pluggy-...
extrainfo: line 1
...
========================= my special section =========================
report something here
======================= no tests ran in 0.00s ========================
```

Outro

Book recommendation

Brian Okken: Python Testing with pytest, Second Edition (The Pragmatic Bookshelf)


The Pragmatic Programmers
Python Testing with pytest
Second Edition
Simple, Rapid, Effective, and Scalable
Brian Okken
edited by Katharine Dvorak

- ISBN 978-1680508604
- https://pragprog.com/titles/bopytest2/
- Discount code: **PyConDEBerlin**
  35% discount on PDF + epub + mobi
  until May 10th
- Full disclosure: I'm technical reviewer
  (but don't earn any money from it)

Upcoming events

- **June 11th to 13th, 2024:**
  Python Academy (python-academy.com):
  Professional Testing with Python
  Remote
- **June 17th to 22nd, 2024:**
  pytest sprint at Omicron Energy
  Klaus, Vorarlberg, Austria
- **March 4th to 6th, 2025**
  Python Academy (python-academy.com):
  Professional Testing with Python
  Leipzig, Germany & Remote

- **Custom training / coaching:**
  - Python
  - pytest
  - GUI programming with Qt
  - Best Practices (packaging, linting, etc.)
  - Git
  - ...

  Remote or on-site
  `florian@bruhin.software`
  `https://bruhin.software/`

Feedback and questions

Florian Bruhin
florian@bruhin.software
https://bruhin.software/
@the_compiler on ~~Twitter~~ 𝕏

BRUHIN
SOFTWARE