pytest tips and tricks

for a better testsuite

Florian Bruhin

Europython 2024

July 9th, 2024

About me

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

2006 Started programming (QBasic, bash)

2009 – 2013 Apprenticeship in electrical engineering

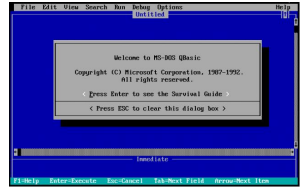
2011 Started using Python

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



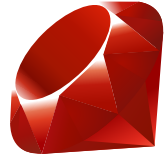
BRUNNER
dedication + innovation

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

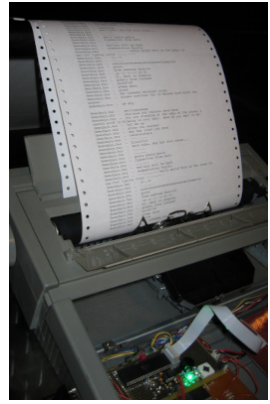


2–3 days/week in autumn (OST): Teaching Python to first-semester BSc. students
rest: open-source and training/consulting (Bruhin Software): Python, pytest, Qt

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

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

Timeline

09:30 – 11:00

15 minute
coffee break

11:15 – 12:45

Training contents

- **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**: Capturing output, patching, temporary files, 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 8.2 ($\geq$ 7.0 is okay).
- Download slides and example code for exercises:
  https://github.com/The-Compiler/pytest-tips-and-tricks

Setup with PyCharm / VS Code

- Open `code/` folder (**not** enclosing folder!) as project

- Open `basic/test_calc.py`, configure Python interpreter
- Wait until "Install requirements" prompt appears and accept

- Ctrl-Shift-P to open command palette, run "Python: Create Environment..."
- Select `venv` and `requirements.txt` for installation

- Open PyCharm / VS Code 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)
- Are the building blocks below high-level tools like `poetry`

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"

`first-proj/.venv`

- pytest 8.2.0
- pytest-cov
- pytest-mock

`second-proj/.venv`

- pytest 7.4.4
- requests
- pytest-recording

Using virtual environments

Installing and creating

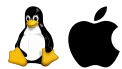
Install venv:

(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`.

Asserting expected exceptions

rpncalc/utils.py

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

basic/test_raises.py

```
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
- 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:

```
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:

```
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`):

```
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` (really!)

Comparing floating point numbers

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

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

```
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` (really!)

Comparing floating point numbers

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

```
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` (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

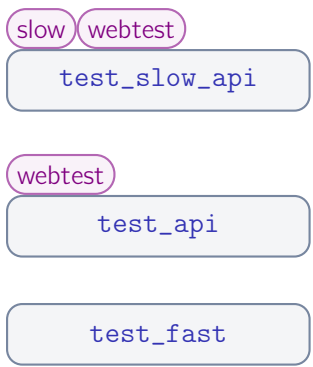
Mark functions or classes:

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
```



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.

pytest.mark: Custom marking

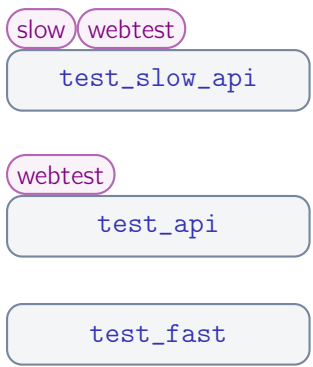
Mark functions or classes:

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
```



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.

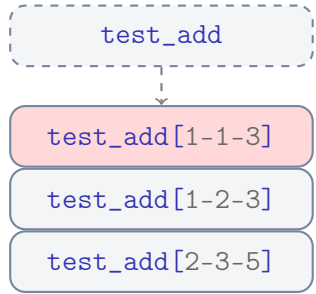
Parametrizing tests

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)
```

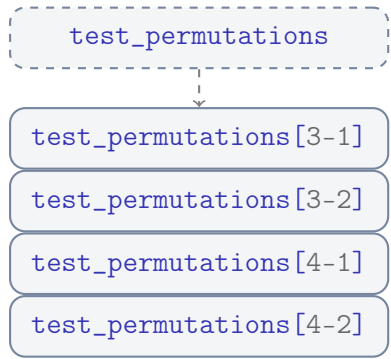


Permutations with parametrize

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
```



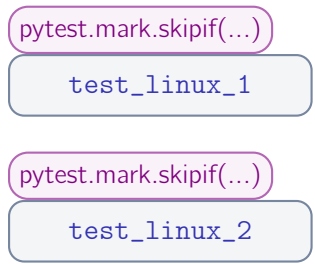
Reusing complex marks

A mark can also be stored in a variable and reused:

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

@skip_windows
def test_linux_1():
    ...

@skip_windows
def test_linux_2():
    ...
```

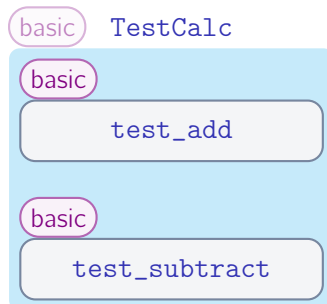


Marking an entire class

Decorators can be applied to classes as well as functions:

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

    def test_subtract(self):
        assert calc(7, 2, "-") == 5
```



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")
```

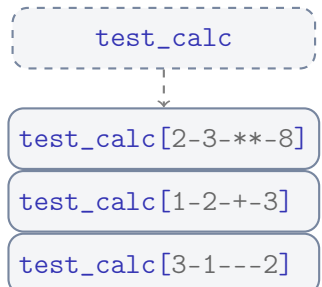
If all tests in a file should be skipped if a library was not found, there's a helper. This is the same as `import pexpect`, but the module is skipped on `ImportError`:

```
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`:

```
@pytest.mark.parametrize(
    "a, b, op, expected", [
        pytest.param(2, 3, "**", 8),
        (1, 2, "+", 3),
        (3, 1, "-", 2),
])
def test_calc(a, b, op, expected):
    assert calc(a, b, op) == expected
```

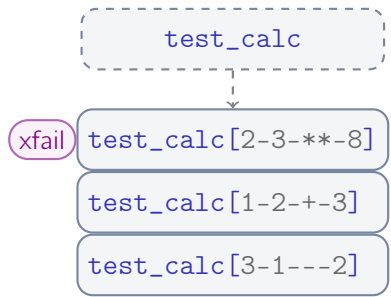


Marking single parameters

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

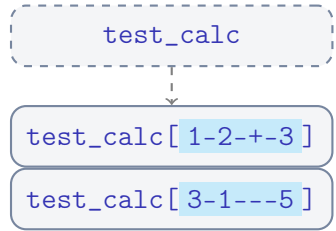
```python
@pytest.mark.parametrize(
    "a, b, op, expected", [
    pytest.param(
        2, 3, "**", 8,
        marks=pytest.mark.xfail(reason="..."),
    ),
    (1, 2, "+", 3),
    (3, 1, "-", 5),
])
def test_calc(a, b, op, expected):
    assert calc(a, b, op) == expected
```



Changing test IDs

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

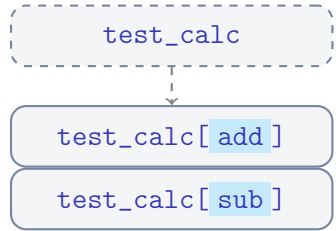
```python
@pytest.mark.parametrize("a, b, op, expected", [
    pytest.param(1, 2, "+", 3),
    pytest.param(3, 1, "-", 5),
])
def test_calc(a, b, op, expected):
    assert calc(a, b, op) == expected
```



Changing test IDs

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

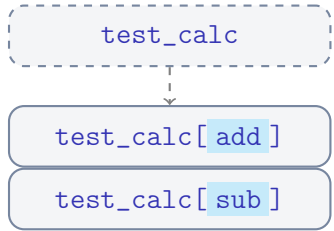
```python
@pytest.mark.parametrize("a, b, op, expected", [
    pytest.param(1, 2, "+", 3, id="add"),
    pytest.param(3, 1, "-", 5, id="sub"),
])
def test_calc(a, b, op, expected):
    assert calc(a, b, op) == expected
```



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
```



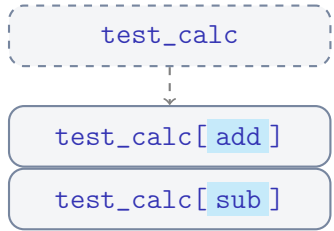
Or pass a list as `ids=` keyword argument to `@pytest.mark.parametrize`:

```
@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:

```
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:

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


Trick: Using dataclasses with parametrize

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

Trick: Using dataclasses with parametrize

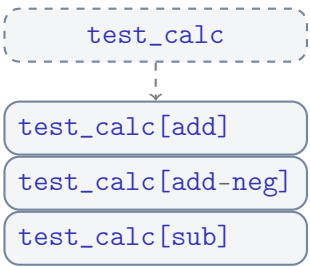
```python
@dataclass
class CalcCase:
    name: str
    a: int
    b: int
    result: int
    op: str = "+"
```

```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 = "+"
```

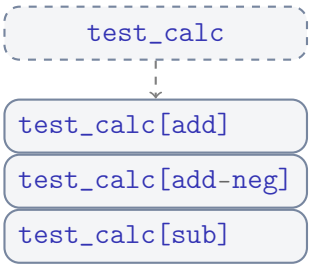
```python
@pytest.mark.parametrize("tc", [
    CalcCase("add", a=1, b=2, result=3),
    CalcCase("add-neg", a=-2, b=-3, result=-5),
    CalcCase("sub", a=2, b=1, op="-", result=1),
], ids=lambda tc: tc.name)
def test_calc(tc):
    assert calc(tc.a, tc.b, tc.op) == tc.result
```



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

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!

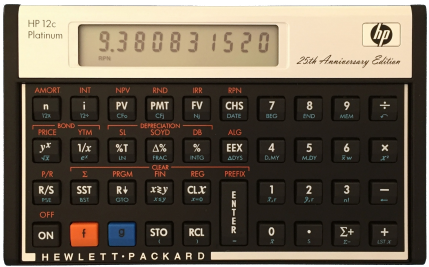
Note: This won't work with imperative `pytest.xfail(...)`, as test gets skipped.

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

Expanding the calculator example

Reverse Polish Notation (RPN)

History

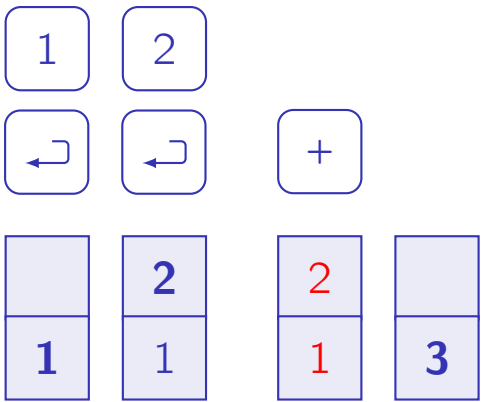


- Using a calculator without needing a = key, and without parentheses
- Makes it much easier to implement, using a stack data structure
- Based on the Polish notation, invented by Jan Łukasiewicz in 1924
- Used by all HP calculators in the 1970s–80s, still used by some today
- Displayed here: HP 12C financial calculator, introduced in 1981, still in production today (HPs longest and best-selling product)

Reverse Polish Notation (RPN)

Explanation

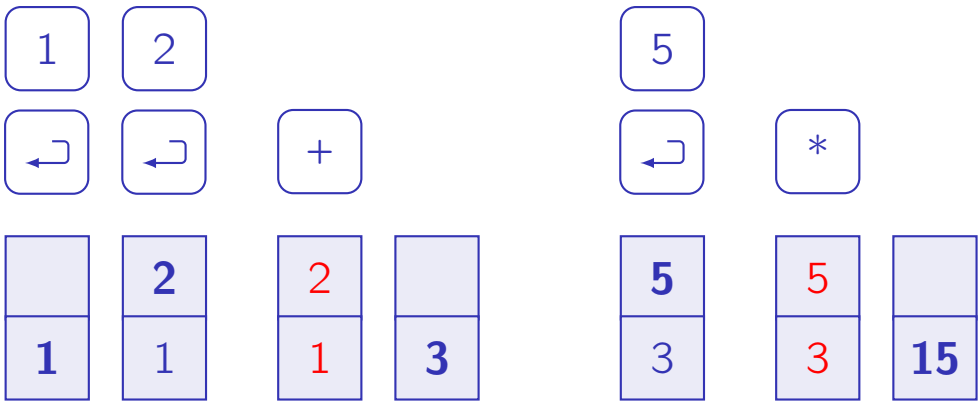
$$1 + 2$$



Reverse Polish Notation (RPN)

Explanation

$$5 \cdot (1 + 2)$$



Reverse Polish Notation (RPN)

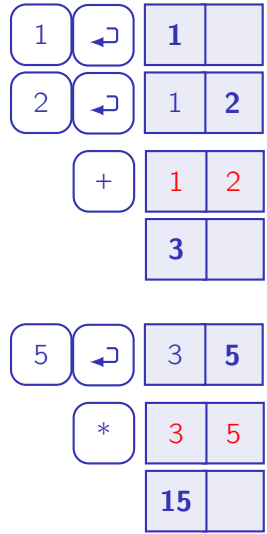
In Python

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) -> None:
        self.stack = []

    def run(self) -> None:
        while True:
            inp = input("> ")
            if inp == "q":
                return
            elif inp == "p":
                print(self.stack)
            else:
                self.evaluate(inp)
```

Reverse Polish Notation (RPN)

rpncalc/rpn_v1.py

```python
from rpncalc.utils import calc

class RPNCalculator:
    def __init__(self) -> None:
        self.stack = []

    def run(self) -> None:
        while True:
            inp = input("> ")
            if inp == "q":
                return
            elif inp == "p":
                print(self.stack)
            else:
                self.evaluate(inp)
```

Reverse Polish Notation (RPN)

rpncalc/rpn_v1.py

```
from rpncalc.utils import calc

class RPNCalculator:
    def __init__(self) -> None:
        self.stack = []

    def run(self) -> None:
        while True:
            inp = input("> ")
            if inp == "q":
                return
            elif inp == "p":
                print(self.stack)
            else:
                self.evaluate(inp)
```

```
    def evaluate(self, inp: str):
        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()
```

Reverse Polish Notation (RPN)

rpncalc/rpn_v1.py

```python
from rpncalc.utils import calc

class RPNCalculator:
    def __init__(self) -> None:
        self.stack = []

    def run(self) -> None:
        while True:
            inp = input("> ")
            if inp == "q":
                return
            elif inp == "p":
                print(self.stack)
            else:
                self.evaluate(inp)

    def evaluate(self, inp: str):
        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()
```

Reverse Polish Notation (RPN)

rpncalc/rpn_v1.py

```python
from rpncalc.utils import calc

class RPNCalculator:
    def __init__(self) -> None:
        self.stack = []

    def run(self) -> None:
        while True:
            inp = input("> ")
            if inp == "q":
                return
            elif inp == "p":
                print(self.stack)
            else:
                self.evaluate(inp)

    def evaluate(self, inp: str):
        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()
```

Reverse Polish Notation (RPN)

rpncalc/rpn_v1.py

```python
from rpncalc.utils import calc

class RPNCalculator:
    def __init__(self) -> None:
        self.stack = []

    def run(self) -> None:
        while True:
            inp = input("> ")
            if inp == "q":
                return
            elif inp == "p":
                print(self.stack)
            else:
                self.evaluate(inp)

    def evaluate(self, inp: str):
        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()
```

Reverse Polish Notation (RPN)

Towards an improved version

- Fix bugs and add additional error handling:
  - Allow negative numbers and floating-point inputs, not just `.isdigit()`
  - Fix `+-` being treated as valid input due to `elif inp in "+-*/":`
  - Print error when using an invalid operator
  - Handle `ZeroDivisionError` when dividing by zero, and `IndexError` with < 2 elements on stack
- Allow passing a `Config` object to the calculator, with a custom prompt, instead of `"> "`
- Support multiple inputs on one line, e.g. `2 1 +`

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**?

Reverse Polish Notation (RPN)

Code: Adding a Config object

rpncalc/rpn_v2.py

```python
from rpncalc.utils import calc, Config

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

...

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

rpncalc/utils.py

```python
class Config:
    def __init__(self, prompt=">"):
        self.prompt = prompt

    ...
```

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.
```

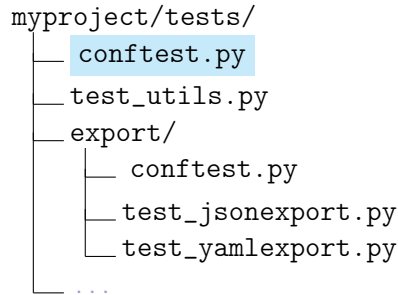
Output:

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

conftest.py fixtures

You can move fixture functions into `conftest.py`:

- Visible to modules in same or sub directory.
- Put hooks and cross-module used fixtures into conftest file(s) but don't import from them.

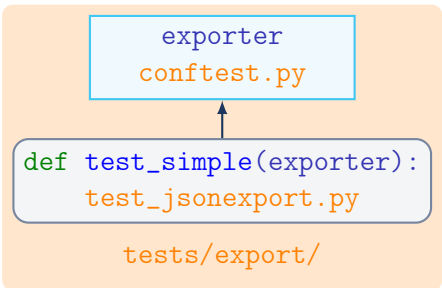


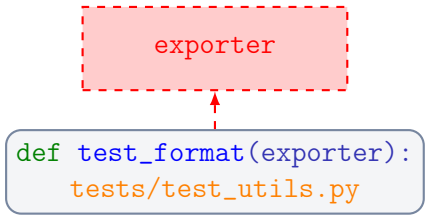
conftest.py fixtures

You can move fixture functions into `conftest.py`:

- Visible to modules in same or sub directory.
- Put hooks and cross-module used fixtures into conftest file(s) but don't import from them.

```
myproject/tests/
├── conftest.py
├── test_utils.py
└── export/
    ├── conftest.py
    ├── test_jsonexport.py
    └── test_yamlexport.py
    └── ...
```





conftest.py fixtures

You can move fixture functions into `conftest.py`:

- Visible to modules in same or sub directory.
- Put hooks and cross-module used fixtures into conftest file(s) but don't import from them.

```
myproject/tests/
├── conftest.py
├── test_utils.py
├── export/
│   ├── conftest.py
│   ├── test_jsonexport.py
│   └── test_yamlexport.py
└── ...
```

Caching fixture results

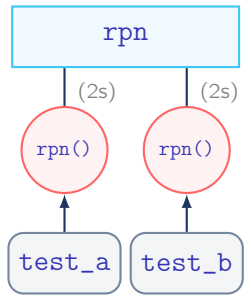
Fixture functions can declare a caching scope:

```python
@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:

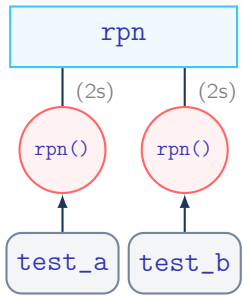
fixtures/test_fixture_scope.py

```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

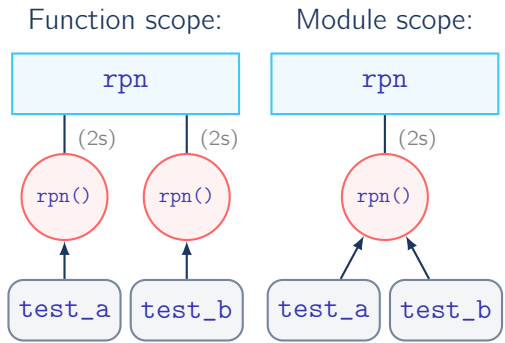
Fixture functions can declare a caching scope:

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

```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())
```

```python
@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 =====
```

Declarative or imperative skip/xfail

Declarative versions on a test function (**decorators**):

```
@pytest.mark.skipif(
    sys.platform == "win32",
    reason="Linux only",
)
def test_linux():
    ...


@pytest.mark.xfail(
    reason="not implemented",
)
def test_new_api():
    ...
```

Declarative or imperative skip/xfail

Declarative versions on a test function (**decorators**):

```
@pytest.mark.skipif(
    sys.platform == "win32",
    reason="Linux only",
)
def test_linux():
    ...


@pytest.mark.xfail(
    reason="not implemented",
)
def test_new_api():
    ...
```

Imperative skipping from **inside** test/fixture:

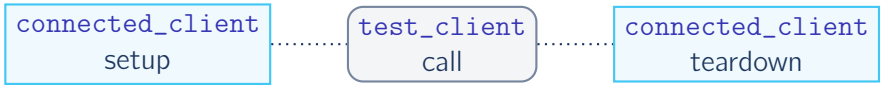
```
@pytest.fixture
def server_connection() -> Connection:
    conn = Connection(...)
    if not conn.server_is_reachable():
        pytest.skip("Server unreachable")
    return conn
```

```
pytest.skip(...)   ⇒  s
pytest.xfail(...)  ⇒  x
pytest.fail(...)   ⇒  F
```

Test is interrupted once called.

Doing cleanup with yield

A fixture can use `yield` instead of `return` to do cleanup:



fixtures/test_yield_fixture.py

```python
@pytest.fixture(scope="function")
def connected_client() -> Iterator[Client]:
    client = Client()
    client.connect()
    yield client
    client.disconnect()

def test_client_1(connected_client: Client):
    print("in the test 1")
```

Alternative: request.addfinalizer

Fixture functions can use `request.addfinalizer` to register teardown functions:

fixtures/test_fixture_finalizer.py

```python
@pytest.fixture
def connected_client(request: pytest.FixtureRequest) -> Client:
    client = Client()
    client.connect()
    request.addfinalizer(client.disconnect)
    return client
```

Using fixtures implicitly

autouse

If you want to prepare the environment for all tests implicitly, use `autouse=True`:

fixtures/test_autouse.py

```
class TestEmptyHomedir:
    @pytest.fixture(autouse=True)
    def tmp_homedir(self, tmp_path, monkeypatch):
        monkeypatch.setenv("HOME", str(tmp_path))
        return tmp_path

    def test_a(self, tmp_homedir):
        assert not list(tmp_homedir.iterdir())

    def test_b(self):
        assert not list(pathlib.Path.home().iterdir())
```

Using fixtures implicitly

autouse

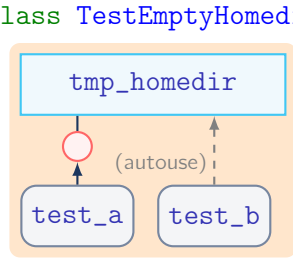
If you want to prepare the environment for all tests implicitly, use `autouse=True`:

fixtures/test_autouse.py

```python
class TestEmptyHomedir:
    @pytest.fixture(autouse=True)
    def tmp_homedir(self, tmp_path, monkeypatch):
        monkeypatch.setenv("HOME", str(tmp_path))
        return tmp_path

    def test_a(self, tmp_homedir):
        assert not list(tmp_homedir.iterdir())

    def test_b(self):
        assert not list(pathlib.Path.home().iterdir())
```



Using fixtures implicitly

autouse

If you want to prepare the environment for all tests implicitly, use `autouse=True`:

fixtures/test_autouse.py

```
class TestEmptyHomedir:
    @pytest.fixture(autouse=True)
    def tmp_homedir(self, tmp_path, monkeypatch):
        monkeypatch.setenv("HOME", str(tmp_path))
        return tmp_path

    def test_a(self, tmp_homedir):
        assert not list(tmp_homedir.iterdir())

    def test_b(self):
        assert not list(pathlib.Path.home().iterdir())
```

The patching will be done implicitly for every test for which the fixture is visible.

We can do something at the beginning of the test run via `scope="session"`.

We still return the `tmp_path` object in case a test wants to access it.

Using fixtures implicitly

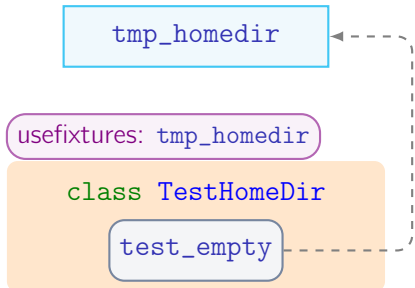
usefixtures marker

Using fixtures without specifying them as function arguments:

fixtures/test_usefixtures.py

```python
@pytest.fixture
def tmp_homedir(tmp_path, monkeypatch):
    monkeypatch.setenv("HOME", str(tmp_path))
    return tmp_path


@pytest.mark.usefixtures("tmp_homedir")
class TestHomeDir:
    def test_empty(self):
        assert not list(pathlib.Path.home().iterdir())
```



Introspecting calling site

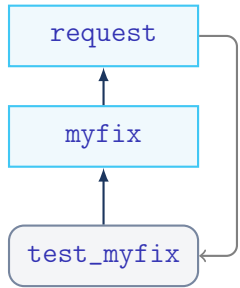
request

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

```python
@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:

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

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

```python
@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(...)`, we can get markers from test nodes:

fixtures/test_builtin_request_markers.py

```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:

```python
@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:

```python
@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>")`.

Configuring fixtures with command line options

Add and use command line options:

fixtures/cli_opt/conftest.py

```python
def pytest_addoption(parser: pytest.Parser) -> None:
    parser.addoption("--server-ip", type=str)

@pytest.fixture
def server_ip(request: pytest.FixtureRequest) -> str:
    return request.config.option.server_ip
```

Demo:

- Use this `conftest.py`: In `fixtures/cli_opt/`, create a `test_server_ip.py` with a test using the `server_ip` fixture
- Invoke with different `--server-ip` option values

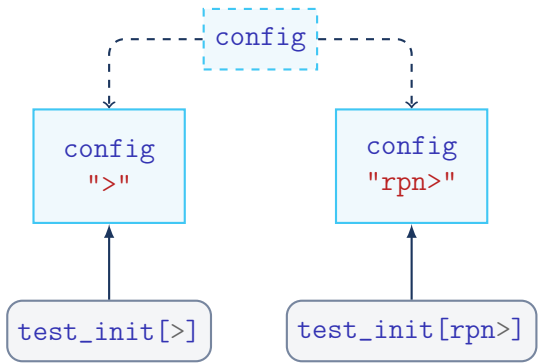
Parametrizing fixtures

— fixtures/test_fixture_param.py —

```python
@pytest.fixture(
    params=[
        ">",
        "rpn>",
    ]
)
def config(request):
    c = Config(prompt=request.param)
    return c

def test_init(config):
    print(config.prompt)
    assert False
```

Now each test using the `config` fixture will run twice!

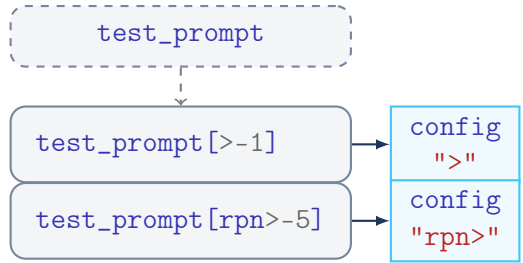


Indirect parametrization

fixtures/test_parametrize_indirect.py

```python
@pytest.fixture
def config(request):
    c = Config(prompt=request.param)
    return c


@pytest.mark.parametrize(
    "config, length", [
        (">", 1), ("rpn>", 4),
    ],
    indirect=["config"],
)
def test_prompt(config: Config, length: int):
    assert len(config.prompt) == length
```

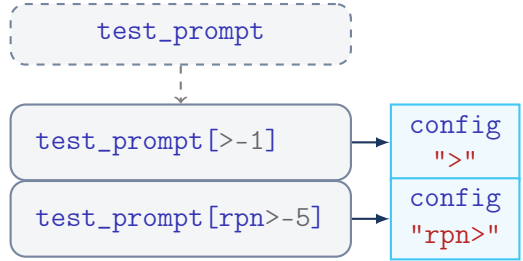


Indirect parametrization

fixtures/test_parametrize_indirect.py

```python
@pytest.fixture
def config(request):
    c = Config(prompt=request.param)
    return c


@pytest.mark.parametrize(
    "config, length", [
        (">", 1), ("rpn>", 4),
    ],
    indirect=["config"],
)
def test_prompt(config: Config, length: int):
    assert len(config.prompt) == length
```



Can take a list of argument names (like `indirect=["config"]` here), or `indirect=True` to make all indirect.

Builtin fixtures

pytest provides builtin fixtures:

`capsys` and `capfd`	Capturing stdout/stderr in a test
`caplog`	Capturing logging output from a test
`monkeypatch`	Temporarily modify state for test duration
`tmp_path` / `tmpdir`	A fresh empty directory for each test invocation
`request`	Get information about the current test
...	

More fixtures are provided by plugins.

Use `pytest --fixtures` to see available fixtures with docs

Builtin fixtures

Capturing

`capsys` Capturing stdout/stderr in a test by overriding `sys.stdout`
e.g. for `print(...)`

`capfd` Capturing stdout/stderr in a test at file descriptor level
e.g. for subprocesses or libraries printing from C code

`caplog` Capturing logging output from a test

Builtin fixtures

Capturing

`capsys` Capturing stdout/stderr in a test by overriding `sys.stdout`
e.g. for `print(...)`

`capfd` Capturing stdout/stderr in a test at file descriptor level
e.g. for subprocesses or libraries printing from C code

`caplog` Capturing logging output from a test

Background: Every process has *two* output streams: `stdout` and `stderr`.

```
$ ls /home doesnotexist
ls: cannot access 'doesnotexist': No such file or directory
/home:
username
```

```
$ ls /home doesnotexist > /dev/null
ls: cannot access 'doesnotexist': No such file or directory
```

Reverse Polish Notation (RPN)

Code: Fixing bugs, improving error handling

rpncalc/rpn_v2.py

```
def err(self, msg: str) -> None:
    print(msg, file=sys.stderr)

def evaluate(self, inp: str) -> None:
    ...
    if inp not in ["+", "-", "*", "/"]:
        self.err(
            f"Invalid input: {inp}")
        return

    if len(self.stack) < 2:
        self.err("Not enough operands")
        return

    b = self.stack.pop()
    a = self.stack.pop()

    try:
        res = calc(a, b, inp)
    except ZeroDivisionError:
        self.err("Division by zero")
        return

    self.stack.append(res)
    print(res)
```

Reverse Polish Notation (RPN)

Code: Fixing bugs, improving error handling

rpncalc/rpn_v2.py

```
def err(self, msg: str) -> None:
    print(msg, file=sys.stderr)

def evaluate(self, inp: str) -> None:
    ...
    if inp not in ["+", "-", "*", "/"]:
        self.err(
            f"Invalid input: {inp}")
        return

    if len(self.stack) < 2:
        self.err("Not enough operands")
        return

    b = self.stack.pop()
    a = self.stack.pop()

    try:
        res = calc(a, b, inp)
    except ZeroDivisionError:
        self.err("Division by zero")
        return

    self.stack.append(res)
    print(res)
```

Reverse Polish Notation (RPN)

Code: Fixing bugs, improving error handling

rpncalc/rpn_v2.py

```
def err(self, msg: str) -> None:
    print(msg, file=sys.stderr)

def evaluate(self, inp: str) -> None:
    ...
    if inp not in ["+", "-", "*", "/"]:
        self.err(
            f"Invalid input: {inp}")
        return

    if len(self.stack) < 2:
        self.err("Not enough operands")
        return

    b = self.stack.pop()
    a = self.stack.pop()

    try:
        res = calc(a, b, inp)
    except ZeroDivisionError:
        self.err("Division by zero")
        return

    self.stack.append(res)
    print(res)
```

Reverse Polish Notation (RPN)

Code: Fixing bugs, improving error handling

rpncalc/rpn_v2.py

```
def err(self, msg: str) -> None:
    print(msg, file=sys.stderr)

def evaluate(self, inp: str) -> None:
    ...
    if inp not in ["+", "-", "*", "/"]:
        self.err(
            f"Invalid input: {inp}")
        return

    if len(self.stack) < 2:
        self.err("Not enough operands")
        return

    b = self.stack.pop()
    a = self.stack.pop()

    try:
        res = calc(a, b, inp)
    except ZeroDivisionError:
        self.err("Division by zero")
        return

    self.stack.append(res)
    print(res)
```

Reverse Polish Notation (RPN)

Tests: Error handling

rpncalc/test_rpn_errors.py

```python
@pytest.mark.parametrize("op", ["**", "+-"])
def test_unknown_operator(rpn: RPNCalculator, op: str):
    rpn.stack = [1, 2]
    rpn.evaluate(op)  # FIXME how to test that this prints an error?

def test_division_by_zero(rpn: RPNCalculator):
    rpn.stack = [1, 0]
    rpn.evaluate("/")  # FIXME how to test that this prints an error?

@pytest.mark.parametrize("stack", [[1], []])
def test_not_enough_operands(rpn: RPNCalculator, stack: list[int]):
    rpn.stack = stack
    rpn.evaluate("+")  # FIXME how to test that this prints an error?
```

Reverse Polish Notation (RPN)

Tests: Error handling

rpncalc/test_rpn_errors.py

```python
@pytest.mark.parametrize("op", ["**", "+-"])
def test_unknown_operator(rpn: RPNCalculator, op: str):
    rpn.stack = [1, 2]
    rpn.evaluate(op)  # FIXME how to test that this prints an error?

def test_division_by_zero(rpn: RPNCalculator):
    rpn.stack = [1, 0]
    rpn.evaluate("/")  # FIXME how to test that this prints an error?

@pytest.mark.parametrize("stack", [[1], []])
def test_not_enough_operands(rpn: RPNCalculator, stack: list[int]):
    rpn.stack = stack
    rpn.evaluate("+")  # FIXME how to test that this prints an error?
```

Reverse Polish Notation (RPN)

Tests: Error handling

rpncalc/test_rpn_errors.py

```python
@pytest.mark.parametrize("op", ["**", "+-"])
def test_unknown_operator(rpn: RPNCalculator, op: str):
    rpn.stack = [1, 2]
    rpn.evaluate(op)  # FIXME how to test that this prints an error?

def test_division_by_zero(rpn: RPNCalculator):
    rpn.stack = [1, 0]
    rpn.evaluate("/")  # FIXME how to test that this prints an error?

@pytest.mark.parametrize("stack", [[1], []])
def test_not_enough_operands(rpn: RPNCalculator, stack: list[int]):
    rpn.stack = stack
    rpn.evaluate("+")  # FIXME how to test that this prints an error?
```

Builtin fixtures

Capturing

`capsys` overrides `sys.stdout`/`err`:

`fixtures/test_builtin_capsys.py`

```python
def test_output(capsys):
    print("Hello World")
    out, err = capsys.readouterr()
    assert out == "Hello World\n"
```

`capfd` captures at file descriptor level:

`fixtures/test_builtin_capfd.py`

```python
def test_output(capfd):
    subprocess.run(["ls"])
    out, err = capfd.readouterr()
    assert out == "..."
```

Builtin fixtures

Capturing

`capsys` overrides `sys.stdout`/`err`:

— fixtures/test_builtin_capsys.py —

```python
def test_output(capsys):
    print("Hello World")
    out, err = capsys.readouterr()
    assert out == "Hello World\n"
```

`capfd` captures at file descriptor level:

— fixtures/test_builtin_capfd.py —

```python
def test_output(capfd):
    subprocess.run(["ls"])
    out, err = capfd.readouterr()
    assert out == "..."
```

	stdout	stderr
`print("out 1")`	out 1	
`print("out 2")`	out 2	
`self.err("err 1")`		err 1

```
out, err = capsys.readouterr()
```

⇒ `out`: `"out 1\nout2\n"`
`err`: `"err 1\n"`

Builtin fixtures

Capturing

`capsys` overrides `sys.stdout`/`err`:

— fixtures/test_builtin_capsys.py —

```python
def test_output(capsys):
    print("Hello World")
    out, err = capsys.readouterr()
    assert out == "Hello World\n"
```

`capfd` captures at file descriptor level:

— fixtures/test_builtin_capfd.py —

```python
def test_output(capfd):
    subprocess.run(["ls"])
    out, err = capfd.readouterr()
    assert out == "..."
```

	stdout	stderr
`print("out 1")`	out 1	
`print("out 2")`	out 2	
`self.err("err 1")`		err 1

```
out, err = capsys.readouterr()
```

⇒ `out`: `"out 1\nout2\n"`
`err`: `"err 1\n"`

`print("out 3")`	out 3	
`self.err("err 2")`		err 2

```
out, err = capsys.readouterr()
```

⇒ `out`: `"out 3\n"`
`err`: `"err 2\n"`

Builtin fixtures

Capturing

`capsys` overrides `sys.stdout`/`err`:

— `fixtures/test_builtin_capsys.py` —

```
def test_output(capsys):
    print("Hello World")
    out, err = capsys.readouterr()
    assert out == "Hello World\n"
```

`capfd` captures at file descriptor level:

— `fixtures/test_builtin_capfd.py` —

```
def test_output(capfd):
    subprocess.run(["ls"])
    out, err = capfd.readouterr()
    assert out == "..."
```

Demo:

- Take a look at the revised `evaluate` method in `rpncalc/rpn_v2.py`, with added error handling.
- Pick one of the incomplete tests in `test_rpn_v2.py`.
- Complete it by capturing the error message printed on `stderr` and asserting on it.

Builtin fixtures

Capturing

`caplog` lets you access logging records:

fixtures/test_builtin_caplog.py

```python
def test_output(caplog):
    logging.warning("Something failed")
    assert caplog.messages == ["Something failed"]
```

Builtin fixtures

Capturing

`caplog` lets you access logging records:

fixtures/test_builtin_caplog.py

```python
def test_output(caplog):
    logging.warning("Something failed")
    assert caplog.messages == ["Something failed"]


def test_record_tuples(caplog):
    logging.warning("Something failed")
    assert caplog.record_tuples == [
        ("root", logging.WARNING, "Something failed")
    ]
```

Builtin fixtures

monkeypatch

The `monkeypatch` fixture allows to temporarily change state for the **duration of a test function execution**:

- Modify attributes on objects, classes or modules (`setattr`, `delattr`).
  Also works with functions/methods, as those are attributes of the respective module/class/instance too.
- Modify environment variables (`setenv`, `delenv`)
- Modify dictionaries (`setitem`, `delitem`)
- Change current directory (`chdir`)
- Prepend to `sys.path` for importing (`syspath_prepend`)

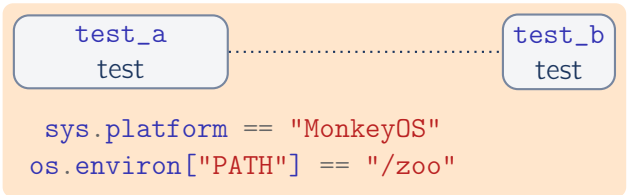
Builtin fixtures

monkeypatch: How to not do things

```python
def print_info():
    path = os.environ.get("PATH", "")
    print(f"platform: {sys.platform}")
    print(f"PATH: {path}")

def test_a():
    sys.platform = "MonkeyOS"    # don't do this!
    os.environ["PATH"] = "/zoo"  # don't do this!
    print_info()
    assert False

def test_b():
    print_info()
    assert False
```



Builtin fixtures

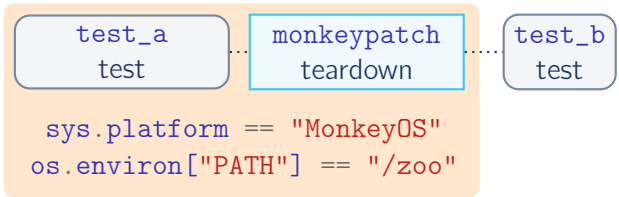
monkeypatch

fixtures/test_builtin_monkeypatch.py

```python
def print_info():
    path = os.environ.get("PATH", "")
    print(f"platform: {sys.platform}")
    print(f"PATH: {path}")

def test_a(monkeypatch: pytest.MonkeyPatch):
    monkeypatch.setattr(sys, "platform", "MonkeyOS")
    monkeypatch.setenv("PATH", "/zoo")
    print_info()
    assert False

def test_b():
    print_info()
    assert False
```



Builtin fixtures

monkeypatch: Patching functions

fixtures/test_builtin_monkeypatch.py

```python
def get_folder_name() -> str:
    user = getpass.getuser()
    return f"pytest-of-{user}"
```

Builtin fixtures

monkeypatch: Patching functions

fixtures/test_builtin_monkeypatch.py

```python
def get_folder_name() -> str:
    user = getpass.getuser()
    return f"pytest-of-{user}"


def fake_getuser() -> str:
    return "fakeuser"

def test_get_folder_name(monkeypatch: pytest.MonkeyPatch):
    monkeypatch.setattr(getpass, "getuser", fake_getuser)
    assert get_folder_name() == "pytest-of-fakeuser"
```

Builtin fixtures

monkeypatch: Patching functions

fixtures/test_builtin_monkeypatch.py

```python
def get_folder_name() -> str:
    user = getpass.getuser()
    return f"pytest-of-{user}"


def test_get_folder_name_lambda(monkeypatch: pytest.MonkeyPatch):
    monkeypatch.setattr(getpass, "getuser", lambda: "fakeuser")
    assert get_folder_name() == "pytest-of-fakeuser"
```

Reverse Polish Notation (RPN)

Code: Supporting multiple inputs

rpnalc/rpn_v1.py

```
class RPNCalculator:
    ...

    def run(self) -> None:
        while True:
            inp = input("> ")
            ...
            self.evaluate(inp)
```

rpncalc/rpn_v2.py

```
class RPNCalculator:
    ...

    def get_inputs(self) -> list[str]:
        inp = input(self.config.prompt + " ")
        return inp.split()

    def run(self) -> None:
        while True:
            for inp in self.get_inputs():
                ...
                self.evaluate(inp)
```

Builtin fixtures

monkeypatch

rpncalc/rpn_v2.py

```
class RPNCalculator:
    ...
    def get_inputs(self) -> list[str]:
        inp = input(self.config.prompt + " ")
        return inp.split()

    def run(self) -> None:
        while True:
            for inp in self.get_inputs():
                ...
```

```
monkeypatch.setattr(target, "name", value)
```

Demo:

- Add a new test to `test_rpn_v2.py`, which will test `run`.
- **Leave `rpn_v2.py` as-is!**
- Use the `monkeypatch` fixture as argument, to patch `get_inputs` on `rpn`.
- Replace it by a function that returns a fixed list of inputs, **ending in `"q"`**. You could define a `def fake_get_inputs():` or use `lambda: ...`.

Builtin fixtures

tmp_path / tmpdir

A **fresh empty directory** for each test invocation

- Store generated input files for tested code
- Store output files, e.g. measurement data, screenshots, etc.
- Access data even after pytest run is done, but no need for manual cleanup

Builtin fixtures

tmp_path / tmpdir

A **fresh empty directory** for each test invocation

- Store generated input files for tested code
- Store output files, e.g. measurement data, screenshots, etc.
- Access data even after pytest run is done, but no need for manual cleanup

`tmp_path`: Based on Python's `pathlib.Path`, a more object-oriented and convenient `os.path` alternative:

```
out_dir_path = pathlib.Path("output")
out_dir_path.mkdir(exist_ok=True)
out_file_path = out_dir_path / "output.txt"
out_file_path.write_text("Hello world!")
```

`tmpdir`:

Based on `py.path.local` from the `py` library (`pylib`) instead, since `pathlib` only exists since Python 3.4.

Should not be used in new code.

Reverse Polish Notation (RPN)

Code: Testing the Config object

rpncalc/rpn_v2.py

```
from rpncalc.utils import calc, Config

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

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

rpncalc/utils.py

```
class Config:
    def __init__(self, prompt=">"):
        self.prompt = prompt

    ...
```

Reverse Polish Notation (RPN)

Code: Saving and loading config objects

rpncalc/utils.py

```
class Config:
    ...

    def load(self, path: pathlib.Path) -> None:
        parser = configparser.ConfigParser()
        parser.read(path)
        self.prompt = parser["rpncalc"]["prompt"]
```

rpncalc.ini

```
[rpncalc]
prompt = rpn>
```

Loading

```
ini_path = Path(
  "rpncalc.ini")
c = Config()
c.load(ini_path)
```

Usage

```
rpn> 1
rpn> 2
rpn> +
3
```

Reverse Polish Notation (RPN)

Code: Saving and loading config objects

rpncalc/utils.py

```
class Config:
    ...
    def load(self, path: pathlib.Path) -> None:
        parser = configparser.ConfigParser()
        parser.read(path)
        self.prompt = parser["rpncalc"]["prompt"]

    def save(self, path: pathlib.Path) -> None:
        parser = configparser.ConfigParser()
        parser["rpncalc"] = {"prompt": self.prompt}
        with path.open("w") as f:
            parser.write(f)
```

Saving

```
ini_path = Path(
  "rpncalc.ini")
c = Config()
c.prompt = "calc>"
c.save(ini_path)
```

rpncalc.ini

```
[rpncalc]
prompt = calc>
```

Builtin fixtures

tmp_path

rpncalc/test_utils.py

```python
@pytest.fixture
def ini_path(tmp_path: Path) -> Path:
    return tmp_path / "rpncalc.ini"
```

```
/tmp/  (system-wide temp directory)
└── pytest-of-florian/
    ├── pytest-1/
    │   └── test_config_save0/
    │       └── rpncalc.ini
    ├── pytest-2/
    │   └── test_config_save0/
    │       └── rpncalc.ini
    └── pytest-3/
        └── ...
```

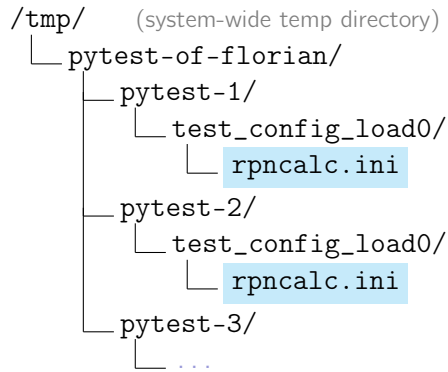
```python
def test_config_save(ini_path: Path, config: Config):
    # call config.save(...), ensure that the ini file is written correctly
    ...
```

Builtin fixtures

tmp_path

rpncalc/test_utils.py

```python
@pytest.fixture
def config_path(ini_path: Path) -> Path:
    contents = (
        "[rpncalc]\n"
        "prompt = rpn>"
    )
    ini_path.write_text(contents)
    return ini_path
```




```python
def test_config_load(config_path: Path, config: Config):
    # call config.load(...), ensure that the prompt is set to "rpn>"
    ...
```

Builtin fixtures

tmp_path

rpncalc/test_utils.py

```python
@pytest.fixture
def ini_path(tmp_path: Path) -> Path:
    return tmp_path / "rpncalc.ini"

@pytest.fixture
def config_path(ini_path: Path) -> Path:
    contents = (
        "[rpncalc]\n"
        "prompt = rpn>"
    )
    ini_path.write_text(contents)
    return ini_path
```

Demo:

- Complete `test_config_load`: Call the `config`'s `load` method, ensure prompt was changed.
- Complete `test_config_save` too, using `ini_path` as output file, then reading and asserting on the contents.

pathlib.Path docs:
docs.python.org/3/library/pathlib.html

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] == 6 , rpn.stack
```

Output:

```
E    AssertionError: [2.0, 4.0]
E    assert 4.0 == 6
```

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

```
(Pdb) list
...
13      elif op == "/":
14  ->      return a / b
...

(Pdb) p a
2

(Pdb) p b
0
```

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.)
```

Mocking

You...

- Used **unittest.mock** or **pytest-mock**?
- Used a mocking lib like **responses**, **freezegun**, **time-machine**?
- Used **VCR.py**?
- Used **pytest-httpserver** or similar?
- Used **hypothesis**?

Mocking: injecting “fake” objects

“Mocking out” dependencies helps to:

- Avoid depending on external resources
- Avoid depending on (network) IO
- Avoid depending on heavy integration with other code
- Have smaller, faster tests focused on one aspect

However, mocking increases test maintenance costs.

Reverse Polish Notation (RPN)

Adding currency conversion

`requests` Simple HTTP client library for Python

`exchangeit.app` Free currency conversion API, no API key needed (thanks!)


```
>>> import requests
>>> url = "https://api.exchangeit.app/v1/currencies/eur/latest"
>>> res = requests.get(url, params={"for": ["chf"]})
>>> res.json()
{
    "copyright": { ... },
    "data": {
        "alias": "eur", "title": "Euro",
        "rates": [{"alias": "chf", "rate": 0.9771274994, ...}]
    }
}
```

exChange it

Reverse Polish Notation (RPN)

Adding currency conversion

In `code/`, using `python -m rpncalc.rpn_v3`:

```
> 100
> eur2chf
Fetching exchange rates...
98.05
> 10
> chf2eur
10.20
> q
```

Demo: Run the calculator, exchange something between EUR/CHF. Take a look at `rpn_v3.py` and `convert.py`, the code will be explained on the next slides.

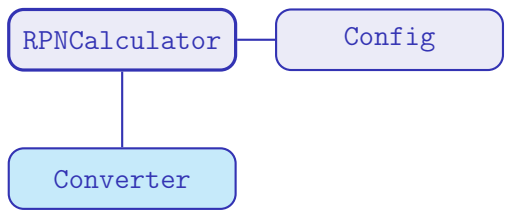
Reverse Polish Notation (RPN)

Code changes

rpncalc/rpn_v3.py

```
from rpncalc.utils import calc, Config
from rpncalc.convert import Converter


class RPNCalculator:
    def __init__(self, config):
        self.converter = Converter()
        self.config = config
        self.stack = []
```



Reverse Polish Notation (RPN)

Code changes

rpncalc/rpn_v3.py

```
class RPNCalculator:
    ...
    def evaluate(self, inp: str) -> None:
        try:
            self.stack.append(float(inp))
            return
        except ValueError:
            pass

        if inp in ["eur2chf", "chf2eur"]:
            self._evaluate_convert(inp)
            return

        ...
```

Reverse Polish Notation (RPN)

Code changes

```
    def _evaluate_convert(self, inp: str) -> None:
        try:
            amount = self.stack.pop()
        except IndexError:
            print("Not enough operands")
            return

        if inp == "eur2chf":
            res = self.converter.eur2chf(amount)
        elif inp == "chf2eur":
            res = self.converter.chf2eur(amount)

        self.stack.append(res)
        print(f"{res:.2f}")
```

Reverse Polish Notation (RPN)

Code changes

```
    def _evaluate_convert(self, inp: str) -> None:
        try:
            amount = self.stack.pop()
        except IndexError:
            print("Not enough operands")
            return

        if inp == "eur2chf":
            res = self.converter.eur2chf(amount)
        elif inp == "chf2eur":
            res = self.converter.chf2eur(amount)

        self.stack.append(res)
        print(f"{res:.2f}")
```

Reverse Polish Notation (RPN)

Code changes

```
    def _evaluate_convert(self, inp: str) -> None:
        try:
            amount = self.stack.pop()
        except IndexError:
            print("Not enough operands")
            return

        if inp == "eur2chf":
            res = self.converter.eur2chf(amount)
        elif inp == "chf2eur":
            res = self.converter.chf2eur(amount)

        self.stack.append(res)
        print(f"{res:.2f}")
```

Reverse Polish Notation (RPN)

Code changes

rpncalc/convert.py

```
class Converter:
    ...
    def eur2chf(self, amount: float) -> float:
        eur2chf_rate = self._fetch()
        return amount * eur2chf_rate

    def chf2eur(self, amount: float) -> float:
        eur2chf_rate = self._fetch()
        return amount / eur2chf_rate

    @cache
    def _fetch(self) -> float:
        # Download and return rate, cached after first download
```

Testing the v3 calculator

Functional test

A functional test might look like this:

mocking/test_real.py

```python
@pytest.fixture
def rpn() -> RPNCalculator:
    return RPNCalculator(Config())

def test_convert(rpn: RPNCalculator):
    rpn.stack = [10]
    rpn.evaluate("eur2chf")
    rpn.evaluate("chf2eur")
    assert rpn.stack[-1] == 10
```

Testing the v3 calculator

Functional test

A functional test might look like this:

mocking/test_real.py

```python
@pytest.fixture
def rpn() -> RPNCalculator:
    return RPNCalculator(Config())

def test_convert(rpn: RPNCalculator):
    rpn.stack = [10]
    rpn.evaluate("eur2chf")
    rpn.evaluate("chf2eur")
    assert rpn.stack[-1] == 10
```

Observations:

+ Testing the real API, like a user would
+ Fixture is very simple
– Depends on reachability of remote HTTP API server
– Somewhat slow, depending on network
– Hard to test actual values

Testing the v3 calculator

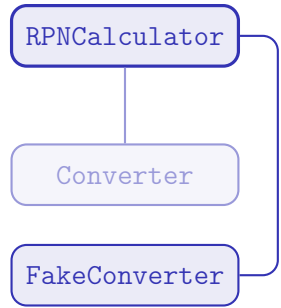
Fake class

mocking/test_fake.py

```
class FakeConverter:
    RATE = 2

    def eur2chf(self, amount: float) -> float:
        return amount * self.RATE

    def chf2eur(self, amount: float) -> float:
        return amount / self.RATE
```



Testing the v3 calculator

Fixture and test

rpncalc/rpn_v3.py

```
class RPNCalculator:
    def __init__(self, config):
        self.converter = Converter()
        self.config = config
        self.stack = []
```

mocking/test_fake.py

```
@pytest.fixture
def rpn(monkeypatch):
    calc = RPNCalculator(Config())
    monkeypatch.setattr(
        calc,
        "converter",
        FakeConverter()
    )
    return calc

def test_convert(rpn):
    rpn.stack = [10]
    rpn.evaluate("eur2chf")
    assert rpn.stack == [20]
```

Testing the v3 calculator

Result

Observations:

+ Faster
+ No network dependency
+ Test is more minimal: Can test EUR ↔ CHF independently, with real values
– More complexity in fixture
– Patching requires care: If we e.g. change `self.converter` to `self._converter`, test code needs changes too.

How can we do better?

Testing the v3 calculator

Result

Observations:

+ Faster
+ No network dependency
+ Test is more minimal: Can test EUR ↔ CHF independently, with real values
– More complexity in fixture
– Patching requires care: If we e.g. change `self.converter` to `self._converter`, test code needs changes too.

How can we do better? Dependency injection!

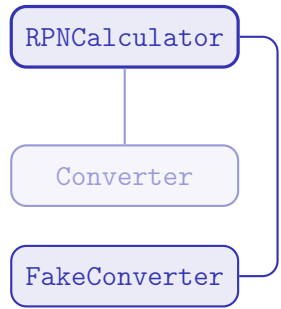
Demo:

- Adjust `RPNCalculator` in `rpncalc/rpn_v3.py`, so that it takes a `converter` argument and stores it in `self.converter`
- Adjust the `if __name__ == "__main__":` block, to create and pass a `Converter()`, like with `Config()` (use keyword arguments?)
- Make sure running the calculator still works (`python -m rpncalc.rpn_v3`, see slide 86)
- Adjust the `rpn` fixture in `mocking/test_fake.py` to pass `FakeConverter()` as argument instead of using `monkeypatch`

Testing the v3 calculator

Ensuring correctness

How do we ensure our `FakeConverter` behaves like the real one?



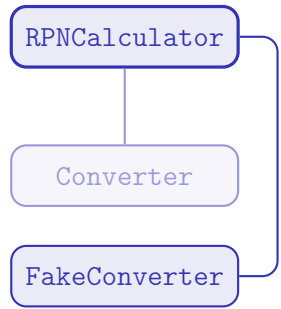
Testing the v3 calculator

Ensuring correctness

How do we ensure our `FakeConverter` behaves like the real one?

Various options:

- Use a library like `unittest.mock` with `spec` or `autospec=True` (weak guarantee)
- Use a type checker like `mypy` to ensure the interface matches (stronger guarantee)
- Periodically run tests with the real converter instead (strong guarantee)



Testing the v3 calculator

Using unittest.mock instead

mocking/test_fake_mock.py

```python
@pytest.fixture
def rpn(mocker: pytest_mock.MockerFixture) -> RPNCalculator:
    mock = mocker.Mock(spec=Converter)
    mock.eur2chf.return_value = 20
    mock.chf2eur.return_value = 5
    return RPNCalculator(config=Config(), converter=mock)


def test_convert(rpn: RPNCalculator):
    rpn.stack = [10]
    rpn.evaluate("eur2chf")
    assert rpn.stack == [20]

    rpn.converter.eur2chf.assert_called_once_with(10)
```

Testing the v3 calculator

Using unittest.mock instead

mocking/test_fake_mock.py

```python
@pytest.fixture
def rpn(mocker: pytest_mock.MockerFixture) -> RPNCalculator:
    mock = mocker.Mock(spec=Converter)
    mock.eur2chf.return_value = 20
    mock.chf2eur.return_value = 5
    return RPNCalculator(config=Config(), converter=mock)


def test_convert(rpn: RPNCalculator):
    rpn.stack = [10]
    rpn.evaluate("eur2chf")
    assert rpn.stack == [20]

    rpn.converter.eur2chf.assert_called_once_with(10)
```

Testing the v3 calculator

Using unittest.mock instead

mocking/test_fake_mock.py

```python
@pytest.fixture
def rpn(mocker: pytest_mock.MockerFixture) -> RPNCalculator:
    mock = mocker.Mock(spec=Converter)
    mock.eur2chf.return_value = 20
    mock.chf2eur.return_value = 5
    return RPNCalculator(config=Config(), converter=mock)


def test_convert(rpn: RPNCalculator):
    rpn.stack = [10]
    rpn.evaluate("eur2chf")
    assert rpn.stack == [20]

    rpn.converter.eur2chf.assert_called_once_with(10)
```

Testing the v3 calculator

Using unittest.mock instead

mocking/test_fake_mock.py

```python
@pytest.fixture
def rpn(mocker: pytest_mock.MockerFixture) -> RPNCalculator:
    mock = mocker.Mock(spec=Converter)
    mock.eur2chf.return_value = 20
    mock.chf2eur.return_value = 5
    return RPNCalculator(config=Config(), converter=mock)


def test_convert(rpn: RPNCalculator):
    rpn.stack = [10]
    rpn.evaluate("eur2chf")
    assert rpn.stack == [20]

    rpn.converter.eur2chf.assert_called_once_with(10)
```

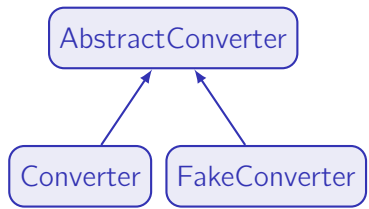
Testing the v3 calculator

Using type checking

```
class AbstractConverter:
    def eur2chf(self, amount: float) -> float:
        raise NotImplementedError

    def chf2eur(self, amount: float) -> float:
        raise NotImplementedError

    def _fetch(self) -> float:
        raise NotImplementedError
```



Testing the v3 calculator

Using type checking

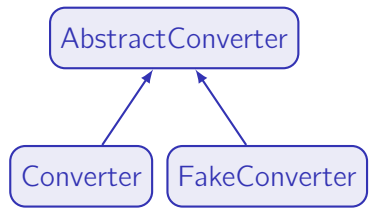
```
class AbstractConverter:
    def eur2chf(self, amount: float) -> float:
        raise NotImplementedError

    def chf2eur(self, amount: float) -> float:
        raise NotImplementedError

    def _fetch(self) -> float:
        raise NotImplementedError
```

rpncalc/convert.py

```
class Converter(AbstractConverter):
    ...
```



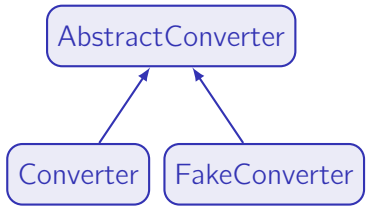
Testing the v3 calculator

Using type checking

```
class AbstractConverter:
    def eur2chf(self, amount: float) -> float:
        raise NotImplementedError

    def chf2eur(self, amount: float) -> float:
        raise NotImplementedError

    def _fetch(self) -> float:
        raise NotImplementedError
```



rpncalc/convert.py

```
class Converter(AbstractConverter):
    ...
```

tests/test_fake.py

```
class FakeConverter(AbstractConverter):
    ...
```

Testing the v3 calculator

Testing against real API

conftest.py

```python
def pytest_addoption(parser: pytest.Parser) -> None:
    parser.addoption("--real-api", action="store_true")


@pytest.fixture
def rpn(request: pytest.FixtureRequest) -> RPNCalculator:
    if request.config.option.real_api:
        converter = Converter()
    else:
        converter = FakeConverter()
    return RPNCalculator(config=Config(), converter=converter)
```

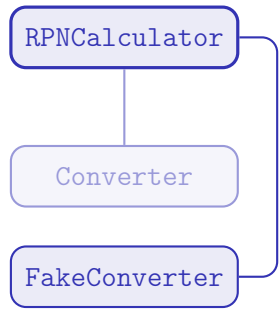
`$ pytest` ⇒ `FakeConverter()` `$ pytest --real-api` ⇒ real `Converter()`

Testing the v3 calculator

Where we are, what's next

We've found various strategies to test the `RPNCalculator` changes:

- Using `monkeypatch` with a hand-written fake class
- Using dependency injection, avoiding need to patch
- Using `unittest.mock` instead via `pytest-mock`
- Ensuring correctness on a type level with an abstract class and e.g. `mypy`
- Ensuring correctness by periodically running tests against the real API



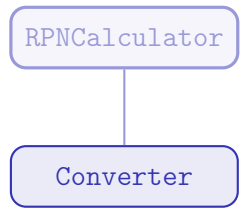
Testing the v3 calculator

Where we are, what's next

We still need to test the `Converter` class itself.

Next up:

- Naive approach: Using `monkeypatch` to patch `requests.get`
- Using `unittest.mock` — still low-level!
- Using mocking libraries: `responses`, `VCR.py`
- Back to functional tests: `pytest-httpserver`



Reverse Polish Notation (RPN)

The Converter class

rpncalc/convert.py

```
class Converter:
    ...
    def eur2chf(self, amount: float) -> float:
        eur2chf_rate = self._fetch()
        return amount * eur2chf_rate

    def chf2eur(self, amount: float) -> float:
        eur2chf_rate = self._fetch()
        return amount / eur2chf_rate

    @cache
    def _fetch(self) -> float:
        ...
```

Reverse Polish Notation (RPN)

The Converter class

rpncalc/convert.py

```
class Converter:
    API_URL = "https://api.exchangeit.app/v1/currencies/eur/latest"
    HEADERS = {"User-Agent": "rpncalc/0.1 (florian@bruhin.software)"}
    PARAMS = {"for": "chf"}
    ...
    @cache
    def _fetch(self) -> float:
        print("Fetching exchange rates...")
        response = requests.get(
            self.API_URL,
            params=self.PARAMS,
            headers=self.HEADERS,
        )
        ...
```

Reverse Polish Notation (RPN)

The Converter class

rpncalc/convert.py

```python
class Converter:
    ...
    @cache
    def _fetch(self) -> float:
        ...
        response = requests.get(...)
        response.raise_for_status()
        d = response.json()
        rates = d["data"]["rates"]
        return rates[0]["rate"]
```

Reverse Polish Notation (RPN)

The Converter class

rpncalc/convert.py

```python
class Converter:
    ...
    @cache
    def _fetch(self) -> float:
        ...
        response = requests.get(...)
        response.raise_for_status()
        d = response.json()
        rates = d["data"]["rates"]
        return rates[0]["rate"]
```

```
{
  ...
  "data": {
    "alias": "eur",
    "title": "Euro",
    "rates": [
      {
        "alias": "chf",
        "rate": 0.977...,
        ...
      },
    ]
  }
}
```

Testing the Converter

Reducing complexity: unittest.mock?

mocking/converter/conftest.py

```python
@pytest.fixture
def exchange_data() -> dict:
    rates = [{"alias": "chf", "rate": 2}]
    return {"data": {"alias": "eur", "rates": rates}}
```

mocking/converter/test_mock.py

```python
@pytest.fixture(autouse=True)
def mock_requests_get(
    mocker: pytest_mock.MockerFixture,
    exchange_data: dict,
):
    ...
```

Testing the Converter

Reducing complexity: unittest.mock?

mocking/converter/test_mock.py

```python
@pytest.fixture(autouse=True)
def mock_requests_get(
    mocker: pytest_mock.MockerFixture,
    exchange_data: dict,
):
    mock_get = mocker.patch.object(requests, "get", autospec=True)
    mock_get(Converter.API_URL).json.return_value = exchange_data
    yield mock_get
    mock_get.assert_called_with(
        Converter.API_URL,
        params=Converter.PARAMS,
        headers=Converter.HEADERS
    )
```

Testing the Converter

Reducing complexity: unittest.mock?

mocking/converter/test_mock.py

```python
@pytest.fixture(autouse=True)
def mock_requests_get(
    mocker: pytest_mock.MockerFixture,
    exchange_data: dict,
):
    mock_get = mocker.patch.object(requests, "get", autospec=True)
    mock_get(Converter.API_URL).json.return_value = exchange_data
    yield mock_get
    mock_get.assert_called_with(
        Converter.API_URL,
        params=Converter.PARAMS,
        headers=Converter.HEADERS
    )
```

Testing the Converter

Reducing complexity: unittest.mock?

mocking/converter/test_mock.py

```python
@pytest.fixture(autouse=True)
def mock_requests_get(
    mocker: pytest_mock.MockerFixture,
    exchange_data: dict,
):
    mock_get = mocker.patch.object(requests, "get", autospec=True)
    mock_get(Converter.API_URL).json.return_value = exchange_data
    yield mock_get
    mock_get.assert_called_with(
        Converter.API_URL,
        params=Converter.PARAMS,
        headers=Converter.HEADERS
    )
```

Testing the Converter

Reducing complexity: unittest.mock?

mocking/converter/test_mock.py

```python
@pytest.fixture(autouse=True)
def mock_requests_get(
    mocker: pytest_mock.MockerFixture,
    exchange_data: dict,
):
    mock_get = mocker.patch.object(requests, "get", autospec=True)
    mock_get(Converter.API_URL).json.return_value = exchange_data
    yield mock_get
    mock_get.assert_called_with(
        Converter.API_URL,
        params=Converter.PARAMS,
        headers=Converter.HEADERS
    )
```

Testing the Converter

Reducing complexity: unittest.mock?

Observations:

+ No need to hand-roll a `FakeResponse`
+ More safety due to `autospec=True`

Testing the Converter

Reducing complexity: unittest.mock?

Observations:

+ No need to hand-roll a `FakeResponse`
+ More safety due to `autospec=True`

– Tedious; “weird” `unittest.mock` API
– Still needs to know about implementation

Maybe there's no need to hand-roll mocks?

Ready-made mocking libraries

requests	**responses**, betamax, requests-mock
aiohttp	aioresponses
HTTP in general	**VCR.py**, HTTPretty, pook
sockets in general	mocket
time/datetime	freezegun, time-machine

For most of those, there's a corresponding pytest plugin for nice integration.

Maybe there's no need to hand-roll mocks?

Ready-made mocking libraries

requests	**responses**, betamax, requests-mock
aiohttp	aioresponses
HTTP in general	**VCR.py**, HTTPretty, pook
sockets in general	mocket
time/datetime	freezegun, time-machine

For most of those, there's a corresponding pytest plugin for nice integration.

Result: More high-level abstraction for mocking
(e.g. HTTP responses instead of the Python API).

Well-tested mocks, usually covering the entire API of a library.

Not many high-level mocking libraries exist (wish more projects would do this!)
But if one does, **use it**!

Testing the Converter

High-level mocking with responses

mocking/converter/test_responses.py

```python
@pytest.fixture(autouse=True)
def patch_requests_get(
    responses: RequestsMock, exchange_data: dict
) -> None:
    responses.get(
        Converter.API_URL,
        json=exchange_data,
        match=[
            matchers.query_param_matcher(Converter.PARAMS),
            matchers.header_matcher(Converter.HEADERS),
        ],
    )
```

Testing the Converter

High-level mocking with responses

mocking/converter/test_responses.py

```python
@pytest.fixture(autouse=True)
def patch_requests_get(
    responses: RequestsMock, exchange_data: dict
) -> None:
    responses.get(
        Converter.API_URL,
        json=exchange_data,
        match=[
            matchers.query_param_matcher(Converter.PARAMS),
            matchers.header_matcher(Converter.HEADERS),
        ],
    )
```

Testing the Converter

High-level mocking with responses

mocking/converter/test_responses.py

```python
@pytest.fixture(autouse=True)
def patch_requests_get(
    responses: RequestsMock, exchange_data: dict
) -> None:
    responses.get(
        Converter.API_URL,
        json=exchange_data,
        match=[
            matchers.query_param_matcher(Converter.PARAMS),
            matchers.header_matcher(Converter.HEADERS),
        ],
    )
```

Testing the Converter

High-level mocking with responses

mocking/converter/test_responses.py

```python
@pytest.fixture(autouse=True)
def patch_requests_get(
    responses: RequestsMock, exchange_data: dict
) -> None:
    responses.get(
        Converter.API_URL,
        json=exchange_data,
        match=[
            matchers.query_param_matcher(Converter.PARAMS),
            matchers.header_matcher(Converter.HEADERS),
        ],
    )
```

Testing the Converter

High-level mocking with responses

```
responses.get(
    Converter.API_URL,
    json={ ... },
    match=...,
)
```

Observations:

+ Much more high-level API
+ Easy to use, easy to validate requests
+ Covers the entire API of `requests`

Testing the Converter

High-level mocking with responses

```
responses.get(
    Converter.API_URL,
    json={ ... },
    match=...,
)
```

Observations:

+ Much more high-level API
+ Easy to use, easy to validate requests
+ Covers the entire API of `requests`
± We still need to provide the data manually
  + Possibility to test corner cases
  – Needs care so data is same as real API

Testing the Converter

High(er)-level mocking with VCR.py

mocking/converter/test_vcr.py

```python
@pytest.mark.vcr
def test_eur2chf(converter: Converter):
    assert converter.eur2chf(1) == pytest.approx(0.98, rel=1e-2)


@pytest.mark.vcr
def test_chf2eur(converter: Converter):
    assert converter.chf2eur(1) == pytest.approx(1.02, rel=1e-2)
```

Note: The `pytest-recording` plugin to integrate VCR.py is disabled in this training, as it runs many fixtures with `autouse=True`, interfering with some exercises.

If you want to try it, remove `-p no:recording` from `addopts` in `pytest.ini`.

Testing the Converter

High(er)-level mocking with VCR.py

mocking/converter/test_vcr.py

```python
@pytest.mark.vcr
def test_eur2chf(converter: Converter):
    assert converter.eur2chf(1) == pytest.approx(0.98, rel=1e-2)


@pytest.mark.vcr
def test_chf2eur(converter: Converter):
    assert converter.chf2eur(1) == pytest.approx(1.02, rel=1e-2)
```

Note: The `pytest-recording` plugin to integrate VCR.py is disabled in this training, as it runs many fixtures with `autouse=True`, interfering with some exercises.

If you want to try it, remove `-p no:recording` from `addopts` in `pytest.ini`.

Testing the Converter

High(er)-level mocking with VCR.py

mocking/converter/test_vcr.py

```python
@pytest.mark.vcr
def test_eur2chf(converter: Converter):
    assert converter.eur2chf(1) == pytest.approx(0.98, rel=1e-2)
```

```
$ pytest test_vcr.py
...
vcr.errors.CannotOverwriteExistingCassetteException: Can't overwrite
    existing cassette ('.../converter/cassettes/test_vcr/
    test_eur2chf.yaml') in your current record mode ('none').
No match for the request (<Request (GET) https://api.exchangeit.app/
    v1/currencies/eur/latest?for=chf>) was found.
No similar requests, that have not been played, found.
```

Testing the Converter

High(er)-level mocking with VCR.py

```
$ pytest --record-mode=once test_vcr.py
...

converter/test_vcr.py ..

========== 2 passed in 0.86s ===========
```

Testing the Converter

High(er)-level mocking with VCR.py

```
$ pytest --record-mode=once test_vcr.py
...

converter/test_vcr.py ..

========== 2 passed in 0.86s ===========
```

```
code/mocking/converter/
├── ...
├── test_vcr.py
└── cassettes/
    └── test_vcr/
        ├── test_eur2chf.yaml
        └── test_chf2eur.yaml
```

Testing the Converter

High(er)-level mocking with VCR.py

```
$ pytest --record-mode=once test_vcr.py
...

converter/test_vcr.py ..

========== 2 passed in 0.86s ===========

$ pytest test_vcr.py
...

converter/test_vcr.py ..

========== 2 passed in 0.05s ===========
```

```
code/mocking/converter/
├── ...
├── test_vcr.py
└── cassettes/
    └── test_vcr/
        ├── test_eur2chf.yaml
        └── test_chf2eur.yaml
```

To re-record existing cassettes: `--record-mode=all`.

Testing the Converter

High(er)-level mocking with VCR.py

```
interactions:
- request:
    uri: https://api.exchangeit.app/v1/currencies/eur/latest?for=chf
    method: GET
    headers:
      ...
      User-Agent:
      - rpncalc/0.1 (florian@bruhin.software)
    body: null
  response:
    body:
      string: '{"data":{"rates":[{"alias":"chf","rate":0.9771274}]}}' *
    headers: ...
    status: {"code": 200, "message": "OK"}
version: 1
```

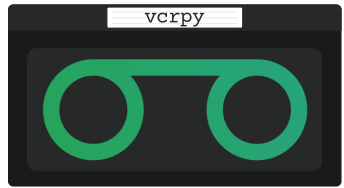
* (simplified)

Testing the Converter

High(er)-level mocking with VCR.py

Observations:

+ Even higher-level: No need for explicit mocking
+ Guaranteed to have a 1:1 recording of server response
+ Covers most Python HTTP APIs!
+ Easy to update cassettes if API changes server-side



Testing the Converter

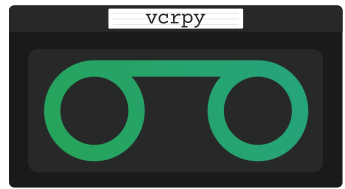
High(er)-level mocking with VCR.py

Observations:

+ Even higher-level: No need for explicit mocking
+ Guaranteed to have a 1:1 recording of server response
+ Covers most Python HTTP APIs!
+ Easy to update cassettes if API changes server-side

– Much harder to test corner cases
(though we can write manual cassettes)

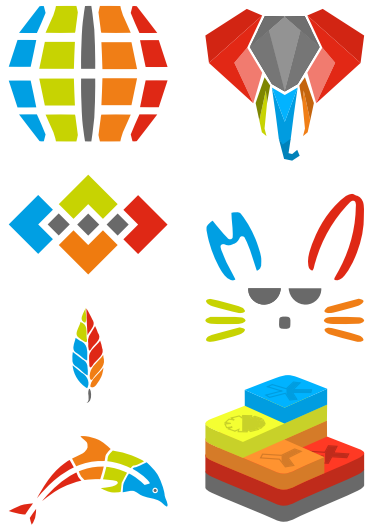
– Tests are less expressive for documentation



Maybe there's no need to mock?

Consider using the real thing

Databases	sqlite, pytest-postgresql, pytest-mongo, pytest-mysql, pytest-dynamodb
HTTP servers	**pytest-httpserver**, pytest-localserver; Flask and cheroot
Others	pytest-rabbitmq, pytest-solr, pytest-elasticsearch, pytest-redis, pytest-grpc, pytest-localserver (SMTP)
Generic	pytest-xprocess, pytest-docker



Different approaches to mocking

- Hand-written stubs with `monkeypatch`
  - Possibly using an abstract base class and `mypy`
  - Possibly using a `--real-thing` option to cross-check
- Automated mocks with `unittest.mock` (via `pytest-mock`)
- Write code in a flexible enough way so you don't need to patch
- Use a pre-made mocking library (e.g. `responses`, `VCR.py`)
- Can you easily use the real thing instead? (e.g. `pytest-httpserver`)

Some plugins

Using the coverage plugin

Install plugin: `pip install pytest-cov`

Options:

`--cov=`*`path`* filesystem path to generate coverage for

`--cov-report=`*`type`* type of report (“term”, “html”, . . .)

`--cov-config=`*`path`* path of `.coveragerc` file

Demo:

- Run `pytest --cov=rpncalc/ --cov-report=term-missing rpncalc/` in the `code/` folder, take a look at the terminal output.
- Rerun with `--cov-report=html`, then open `htmlcov/index.html`, look at the report

pytest-bdd: Behavior driven testing

Gherkin (Cucumber)

article.feature

```
Scenario: Publishing the article
  Given I'm an author user
  And I have an article

  When I go to the article page
  And I press the publish button

  Then no error should be shown
  And the article should be published
```

pytest-bdd: Behavior driven testing

Gherkin (Cucumber)

article.feature

```
Scenario: Publishing the article
  Given I'm an author user
  And I have an article

  When I go to the article page
  And I press the publish button

  Then no error should be shown
  And the article should be published
```

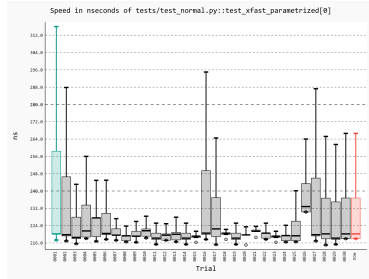
test_article.py

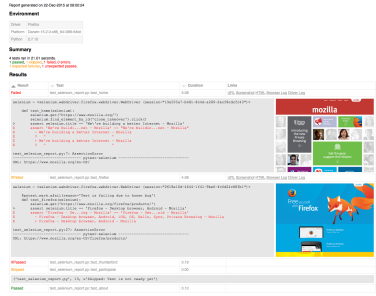
```python
@when("I go to the article page")
def go_to_article(article, browser):
    browser.visit(article.url())

@when("I press the publish button")
def publish_article(browser):
    browser.find_by_id(...).click()
```

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





Property-based testing with Hypothesis

Motivation

From the Hypothesis website, `hypothesis.works`:

> *Most testing is ineffective*
>
> *Normal "automated" software testing is surprisingly manual.*
> *Every scenario the computer runs, someone had to write by hand.*
> *Hypothesis can fix this.*

Property-based testing with Hypothesis

Motivation

From the Hypothesis website, `hypothesis.works`:

> *Most testing is ineffective*
>
> *Normal "automated" software testing is surprisingly manual.*
> *Every scenario the computer runs, someone had to write by hand.*
> *Hypothesis can fix this.*

Idea behind "Property-Based testing" (popularized by QuickCheck/Haskell):

- Generate input data based on a *strategy* (how should the data look?)
- Run the test case many times (say, 100), with different random generated data
- Check for *properties* / *invariants* that should hold for all input data
- If there is a failure, *minimize* the input data to a minimal failing example

Property-based testing with Hypothesis

Old buggy RPN calculator

rpncalc/rpn_v1.py

```
class RPNCalculator:
    ...

    def evaluate(self, inp: str):
        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)
```

```
> 1
> +
Traceback (most recent call last):
  ...
  File ..., in <module>
    rpn.run()
  File ..., in run
    self.evaluate(inp)
  File ..., in evaluate
    a = self.stack.pop()
IndexError: pop from empty list
```

Property-based testing with Hypothesis

Old buggy RPN calculator: Hypothesis test

plugins/test_hypothesis_rpncalc_v1.py

```
from hypothesis import given, strategies as st
from rpncalc.rpn_v1 import RPNCalculator

@given(st.text())
def test_random_strings(s):
    rpn = RPNCalculator()
    rpn.evaluate(s)
```

Property-based testing with Hypothesis

Old buggy RPN calculator: Hypothesis test

plugins/test_hypothesis_rpncalc_v1.py

```python
from hypothesis import given, strategies as st
from rpncalc.rpn_v1 import RPNCalculator

@given(st.text())
def test_random_strings(s):
    rpn = RPNCalculator()
    rpn.evaluate(s)
```

```
>   b = self.stack.pop()
E   IndexError: pop from empty list
E   Falsifying example: test_random_strings(
E       s='',
E   )
```

Property-based testing with Hypothesis

Old buggy RPN calculator: Hypothesis test

plugins/test_hypothesis_rpncalc_v1.py

```python
from hypothesis import given, strategies as st
from rpncalc.rpn_v1 import RPNCalculator

@given(st.text())
def test_random_strings(s):
    rpn = RPNCalculator()
    rpn.evaluate(s)
```

```
>   b = self.stack.pop()
E   IndexError: pop from empty list
E   Falsifying example: test_random_strings(
E       s='',
E   )
```

```python
elif inp in "+-*/":
    ...
```

```
>>> "" in "+-*/"
True
```

Property-based testing with Hypothesis

Surely v3 is bug-free?

rpncalc/rpn_v3.py

```
class RPNCalculator:
    ...

    def evaluate(self, inp):
        ...
        if len(self.stack) < 2:
            print("Not enough operands")
            return

        b = self.stack.pop()
        a = self.stack.pop()
```

Property-based testing with Hypothesis

Surely v3 is bug-free?

rpncalc/rpn_v3.py

```
class RPNCalculator:
    ...

    def evaluate(self, inp):
        ...
        if len(self.stack) < 2:
            print("Not enough operands")
            return

        b = self.stack.pop()
        a = self.stack.pop()
```

```
test_random_strings PASSED
```

Property-based testing with Hypothesis

Surely v3 is bug-free?

plugins/test_hypothesis_rpncalc_v3.py

```python
@given(st.integers(), st.integers())
def test_operators(n1, n2):
    rpn = RPNCalculator(Config())
    rpn.evaluate(str(n1))
    rpn.evaluate(str(n2))
    rpn.evaluate("+")
    assert rpn.stack == [n1 + n2]
```

Property-based testing with Hypothesis

Surely v3 is bug-free?

plugins/test_hypothesis_rpncalc_v3.py

```
@given(st.integers(), st.integers())
def test_operators(n1, n2):
    rpn = RPNCalculator(Config())
    rpn.evaluate(str(n1))
    rpn.evaluate(str(n2))
    rpn.evaluate("+")
    assert rpn.stack == [n1 + n2]
E   assert [1.055531162664984 8 e+16] == [1055531162664984 9 ]
E   ...
E   Falsifying example: test_operators(
E       n1=0,
E       n2=10555311626649849,
E   )
```

Property-based testing with Hypothesis

Surely v3 is bug-free?

plugins/test_hypothesis_rpncalc_v3.py

```
@given(st.integers(), st.integers())
def test_operators(n1, n2):
    rpn = RPNCalculator(Config())
    rpn.evaluate(str(n1))
    rpn.evaluate(str(n2))
    rpn.evaluate("+")
    assert rpn.stack == [n1 + n2]
E   assert [1.055531162664984 8 e+16] == [1055531162664984 9 ]
E   ...
E   Falsifying example: test_operators(
E       n1=0,
E       n2=10555311626649849,
E   )
```

```
> 10555311626649849
> 0
> +
1.055531162664984 8 e+16
```

Property-based testing with Hypothesis

Surely v3 is bug-free?

plugins/test_hypothesis_rpncalc_v3.py

```
@given(st.integers(), st.integers())
def test_operators(n1, n2):
    rpn = RPNCalculator(Config())
    rpn.evaluate(str(n1))
    rpn.evaluate(str(n2))
    rpn.evaluate("+")
    assert rpn.stack == [n1 + n2]
E   assert [1.0555311626649848e+16] == [10555311626649849]
E   ...
E   Falsifying example: test_operators(
E       n1=0,
E       n2=10555311626649849,
E   )
```

```
> 10555311626649849
> 0
> +
1.0555311626649848e+16
```

rpncalc/rpn_v3.py

```
...
self.stack.append(float(inp))
```

Property-based testing with Hypothesis

More sophisticated strategies

```
@given(
    st.lists(
        st.one_of(
            st.integers().map(str),
            st.floats().map(str),
            st.just("+"), st.just("-"),
            st.just("*"), st.just("/"),
        )
    )
)
def test_usage(inputs):
    rpn = RPNCalculator(Config())
    for inp in inputs:
        rpn.evaluate(inp)
```

Property-based testing with Hypothesis

More sophisticated strategies

```python
@given(
    st.lists(
        st.one_of(
            st.integers().map(str),
            st.floats().map(str),
            st.just("+"), st.just("-"),
            st.just("*"), st.just("/"),
        )
    )
)
def test_usage(inputs):
    rpn = RPNCalculator(Config())
    for inp in inputs:
        rpn.evaluate(inp)
```

```
Trying example: test_usage(
  inputs=[
    '-2.2250738585e-313',
    '/', '-10', '110', '+', '+'])
Not enough operands
100.0
100.0
```

Property-based testing with Hypothesis

More sophisticated strategies

```python
@given(
    st.lists(
        st.one_of(
            st.integers().map(str),
            st.floats().map(str),
            st.just("+"), st.just("-"),
            st.just("*"), st.just("/"),
        )
    )
)
def test_usage(inputs):
    rpn = RPNCalculator(Config())
    for inp in inputs:
        rpn.evaluate(inp)
```

```
Trying example: test_usage(
  inputs=[
    '-2.2250738585e-313',
    '/', '-10', '110', '+', '+'])
Not enough operands
100.0
100.0

Trying example: test_usage(
  inputs=['+', '+', '-', '*'],
)
Not enough operands
Not enough operands
Not enough operands
Not enough operands
```

Property-based testing with Hypothesis

What it can do

- Generate data based on types/strategies
- Combine/filter various strategies to generate more sophisticated data
- Generate names, etc. via fakefactory
- Generate data matching a Django model
- Generate data matching a grammar
- Generate test code (rather than data) based on a state machine

Writing your own plugins

Writing pytest plugins

Plugins come in two flavours:

- Local plugins: `conftest.py` files
- Installable plugins via `setuptools` entry points

They can contain fixtures, plus hook implementations for:

- configuration
- collection
- test running
- reporting

A hook is auto-discovered by its `pytest_···` name.

Hook mechanism

All hook implementations have a `pytest_` prefix.
The argument names **must** match the precise signature names:

ok:

```
def pytest_configure(config):
    ...
```

You can accept fewer parameters:

```
def pytest_configure():
    ...
```

mismatch:

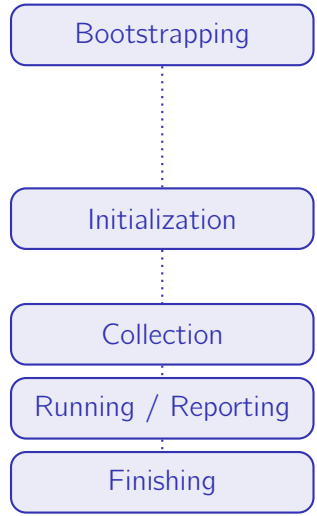
```
def pytest_configure(cfg):
    ...
```

As with fixtures, imagine pytest calling your function with keyword arguments:

```
pytest_configure(config=pytest.Config(...))
```

Anatomy of a pytest run

- `main()`
  - Import and register built-in plugins:
    `pytest_plugin_registered`
  - `pytest_addoption`
  - `pytest_cmdline_parse`
  - `pytest_cmdline_main`
    - `pytest_configure`
    - `pytest_sessionstart`
      - `pytest_report_header`
    - `pytest_collection`
      - ...
    - `pytest_runtestloop`
      - ...
    - `pytest_sessionfinish`
      - `pytest_terminal_summary`
    - `pytest_unconfigure`



The pytest_addoption hook

```
def pytest_addoption(
    parser: Parser,
    pluginmanager: PytestPluginManager,
) -> None:
    """Register argparse-style options and ini-style config values,
    called once at the beginning of a test run.

    [...]
    """
```

Configuring fixtures with command line options

Add and use command line options:

fixtures/cli_opt/conftest.py

```python
def pytest_addoption(parser: pytest.Parser) -> None:
    parser.addoption("--server-ip", type=str)


@pytest.fixture
def server_ip(request: pytest.FixtureRequest) -> str:
    return request.config.option.server_ip
```

Demo: `fixtures/cli_opt/`

- Using this `conftest.py`, writing a test using the `server_ip` fixture
- Invoking with different `--server-ip` option values

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

pytest_report_header

hooks/reporting/conftest.py

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

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

Adding to header and summary

pytest_terminal_summary

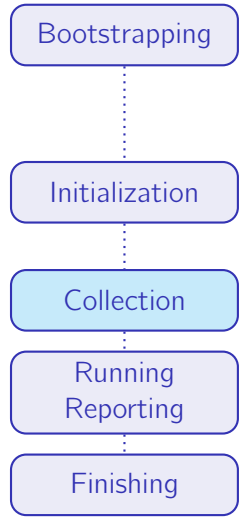
hooks/reporting/conftest.py

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

```
$ pytest -v
======================== test session starts ========================
...
------------------------ my special section ------------------------
report something here
======================= no tests ran in 0.00s =======================
```

Anatomy of a pytest run

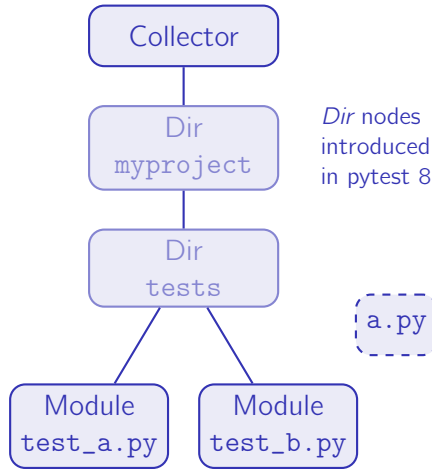
- `main()`
  - Import and register built-in plugins:
    `pytest_plugin_registered`
  - `pytest_addoption`
  - `pytest_cmdline_parse`
  - `pytest_cmdline_main`
    - `pytest_configure`
    - `pytest_sessionstart`
      - `pytest_report_header`
    - `pytest_collection`
      - `...`
    - `pytest_runtestloop`
      - `...`
    - `pytest_sessionfinish`
      - `pytest_terminal_summary`
    - `pytest_unconfigure`



Anatomy of a pytest run

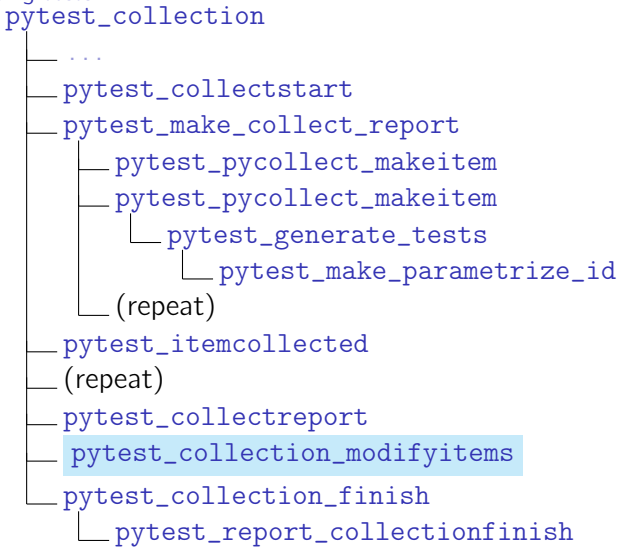
Collecting files

- pytest_collection
  - pytest_collectstart
  - pytest_make_collect_report
    - pytest_ignore_collect
    - pytest_collect_file
      - pytest_pycollect_makemodule
    - pytest_ignore_collect
    - pytest_collect_file
      - pytest_pycollect_makemodule
    - (repeat)
  - pytest_itemcollected
  - (repeat)
  - pytest_collectreport
  - ...



Anatomy of a pytest run

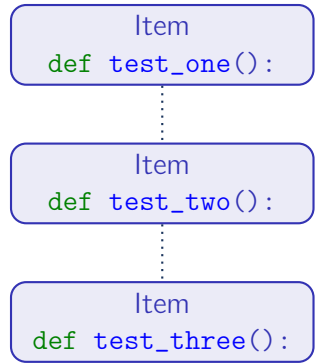
Collecting tests



Anatomy of a pytest run

Collecting tests

- pytest_collection
  - ...
  - pytest_collectstart
  - pytest_make_collect_report
    - pytest_pycollect_makeitem
    - pytest_pycollect_makeitem
      - pytest_generate_tests
        - pytest_make_parametrize_id
    - (repeat)
  - pytest_itemcollected
  - (repeat)
  - pytest_collectreport
  - pytest_collection_modifyitems
  - pytest_collection_finish
    - pytest_report_collectionfinish



The pytest_collection_modifyitems hook

```
def pytest_collection_modifyitems(
    session: pytest.Session,
    config: pytest.Config,
    items: list[pytest.Item]
) -> None:
    """Called after collection has been performed.
    May filter or re-order the items in-place.
    """
```

The pytest_collection_modifyitems hook

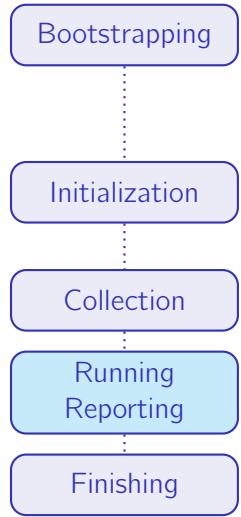
```python
def pytest_collection_modifyitems(
    session: pytest.Session,
    config: pytest.Config,
    items: list[pytest.Item]
) -> None:
    """Called after collection has been performed.
    May filter or re-order the items in-place.
    """
```

We can use the hook to e.g. add a marker to all tests using a certain fixture:

```python
def pytest_collection_modifyitems(items: list[pytest.Item]) -> None:
    for item in items:
        if "qapp" in getattr(item, "fixturenames", []):
            item.add_marker("gui")
```

Anatomy of a pytest run

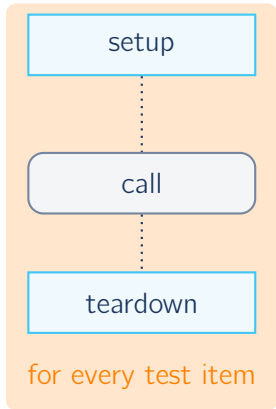
- `main()`
  - Import and register built-in plugins:
    `pytest_plugin_registered`
  - `pytest_addoption`
  - `pytest_cmdline_parse`
  - `pytest_cmdline_main`
    - `pytest_configure`
    - `pytest_sessionstart`
      - `pytest_report_header`
    - `pytest_collection`
      - ...
    - `pytest_runtestloop`
      - ...
    - `pytest_sessionfinish`
      - `pytest_terminal_summary`
    - `pytest_unconfigure`



Anatomy of a pytest run

Running tests

- pytest_runtestloop
  - pytest_runtest_protocol
    - pytest_runtest_logstart
    - pytest_runtest_setup
    - pytest_runtest_makereport
    - pytest_runtest_logreport
      - pytest_runtest_teststatus
    - pytest_runtest_call
      - pytest_pyfunc_call
    - ...(makereport, logreport)
    - pytest_runtest_teardown
    - ...(makereport, logreport)
    - pytest_runtest_logfinish
  - (repeat)



Making a plugin installable

code/hooks/yaml/pyproject.toml

```toml
[project]
name = "pytest-yamlcalc"
description = "yaml-based rpn calculator tests"
...
classifiers = ["Framework :: Pytest", ...]
dependencies = ["pytest>=7.0.0", ...]

[project.entry-points.pytest11]
yamlcalc = "pytest_yamlcalc.plugin"

[...]
...
```

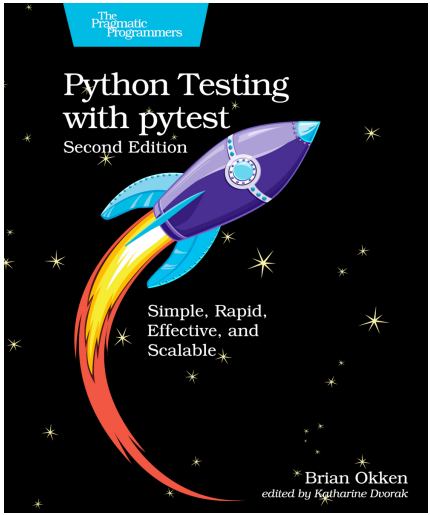
Cookiecutter makes things easier

- `pip install cookiecutter`
- `cookiecutter https://github.com/pytest-dev/cookiecutter-pytest-plugin`


```
pytest-yamlcalc/
├── .gitignore
├── .github/workflows/main.yml
├── docs
│   ├── conf.py
│   ├── index.rst
│   └── ...
├── LICENSE
├── MANIFEST.in
├── src/pytest-yamlcalc
│   ├── __init__.py
│   └── plugin.py
├── README.rst
├── pyproject.toml
├── tests
│   ├── conftest.py
│   └── test_yamlcalc.py
└── tox.ini
```

Book recommendation

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



- ISBN 978-1680508604
- https://pragprog.com/titles/bopytest2/
- Discount code: **EuroPythonPrague**
  35% discount on PDF + epub + mobi
  until July 20th
- Full disclosure: I'm technical reviewer
  (but don't earn any money from it)

Upcoming events

- **September 2nd, 2024:**
  pytest: Professionelles Testen (nicht nur) für Python
  At CH Open Workshoptage 2024
  HSLU Rotkreuz, Switzerland

- **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
bruhin.software
@the_compiler
linkedin.com/in/florian-bruhin

BRUHIN
SOFTWARE