

pytest

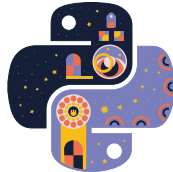
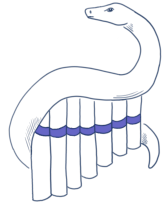
Simple, rapid and fun testing with Python



 [The-Compiler/pytest-basics](#)

Florian Bruhin

July 14th, 2025

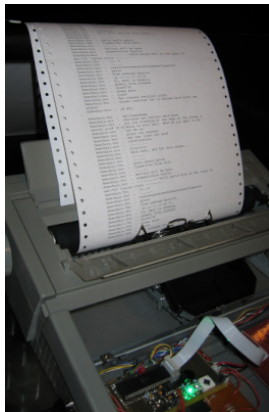


About me

 The-Compiler/pytest-basics



2011:



2013, 2015:



2019, 2020:



You:

Used **pytest** before?

Used **unittest/nose/...** before?

Used pytest **fixtures**?

Used Python **decorators**?

Used **context managers**?

Used **“yield”**?

Used **virtualenv**?

Course content (I)

- **About testing:** Why and how to write tests
- **About pytest:** Why pytest, popularity, history overview
- **The basics:** Fundamental pytest features, test discovery and plain asserts
- **Configuration:** Typical directory structure, configs, options
- **Marks:** Marking/grouping tests, skipping tests
- **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

Course content (II)

- **Debugging failing tests:** Controlling output, selecting tests, tracing fixtures, using breakpoints, showing durations, dealing with hanging and flaky tests
- **Migrating to pytest:** Running existing testsuites, incremental rewriting, tooling
- **Mocking:** Dealing with dependencies which are in our way, monkeypatch and unittest.mock, mocking libraries and helpers, alternatives
- **Plugin tour:** Coverage, distributed testing, test reporting, alternative test syntax, testing C libraries, asyncio integration, plugin overview
- **Property-based testing:** Using *hypothesis* to generate test data
- **Writing plugins:** Extending pytest via custom hooks, domain-specific languages

About testing



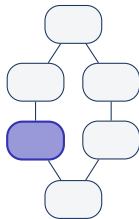
Why automated testing?

- ✓ To raise confidence that code works
- ✍ To allow for changes without fear
- 📄 To specify and document behaviour
- 👥 Collaborative and faster development cycles

Testing terminology

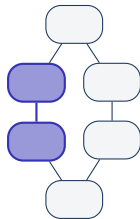
Different sizes of tests

Unit tests



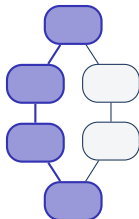
Units react well to input
(micro-tests)

Integration tests



Multiple components
co-operate nicely

Functional tests



Full code works in user
environments
(end-to-end tests,
system tests,
acceptance tests)



A test function usually has three parts:

Arrange **Set up** objects/values needed for the test
(with pytest, often done via *fixtures* instead of in the test function)

Act **Call** the function/method to be tested

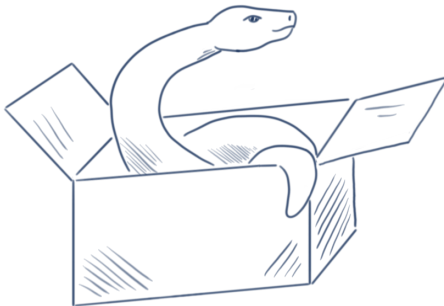
Assert **Check** the result (with pytest, using the Python `assert` statement)

Act/assert might be repeated, but it's usually good practice to only test *one* thing per test function.







Setup



Setup overview

-  Download slides and example code for exercises:
 [The-Compiler/pytest-basics](https://github.com/The-Compiler/pytest-basics)
- We'll use Python 3.9 or newer, with pytest 8.3 (≥ 7.0 is okay).
 - Use `python3 --version` or `py -3 --version` (Windows) to check your version.
- You can use whatever editor/IDE you'd like – if you don't use one yet, PyCharm (Community Edition) or VS Code are good choices.
- However, we'll first start exploring pytest on the command line, in order to see how it works “under the hood” and explore various commandline arguments.




Setup with PyCharm / VS Code


- Open `code/` folder (**not** enclosing folder!) as project, open `basic/test_calc.py`



- Select “Configure Python interpreter” when prompted
- 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
- Click  in the sidebar, you should see a tree of tests (some will fail)

- Open PyCharm / VS Code terminal at the bottom
- You should be able to run `pytest --version` and see at least `7.0.0`
- Also try  next to a test function (not entire file)

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 for high-level tools like [poetry](#)/[uv](#)

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”



pyte.st/venv

[first-proj/.venv](#)

- pytest 8.3.0
- 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
```

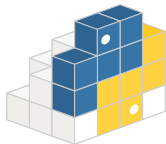


```
venv\Scripts\Activate
```



```
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

Fundamental features of pytest

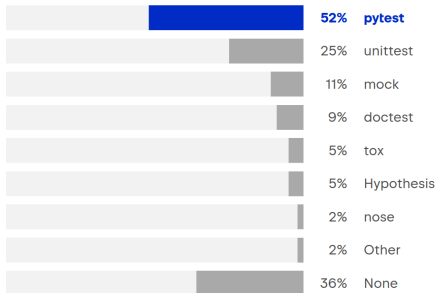


Popularity

- Automatic test discovery, no-boilerplate test code
(boilerplate: repeated code without any “real” use)
- Useful information when a test fails
- Test parametrization
- Modular setup/teardown via fixtures
- Customizable: Many options,
> 1600 plugins (easy to write your own!)

Unit-testing frameworks

100+



JetBrains Python Developers Survey 2023
 $n > 25\,000$



- Tests are run via `pytest` command line tool (called `py.test` before v3.0)
- Testing starts from given files/directories, or the current directory

Examples:

- `pytest`
- `pytest path/to/tests`
- `pytest test_file.py`
- `pytest test_file.py::test_func`
- `pytest test_file.py::TestClass`
- `pytest test_file.py::TestClass::test_meth`



pytest walks over the filesystem and:

- Discovers `test_...py` and `..._test.py` test files
- Discovers `test_...` functions and `Test...` classes
- Discovers classes deriving from `unittest.TestCase`

Automatic discovery avoids boilerplate

Calculator project

— `rpncalc/utils.py` —

```
def calc(a, b, op):  
    if op == "+":  
        return a + b  
    elif op == "-":  
        return a - b  
    elif op == "*":  
        return a * b  
    elif op == "/":  
        return a / b  
    raise ValueError("Invalid operator")
```

No boilerplate Python test code

— rpncalc/utils.py —————

```
def calc(a, b, op):  
    if op == "+":  
        return a + b  
    ...
```

— basic/test_calc.py —————

```
from rpncalc.utils import calc
```

```
def test_add():  
    res = calc(1, 3, "+")  
    assert res == 4
```

— \$ pytest basic/test_calc.py —————

```
===== test session starts =====  
collected 1 item
```

```
basic/test_calc.py .
```

```
===== 1 passed in ... =====
```

No boilerplate Python test code

— rpncalc/utils.py —

```
def calc(a, b, op):  
    if op == "+":  
        return a + b  
    ...
```

— basic/test_calc.py —

```
from rpncalc.utils import calc
```

```
def test_add():  
    res = calc(1, 3, "+")  
    assert res == 4
```

```
import unittest  
from rpncalc.utils import calc
```

```
class TestCalc(unittest.TestCase):  
    def test_add(self):  
        res = calc(1, 3, "+")  
        self.assertEqual(res, 4)
```

```
if __name__ == "__main__":  
    unittest.main()
```



```
class TestCalc:
    def test_add(self):
        assert calc(1, 3, "+") == 4

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

```
class TestCalc:
```

test_add

test_subtract

With pytest:

- Test classes have a `Test` prefix, are autodiscovered
- There is no need to subclass anything, test functions don't have to be in classes
- Test classes are for logical grouping of your tests
- Fixtures are typically used in place of unittest's `setUp()` and `tearDown()`



```
assert x
assert x == 1
assert x != 2
assert not x
assert x < 3 or y > 5
```

```
self.assertTrue(x)
self.assertEqual(x, 1)
self.assertNotEqual(x, 2)
self.assertFalse(x)
?
```

Demo on failure reporting ([basic/failure_demo.py](#))

Test outcomes

Every test can have one of the following outcomes:

PASSED	.	All assertions passed, no exceptions occurred
FAILED	F	An assertion failed or an exception occurred
ERROR	E	An exception occurred outside of the test (e.g. in a <i>fixture</i>)
SKIPPED	s	The test was skipped, e.g. because of a missing optional dependency
XFAILED	x	An expected failure occurred
XPASS	X	An unexpected success occurred (expected to fail but passed)

Tracebacks

— rpncalc/utils.py —

```
def calc(a, b, op):  
    if op == "+":  
        return a + b  
    elif op == "-":  
        return a - b  
    elif op == "*":  
        return a * b  
    elif op == "/":  
        return a / b  
    raise ValueError("Invalid operator")
```

— basic/test_traceback.py —

```
from rpncalc.utils import calc  
  
def test_divide():  
    # This will raise ZeroDivisionError  
    assert calc(2, 0, "/") == 0  
  
def test_good():  
    pass
```

Some important options

- `-v / --verbose` More **v**erbose output (can be given multiple times)
- `-q / --quiet` More **q**uiet output (negates `-v`)
- `-k expression` Run tests whose names contain the given **k**eyword

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

Exercise:

getting-started



- Add a `test_subtract` function to `basic/test_calc.py`
- Run `pytest basic/test_calc.py` to run tests
- Add `-k` to only run your new test, play with `-v` and `-q`
- Take a first look at the `--help` output

Every exercise has a label like this on the right, which helps with finding it in the code and solutions: Search for `[getting-started]`.

Typical directory structure



`tests/` outside of `src/`, file layout 1:1 mirroring (roughly) of `src/mypackage/`:

```
myproject/
├── pytest.ini
├── pyproject.toml
├── src/ myproject/
│   ├── __init__.py
│   ├── utils.py
│   └── export/
│       ├── __init__.py
│       ├── jsonexport.py
│       └── yamlexport.py
└── tests/
    └── ...
```

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

Typical directory structure



`conftest.py`: fixtures, plugin hooks, etc.

```
myproject/
├── pytest.ini
├── pyproject.toml
├── src/ myproject/
│   ├── __init__.py
│   ├── utils.py
│   └── export/
│       ├── __init__.py
│       ├── jsonexport.py
│       └── yamlexport.py
└── tests/
    └── ...
```

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

Typical directory structure



`pytest.ini` or `pyproject.toml`: (declarative) config

```
myproject/
├── pytest.ini
├── pyproject.toml
├── src/ myproject/
│   ├── __init__.py
│   ├── utils.py
│   └── export/
│       ├── __init__.py
│       ├── jsonexport.py
│       └── yamlexport.py
└── tests/
    └── ...
```

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

Typical directory structure



`__init__.py` files both in `src/myproject` and `tests/`

```
myproject/
├── pytest.ini
├── pyproject.toml
├── src/ myproject/
│   ├── __init__.py
│   ├── utils.py
│   └── export/
│       ├── __init__.py
│       ├── jsonexport.py
│       └── yamlexport.py
└── tests/
    └── ...
```

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


Typical directory structure



```
myproject/
├── pytest.ini
├── pyproject.toml
├── src/ myproject/
│   ├── __init__.py
│   ├── utils.py
│   └── export/
│       ├── __init__.py
│       ├── jsonexport.py
│       └── yamlexport.py
└── tests/
    └── ...
```

Based on the “[src](#) layout”:

Ionel Cristian Mărieș ([ionelmc.ro](#)):
“[Packaging a python library](#)”
(also for applications!)



Hynek Schlawack ([hynek.me](#)):
“[Testing & Packaging](#)”



```
myproject/
├── pytest.ini
├── pyproject.toml
├── src/ myproject/
│   ├── utils.py
│   └── ...
└── tests/
    ├── conftest.py
    ├── test_utils.py
    └── ...
```

You can create a `pytest.ini` file:

```
[pytest]
option = value
...
```

pytest also can read a `[pytest]` section in `tox.ini` or `[tool.pytest.ini_options]` from `pyproject.toml`, in case you prefer having multiple tools configured in one file.

Available options are listed in the `pytest -h` output, or in the [reference documentation](#):

[pytest.org](#) → Reference guides →
API Reference → Configuration Options

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, "/")
```

Exercise:

raises

- In `basic/test_raises.py`, write another test with `pytest.raises`, to ensure that `ValueError` is raised when calling `calc` with an invalid operator.
- Rerun test, but edited so that no exception or a different exception is raised, e.g. `calc(1, 2, "+")` and `calc(1, 0, "/")`.
- Pass a regex pattern to the `match` argument to check the exception message:

```
with pytest.raises(ValueError, match=r"..."): (optional)
```

Marks

Mark functions or classes:

— marking/test_marking.py

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

slow webtest

test_slow_api

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

webtest

test_api

```
def test_fast():
    pass
```

test_fast

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.



Mark functions or classes:

Register the custom markers:

— `marking/test_marking.py`

```
@pytest.mark.slow
@pytest.mark.webtest
```

```
def test_slow_api():
    time.sleep(1)
```

```
@pytest.mark.webtest
```

```
def test_api():
    pass
```

```
def test_fast():
    pass
```

— `pytest.ini`

```
[pytest]
markers =
    slow: Tests which take some time to run
    webtest: Tests making web requests
```

Then pass e.g. `-m "slow"` to pytest to filter by marker.

Use `--markers` to show a list of all known markers.

Parametrizing tests



Tests can be parametrized to run them with various values:

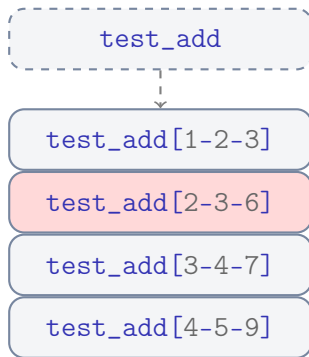
— `marking/test_parametrization.py` —

```
@pytest.mark.parametrize("a, b, expected", [
    (1, 2, 3),    # 1 + 2 = 3
    (2, 3, 6),    # 2 + 3 = 6 (?)
    (3, 4, 7),    # 3 + 4 = 7
    (4, 5, 9),    # 4 + 5 = 9
])
```

```
def test_add(a: int, b: int, expected: int):
    assert calc(a, b, "+") == expected
```

```
@pytest.mark.parametrize(
    "op", ["+", "-", "*", "/", "@"])
```

```
def test_smoke(op: str):
    calc(1, 2, op)
```



Parametrizing tests



Tests can be parametrized to run them with various values:

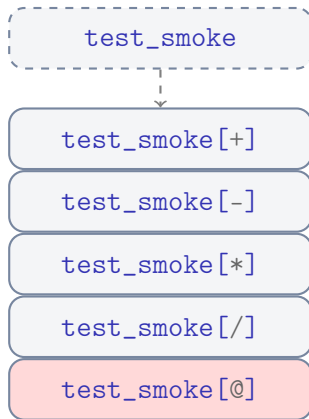
— `marking/test_parametrization.py` —

```
@pytest.mark.parametrize("a, b, expected", [
    (1, 2, 3), # 1 + 2 = 3
    (2, 3, 6), # 2 + 3 = 6 (?)
    (3, 4, 7), # 3 + 4 = 7
    (4, 5, 9), # 4 + 5 = 9
])
```

```
def test_add(a: int, b: int, expected: int):
    assert calc(a, b, "+") == expected
```

```
@pytest.mark.parametrize(
    "op", ["+", "-", "*", "/", "@"])
```

```
def test_smoke(op: str):
    calc(1, 2, op)
```



Parametrizing tests



Tests can be parametrized to run them with various values:

— `marking/test_parametrization.py` —

```
@pytest.mark.parametrize("a, b, expected", [
    (1, 2, 3),    # 1 + 2 = 3
    (2, 3, 6),    # 2 + 3 = 6 (?)
    (3, 4, 7),    # 3 + 4 = 7
    (4, 5, 9),    # 4 + 5 = 9
])

def test_add(a: int, b: int, expected: int):
    assert calc(a, b, "+") == expected

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

Exercise:

parametrize

- Write a `test_multiply` with a single, hardcoded value
- Parametrize the test to test multiple inputs and expected outputs
- Run pytest with `-v`

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.

“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).



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

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

xfail vs. raises

`pytest.raises`

Testing a “bad case”, but **intended** behavior:

⇒ When we call `calc(1, 2, "@")`, we expect a `ValueError`.

`pytest.mark.xfail`

Marking a test where the implementation behaves in **unintended** ways:

This test **should work**, but currently does not (e.g. upstream bug).

⇒ e.g. if the user asks for the result of `1 / 0` and gets an unhandled exception.

Marks summary

- At a basic level, marks let us categorize tests by using them as “tags” / “labels”.
- However, marks also lets us attach “meta-information” to a test.
This information is used by pytest in various ways:

`@pytest.mark.parametrize` Run the same test against different data

`@pytest.mark.skip` Skip a test (not applicable)

`@pytest.mark.xfail` Expect a test to fail (broken, out of our control)

- (In fixtures or plugin hooks, we can also access a marker’s arguments to customize behavior.)
- Test IDs are auto-generated but can be overridden

Expanding the calculator example

Reverse Polish Notation (RPN)

History

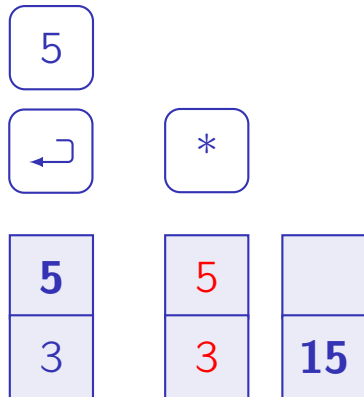
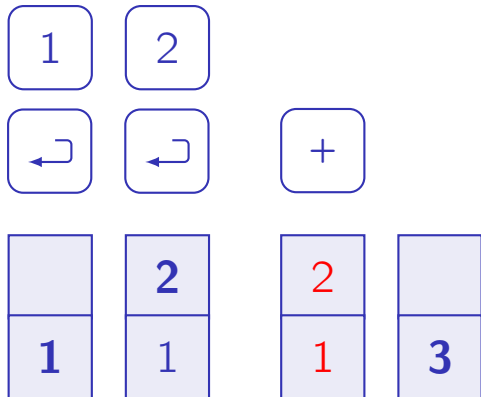


- Using a calculator without needing a = key, and without parentheses
- Makes it much easier to implement, using a stack data structure
- Used by all HP calculators in the 1970s–80s, still used by some today
- Displayed here: HP 12C financial calculator, introduced in 1981, still in production today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

Reverse Polish Notation (RPN)

Explanation

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

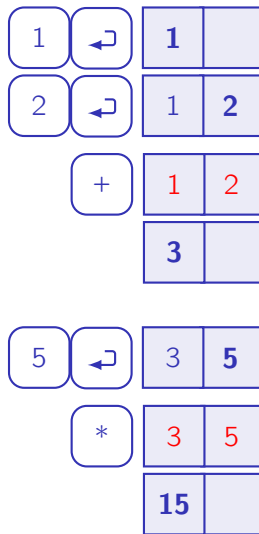


Reverse Polish Notation (RPN)

In Python

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

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



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()
```

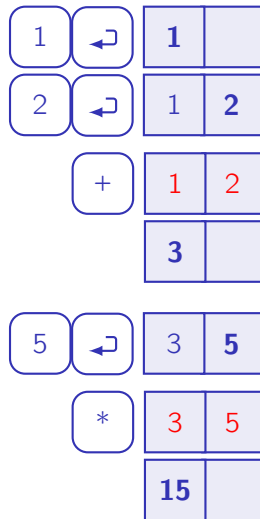
Reverse Polish Notation (RPN)

Tests: Recycling the example

— `rpncalc/test_rpn_v1.py` —

```
def test_complex_example():  
    rpn = RPNCalculator()  
  
    rpn.evaluate("1")  
    assert rpn.stack == [1]  
    rpn.evaluate("2")  
    assert rpn.stack == [1, 2]  
    rpn.evaluate("+")  
    assert rpn.stack == [3]  
    rpn.evaluate("5")  
    assert rpn.stack == [3, 5]  
    rpn.evaluate("*")  
    assert rpn.stack == [15]
```

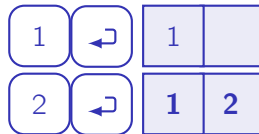
$$5 \cdot (1 + 2)$$



Reverse Polish Notation (RPN)

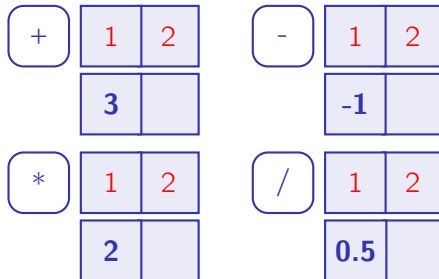
Tests: Getting a bit smaller

```
def test_stack_push():  
    rpn = RPNCalculator()  
    rpn.evaluate("1")  
    rpn.evaluate("2")  
    assert rpn.stack == [1, 2]
```



```
@pytest.mark.parametrize("op, expected", [  
    ("+", 3), ("-", -1),  
    ("*", 2), ("/", 0.5),  
)
```

```
def test_operations(op, expected):  
    rpn = RPNCalculator()  
    rpn.stack = [1, 2]  
    rpn.evaluate(op)  
    assert rpn.stack == [expected]
```



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 + ↵` (not `2 ↵ 1 ↵ + ↵`)

Reverse Polish Notation (RPN)

Fixing bugs, improving error handling

— rpncalc/rpn_v1.py —

```
...
def evaluate(self, inp: str):
    if inp.isdigit():
        n = float(inp)
        self.stack.append(n)
    elif inp in "+-*/":
        ...
```

— rpncalc/test_rpn_v2.py —

```
@pytest.mark.parametrize("n", [1.5, -1])
def test_number_input(n: float):
    rpn = RPNCalculator(Config())
    rpn.evaluate(str(n))
    assert rpn.stack == [n]
```

With rpn_v1.py:

```
E      assert [] == [1.5]
E      assert [] == [-1]
```

With rpn_v2.py:

```
...::test_number_input[1.5] PASSED
...::test_number_input[-1]  PASSED
```

Reverse Polish Notation (RPN)

Fixing bugs, improving error handling

— rpncalc/rpn_v1.py —

```
...  
  
def evaluate(self, inp: str):  
    if inp.isdigit():  
        n = float(inp)  
        self.stack.append(n)  
    elif inp in "+-*/":  
        ...
```



pyte.st/mathsp-p-error

LBYL: Look before you leap

— rpncalc/rpn_v2.py —

```
...  
  
def evaluate(self, inp: str) -> None:  
    try:  
        self.stack.append(float(inp))  
        return  
    except ValueError:  
        pass  
  
    if inp not in ["+", "-", "*", "/]:  
        ...
```

EAFP: It's easier to ask for forgiveness,
than for permission

Reverse Polish Notation (RPN)

Fixing bugs, improving error handling

— rpncalc/rpn_v1.py —

```
...
def evaluate(self, inp: str):
    if inp.isdigit():
        n = float(inp)
        self.stack.append(n)
    elif inp in "+-*/*":
        ...
```

— rpncalc/test_rpn_v2.py —

```
@pytest.mark.parametrize(
    "op", ["@", "+-"])
def test_unknown_operator(op: str):
    rpn = RPNCalculator(Config())
    rpn.stack = [1, 2]
    rpn.evaluate(op)
```

With rpn_v1.py:

```
...::test_unknown_operator[@] PASSED
...::test_unknown_operator[+-] FAILED
E      ValueError: Invalid operator
```

With rpn_v2.py: rpncalc/test_rpn_v2.py ..

Reverse Polish Notation (RPN)

Fixing bugs, improving error handling

— rpncalc/rpn_v1.py —

...

```
def evaluate(self, inp: str):
    if inp.isdigit():
        n = float(inp)
        self.stack.append(n)
    elif inp in "+-*/*":
        ...
```

— rpncalc/rpn_v2.py —

...

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

```
if inp not in ["+", "-", "*", "/]:
```

...

Fixtures

Reverse Polish Notation (RPN)

Adding a 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_cwd()  
    rpn = RPNCalculator(config)  
    rpn.run()
```

— `rpncalc/utils.py` —————

```
class Config:
```

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

```
...
```

RPNCalculator

Config

Beyond simple testing: fixtures!

```
def test_example():  
    config = Config()  
    rpn = RPNCalculator(config)  
    ...
```

```
def test_stack_push():  
    config = Config()  
    rpn = RPNCalculator(config)  
    ...
```

```
def test_operations(...):  
    config = Config()  
    rpn = RPNCalculator(config)  
    ...
```

A test fixture:

- Sets up objects or apps for testing
- Provides test code with “base” app objects
- Is very important to avoid repetitive test code

In pytest realized via dependency injection:

- Fixture functions create and return fixture values
- They are registered with pytest by using a `@pytest.fixture` decorator
- Test functions and classes can request and use them

Example of pytest fixture injection

pytest calls the fixture function to inject a dependency into the test function:

— fixtures/test_fixture.py —

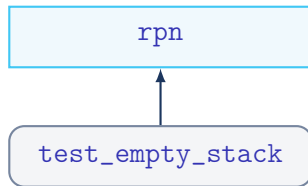
```
@pytest.fixture
def rpn():
    return RPNCalculator(Config())
```

```
def test_empty_stack(rpn):
    assert rpn.stack == []
```

Imagine pytest running your test like:

```
test_empty_stack(rpn=rpn())
```

pyte.st/fixtures



Exercise:

fixtures

- Add an `rpn` fixture to `rpncalc/test_rpn_v2.py`
- Rewrite `test_operations` to use it

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):
    ...
```

Your IDE will typically show an error if the type annotations aren't correct, but you should probably use a tool like [mypy](https://mypy-lang.org) to validate them in a CI job.



mypy-lang.org

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.
```

Modularity: Using fixtures from fixtures

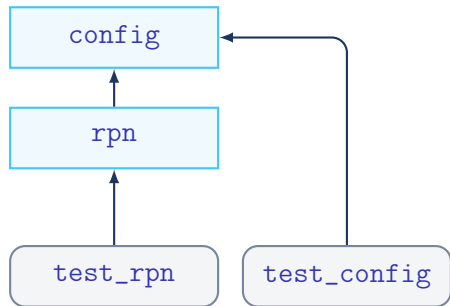
— fixtures/test_fixtures_using_fixtures.py

```
@pytest.fixture
def config() -> Config:
    return Config()

@pytest.fixture
def rpncalculator(config: Config) -> RPNCalculator:
    return RPNCalculator(config)

def test_config(config: Config):
    assert config.prompt == ">"

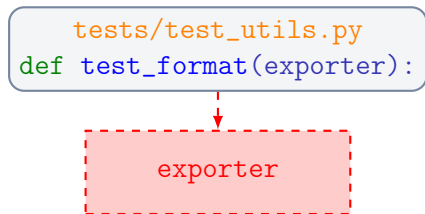
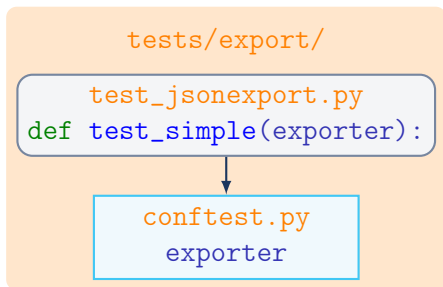
def test_rpn(rpncalculator: RPNCalculator):
    assert rpncalculator.stack == []
```



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.



pyte.st/conftest



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


Fixture visibility



Fixtures defined...

- ...as methods of a test class are available only to test methods on that class.
- ...in a test module are available only to tests in that module.
- ...in a `conftest.py` file are available to tests in that directory and subdirectories.
- ...in a plugin (or built into pytest) are available everywhere.

Use `pytest --fixtures` to see which fixtures have been picked up from where.

Fixtures in a more specific location shadow those in a more general location.

```
class TestConfig:
    @pytest.fixture
    def config(self):
        return Config()

    def test_config(self, config):
        ...
```

Fixture visibility



Fixtures defined...

- ...as methods of a test class are available only to test methods on that class.
- ...in a test module are available only to tests in that module.
- ...in a `conftest.py` file are available to tests in that directory and subdirectories.
- ...in a plugin (or built into pytest) are available everywhere.

Use `pytest --fixtures` to see which fixtures have been picked up from where.

Fixtures in a more specific location shadow those in a more general location.

— `test_config.py` —

```
@pytest.fixture
def config():
    return Config()
```

```
def test_config(config):
    ...
```

Fixture visibility

Fixtures defined...

- ...as methods of a test class are available only to test methods on that class.
- ...in a test module are available only to tests in that module.
- ...in a `conftest.py` file are available to tests in that directory and subdirectories.
- ...in a plugin (or built into pytest) are available everywhere.

Use `pytest --fixtures` to see which fixtures have been picked up from where.

Fixtures in a more specific location shadow those in a more general location.



— `conftest.py` —

```
@pytest.fixture
def rpn_config():
    return Config()
```

This makes `rpn_config` available for other files, **without any importing**:

— `test_config.py` —

```
def test_config(rpn_config):
    ...
```

— `test_rpn.py` —

```
def test_rpn(..., rpn_config):
    ...
```

Fixture visibility



Fixtures defined...

- ...as methods of a test class are available only to test methods on that class.
- ...in a test module are available only to tests in that module.
- ...in a `conftest.py` file are available to tests in that directory and subdirectories.
- ...in a plugin (or built into pytest) are available everywhere.

Use `pytest --fixtures` to see which fixtures have been picked up from where.

Fixtures in a more specific location shadow those in a more general location.

```
$ pip install pytest-someplugin
```

```
...
```

```
Successfully installed  
pytest-someplugin-1.2.3
```

```
$ pytest
```

```
==== test session starts =====
```

```
...
```

```
plugins: someplugin-1.2.3
```



pytest provides builtin fixtures:

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

More fixtures are provided by plugins.

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



```
def test_ok():  
    print("This isn't printed")
```

```
def test_bad():  
    print("This is printed")  
    assert False
```

```
$ pytest -v basic/test_capturing.py
```

```
...
```

```
...::test_ok PASSED
```

```
...::test_bad FAILED
```

```
===== FAILURES =====
```

```
----- test_bad -----
```

```
...
```

```
----- Captured stdout call -----
```

```
This is printed
```

```
===== short test summary info =====
```

```
...
```

Builtin fixtures

Capturing



```
def test_ok():  
    print("This isn't printed")
```

```
def test_bad():  
    print("This is printed")  
    assert False
```

```
$ pytest -s -v basic/test_capturing.py  
...
```

```
...::test_ok This isn't printed  
PASSED
```

```
...::test_bad This is printed  
FAILED
```

```
===== FAILURES =====  
----- test_bad -----  
...  
===== short test summary info =====  
...
```



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.

Type for both: `pytest.CaptureFixture[str]`

capsysbinary/ Like **capsys/capfd**, but for binary data.

capfdbinary Type for both: `pytest.CaptureFixture[bytes]`

caplog Capturing logging output from a test.

Type: `pytest.LogCaptureFixture`

capsys vs. **capfd**: If in doubt, use **capfd** to ensure everything is captured
(built-in pytest capturing also uses **fd** by default).



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`)
- Change current directory (`chdir`)
- Modify dictionaries (`setitem`, `delitem`)
- Prepend to `sys.path` for importing (`syspath_prepend`)



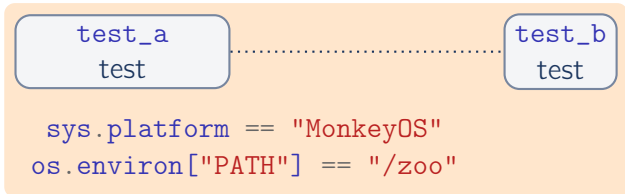


```
def print_info():  
    path = os.environ.get("PATH", "")  
    print(f"platform: {sys.platform}")  
    print(f"PATH: {path}")
```

```
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
```





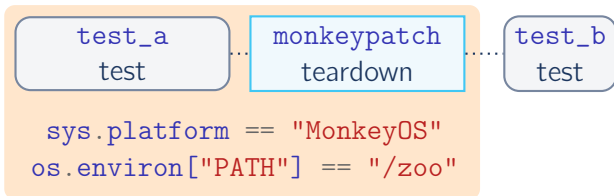
monkeypatch

— fixtures/test_builtin_monkeypatch.py

```
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
```





monkeypatch: Patching functions

— fixtures/test_builtin_monkeypatch.py —

```
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", # target, "name"
        fake_getuser        # value
    )
    assert get_folder_name() == "pytest-of-fakeuser"
```



monkeypatch: Patching functions

— fixtures/test_builtin_monkeypatch.py —

```
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)

Supporting multiple inputs

— rpncalc/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

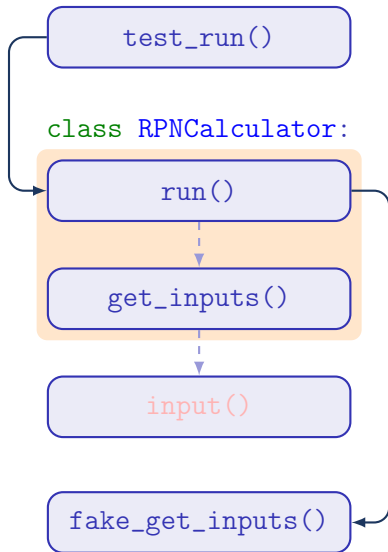
— rpncalc/rpn_v2.py —

```
class RPNCalculator:
    ...

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

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

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





monkeypatch

— `rpncalc/rpn_v2.py` —

```
class RPNCalculator:
    ...

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

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

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

Exercise:

monkeypatch

- Add a new test to `test_rpn_v2.py`, which will test `run`.
- Leave `rpn_v2.py` as-is!
- Use the `monkeypatch` fixture as arg, to patch the `get_inputs` method on `rpn_calc` (instance, not class).
- Replace it by a function that returns a fixed **list of strings, ending in "q"**. E.g. `def fake_get_inputs():` or use `lambda:`
- Remember: Arrange, Act, **Assert**.

Builtin fixtures



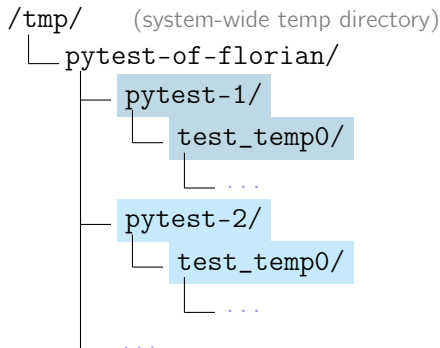
tmp_path / tmpdir

A **fresh empty directory** for every pytest invocation and every test.

- 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

```
from pathlib import Path
```

```
def test_temp(tmp_path: Path):  
    ...
```





tmp_path / tmpdir

A **fresh empty directory** for every pytest invocation and every test.

- 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:

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



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

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)

The Config object

— `rpncalc/rpn_v2.py` —————

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

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

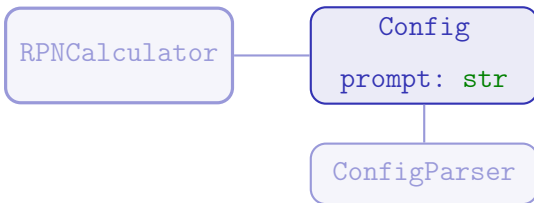
...

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

— `rpncalc/utils.py` —————

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

...



Reverse Polish Notation (RPN)

Changing the prompt

Imagine you could change the prompt inside rpncalc:

```
$ python -m rpncalc.rpn_v2  
> set prompt=calc>  
calc> q
```

The change would automatically be persisted in a config:

```
$ cat rpncalc.ini  
[rpncalc]  
prompt = calc>
```

And loaded again when running rpncalc a second time:

```
$ python -m rpncalc.rpn_v2  
calc>
```

```
class RPNCalculator
```

```
    config = ...
```

```
class Config
```

```
    prompt = ">"
```

```
    c.save(...)
```

```
[rpncalc]
```

```
prompt = calc>
```

```
rpncalc.ini
```

Reverse Polish Notation (RPN)

Changing the prompt

Imagine you could change the prompt inside rpncalc:

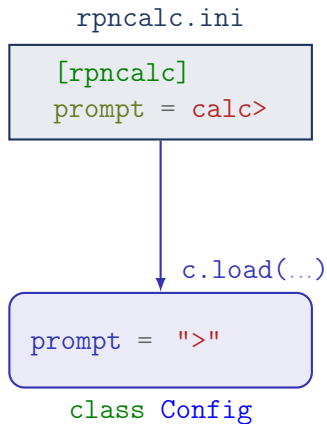
```
$ python -m rpncalc.rpn_v2  
> set prompt=calc>  
calc> q
```

The change would automatically be persisted in a config:

```
$ cat rpncalc.ini  
[rpncalc]  
prompt = calc>
```

And loaded again when running rpncalc a second time:

```
$ python -m rpncalc.rpn_v2  
calc>
```



Reverse Polish Notation (RPN)

Changing the prompt

Imagine you could change the prompt inside rpncalc:

```
$ python -m rpncalc.rpn_v2
> set prompt=calc>
calc> q
```

The change would automatically be persisted in a config:

```
$ cat rpncalc.ini
[rpncalc]
prompt = calc>
```

And loaded again when running rpncalc a second time:

```
$ python -m rpncalc.rpn_v2
calc>
```

There could be more settings in the future as well (e.g. precision or rounding), all stored by the `Config` class.

For simplicity, the `set` command is not actually implemented in `rpncalc/rpncalc_v2.py`.

However, the necessary `Config.load()` and `Config.save()` methods in `utils.py` are, and we want to test them now.

Reverse Polish Notation (RPN)

Saving and loading config

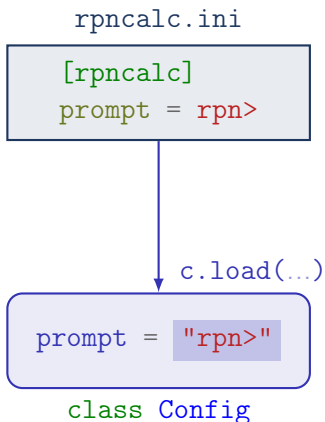
— `rpncalc/utils.py` —

```
class Config:
    ...

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

— Loading —

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



Reverse Polish Notation (RPN)

Saving and loading config

— `rpncalc/utils.py` —

```
class Config:
    ...

    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(prompt="calc>")
c.save(ini_path)
```

`class Config`

`prompt = "calc>"`

`c.save(...)`

`[rpncalc]`

`prompt = calc>`

`rpncalc.ini`

Builtin fixtures

pyte.st/tmp-path



tmp_path

— rpncalc/test_utils.py —

```
@pytest.fixture
```

```
def ini_path(tmp_path: Path) -> Path:
```

```
    return tmp_path / "rpncalc.ini"
```

```
def test_config_save(
```

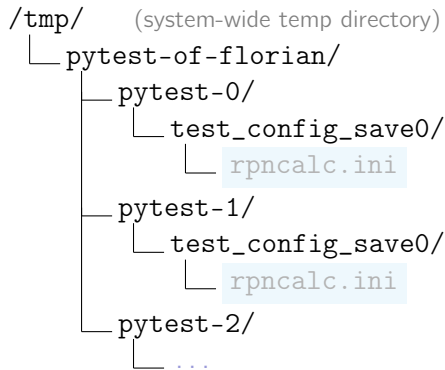
```
    ini_path: Path, config: Config
```

```
):
```

```
    # call config.save(...), ensure that
```

```
    # the ini file is written correctly
```

```
    ...
```



Builtin fixtures

pyte.st/tmp-path



tmp_path

— rpncalc/test_utils.py —

```
@pytest.fixture
```

```
def example_ini(ini_path: Path) -> Path:
```

```
    # creates rpncalc.ini with pathlib
```

```
    ini_path.write_text(
```

```
        "[rpncalc]\n"
```

```
        "prompt = rpn>\n")
```

```
    return ini_path
```

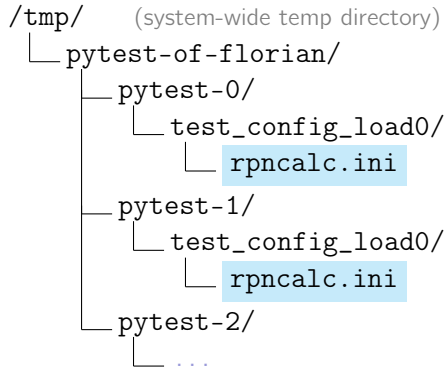
```
def test_config_load(
```

```
    example_ini: Path, config: Config
```

```
):
```

```
    # call config.load(...), ensure that the prompt is set to "rpn>"
```

```
    ...
```



Builtin fixtures

pyte.st/tmp-path



tmp_path

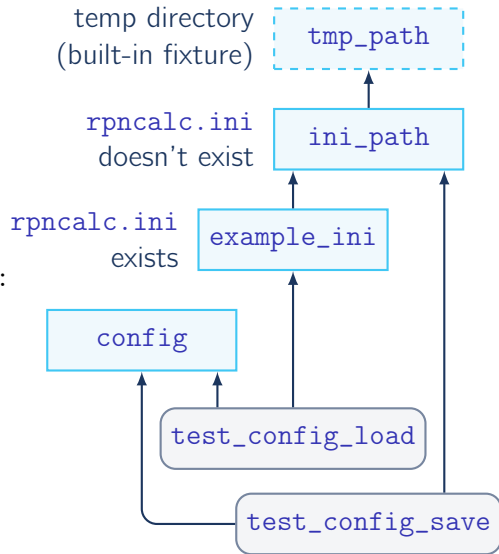
— rpncalc/test_utils.py —

```
@pytest.fixture
```

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

```
@pytest.fixture
```

```
def example_ini(ini_path: Path) -> Path:  
    # creates rpncalc.ini with pathlib  
    ini_path.write_text(  
        "[rpncalc]\n"  
        "prompt = rp>\n")  
    return ini_path
```

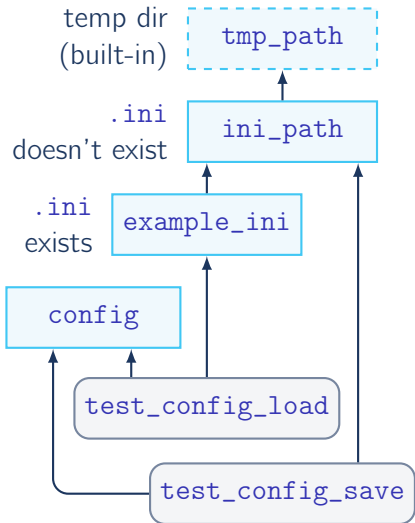


Builtin fixtures

pyte.st/tmp-path



tmp_path



Exercise: `rpncalc/test_utils.py`

load-save

- Complete `test_config_load`:
 - Call the `load` method with the prepared config file
 - Ensure that `config.prompt` has changed from the default value (`>` → `rpn>`)
- Complete `test_config_save`:
 - Set `config.prompt` to a value
 - Call the `save` method with the non-existing ini path
 - Ensure the written file looks correct (`.exists()`, `.read_text()`, or `configparser`)
- **(optional)** Test calling `config.load` with an `ini` containing just `rpn>` (exc.: `configparser.Error`)

pytest fixtures: Where we are, what's next

We've seen how:

Next:

- Fixture values are returned from a fixture function (often just “fixture”)
- Each fixture has a name (the function name)
- Test functions get its value injected as an argument by name
- Fixtures can use other fixtures
- Fixtures can be defined in a class, file, `conftest.py`, plugin, or built into pytest
- pytest provides various built-in fixtures (`monkeypatch`, `tmp_path`)
- Fixture values can be cached on a per scope basis
- `pytest.skip/.xfail` in fixtures
- Doing cleanup / teardown
- Using fixtures implicitly (`autouse`)
- Fixture functions can introspect calling side (`request`)
- Configuring fixtures via markers/CLI
- Fixture functions can be parametrized

Caching fixture results



Fixture functions can declare a caching scope:

— fixtures/test_fixture_scope.py —

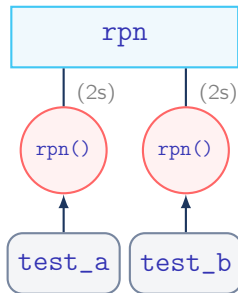
```
@pytest.fixture(scope="module")
```

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

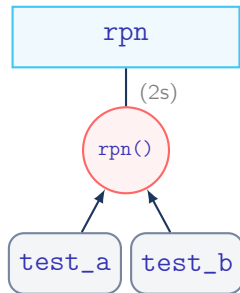
```
def test_a(rpn: RPNCalculator):
    rpn.stack.append(42)
    assert rpn.stack == [42]
```

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

Function scope:



Module scope:



Available scopes:

"function" (default), "class",
"module", "package", "session"

Caching fixture results



Fixture functions can declare a caching scope:

— fixtures/test_fixture_scope.py —

```
@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]
```



Fixture functions can declare a caching scope:

— `fixtures/test_fixture_scope.py` —

```
@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
```

How to avoid this pitfall?

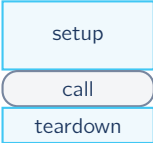
- Can you make the return value immutable somehow?
- Can you copy it (e.g. `deepcopy`)?
- Can you reset the state between tests with a second fixture?
- ...or maybe even take a snapshot before the test and restore it after?

Doing cleanup with yield

A fixture can use `yield` instead of `return` to run cleanup steps after the test:

— `fixtures/test_yield_fixture.py` —

```
@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")
```

Demo:

yield

- Observe the teardown behaviour using `-s` and/or `--setup-show`
- Modify caching scope, check how the behavior changes
- Remove caching again, call `pytest.skip(...)` before `client.connect()`, see what happens

Fixture summary

pytest fixtures are a modular, extensible mechanism to:

- Inject configurable resources into test functions
- Manage life-time / caching scope of resources
- Setup resources implicitly (autouse)
- Interact with tests requiring the resource
- Re-run tests with differently configured resources

Debugging failing tests

Arguments to control output



	<code>--tb</code>	Control traceback generation
		<code>--tb=auto / long / short / line / native / no</code>
<code>-l</code>	<code>--showlocals</code>	Show locals in tracebacks
<code>-s</code>	<code>--capture=no</code>	Disable output capturing
<code>-q</code>	<code>--quiet</code>	Decrease verbosity
<code>-v</code>	<code>--verbose</code>	Increase verbosity (can be given multiple times)

```
$ pytest --tb=short basic/test_traceback.py
```

```
...
----- test_divide -----
basic/test_traceback.py:5: in test_divide
    assert calc(2, 0, "/") == 0
rpncalc/utils.py:14: in calc
    return a / b
E   ZeroDivisionError: division by zero
```

Arguments to select tests

pyte.st/mathspp-select



after  —  —  —  — 

pass: 

fail: 

new: 

`--lf` `--last-failed` Run last-failed only



`--ff` `--failed-first` Run last-failed first



`--nf` `--new-first` Run new tests first



`-x` `--maxfail=n` Exit on first / n -th failure



`--sw` `--stepwise` Look at failures step by step



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.

Reminder: `--fixtures` and `--fixtures-per-test` can be useful as well.

Output:

```
fixtures/test_fixture.py
```

```
  SETUP      F  rpnp
```

```
fixtures/test_fixture.py::test_empty_stack (fixtures used: rpnp) .
```

```
  TEARDOWN  F  rpnp
```

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
```

Some plugins

Using the coverage plugin

Install plugin: `pip install pytest-cov`

Options:

`--cov=path` filesystem path to generate coverage for
`rpncalc/` as our code under test is there

`--cov-report=type` type of report ("term", "html", ...)
`term-missing` to show lines not covered

`testpath` path to tests
`rpncalc/` as our tests are there too



coverage

Exercise:

- 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

Reporting plugins / features

- **pytest-instafail**
reports failure
details while tests
are running.
- **pytest-html**
- **-junitxml**
- **pytest-reportlog**
- **record_property**
- **custom plugin**

```
code/plugins/reporting$ pytest --instafail
```

```
...  
test_instafail.py .....F
```

```
----- test_bad -----
```

```
    def test_bad():  
>         assert False  
E         assert False
```

```
test_instafail.py:8: AssertionError
```

```
test_instafail.py .....
```

Reporting plugins / features

- **pytest-instafail**

- **pytest-html**

generates
(customizable)
HTML reports.

- **-junitxml**

- **pytest-reportlog**

- **record_property**

- **custom plugin**

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



Environment

Driver	Firefox
Platform	Darwin-15.2.0-x86_64-1386-64bit
Python	2.7.10

Summary

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

Results

Result	Test	Duration	Links
Failed	test_selenium_report.py::test_home	5.08	URL, Screenshot HTML, Browser Log, Driver Log
<pre>selenium - <selenium.webdriver.firefox.webdriver.WebDriver (session="19a505a7-b481-8c4d-a289-2ecf8cdcfc43")> def test_home(selenium): selenium.get('https://www.mozilla.org/') selenium.find_element_by_id('close_takeover').click() > assert selenium.title == 'We're building a better Internet - Mozilla' E assert 'We're build...net - Mozilla' == 'We're buildin...net - Mozilla' E - We're building a better Internet - Mozilla E ? ^ E + We're building a better Internet - Mozilla E ? ^ test_selenium_report.py:7: AssertionError URL: https://www.mozilla.org/en-US/</pre> 			
XFailed	test_selenium_report.py::test_firefox	4.56	URL, Screenshot HTML, Browser Log, Driver Log
<pre>selenium - <selenium.webdriver.firefox.webdriver.WebDriver (session="2619e164-4644-1f41-9be4-fc0d24c893b1")> @pytest.mark.xfail(reason="Test is failing due to known bug") def test_firefox(selenium): selenium.get('https://www.mozilla.org/firefox/products/') > assert selenium.title == 'Firefox - Desktop browser, Android - Mozilla' E assert 'Firefox - De...ing - Mozilla' == 'Firefox - Des...old - Mozilla' E - Firefox - Desktop browser, Android, iOS, OS, Hello, Sync, Private Browsing - Mozilla E + Firefox - Desktop browser, Android - Mozilla test_selenium_report.py:20: AssertionError URL: https://www.mozilla.org/en-US/firefox/products/</pre> 			
XPassed	test_selenium_report.py::test_thunderbird	0.19	
Skipped	test_selenium_report.py::test_participate	0.00	
({'test_selenium_report.py', 13, u'Skipped: Test is not ready yet'})			
Passed	test_selenium_report.py::test_about	0.10	

Reporting plugins / features

- **pytest-instafail**

- **pytest-html**

- **-junitxml**

generates JUnit
XML reports.

- **pytest-reportlog**

- **record_property**

- **custom plugin**

```
<?xml version="1.0" encoding="utf-8"?>
<testsuites>
  <testsuite name="pytest" errors="0" failures="1"
    skipped="0" tests="2" time="0.029" ... >
    <testcase name="test_ok" ... />
    <testcase name="test_bad" ... >
      <failure message="ZeroDivisionError: ...">
        def test_bad():
          >      1/0
          E      ZeroDivisionError: division by zero

          test_reporting.py:5: ZeroDivisionError
      </failure>
    </testcase>
  </testsuite>
</testsuites>
```



Reporting plugins / features

- **pytest-instafail**

```
{..., "$report_type": "SessionStart"}
```

- **pytest-html**

```
{..., "$report_type": "CollectReport"}
```

- **-junitxml**

```
{
```

- **pytest-reportlog**

JSON reports in a
pytest-specific
format.

```
  "nodeid": "test_rep.py::test_ok",  
  "location": ["test_rep.py", 0, "test_ok"],  
  "keywords": {...},  
  "outcome": "passed", "longrepr": null,  
  "when": "setup", # setup / call / teardown  
  "user_properties": [], "sections": [],  
  "duration": ..., "start": ..., "stop": ...,  
  "$report_type": "TestReport"
```

- **record_property**

- **custom plugin**

```
}
```

```
...
```

```
{"exitstatus": 1, "$report_type": "SessionFinish"}
```

Reporting plugins / features

- **pytest-instafail**

```
def test_function(record_property):  
    record_property("example_key", 1)
```

- **pytest-html**

- **-junitxml**

- **pytest-reportlog**

- **record_property**

fixture to add
custom data to
XML / JSON
reports.

```
<testcase ...>
```

```
    <properties>
```

```
        <property name="example_key" value="1" />
```

```
    </properties>
```

```
</testcase>
```

```
{
```

```
    ...,
```

```
    "user_properties": [["example_key", 1]],
```

```
    "$report_type": "TestReport"
```

```
}
```

Reporting plugins / features

- **pytest-instafail**

```
def pytest_sessionstart(session: Session):
```

- **pytest-html**

```
...
```

- **-junitxml**

```
def pytest_collectreport(report: CollectReport):
```

- **pytest-reportlog**

```
...
```

- **record_property**

```
def pytest_runtest_logreport(report: TestReport):
```

- **custom plugin**

```
...
```

Hooks to get data
as Python objects.

```
def pytest_warning_recorded(...):
```

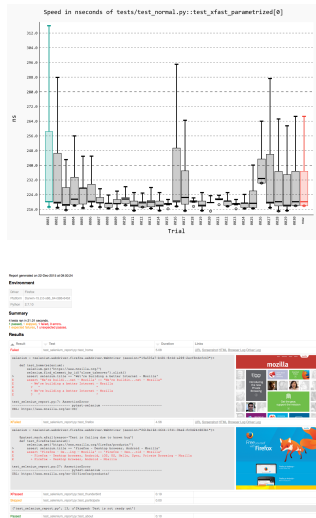
```
...
```

```
def pytest_sessionfinish(session, exitstatus):
```

```
...
```



- **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)
- ... > 1600 more: <https://pyte.st/plugins>



Writing your own plugins



Plugins come in two flavours:

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

They can contain fixtures, plus hook implementations for:

- configuration
- collection
- test running
- reporting

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



Adding to header and summary



pytest_report_header

— hooks/reporting/conftest.py —

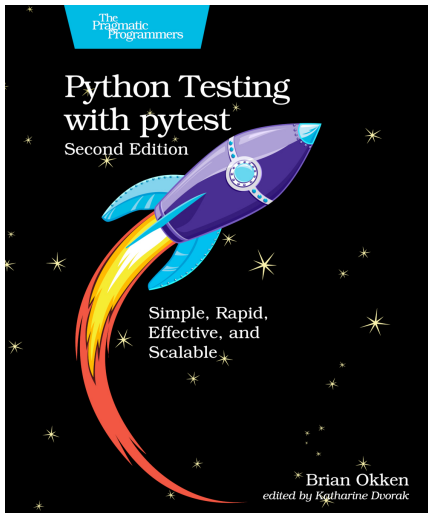
```
def pytest_report_header() -> list[str]:  
    return ["extrainfo: line 1"]
```

```
$ pytest
```

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

Book recommendation

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



- ISBN 978–1680508604
- <https://pragprog.com/titles/bopytest2/>
- Discount code: **EuropythonJuly2025**
35% off until end of July
(DRM-free PDF/epub/mobi ebook)
- Full disclosure: I'm technical reviewer
(but don't earn any money from it)



Also by Brian Okken:
“Test & Code” podcast, testandcode.com

Where to go next

aka “shameless self promotion”

pytest tips and tricks
for a better testsuite

Florian Bruhin



PyConDE & PyData Berlin

April 22nd, 2024



Description

2024.pycon.de/program/DSFWRC/



Repository

 [The-Compiler/pytest-tips-and-tricks](https://github.com/The-Compiler/pytest-tips-and-tricks)



Recording (Youtube)

pyte.st/talk-tips

Where to go next

aka “shameless self promotion”

Property based testing with Hypothesis



 The-Compiler/hypothesis-talk

Florian Bruhin


October 17th, 2024



At Swiss Python Summit 2024:



Repository

 The-Compiler/hypothesis-talk



Recording (media.ccc.de)
pyte.st/talk-hypothesis



Recording (Youtube)
youtu.be/6a1RvMKj0ws

Upcoming events

- **March 3rd – 5th 2026**

Python Academy (python-academy.com)

Professional Testing with Python (3 days)

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



@the_compiler



@the_compiler@mastodon.social



@the-compiler.org



BRUHIN
SOFTWARE



The-Compiler/pytest-basics

Copyright 2015 – 2025 Florian Bruhin



CC BY 4.0



Originally based on materials copyright 2013 – 2015 by Holger Krekel, used with permission.