# The basics

## getting-started

basic/test_calc.py

```
def test_subtract():
    res = calc(3, 1, "-")
    assert res == 2
```

Running the entire file:

$ pytest basic/test_calc.py

```
=================== test session starts ====================
collected 2 items

basic/test_calc.py ..

==================== 2 passed in 0.06s =====================
```

Verbose and quiet output:

$ pytest -v basic/test_calc.py

```
=================== test session starts ====================
collecting ... collected 2 items

code/basic/test_calc.py::test_add PASSED
code/basic/test_calc.py::test_subtract PASSED

==================== 2 passed in 0.06s =====================
```

$ pytest -q basic/test_calc.py

```
..
2 passed in 0.06s
```

Passing `-k` to filter tests by name:

$ pytest -k subtract -v basic/test_calc.py

```
...
basic/test_calc.py::test_subtract PASSED

============= 1 passed, 1 deselected in 0.06s ==============
```

## raises

We expect a `ValueError` and one gets raised, so the test passes:

— basic/test_raises.py

```
def test_invalid_operator():
    with pytest.raises(ValueError):
        calc(1, 2, "?")
```

```
basic/test_raises.py::test_invalid_operator PASSED
```

If no exception gets raised by our code, the test fails:

```
def test_no_exception():
    with pytest.raises(ValueError):
        calc(1, 2, "+")
```

```
basic/test_raises.py::test_no_exception FAILED
...
    def test_no_exception():
>       with pytest.raises(ValueError):
             ^^^^^^^^^^^^^^^^^^^^^^^^^
E       Failed: DID NOT RAISE <class 'ValueError'>

...
```

Any other exception fails the test as normal:

```
def test_different_exception():
    with pytest.raises(ValueError):
        calc(1, 0, "/")
```

```
basic/test_raises.py::test_different_exception FAILED
...
basic/test_raises.py:24: in test_different_exception
    calc(1, 0, "/")
rpncalc/utils.py:13: in calc
    return a / b
           ^^^^^
E   ZeroDivisionError: division by zero
...
```

We can additionally match on the exception message:

```
def test_match():
    with pytest.raises(ValueError, match="Invalid operator"):
        calc(1, 2, "?")
```

```
...
code/basic/test_raises.py::test_match PASSED
```

## Marks

### parametrize

— marking/test_parametrization.py —

```
def test_multiply():
    assert calc(2, 3, "*") == 6


@pytest.mark.parametrize("a, b, expected", [
    (2, 3, 6),    # 2 * 3 = 6
    (3, 4, 12),   # 3 * 4 = 12
    (4, 5, 20),   # 4 * 5 = 20
])
def test_multiply_parametrized(a: int, b: int, expected: int):
    assert calc(a, b, "*") == expected
```

— $ pytest -v -k multiply marking/test_parametrization.py —

```
...
marking/test_parametrization.py::test_multiply PASSED
marking/test_parametrization.py::test_multiply_parametrized[2-3-6] PASSED
marking/test_parametrization.py::test_multiply_parametrized[3-4-12] PASSED
marking/test_parametrization.py::test_multiply_parametrized[4-5-20] PASSED
...
```

## Expanding the calculator example

### run-v1

No handling of `ZeroDivisionError`:

```
> 1
> 0
> /
Traceback (most recent call last):
  ...
  File ".../rpncalc/rpn_v1.py", line 15, in run
    self.evaluate(inp)
  File ".../rpncalc/rpn_v1.py", line 24, in evaluate
    res = calc(a, b, inp)
  File ".../rpncalc/utils.py", line 13, in calc
    return a / b
ZeroDivisionError: float division by zero
```

Same for not having enough values on the stack:

```
> 1
> +
Traceback (most recent call last):
  ...
  File ".../rpncalc/rpn_v1.py", line 15, in run
    self.evaluate(inp)
  File ".../rpncalc/rpn_v1.py", line 23, in evaluate
    a = self.stack.pop()
IndexError: pop from empty list
```

`+-` gets treated as valid operator but isn't:

```
> 1
> 2
> +-
Traceback (most recent call last):
  ...
  File ".../rpncalc/rpn_v1.py", line 15, in run
    self.evaluate(inp)
  File ".../rpncalc/rpn_v1.py", line 24, in evaluate
    res = calc(a, b, inp)
  File ".../rpncalc/utils.py", line 14, in calc
    raise ValueError("Invalid operator")
ValueError: Invalid operator
```

Invalid inputs silently get ignored:

```
> abcd
> efg
> p
[]
> q
```

Negative numbers and floats don't work:

```
> 0.5
> -1
> p
[]
> q
```

Digit-like characters cause issues as well:

Same underlying issue as above, but different manifestation:

```
> ²           (superscript digit)
Traceback (most recent call last):
  ...
  File ".../rpncalc/rpn_v1.py", line 31, in <module>
    rpn.run()
  File ".../rpncalc/rpn_v1.py", line 15, in run
```

```
    self.evaluate(inp)
  File ".../rpncalc/rpn_v1.py", line 19, in evaluate
    n = float(inp)
ValueError: could not convert string to float: '²'
```

### run-v2

Division by zero handled:

```
> 1
> 0
> /
Division by zero
> q
```

Same for an empty stack:

```
> 1
> +
Not enough operands
> q
```

Invalid inputs are reported:

```
> abcd
Invalid input: abcd
> q
```

`+-` and `²` are now invalid inputs:

```
> 1
> 2
> +-
Invalid input: +-
> ²
Invalid input: ²
> q
```

Negative numbers and floats are now correctly handled:

```
> 0.5
> -1
> p
[0.5, -1.0]
> q
```

Multiple input support:

```
> 1 2 +
3.0
> q
```

## Fixtures

### fixtures

— rpncalc/test_rpn_v2.py

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

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

## Built-in fixtures

### capturing

rpncalc/test_rpn_errors.py

```
def test_unknown_operator(
    rpn: RPNCalculator,
    op: str,
    capfd: pytest.CaptureFixture[str],
):
    rpn.stack = [1, 2]
    rpn.evaluate(op)
    captured = capfd.readouterr()
    assert captured.err == f"Invalid input: {op}\n"
```

How do we best deal with the final `\n` after the message, and with the expected operator being part of it? A seemingly simpler solution would have been to check `captured.err.startswith("Invalid input:")` or even `assert "Invalid input:" in captured.err`, but both make the test a bit less strict. Something like `assert captured.err.rstrip() == f"Invalid input: {op}"` is a bit better, but arguably just doing an exact `==` match with the expected string is simplest. In the end, we also want to ensure the newline *is actually there*, otherwise the next prompt would be printed in the same line as our error message!

```
def test_division_by_zero(
    rpn: RPNCalculator,
    capfd: pytest.CaptureFixture[str],
):
    rpn.stack = [1, 0]
    rpn.evaluate("/")
    captured = capfd.readouterr()
    assert captured.err == "Division by zero\n"


@pytest.mark.parametrize("stack", [[1], []])
def test_not_enough_operands(
    rpn: RPNCalculator,
    stack: list[int],
    capfd: pytest.CaptureFixture[str],
):
    rpn.stack = stack
    rpn.evaluate("+")
    captured = capfd.readouterr()
    assert captured.err == "Not enough operands\n"
```

## monkeypatch

### Simple solution testing 1+2

- We need to return a list of strings from our fake `get_inputs()`!
  While returning `[1, 2, "+", "q"]` works, it would be different to what actually happens in the real code, which makes our test somewhat useless.
- We need to have `"q"` as the last element, otherwise we would still be in the `while True:` in the `run` method and the test hangs.
- We don't actually quit anything (the `"q"` just returns from `run`), so we can easily check the stack after the run.

— rpncalc/test_rpn_v2.py —

```python
def test_run(rpn: RPNCalculator, monkeypatch: pytest.MonkeyPatch):
    # arrange
    monkeypatch.setattr(rpn, "get_inputs", lambda: ["1", "2", "+", "q"])
    # act
    rpn.run()
    # assert
    assert rpn.stack == [3]
```

We also could take a look at the printed output instead of (or in addition to) the stack:

```python
def test_run(
    rpn: RPNCalculator,
    monkeypatch: pytest.MonkeyPatch,
    capfd: pytest.CaptureFixture[str],
):
    # arrange
    monkeypatch.setattr(rpn, "get_inputs", lambda: ["1", "2", "+", "q"])
    # act
    rpn.run()
    # assert
    out, err = capfd.readouterr()
    assert out == "3.0\n"
    assert not err
```

### More complex solution with parametrization

We can now build on top of this idea for more sophisticated tests. With this approach, we get a little "framework" to easily write integration tests to test almost our complete calculator just by extending a parameterized test.

For every test case, we:

- Run the calculator with the given list of fake inputs.
- Make sure the stack looks as expected.
- Make sure the expected output and/or error output is printed.

We also use `pytest.param(...)` to give each test case a nice name.

— rpncalc/test_rpn_v2.py —

```python
@pytest.mark.parametrize("inputs, stack, output, error", [
    # calculations
    pytest.param(["1", "2", "+", "q"], [3], "3.0\n", "", id="add"),
    # printing the stack
    pytest.param(["1", "2", "p", "q"], [1, 2], "[1.0, 2.0]\n", "", id="print"),
    # error conditions
    pytest.param(["1", "0", "/", "q"], [], "", "Division by zero\n", id="div-zero"),
    pytest.param(
        ["1", "2", "+-", "q"], [1, 2], "", "Invalid input: +-\n", id="invalid-input"
    ),
    pytest.param(
        ["1", "+", "q"], [1], "", "Not enough operands\n", id="not-enough-operands"
    ),
],
)
def test_run(
    rpn: RPNCalculator,
    monkeypatch: pytest.MonkeyPatch,
    capsys: pytest.CaptureFixture[str],
    inputs: list[str],
    stack: list[int],
    output: str,
    error: str,
):
    monkeypatch.setattr(rpn, "get_inputs", lambda: inputs)
    rpn.run()
    out, err = capsys.readouterr()
    assert rpn.stack == stack
    assert out == output
    assert err == error
```

## Fixtures advanced

### yield

The `Client` just prints on `.connect()` and `.disconnect()`:

fixtures/test_yield_fixture.py

```python
class Client:
    def connect(self):
        print("\nConnecting...")

    def disconnect(self):
        print("\nDisconnecting...")
```

We can see how the teardown happens with `--setup-show`:

$ pytest fixtures/test_yield_fixture.py -v --setup-show

```
...
fixtures/test_yield_fixture.py::test_client_1
        SETUP    F connected_client
        fixtures/test_yield_fixture.py::test_client_1
            (fixtures used: connected_client)PASSED
        TEARDOWN F connected_client
fixtures/test_yield_fixture.py::test_client_2
        SETUP    F connected_client
        fixtures/test_yield_fixture.py::test_client_2
            (fixtures used: connected_client)PASSED
        TEARDOWN F connected_client
...
```

Or we could use `-s` instead to see the prints:

$ pytest fixtures/test_yield_fixture.py -v -s

```
...
fixtures/test_yield_fixture.py::test_client_1
Connecting...
in the test 1
PASSED
Disconnecting...

fixtures/test_yield_fixture.py::test_client_2
Connecting...
in the test 2
PASSED
Disconnecting...
...
```

Then we change the scope:

— fixtures/test_yield_fixture.py —

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

Setup and teardown will always be symmetric:

— $ pytest fixtures/test_yield_fixture.py -v --setup-show —

```
...
fixtures/test_yield_fixture.py::test_client_1
    SETUP    M connected_client
        fixtures/test_yield_fixture.py::test_client_1 ... PASSED
        fixtures/test_yield_fixture.py::test_client_2 ... PASSED
    TEARDOWN M connected_client
...
```

— $ pytest fixtures/test_yield_fixture.py -v -s —

```
...
fixtures/test_yield_fixture.py::test_client_1
Connecting...
in the test 1 ... PASSED
fixtures/test_yield_fixture.py::test_client_2 in the test 2 ... PASSED
Disconnecting...
...
```

Adding skipping:

```
@pytest.fixture
def connected_client() -> Iterator[Client]:
    client = Client()
    pytest.skip("Client not available")
    client.connect()
    yield client
    client.disconnect()
```

— $ pytest fixtures/test_yield_fixture.py -v —

```
...
fixtures/test_yield_fixture.py::test_client_1 SKIPPED
fixtures/test_yield_fixture.py::test_client_2 SKIPPED
...
```

## Some plugins

### coverage

```
$ pytest --cov=rpncalc/ --cov-report=term-missing rpncalc/
```

```
...
rpncalc/test_rpn_errors.py .....
rpncalc/test_rpn_v1.py ......
rpncalc/test_rpn_v2.py .................
rpncalc/test_utils.py .......

---------- coverage: platform linux, python ... -----------
Name                          Stmts   Miss  Cover   Missing
-----------------------------------------------------------
rpncalc/__init__.py               0      0   100%
rpncalc/convert.py               24     24     0%   1-35
rpncalc/rpn_v1.py                25      9    64%   8-15, 30-31
rpncalc/rpn_v2.py                46      6    87%   13-14, 59-62
rpncalc/rpn_v3.py                64     64     0%   1-86
rpncalc/test_rpn_errors.py       24      0   100%
rpncalc/test_rpn_v1.py           25      0   100%
rpncalc/test_rpn_v2.py           61      0   100%
rpncalc/test_utils.py            55      0   100%
rpncalc/utils.py                 32      2    94%   14, 22
-----------------------------------------------------------
TOTAL                           356    105    71%

...
```

```
$ pytest --cov=rpncalc/ --cov-report=html rpncalc/
```

```
...
---------- coverage: platform linux, python ... -----------
Coverage HTML written to dir htmlcov
...
```