pytest Cheat Sheet

Resources:
https://pragprog.com/titles/bopytest2/python-testing-with-pytest-second-edition/
https://cheatography.com/nanditha/cheat-sheets/pytest/
https://gist.github.com/kwmiebach/3fd49612ef7a52b5ce3a#file-pytest-md

Getting Started

Using virtual environment venv

Python virtual environments enable you to set up a Python sandbox with its own set of packages separate from the system site-packages in which to work.

Create

```
$ python -m venv env_dir_name
```

Activate

To activate on macOS and Linux.

```
$ source env_dir_name/bin/activate
```

To activate on Windows.

```
$ env_dir_name\Scripts\activate.bat
```

To activate on Windows with PowerShell.

```
$ env_dir_name\Scripts\Activate.ps1
```

Deactivate

When done, run:

```
$ deactivate
```

Using pip to install pytest

pip is the tool used to install Python packages, and it is installed as part of your Python installation.

Confirm pip version by running:

```
$ pip --version
```

You should see:

```
pip 24.0 from /Users/ ...
/env_dir_name/lib/python3.11/site-packages/pip (python 3.11)
```

Install pytest by running:

```
$ pip install pytest
```

Confirm the pytest version by running:

```
$ pytest --version
```

You should see:

```
pytest 8.1.1
```

Running the pytest testing framework

Run pytest with the following command:

```
$ pytest [options] [file_or_dir] [file_or_dir]
```

Show help message and configuration info:

```
$ pytest --help
```

Commands to run tests

Command	Description
`pytest test_one.py`	Run tests in a module.
`pytest tests/`	Run tests in a directory.
`pytest test_one.py::test_function`	Run a specific test within a module.
`pytest test_one.py::TestClass:test_method`	Run a specific method of a class.

Passing test output

```
$ pytest test_one.py
====================== test session starts =======================
collected 1 item
test_one.py . [100%]
======================= 1 passed in 0.01s ========================
```

The dot after `test_one.py` means that one test was run and it passed. The `[100%]` is a percentage indicator showing how much of the test suite is done so far. Because there is just one test in the test session, one test equals 100% of the tests. If you need more information, you can use `-v` or `--verbose`.

Failing test output

```
$ pytest test_two.py
===================== test session starts ======================
collected 1 item
test_two.py F [100%]
=========================== FAILURES ===========================
_________________________ test_failing _________________________
def test_failing():
> assert (1, 2, 3) == (3, 2, 1)
E assert (1, 2, 3) == (3, 2, 1)
E At index 0 diff: 1 != 3
E Use -v to get the full diff
test_two.py:2: AssertionError
=================== short test summary info ====================
FAILED test_two.py::test_failing - assert (1, 2, 3) == (3, 2, 1)
===================== 1 failed in 0.03s ========================
```

The failing test, `test_failing`, gets its own section to show us why it failed. And `pytest` tells us exactly what the first failure is: index 0 is a mismatch. If you have a color terminal, much of this appears in red to make it really stand out. This extra section shows exactly where the test failed and some of the surrounding code is called a *traceback*.

More or less verbosity

Option	Description
`-v, –verbose`	Increase verbosity
`-q, --quiet`	Decrease verbosity

Why pytest?

Here are a few of the reasons `pytest` stands out above many other testing frameworks:

- Simple tests are simple to write in `pytest`.
- Complex tests are still simple to write.
- Tests are easy to read.
- You can get started in seconds.
- You use `assert` in tests for verifications, not things like `self.assertEqual()` or `self.assertLessThan()`
- You can use `pytest` to run tests written for `unittest` or `nose`.

Features

This section covers a few features of `pytest`.

Test Discovery

Part of `pytest` execution finds which tests to run using *test discovery*. If we name them according to the default `pytest` naming conventions, `pytest` looks at your current directory and all subdirectories for test files and runs the test code it finds. The default naming conventions:

- Test files named `test_<something>.py` or `<something>_test.py`
- Test methods and functions should be named `test_<something>`
- Test classes should be named `Test<Something>`

There are ways to alter these discovery rules via `pytest` configuration.

Using assert Statements

`pytest` allows you to use the standard Python `assert` for verifying expectations and values in Python tests (`pytest` uses assert rewriting). These assert statements communicate test failure. Other testing frameworks use assert helper functions.

Assertions in pytest versus unittest

pytest	unittest
`assert something`	`assertTrue(something)`
`assert not something`	`assertFalse(something)`
`assert a == b`	`assertEqual(a, b)`
`assert a != b`	`assertNotEqual(a, b)`
`assert a is None`	`assertIsNone(a)`
`assert a is not None`	`assertIsNotNone(a)`
`assert a <= b`	`assertLessEqual(a, b)`
`assert 1 in [2, 3, 4]`	`assertNotIn(1, [2, 3, 4])`
`assert 'fizz' not in 'fizzbuzz'`	`assertNotIn('fizz', 'fizzbuzz')`

Failing Test Code

Tests can fail from:

- assertion failures
- from calls to `pytest.fail()`
- from any uncaught exception

When the test code calls `pytest.fail()`, it raises an exception and allows you to explicitly fail an executing test with a given message.

Test Outcomes

Pass and fail are not the only outcomes possible. Possible outcomes include:

- PASSED (.) — The test ran successfully.
- FAILED (F) — The test did not run successfully.
- SKIPPED (s) — The test was skipped. Skip a test by using decorators:
  - `@pytest.mark.skip()`
  - `@pytest.mark.skipif()`
- XFAIL (x) — The test was not supposed to pass, and it ran and failed.
  - Tell `pytest` that a test is expected to fail by using the `@pytest.mark.xfail()`
- XPASS (X) — The test was marked with `xfail`, but it ran and passed.
- ERROR (E) — An exception happened either during the execution of a fixture or hook function, and not during the execution of a test function.

For complete details on skip and xfail, see:
https://docs.pytest.org/en/latest/how-to/skipping.html

Capturing stdout/stderr Output

By default, during test execution any output sent to stdout and stderr is captured.
If a test or a setup method fails its corresponding captured output will usually be shown along with the failure traceback.

Use the `--capture=no` option to allow print statements to be seen even in passing tests.

NOTE: the option `-s` is a shortcut for `--capture=no`

For complete details, see:
https://docs.pytest.org/en/latest/how-to/capture-stdout-stderr.html

Traceback print mode

Use the `--tb=no` flag to turn off tracebacks, which is useful when you don't need the full output.

```
$ pytest --tb=no
===================== test session starts ======================
collected 2 items
test_one.py . [ 50%]
test_two.py F [100%]
=================== short test summary info ====================
FAILED test_two.py::test_failing - assert (1, 2, 3) == (3, 2, 1)
================= 1 failed, 1 passed in 0.03s ==================
```

The options are:

`--tb=auto` — (default) 'long' tracebacks for the first and last entry, but 'short' style for the other entries

`--tb=no` — no traceback at all
`--tb=long` — exhaustive, informative traceback formatting
`--tb=short` — shorter traceback format
`--tb=line` — only one line per failure
`--tb=native` — Python standard library formatting

For complete details on managing pytest's output, see:
https://docs.pytest.org/en/latest/how-to/output.html

Expected Exceptions

Use `pytest.raises()` to test for expected exceptions.

Example:
```
import pytest
import vending_machine

def test_insert_coin_when_quarter_is_string_raises():
    with pytest.raises(TypeError):
        machine.insert_coins('foobar')
```

If no exception is raised, the test fails.
If the test raises a different exception, it fails.

Grouping Tests with Classes

In most cases, use test classes only for the purpose of grouping tests. This will allow you to easily run the test methods together. Having test classes that use inheritance will likely confuse people. Running all tests in a class:
```
$ pytest path/test_module.py::TestClass
```

Running a Subset of Tests

Running small subsets of tests is useful, especially when debugging tests. pytest allows you to run a small batch of tests in many different ways:

All tests in a directory — `pytest path`
All tests in a module — `pytest path/test_module.py`
Single test function — `pytest path/test_module.py::test_function`
All tests in a class — `pytest path/test_module.py::TestClass`
Single test method — `pytest path/test_module.py::TestClass::test_method`
Tests matching a name pattern — `pytest -k EXPRESSION`

For the `-k` option, `pytest` will only run tests which match the given substring expression.

Example: `-k 'test_method or test_other'` matches all test functions and classes whose name contains 'test_method' or 'test_other'. And `-k 'not test_method'` matches those that don't contain 'test_method' in their names.

Fixtures

Fixtures are identified by the `@pytest.fixture()` decorated functions.

```
import pytest

@pytest.fixture()
def fixture_name():
    fixture_value = 'some data'
    return fixture_value

def test_something_with_fixture(fixture_name):
    assert fixture_name == 'some data'
```

Test functions or other fixtures depend on a fixture by putting its name in their parameter list.

Fixtures can return data using return or yield.

```
import pytest

@pytest.fixture()
def cards_db():
    with TemporaryDirectory() as db_dir:
        db_path = Path(db_dir)
        db = cards.CardsDB(db_path)
        yield db
        db.close()

def test_empty(cards_db):
    assert cards_db.count() == 0
```

Code before the yield is the setup code. Code after the yield is the teardown code.

Fixtures can be set to function, class, module, package, or session scope.
The default is function scope. You can even define the scope dynamically.

Multiple test functions can use the same fixture.
Multiple test modules can use the same fixture if it's in a `conftest.py` file.
Multiple fixtures at different scope can speed up test suites while maintaining test isolation.
Tests and fixtures can use multiple fixtures.

`pytest --setup-show` is used to see the order of execution.
`pytest --fixtures` is used to list available fixtures and where the fixture is located.

Test Parameterization

NOTE: the pytest keyword is spelled `parametrize`.

Three ways to parametrize tests:

1. Parametrize test functions, creating many test cases, by applying the `@pytest.mark.parametrize()` decorator.
2. Parametrize fixtures with `@pytest.fixture(params=())`
3. Generate complex parameterization sets with `pytest_generate_tests`.

Parametrize Test Functions

To parametrize a test function, add parameters to the test definition and use the `@pytest.mark.parametrize()` decorator to define the sets of arguments to test.

Example:

```
import pytest

@pytest.mark.parametrize(
   ('n', 'expected'), [
       (1, 2),
       (2, 3),
       (3, 4),
       (4, 5),
   ]
)
def test_increment(n, expected):
   assert n + 1 == expected
```

Run subsets of parameterized test cases using the `-k` option.

For details on parameterizing fixtures, see:
https://docs.pytest.org/en/latest/how-to/fixtures.html#parametrizing-fixtures

For details on complex parameterization, see:
https://docs.pytest.org/en/latest/how-to/parametrize.html#basic-pytest-generate-tests-example

For complete details on parametrization, see:
https://docs.pytest.org/en/latest/how-to/parametrize.html