METASNAKE

Testing in Python

Pytest

- Easy test creation
- Test runner
- Test selection
- Test parametrization
- Test fixtures
- Plugins

Installation

```
uv add --dev pytest
```

Running tests

```
uv run pytest
```

Test Creation

4 steps of a test:

1. Setup
2. Exercise
3. Verify
4. Teardown

In pytest these steps are usually done with:

1. Setup: Fixtures or setup methods
2. Exercise: Call the function or method to be tested
3. Verify: Use assert statements
4. Teardown: Fixtures or teardown methods

Example

```
!pip install ipytest
```

```
import ipytest
ipytest.autoconfig()
```

```
%%ipytest -qq

def add(a, b):
    return a + b
```

```
def test_add():
    assert add(1, 2) == 3
    assert add(2, 3) == 5
```

```
%%ipytest -qq

def add(a, b):
    return a + b

def test_that_fails():
    assert add(1, 2) == 3
    assert add(2, 3) == 6
```

Test Layout

Tests are usually placed in a `tests` directory. Generally it does not need to be in the same directory as the code being tested. But the code that is being tested should be importable from the test directory. (This is why I like to do an editable install of the package I am testing.)

add `tests/test_basic.py`:

```
import sk_stepwise as sw


def test_initialization():
    model = None
    rounds = []
    optimizer = sw.StepwiseHyperoptOptimizer(model, rounds)
    assert optimizer is not None
```

Then run:

```
uv run pytest
```

The output should be:

```
% uv run pytest
==================== test session starts ============================================
platform darwin -- Python 3.12.5, pytest-8.3.3, pluggy-1.5.0
rootdir: /private/tmp/sk-stepwise
configfile: pyproject.toml
collected 1 item

tests/test_basic.py .                                                          [100%]

================ warnings summary ===================================================
.venv/lib/python3.12/site-packages/hyperopt/atpe.py:19
  /private/tmp/sk-stepwise/.venv/lib/python3.12/site-packages/hyperopt/atpe.py:19: DeprecationWarning: pkg_resources is deprecated as an
API. See https://setuptools.pypa.io/en/latest/pkg_resources.html
    import pkg_resources

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
============ 1 passed, 1 warning in 0.66s ===========================================
matt@Matts-MacBook-Pro-4 sk-stepwise % cat tests/test_basic.py
```

Test Output

- `.` - test passed
- `F` - test failed
- `E` - test had an exception during fixture setup or teardown
- `s` - test was skipped
- `x` - expected failure
- `X` - unexpected success (should have failed)

```
%%ipytest?
```

```
%%ipytest -qq
# the -qq is for quiet mode, which suppresses the output of the tests

import os
import pytest


def test_period():
    assert 1 == 1

@pytest.fixture
def fail_fixture():
    raise NotImplementedError

def test_E(fail_fixture):
    assert 1 == 1

def test_skip():
    if os.name == 'posix':
        pytest.skip('skipping this test on posix')

@pytest.mark.xfail
def test_x():
    # assuming 3.14 adds the method .fancy_split() to the str class
    assert '1,2-4'.fancy_split() == ['1', '2', '3', '4']

@pytest.mark.xfail
def test_X():
    assert ' 1'.lstrip() == '1'
```

Different Outputs

```
import sk_stepwise as sw
import pytest


def test_initialization():
    model = None
    rounds = []
    optimizer = sw.StepwiseHyperoptOptimizer(model, rounds)
    assert optimizer is not None

def test_that_fails():
    assert 'matt' == 'fred'

@pytest.fixture
def one():
    return 1/0
```

```
def test_with_exception(one):
    assert one == 1

@pytest.mark.xfail(raises=TypeError)
def test_logistic():
    from sklearn import linear_model
    model = linear_model.LinearRegression()
    rounds = []
    opt = sw.StepwiseHyperoptOptimizer(model, rounds)
    X = [[0,1], [0,2]]
    y = [1, 0]
    opt.fit(X, y)
```

The assert Statement

Pytest rewrote the assert statement to give more information when it fails. It works with different types to give relevant information.

```
%%ipytest -vv

import pytest

def test_number():
    assert 1 == 1

def test_string():
    assert 'The cold brown fox ate a bird' == 'The jumping brown fox ate a bird'

def test_string_in():
    assert 'b' in 'matt'

def test_list_in():
    assert 5_000 in list(range(1000))

def test_raise():
    # no assertion in this one
    with pytest.raises(ZeroDivisionError):
        1 / 0
```

```
pytest.raises??
```

```
from _pytest import python_api
```

```
python_api.RaisesContext??
```

```
%%ipytest -qq

# Can provide addtional text upon failure
import pandas as pd

@pytest.fixture
def sales():
    return pd.DataFrame({'sales': [1, 2, 3, 1, 2, 3],
                         'date': pd.date_range('2020-01-01', periods=6)})

def test_number(sales):
    assert isinstance(sales.index, pd.DatetimeIndex), 'Should be a timeseries index'
```

Test Runner

`pytest` assumes that the code you are testing is installed. It does not add the current directory to the path. So you need to install the package you are testing. You can do this with `pip install -e .` or `uv run pip install -e .`. If you run `python -m pytest` it will add the current directory to the path.

Pytest searches the current directory and subdirectories for files that start with `test_` or end with `_test.py`. You can specify `testpaths` in the `pytest.ini` file to specify where to look for tests.

Command line options:

- `--doctest-modules` - run doctests in the module
- `--doctest-glob=*.md` - run doctests in markdown files
- `--pdb` - drop into the debugger on test failure
- `-v` - verbose output (show NODEID)
- `-q` - quiet output
- `-m EXPR` - run tests with marks that match the expression
- `-k EXPRESSION` - run tests with names that match the expression
- `NODEID` - run a specific test

Debugging Tests

By default, pytest will hide the output of a test that passes. You can use the `-s` option to show the output of a passing test.

You can use the `--pdb` option to drop into the debugger on a test failure.

Other options:

- `-l` - show local variables
- `--lf` - run the last failed test
- `--maxfail=2` - stop after 2 failures
- `-v` - show NODEID of tests
- `-x` - stop after the first failure (`--maxfail=1`)

Careful with Output

You don't want to print `-l` in CI if you have sensitive information in your tests. (Like secret keys.)

Hint

Consider combining `-x` and `--lf` to stop after the first failure and rerun the last failed test.

Doctests

Python has a built-in doctest module that can be used to test code in docstrings. Any code in a docstring that starts with `>>>` will be run and the output will be compared to the following lines.

Here is a function with a simple doctest:

```
def add(a, b):
    """
```

```
    Add two numbers together.

    >>> add(1, 2)
    3
    >>> add(3, 4)
    7
    """
    return a + b
```

You can run the doctests with `python -m doctest -v file.py` or `pytest --doctest-modules`.

```
%%ipytest --doctest-modules

def add(a, b):
    """
    Add two numbers together

>>> add(1, 2)
    3
>>> add(2, 3)
    5
    """
    return a + b

def code_with_bad_docs(x,y):
    """
    Add two numbers together

>>> add(1, 2)
    1
>>> add(2, 3)
    5
    """
    return x + y
```

Fixtures in Doctest

If you have a fixture defined in `conftest.py` you can use it in a doctest with the `getfixture` function.

```
%%writefile conftest.py

import pytest
import pandas as pd

@pytest.fixture
def sales():
    return pd.DataFrame({'sales': [1, 2, 3, 1, 2, 3],
                         'date': pd.date_range('2020-01-01', periods=6)})
```

```
%%writefile test_sales.py

def agg_sales(df):
    """
    Aggregate sales data
>>> data = getfixture('sales') # needs to be a string
>>> agg_sales(data)
                sales
    date
    2020-01-01      1
    2020-01-02      2
    2020-01-03      3
    2020-01-04      1
```

```
    2020-01-05      2
    2020-01-06      3
    """
    return (df.groupby('date').sum())
```

```
!pytest --doctest-modules test_sales.py -v
============================= test session starts
     ==============================
platform darwin -- Python 3.10.14, pytest-7.2.0, pluggy-1.0.0 --
     /Users/matt/.envs/menv/bin/python3.10
cachedir: .pytest_cache
hypothesis profile 'default' ->
     database=DirectoryBasedExampleDatabase('/Users/matt/Dropbox/work/courses
    /ms-courses/professionalpython/03-Testing/.hypothesis/examples')
rootdir:
     /Users/matt/Dropbox/work/courses/ms-courses/professionalpython/03-Testin
    g
plugins: dash-2.11.1, timeout-2.1.0, pytest_check_links-0.8.0,
     cov-4.0.0, hypothesis-6.81.2, console-scripts-1.3.1, anyio-3.6.2,
     typeguard-4.0.0, mock-3.14.0
collected 1 item

test_sales.py::test_sales.agg_sales PASSED
     [100%]

============================== 1 passed in 0.12s
     ===============================

!pytest test_sales.py -v
============================= test session starts
     ==============================
platform darwin -- Python 3.10.14, pytest-7.2.0, pluggy-1.0.0 --
     /Users/matt/.envs/menv/bin/python3.10
cachedir: .pytest_cache
hypothesis profile 'default' ->
     database=DirectoryBasedExampleDatabase('/Users/matt/Dropbox/work/courses
    /ms-courses/professionalpython/03-Testing/.hypothesis/examples')
rootdir:
     /Users/matt/Dropbox/work/courses/ms-courses/professionalpython/03-Testin
    g
plugins: dash-2.11.1, timeout-2.1.0, pytest_check_links-0.8.0,
     cov-4.0.0, hypothesis-6.81.2, console-scripts-1.3.1, anyio-3.6.2,
     typeguard-4.0.0, mock-3.14.0
collected 0 items

============================ no tests ran in 0.11s
     ==============================
```

Doctest Warts

Doctests are whitespace sensitive. If you have a function that returns a string with a newline at the end, you need to include that newline in the doctest. If you have trailing whitespace in a doctest, it will fail.

```
%%ipytest --doctest-modules -k trailing_whitespace

import pytest

def trailing_whitespace():
    """
    Test that trailing whitespace is removed

>>> print(trailing_whitespace())
```

```
    no whitespace
    """
    return 'no whitespace '
```

```
%%ipytest --doctest-modules -k heading

import pytest

def heading(value):
    """
    Test that heading is added

    This works:
>>> print(heading('GOOD'))
    <BLANKLINE>
    # GOOD
    <BLANKLINE>

    This does not:
>>> print(heading('heading'))

    # heading

    """
    return f"\n# {value}\n"
```

Test Selection

Marking Tests

You can *mark* tests with a decorator to give them attributes. You can then run tests based on these attributes.

```
%%ipytest -k slow
import pytest
import time

@pytest.mark.slow
def test_slow():
    time.sleep(1) # simulate a slow test
    assert 1 == 1

@pytest.mark.slow
def test_slow2():
    time.sleep(1)
    assert 2 == 2

def test_normal():
    assert 3 == 3
```

```
%%ipytest -k "not slow"
import pytest
import time

@pytest.mark.slow
def test_slow():
    time.sleep(1) # simulate a slow test
    assert 1 == 1

@pytest.mark.slow
```

```
def test_slow2():
    time.sleep(1)
    assert 2 == 2

def test_normal():
    assert 3 == 3
```

```
%%ipytest -k "time and not normal"
import pytest
import time

# mark the whole module
pytestmark = pytest.mark.time

@pytest.mark.slow
def test_slow():
    time.sleep(1) # simulate a slow test
    assert 1 == 1

@pytest.mark.slow
def test_slow2():
    time.sleep(1)
    assert 2 == 2

@pytest.mark.normal
def test_normal():
    assert 3 == 3
```

Registering Marks

Python is a language of typos. If you mistype a mark, pytest will not complain. You can register marks in a `pytest.ini` file to catch these typos.

```
[pytest]
markers =
    slow: mark a test as slow
    fast: mark a test as fast
```

Run `pytest --markers` to see the registered marks.

If you run `pytest --strict-markers` it will fail if you use an unregistered mark.

```
%%ipytest -k "time and not normal" --strict-markers
import pytest
import time

# mark the whole module
pytestmark = pytest.mark.time

@pytest.mark.slow
def test_api():
    time.sleep(1) # simulate a slow test
    assert 1 == 1

@pytest.mark.slow
def test_db():
    time.sleep(1)
    assert 2 == 2

@pytest.mark.normal
def test_local():
    assert 3 == 3
```

Built-in Marks

Pytest has some built-in marks:

- `skip` - skip a test
- `skipif` - skip a test if a condition is true
- `xfail` - expect a test to fail

```
%%ipytest

import pytest

@pytest.mark.skip(reason="no way of currently testing this")
def test_the_unknown():
    assert 1 == 1


@pytest.mark.skipif(os.environ.get('USER') == 'matt',
                    reason='Matt is not allowed to run this test')
def test_user():
    assert 1 == 1


@pytest.mark.xfail
def test_x():
    assert ' 1'.lstrip() == '1'
```

Test Parametrization

You can run the same test with different parameters using the `@pytest.mark.parametrize` decorator.

```
%%ipytest

def parse_num_seq(txt):
    """
    Parse a string of numbers separated by commas

>>> parse_num_seq('1,2,3')
    [1, 2, 3]
>>> parse_num_seq('1, 2, 3')
    [1, 2, 3]
    """
    return [int(x) for x in txt.split(',')]

def test_parse_num_seq():
    assert parse_num_seq('1,2,3') == [1, 2, 3]

def test_parse_num_seq2():
    assert parse_num_seq('1, 2, 4') == [1, 2, 4]

def test_parse_num_seq3():
    assert parse_num_seq('3,10, 20') == [3, 10, 20]
```

```
%%ipytest -v

def parse_num_seq(txt):
    """
    Parse a string of numbers separated by commas
```

```
>>> parse_num_seq('1,2,3')
    [1, 2, 3]
>>> parse_num_seq('1, 2, 3')
    [1, 2, 3]
    """
    return [int(x) for x in txt.split(',')]

@pytest.mark.parametrize('txt, expected', [
    ('1,2,3', [1, 2, 3]),
    ('1, 2, 4', [1, 2, 4]),
    ('3,10, 20', [3, 10, 20])
])
def test_parse_num_seq(txt, expected):
    assert parse_num_seq(txt) == expected
```

Note the NODEID in the output. This is the name of the test that was run.

```
% pytest -v
================================== test session starts ==========================================================================
platform darwin -- Python 3.10.14, pytest-7.2.0, pluggy-1.0.0 -- /Users/matt/.envs/menv/bin/python3.10
cachedir: .pytest_cache
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/private/tmp/foo/.hypothesis/examples')
rootdir: /private/tmp/foo
plugins: dash-2.11.1, timeout-2.1.0, pytest_check_links-0.8.0, cov-4.0.0, hypothesis-6.81.2, console-scripts-1.3.1, anyio-3.6.2,
typeguard-4.0.0, mock-3.14.0
collected 3 items

test_parse.py::test_parse_num_seq[1,2,3-expected0] PASSED            [ 33%]
test_parse.py::test_parse_num_seq[1, 2, 4-expected1] PASSED          [ 66%]
test_parse.py::test_parse_num_seq[3,10, 20-expected2] PASSED         [100%]
```

Fixtures

Fixtures are a way to set up and tear down resources for tests. They can be used to set up a database connection, create a temporary directory, or set up a model for testing.

```
%%ipytest

import pytest

def add(a, b):
    return a + b

@pytest.fixture
def large_num():
    return 1e20

def test_large(large_num):
    assert add(large_num, 1) == \
        large_num
```

```
type(1e20)
```

```
%%ipytest

#method fixture

def adder(a, b):
    return a + b
```

```
class TestAdder:

    @pytest.fixture
    def other_num(self):
        return 42

    def test_other(self, other_num):
        assert adder(other_num, 1) == 43
```

```
%%ipytest --fixtures

# show out of the box fixtures and installed fixtures
```

Fixture Tear Down

There are a few ways to tear down a fixture:

- `yield` - yield the fixture and run the teardown code after the test
- `addfinalizer` - add a finalizer to the fixture
- Use `setup` and `teardown` methods (`setup_module`/`setup_function`/`setup_class`/`setup_method`)

```
# stick some data in parquet

import pandas as pd

df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})
df.to_parquet('test.parquet')
```

```
%%ipytest
import duckdb
import pytest

@pytest.fixture
def duckdb_con():
    con = duckdb.connect()
    yield con
    con.close()

def test_query(duckdb_con):
    df = duckdb_con.execute('SELECT * FROM test.parquet').fetchdf()
    assert df.shape == (3, 2)
```

```
%%ipytest

# use addfinalizer to close the connection
@pytest.fixture
def duckdb_con(request):
    con = duckdb.connect()
    request.addfinalizer(con.close)
    return con

def test_query(duckdb_con):
    df = duckdb_con.execute('SELECT * FROM test.parquet').fetchdf()
    assert df.shape == (3, 2)
```

```
%%ipytest
```

```
import duckdb
import pytest

con = None

def setup_module():
    # called once for the module
    global con
    con = duckdb.connect(database=':memory:')

def teardown_module():
    con.close()

def test_query():
    result = con.execute('SELECT * FROM test.parquet')
    assert result.fetchone() == (1,4)

def test_query2():
    result = con.execute('SELECT sum(a) FROM test.parquet')
    assert result.fetchone() == (6,)
```

Fixture Scope

Fixtures can have different scopes:

- `function` - run once per test (default)
- `class` - run once per class
- `module` - run once per module
- `session` - run once per session

```
%%ipytest
import duckdb
import pytest

@pytest.fixture(scope='session')
def duckdb_con():
    con = duckdb.connect()
    yield con
    con.close()

def test_query(duckdb_con):
    df = duckdb_con.execute('SELECT * FROM test.parquet').fetchdf()
    assert df.shape == (3, 2)
```

```
%%ipytest
# bad fixture depend
@pytest.fixture(scope='function')
def two():
    return 2

@pytest.fixture(scope='session')
def four(two):
    return two * two

def test4(four):
    assert four == 4
```

```
%%ipytest
# trigger skip from fixture
import os
import duckdb
import pytest

@pytest.fixture(scope='session')
def duckdb_con():
    if not os.path.exists('test.parquet'):
        pytest.skip('no test database')
    con = duckdb.connect()
    yield con
    con.close()

def test_query(duckdb_con):
    df = duckdb_con.execute('SELECT * FROM test.parquet').fetchdf()
    assert df.shape == (3, 2)
```

```
%%ipytest -s
# pass data from marks to fixture

import os
import duckdb
import pytest

@pytest.fixture
def duckdb_con(request):
    # doesn't work if scope is session or module
    mark = request.node.get_closest_marker('dbfile')
    if mark is not None:
        name = mark.args[0]
    else:
        name = None
    if name is None or not os.path.exists(name):
        pytest.skip('no test database')
    con = duckdb.connect()
    return con

@pytest.mark.dbfile('test.parquet')
def test_query(duckdb_con, request):
    db_name = request.node.get_closest_marker('dbfile').args[0]
    df = duckdb_con.execute(f'SELECT * FROM {db_name}').fetchdf()
    assert df.shape == (3, 2)

@pytest.mark.dbfile('test.csv')
def test_query2(duckdb_con):
    db_name = request.node.get_closest_marker('dbfile').args[0]
    df = duckdb_con.execute(f'SELECT * FROM {db_name}').fetchdf()
    assert df.shape == (3, 2)

def test_query3(duckdb_con):
    df = duckdb_con.execute('SELECT * FROM test.parquet').fetchdf()
    assert df.shape == (3, 2)
```

Monkeypatch

You can use the `monkeypatch` fixture to change the behavior of a function. This is useful for testing functions that call external services or functions that have side effects.

I prefer to use this instead of mocking because I find it easier to understand what is happening.

- *`monkeypatch.setattr()`*: Replaces functions or class methods with custom versions (e.g., lambdas), useful for mocking behaviors and testing edge cases.
- *`monkeypatch.setenv()`*: Modifies environment variables, making it easy to configure test settings that depend on external environments.
- *`monkeypatch.delattr()`*: Removes attributes temporarily, ideal for testing scenarios where attributes are absent.
- *`monkeypatch.delenv()`*: Deletes environment variables temporarily for testing purposes.
- All changes made with `monkeypatch` are temporary and only apply for the duration of the test, ensuring no side effects on other tests.

```
%%ipytest
import math

def test_sin(monkeypatch):
    monkeypatch.setattr(math, 'sin', lambda x: 42)
    assert math.sin(0) == 42

def test_sin_normal():
    assert math.sin(0) == 0
```

Pytest Configuration

- Rootdir
  - Nodeid determined by the rootdir
  - Plugins may store data in the rootdir
  - Normally the rootdir is the directory where you run `pytest`

Can put configuration in `pytest.ini` or in `pyproject.toml` (as of pytest 6.0)

Common configuration options:

- `minversion` - minimum version of pytest
- `addopts = -v --strict-markers` - additional command line options
- `testpaths` - directories to search for tests
- `markers` - marks to register

Example `pyproject.toml`:

```
[tool.pytest.ini_options]
minversion = "6.0"
addopts = "-v --strict-markers"
testpaths = ["tests"]
markers = [
    "slow: mark a test as slow",
    "fast: mark a test as fast"
]
```

`conftest.py`

`conftest.py` is a file that pytest looks for in the current directory and all parent directories. It can be used to define fixtures, marks, hooks, and plugins.

Pytest Plugins

Pytest has a rich plugin ecosystem. You can find plugins for:

- pytest-cov - Code coverage
- pytest-xdist - run tests in parallel
- pytest-asyncio - asyncio support
- pytest-timeout - add a timeout to tests