Using Pytest

please first ensure you have installed pytest in Venv via
python -m pip install pytest

Unit test:
- implement function parameters, return
- find examples and the expected return value
- write a test calling the examples
- fails → correct code
- success → finish

you may want to test edge cases. (usual cases are called *use cases*)

Setting up your PyTest

Step 1. create a new file with _test in the name.
e.g. exercise1_test.py

Step 2. set up test module
import pytest
import tested fucntions
from xxx import function
or
import xxx (the file with functions to test)

name your test by "test_function"
def test_function():
    ... # use the function
    assert ... # write the condition for test to pass

if you have several cases to test on a function, you may want to add additional information in test name. e.g.
test_matmul_empty, test_matmul_sparse, test_matmul_non_symmetric

if there are too many tests, group them into classes
class TestBunch1:
    # define some class attributes
    value1 = 0
    value2 = 10

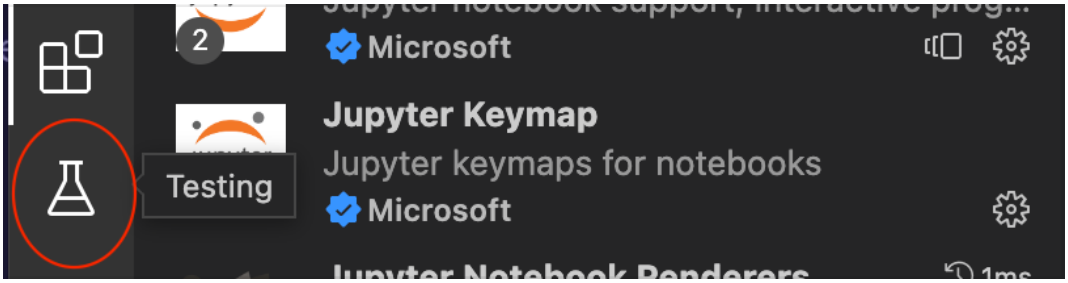
    def test:

....

attributes will be shared between tests

Step 3. go to terminal
$ python -m pytest filePath

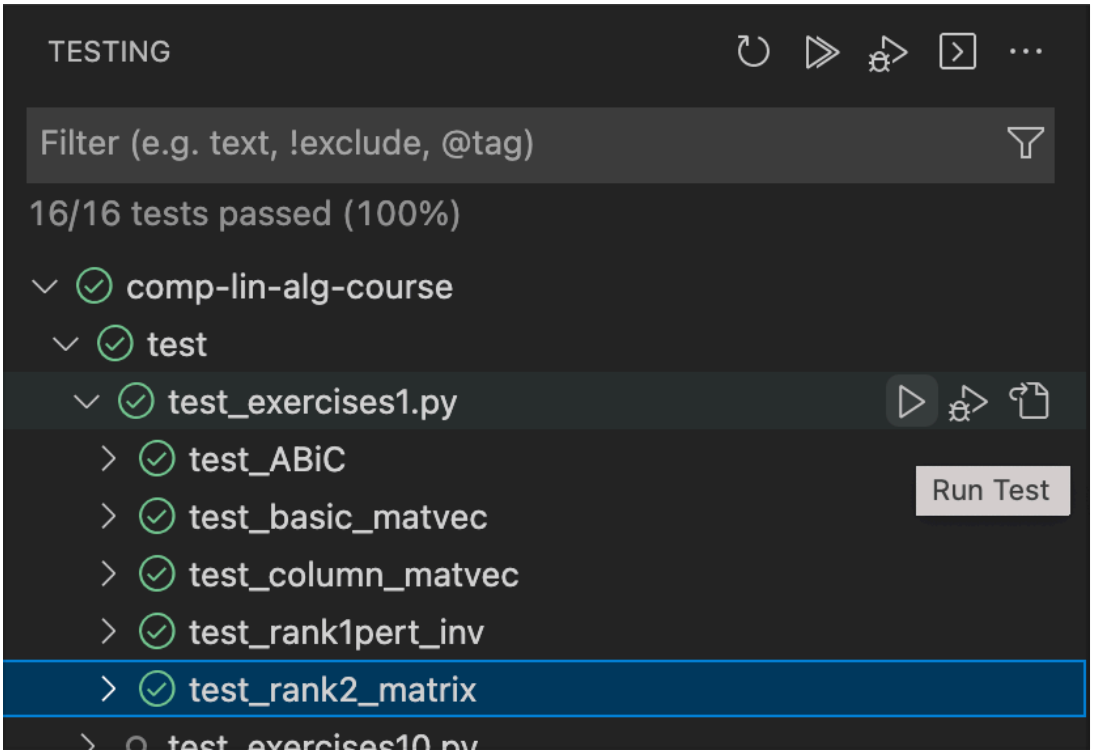
If you have python extension by microsoft in VScode. (click setting — extensions to check), then test tool will display on left sidebar



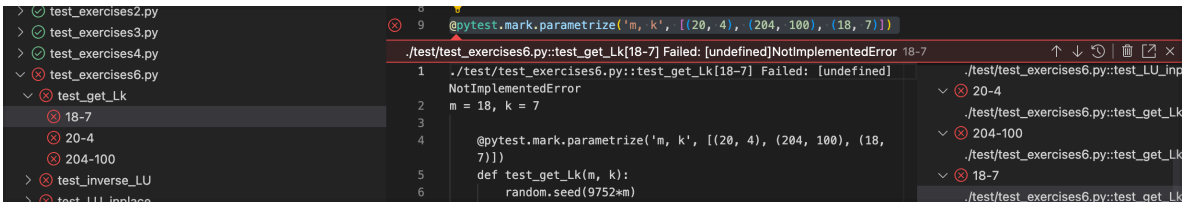
You may have to configure (tell vsc where are the test files and how are they named) first

- for this course, select pytest framework NOT standard python testing framwork!!

it automatically runs through all tests and indicate which cases fail.



clicking failed cases shows the test function and shows you the exact error



possible test discovery errors:

- due to syntax error
- not configured correctly, check settings.json

Implementation

Good implementation of function reduces time of debugging:
Think before writing code:

1. usual input, invalid inputs (throw exception)
2. valid but edge cases e.g. array of length 1, or empty list
3. expected return value

write input, output meaning and their data types clearly in docstring

KeyPoints

your own test must cover enough cases:
2+ use cases, 1+ edge cases

Using Marks

marks are added above functions, or for the whole class, to give better control on tests.
pytest —markers # check all built-in markers

register a custom mark
def pytest_configure(config):
  config.addinivalue_line(
    "markers", “slow: mark tests as slow”
)

common built-in useful fixtures:

- fixture
- filterwarnings
- skip, skipif
- parametrize

Filter warnings

method 1: use -W
pytest -W “ignore:...” filePath

method 2: mark a test function
@pytest.mark.filterwarnings(“ignore: ...”)
def test_function():
    ...

warning filters are in the format *action:message:category:module:line*
possible actions:
default - print first warning
error - turn warnings into exceptions
ignore - never show warnings
always - print all warnings
module - print first warning in each module

other four arguments can be ignored:
message: a string that start of warning message must match, case-sensitive
category: choose a class of waning e.g. RuntimeWarning
module: matching module name, case-sensitive
line: integer, match warning occurrence line. if line = 0, all lines matched

common filters:
ignore - ignore all warnings
error - convert all warnings to error
error::DeprecationWarning - treat convert DeprecationWarning as errors
ignore, default:::module1 - only show warnings in module1
error:::module1[.*] - convert warnings in module1 to errors

Categories of warnings:
Warning, UserWarning, DeprecationWarning, SyntaxWarning, RuntimeWarning, FutureWarning, ImportWarning, UnicodeWarning, BytesWarning, ResourceWarning

skip, skipif
skip with reason, can skip without reason
@pytest.mark.skip(reason=“test later”)
skip whole module
@pytest.mark.skip(reason=“test later”, allow_module_level=True)

skip with condition
@pytest.mark.skip(sys.version_info < (3, 10), reason=“requires python3.10

or higher")
@pytest.mark.skip(sys.platform == "win32", reason="not compatible with windows")

Parametrize

send multiple parameters to test function
@pytest.mark.parametrize("m, n, p", [(3, 2, 2), (10, 7, 8), (20, 15, 14)]
def test_matmul(m, n, p)
    ... # test matrix multiplication for matrices of size m × n with n × p

Note: passed values will be in-place(shallow copy). e.g. if a list *fruits* is passed, and test codes change the list, the original list *fruits* will be changed.

parametrize all tests in a class
@pytest.mark.parametrize("m, n, p", [(3, 2, 2), (10, 7, 8), (20, 15, 14)]

class TestMatmul
    def test_matmul_usual(self, m, n, p):
        ...
    def test_matmul_edge(self, m, n, p):
        ...

Fixtures

set some fixed things to tests, e.g. parameters, settings

@pytest.fixture
def initial_values():
    # set up some initial values for tests
    v = np.random.rand(10)
    w = np.random.rand(10)
    A = np.random.rand((10, 10))

def test_dot(initialvalues):
    assert ... # here you can use v, w, A defined by fixture initial_values above.

the following activates dataBase automatically, without putting it in every test function
@pytest.fixture(autouse=True)
def dataBase():
    # get data base required

fixtures are useful for connecting to networks, getting test cases for machine learning model training.

set scope, how long fixture will be activated
@pytest.fixture(scope='...')
function — terminates after function calling this fixture is closed
module
class
package
session — only terminates after pytest session is closed