Supplementary PyTest Section

John Strickler

Version 1.0, March 2020

Table of Contents

Chapter 1: Unit Tests with pytest

Objectives

- Understand the purpose of unit tests
- Design and implement unit tests with pytest
- Run tests in different ways
- Use builtin fixtures
- Create and use custom fixtures
- Mark tests for running in groups
- Learn how to mock data for tests

What is a unit test?

- Tests *unit* of code in isolation
- Ensures repeatable results
- Asserts expected behavior

A *unit test* is a test which asserts that an isolated piece of code (one function, method, class, or module) has some expected behavior. It is a way of making sure that code provides repeatable results.

There are four main components of a unit testing system:

1. Unit tests – individual assertions that an expected condition has been met
2. Test cases – collections of related unit tests
3. Fixtures — provide data to set up tests in order to get repeatable results
4. Test runners – utilities to execute the tests in one or more test cases

Unit tests should each test one aspect of your code, and each test should be independent of all other tests, including the order in which tests are run.

Each test asserts that some condition is true.

Unit tests may collected into a **test case**, which is a related group of unit tests. With pytest, a test case can be either a module or a class. This is optional in **pytest**.

Fixtures provide repeatable, known input to a test.

The final component is a **Test runner**, which executes one, some, or all tests and reports on the results. There are many different test runners for pytest. The builtin runner is very flexible.

The pytest module

- Provides
  - test runner
  - fixtures
  - special assertions
  - extra tools
- Not based on xUnit[1]

The **pytest** module provides tools for creating, running, and managing unit tests.

Each test supplies one or more **assertions**. An assertion confirms that some condition is true.

Here's how **pytest** implements the main components of unit testing:

unit test
: A normal Python function that uses the **assert** statement to assert some condition is true

test case
: A class *or* a module than contains unit tests (tests can be grouped with *markers*.

fixture
: A special parameter of a unit test function that provides test resources (fixtures can be nested)

test runner
: A text-based test runner is built in, and there are many third-party test runners

pytest is more flexible than classic xUnit implementations. For example, fixtures can be associated with any number of individual tests, or with a test class. Test cases need not be classes.

[1] The builtin unit testing module, **unittest**, *is* based on **xUnit** patterns, as implemented in Java and other languages.

Creating tests

- Create test functions
- Use builtin **assert**
- Confirm something is true
- Optional message

To create a test, create a function whose name begins with "test". These should normally be in a separate script, whose name begins with "test_" or ends with "_test". For the simplest cases, tests do not even need to import **pytest**.

Each test function should use the builtin **assert** statement one or more times to confirm that the test passes. If the assertion fails, the test fails.

pytest will print an appropriate message by introspecting the expression, or you can add your own message after the expression, separated by a comma

It is a good idea to make test names verbose. This will help when running tests in verbose mode, so you can see what tests are passing (or failing).

```
assert result == 'spam'
assert 2 == 3, "Two is not equal to three!"
```

Example

pytests/test_simple.py

```
#!/usr/bin/env python

def test_two_plus_two_equals_four():  ①
    assert 2 + 2 == 4   #  ②
```

① tests should begin with "test" (or will not be found automatically)

② if **assert** statement succeeds, the test passes

Running tests (basics)

- Needs a test runner
- **pytest** provides **pytest** script

To actually run tests, you need a *test runner*. A test runner is software that runs one or more tests and reports the results.

pytest provides a script (also named **pytest**) to run tests.

You can run a single test, a test case, a module, or all tests in a folder and all its subfolders.

```
pytest test_⋯..py
```

to run the tests in a particular module, and

```
pytest -v test_⋯..py
```

to add verbose output.

By default, pytest captures anything written to stdout/stderr. If you want to see the output of print() statements in your tests, add the **-s** option, which turns off output capture.

```
pytest -s ⋯
```

NOTE In older versions of pytest, the test runner script was named **py.test**. Newer versions support that name, but the developers recommend only using **pytest**.

TIP PyCharm automatically detects a script containing test cases. When you run the script the first time, PyCharm will ask whether you want to run it normally or use its builtin test runner. Use **Edit Configurations** to modify how the script is run. Note: in PyCharm's settings, you can select the default test runner to be **pytest**, **Unittest**, or other test runners.

Special assertions

- Special cases
  - pytest.raises()
  - pytest.approx()

There are two special cases not easily handled by **assert**.

pytest.raises

For testing whether an exception is raised, use **pytest.raises()**. This should be used with the **with** statement:

```
with pytest.raises(ValueError):
    w = Wombat('blah')
```

The assertion will succeed if the code inside the **with** block raises the specified error.

pytest.approx

For testing whether two floating point numbers are *close enough* to each other, use **pytest.approx()**:

```
assert result == pytest.approx(1.55)
```

The default tolerance is 1e-6 (one part in a million). You can specify the relative or absolute tolerance to any degree. Infinity and NaN are special cases. NaN is normally not equal to anything, even itself, but you can specify `nanok=True` as an argument to approx().

NOTE See https://docs.pytest.org/en/latest/reference.html#pytest-approx for more information on pytest.approx()

Example

pytests/test_special_assertions.py

```
#!/usr/bin/env python
import pytest
import math

FILE_NAME = 'IDONOTEXIST.txt'

def test_missing_filename():
    with pytest.raises(FileNotFoundError):  ①
        open(FILE_NAME)  ②

def test_list():
    print()
    assert (.1 + .2) == pytest.approx(.3)  ③

def test_approximate_pi():
    assert 22 / 7 == pytest.approx(math.pi, .001)  ④
```

① assert FileNotFoundError is raised inside block

② will fail test if file is not found

③ fail unless values are within 0.000001 of each other (actual result is 0.30000000000000004)

④ Default tolerance is 0.000001; smaller (or larger) tolerance can be specified

Fixtures

- Provide resources for tests
- Implement as functions
- Scope
  - Per test
  - Per class
  - Per module
- Source of fixtures
  - Builtin
  - User-defined

When writing tests for a particular object, many tests might require an instance of the object. This instance might be created with a particular set of arguments.

What happens if twenty different tests instantiate a particular object, and the object's API changes? Now you have to make changes in twenty different places.

To avoid duplicating code across many tests, pytest supports *fixtures*, which are functions that provide information to tests. The same fixture can be used by many tests, which lets you keep the fixture creation in a single place.

A fixture provides items needed by a test, such as data, functions, or class instances.

Fixtures can be either builtin or custom.

TIP Use `py.test --fixtures` to list all available builtin and user-defined fixtures.

User-defined fixtures

- Decorate with **pytest.fixture**
- Return value to be used in test
- Fixtures may be nested

To create a fixture, decorate a function with **pytest.fixture**. Whatever the function returns is the value of the fixture.

To use the fixture, pass it to the test function as a parameter. The return value of the fixture will be available as a local variable in the test.

Fixtures can take other fixtures as parameters as well, so they can be nested to any level.

Put fixtures into their own module so they can be shared across multiple test scripts.

TIP Add docstrings to your fixtures and the docstrings will be displayed via `pytest --fixtures`

Example

pytests/test_simple_fixture.py

```
#!/usr/bin/env python
from collections import namedtuple
import pytest

Person = namedtuple('Person', 'first_name last_name')  ①

FIRST_NAME = "Guido"
LAST_NAME = "Von Rossum"

@pytest.fixture  ②
def person():
    """
    Return a 'Person' named tuple with fields 'first_name' and 'last_name'
    """
    return Person(FIRST_NAME, LAST_NAME)  ③


def test_first_name(person):  ④
    assert person.first_name == FIRST_NAME

def test_last_name(person):  ④
    assert person.last_name == LAST_NAME
```

① create object to test

② mark **person** as a fixture

③ return value of fixture

④ pass fixture as test parameter

Builtin fixtures

- Variety of common fixtures
- Provide
  - Temp files and dirs
  - Logging
  - STDOUT/STDERR capture
  - Monkeypatching tools

Pytest provides a large number of builtin fixtures for common testing requirements.

Using a builtin fixture is like using user-defined fixtures. Just specify the fixture name as a parameter to the test. No imports are needed for this.

See https://docs.pytest.org/en/latest/reference.html#fixtures for details on builtin fixtures.

Example

pytests/test_builtin_fixtures.py

```
COUNTER_KEY = 'test_cache/counter'

def test_cache(cache):  ①
    value = cache.get(COUNTER_KEY, 0)
    print("Counter before:", value)
    cache.set(COUNTER_KEY, value + 1)  ②
    value = cache.get(COUNTER_KEY, 0)  ②
    print("Counter after:", value)
    assert True   ③

def hello():
    print("Hello, pytesting world")

def test_capsys(capsys):
    hello()  ④
    out, err = capsys.readouterr()  ⑤
    print("STDOUT:", out)

def bhello():
    print(b"Hello, binary pytesting world\n")

def test_capsysbinary(capsys):
    bhello()  ⑥
    out, err = capsys.readouterr()  ⑦
    print("BINARY STDOUT:", out)

def test_temp_dir1(tmpdir):
    print("TEMP DIR:", str(tmpdir))  ⑧

def test_temp_dir2(tmpdir):
    print("TEMP DIR:", str(tmpdir))

def test_temp_dir3(tmpdir):
    print("TEMP DIR:", str(tmpdir))
```

① cache persists values between test runs

② cache fixture is similar to dictionary, but with **.set()** and **.get()** methods

③ Make test successful

④ Call function that writes text to STDOUT

⑤ Get captured output

⑥ Call function that writes binary text to STDOUT

⑦ Get captured output

⑧ tmpdir fixture provides unique temporary folder name

Table 1. Pytest Builtin Fixtures

Fixture	Brief Description
cache	Return cache object to persist state between testing sessions.
capsys	Enable capturing of writes (text mode) to **sys.stdout** and **sys.stderr**
capsysbinary	Enable capturing of writes (binary mode) to **sys.stdout** and **sys.stderr**
capfd	Enable capturing of writes (text mode) to file descriptors **1** and **2**
capfdbinary	Enable capturing of writes (binary mode) to file descriptors **1** and **2**
doctest_namespace	Return **dict** that will be injected into namespace of doctests
pytestconfig	Session-scoped fixture that returns **_pytest.config.Config** object.
record_property	Add extra properties to the calling test.
record_xml_attribute	Add extra xml attributes to the tag for the calling test.
caplog	Access and control log capturing.
monkeypatch	Return **monkeypatch** fixture providing monkeypatching tools
recwarn	Return **WarningsRecorder** instance that records all warnings emitted by test functions.
tmp_path	Return **pathlib.Path** instance with unique temp directory
tmp_path_factory	Return a **_pytest.tmpdir.TempPathFactory** instance for the test session.
tmpdir	Return **py.path.local** instance unique to each test
tmpdir_factory	Return **TempdirFactory** instance for the test session.

Configuring fixtures

- Create **conftest.py**
- Automatically included
- Provides
  - Fixtures
  - Hooks
  - Plugins
- Directory scope

The **conftest.py** file can be used to contain user-defined fixtures, as well has hooks and plugins. Subfolders can have their own conftest.py, which will only apply to tests in that folder.

In a test folder, define one or more fixtures in conftest.py, and they will be available to all tests in that folder, as well as any subfolders.

Hooks

Hooks are predefined functions that will automatically be called at various points in testing. All hooks start with *pytest_*. A pytest.Function object, which contains the actual test function, is passed into the hook.

For instance, `pytest_runtest_setup()` will be called before each test.

NOTE A complete list of hooks can be found here: https://docs.pytest.org/en/latest/reference.html#hooks

Plugins

There are many pytest plugins to provide helpers for testing code that uses common libraries, such as **Django** or **redis**.

You can register plugins in conftest.py like so:

```
pytest_plugins = "plugin1", "plugin2",
```

This will load the plugins.

Example

pytests/stuff/conftest.py

```
#!/usr/bin/env python
from pytest import fixture

@fixture
def common_fixture():  ①
    return "DATA"


def pytest_runtest_setup(item):  ②
    print("Hello from setup,", item)
```

① user-defined fixture

② predefined hook (all hooks start with *pytest_*

Example

pytests/stuff/test_stuff.py

```
#!/usr/bin/env python
import pytest

def test_one():  ①
    print("WHOOPEE")
    assert(1)

def test_two(common_fixture):   ②
    assert(common_fixture == "DATA")

if __name__ == '__main__':
    pytest.main([__file__, "-s"])   ③
```

① unit test that writes to STDOUT

② unit test that uses fixture from conftest.py

③ run tests (without stdout/stderr capture) when this script is run

pytests/stuff/test_stuff.py

```
============================= test session starts ==============================
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: /Users/jstrick/curr/courses/python/examples3, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, mock-2.0.0, doctestplus-0.2.0,
arraydiff-0.3
collected 2 items

pytests/stuff/test_stuff.py Hello from setup, <Function 'test_one'>
WHOOPEE
.Hello from setup, <Function 'test_two'>
.

========================== 2 passed in 0.01 seconds ===========================
```

Parametrizing tests

- Run same test on multiple values
- Add parameters to fixture decorator
- Test run once for each parameter
- Use `pytest.mark.parametrize()`

Many tests require testing a method or function against many values. Rather than writing a loop in the test, you can automatically repeat the test for a set of inputs via **parametrizing**.

Apply the `@pytest.mark.parametrize` decorator to the test. The first argument is a string with the comma-separated names of the parameters; the second argument is the list of parameters. The test will be called once for each item in the parameter list. If a parameter list item is a tuple or other multi-value object, the items will be passed to the test based on the names in the first argument.

NOTE For more advanced needs, when you need some extra work to be done before the test, you can do indirect parametrizing, which uses a parametrized fixture. See `test_parametrize_indirect.py` for an example.

NOTE The authors of pytest deliberately spelled it "parametrizing", not "parameterizing".

Example

pytests/test_parametrization.py

```
#!/usr/bin/env python
import pytest


def triple(x):  ①
    return x * 3

test_data = [(5, 15), ('a', 'aaa'), ([True], [True, True, True])]  ②

@pytest.mark.parametrize("input,result", test_data)  ③
def test_triple(input, result):  ④
    print("input {} result {}:".format(input, result))  ④
    assert triple(input) == result   ⑤


if __name__ == "__main__":
    pytest.main([__file__, '-s'])
```

① Function to test

② List of values for testing containing input and expected result

③ Parametrize the test with the test data; the first argument is a string defining parameters to the test and mapping them to the test data

④ The test expects two parameters (which come from each element of test data)

⑤ Test the function with the parameters

pytests/test_parametrization.py

```
============================= test session starts ==============================
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: /Users/jstrick/curr/courses/python/examples3, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, mock-2.0.0, doctestplus-0.2.0,
arraydiff-0.3
collected 3 items

pytests/test_parametrization.py input 5 result 15:
.input a result aaa:
.input [True] result [True, True, True]:
.

=========================== 3 passed in 0.01 seconds ===========================
```

Marking tests

- Create groups of tests ("test cases")
- Can create multiple groups
- Use `@pytest.mark.somemark()`

You can mark tests with labels so that they can be run as a group. Use `@pytest.mark.mark()`, where *mark* is the name of the mark, which can be any alphanumeric string.

Then you can run select tests which contain or match the mark, as described in the next topic.

In addition, you can register markers in the **[pytest]** section of **pytest.ini**:

```
[pytest]
markers =
  internet: test requires internet connection
  slow: tests that take more time (omit with '-m "not slow"")
```

```
pytest -m "mark"
pytest -m "not mark"
```

Example

pytests/test_mark.py

```
#!/usr/bin/env python
import pytest

@pytest.mark.alpha  ①
def test_one():
    assert 1

@pytest.mark.alpha  ①
def test_two():
    assert 1

@pytest.mark.beta  ②
def test_three():
    assert 1

if __name__ == '__main__':
    pytest.main([__file__, '-m alpha'])  ③
```

① Mark with label **alpha**

② Mark with label **beta**

③ Only tests marked with **alpha** will run (equivalent to 'pytest -m alpha' on command line)

pytests/test_mark.py

```
============================= test session starts ==============================
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: /Users/jstrick/curr/courses/python/examples3, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, mock-2.0.0, doctestplus-0.2.0,
arraydiff-0.3
collected 3 items / 1 deselected

pytests/test_mark.py ..                                                  [100%]

==================== 2 passed, 1 deselected in 0.01 seconds ====================
```

Running tests (advanced)

- Run all tests
- Run by
  - function
  - class
  - module
  - name match
  - group

pytest provides many ways to select which tests to run.

Running all tests

To run all tests in the current and any descendent directories, use

Use **-s** to disable capturing, so anything written to STDOUT is displayed. Use **-s** for verbose output.

```
pytest
pytest -v
pytest -s
pytest -vs
```

Running by component

Use the node ID to select by component, such aas module, class, method, or function name:

```
file::class
file::class::test
file::::test
```

```
pytest test_president.py::test_dates
pytest test_president.py::test_dates::test_birth_date
```

Running by name match

Use **-k** to run all tests whose name includes a specified string

`pytest -k date` *run all tests whose name includes 'date'*

Skipping and failing

- Conditionally skip tests
- Completely ignore tests
- Decorate with
  - @pytest.mark.xfail
  - @pytest.mark.skip

To skip tests conditionally (or unconditionally), use `@pytest.mark.skip()`. This is useful if some tests rely on components that haven't been developed yet, or for tests that are platform-specific.

To fail on purpose, use `@pytest.mark.xfail)`. This reports the test as "XPASS" or "xfail", but does not provide tracback. Tests marked with xfail will not fail the test suite. This is useful for testing not-yet-implemented features, or for testing objects with know bugs that will be resolved later.

Example

pytests/test_skip.py

```
#!/usr/bin/env python
import sys
import pytest

def test_one():  ①
    assert 1

@pytest.mark.skip(reason="can not currently test")  ②
def test_two():
    assert 1

@pytest.mark.skipif(sys.platform != 'win32', reason="only implemented on Windows")
③
def test_three():
    assert 1

@pytest.mark.xfail  ④
def test_four():
    assert 1

@pytest.mark.xfail  ④
def test_five():
    assert 0

if __name__ == '__main__':
    pytest.main([__file__, '-v'])
```

① Normal test

② Unconditionally skip this test

③ Skip this test if current platform is not Windows

pytests/test_skip.py

```
============================= test session starts ==============================
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0 --
/Users/jstrick/.pyenv/versions/anaconda3-2018.12/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/examples3, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, mock-2.0.0, doctestplus-0.2.0,
arraydiff-0.3
collecting ... collected 5 items

pytests/test_skip.py::test_one PASSED                                    [ 20%]
pytests/test_skip.py::test_two SKIPPED                                   [ 40%]
pytests/test_skip.py::test_three SKIPPED                                 [ 60%]
pytests/test_skip.py::test_four XPASS                                    [ 80%]
pytests/test_skip.py::test_five xfail                                    [100%]

========== 1 passed, 2 skipped, 1 xfailed, 1 xpassed in 0.07 seconds ===========
```

Mocking data

- Simulate behavior of actual objects
- Replace expensive dependencies (time/resources)
- Use unittest.mock or pytest-mock

Some objects have dependencies which can make unit testing difficult. These dependencies may be expensive in terms of time or resources.

The solution is to use a **mock** object, which pretends to be the real object. A mock object behaves like the original object, but is restricted and controlled in its behavior.

For instance, a class may have a dependency on a database query. A mock object may accept the query, but always returns a hard-coded set of results.

A mock object can record the calls made to it, and assert that the calls were made with correct parameters.

A mock object can be preloaded with a return value, or a function that provides dynamic (or random) return values.

A *stub* is an object that returns minimal information, and is also useful in testing. However, a mock object is more elaborate, with record/playback capability, assertions, and other features.

pymock objects

- Use pytest-mock plugin
  - Can also use unittest.mock.Mock
- Emulate resources

pytest can use unittest.mock, from the standard library, or the pytest-mock plugin, which provides a wrapper around unittest.mock

In either case, there are two primary ways of using mock. One is to provide a replacement class, function, or data object that mimics the real thing. The second is to monkey-path a library, which temporarily (just during the test) replaces a component with a mock version.

Once the pytest-mock module is installed, it provides a fixture named **mocker**, from which you can create mock objects.

Example

pytests/test_mock_unittest.py

```
#!/usr/bin/env python
#
import pytest
from unittest.mock import Mock

ham = Mock()  ①


# system under test
class Spam():  ②
    def __init__(self, param):
        self._value = ham(param)  ③

    @property
    def value(self):  ④
        return self._value

# dependency to be mocked -- not used in test
# def ham(n):
#     pass

def test_spam_calls_ham():  ⑤
    _ = Spam(42)  ⑥
    ham.assert_called_once_with(42)  ⑦


if __name__ == '__main__':
    pytest.main([__file__])
```

① Create mock version of ham() function

② System (class) under test

③ Calls ham() (doesn't know if it's fake)

④ Property to return result of ham()

⑤ Actual unit test

⑥ Create instance of Spam, which calls ham()

⑦ Check that spam.value correctly returns return value of ham()

pytests/test_mock_unittest.py

```
============================= test session starts ==============================
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: /Users/jstrick/curr/courses/python/examples3, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, mock-2.0.0, doctestplus-0.2.0,
arraydiff-0.3
collected 1 item

pytests/test_mock_unittest.py .                                          [100%]

=========================== 1 passed in 0.01 seconds ===========================
```

Example

pytests/test_mock_pymock.py

```
#!/usr/bin/env python
import pytest  ①
import re  ②

class SpamSearch():  ③
    def __init__(self, search_string, target_string):
        self.search_string = search_string
        self.target_string = target_string


    def findit(self):  ④
        return re.search(self.search_string, self.target_string)

def test_spam_search_calls_re_search(mocker):   ⑤
    mocker.patch('re.search')  ⑥
    s = SpamSearch('bug', 'lightning bug')  ⑦
    _ = s.findit()   ⑧
    re.search.assert_called_once_with('bug', 'lightning bug')  ⑨

if __name__ == '__main__':
    pytest.main([__file__, '-s'])   ⑩
```

① Needed for test runner

② Needed for test (but will be mocked)

③ System under test

④ Specific method to test (uses re.search)

⑤ Unit test

⑥ Patch re.search (i.e., replace re.search with a Mock object that records calls to it)

⑦ Create instance of SpamSearch

⑧ Call the method under test

⑨ Check that method was called just once with the expected parameters

⑩ Start the test runner

pytests/test_mock_pymock.py

```
============================= test session starts ==============================
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: /Users/jstrick/curr/courses/python/examples3, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, mock-2.0.0, doctestplus-0.2.0,
arraydiff-0.3
collected 1 item

pytests/test_mock_pymock.py .

=========================== 1 passed in 0.08 seconds ===========================
```

Example

pytests/test_mock_play.py

```
#!/usr/bin/env python
import pytest
from unittest.mock import Mock


@pytest.fixture
def small_list():   ①
    return [1, 2, 3]


def test_m1_returns_correct_list(small_list):
    m1 = Mock(return_value=small_list)  ②
    mock_result = m1('a', 'b')  ③
    assert mock_result == small_list  ④


m2 = Mock()  ⑤

m2.spam('a', 'b')  ⑥
m2.ham('wombat')  ⑥
m2.eggs(1, 2, 3)  ⑥

print("mock calls:", m2.mock_calls)  ⑦

m2.spam.assert_called_with('a', 'b')  ⑧
```

① Create fixture that provides a small list

② Create mock object that "returns" a small list

③ Call mock object with arbitrary parameters

④ Check the mocked result

⑤ Create generic mock object

⑥ Call fake methods on mock object

⑦ Mock object remembers all calls

⑧ Assert that spam() was called with parameters 'a' and 'b'

pytests/test_mock_play.py

```
mock calls: [call.spam('a', 'b'), call.ham('wombat'), call.eggs(1, 2, 3)]
```

Pytest and Unittest

- Run Unittest-based tests
- Use Pytest test runner

The Pytest builtin test runner will detect Unittest-based tests as well. This can be handy for transitioning legacy code to Pytest.

Chapter 1 Exercises

Exercise 1-1 (test_president_pytest.py)

Using **pytest**, Create a unit test for the President class you created earlier.[1]

Suggestions for tests:

- What happens when an out-of-range term number is given?
- President 1's first name is "George"
- All 45 presidential terms match the correct last name (use list of last names and **parametrize**)
- Confirm date fields return an object of type **datetime.date**

[1] If there was not an exercise where you created a President class, there is one in the top-level folder of the student guide.