**Pytest**

(Learning from Eden Marco’s *Real World Python Test Automation with Pytest*)

What is pytest

* Feature-rich, plugin based ecosystem for testing Python
* **Not the same as *unittest* package**
* Provides new approach to writing tests (e.g. functional testing for applications)
* No need for boilerplate code

Running pytest

|  |  |  |
| --- | --- | --- |
| **Shell command** | **Description** | **Remarks** |
| pytest . | To run all tests in current directory | By default, *pytest* looks for filenames starting with ‘*test\_*’ |
| pytest /path/to/test\_file.py | To run tests within a specific file |  |
| pytest /path/to/test\_file.py /path/to/test\_dir | To run tests in multiple files and/or directories |  |
| pytest -v /path/to/test\_file.py | To run *pytest* command with flags   * Refer to table below for more flags |  |
| pytest /path/to/test\_file.py**::**<test\_classname> | Run a specific test class |  |
|  | Run a specific test from a specific test class |  |
| pytest <filepath\_to\_test\_file>**::**<test\_classname>**::**<test\_name> |  |  |
|  |  |  |
|  |  |  |

Common flags in *pytest* command:

|  |  |
| --- | --- |
| **Flag** | **Description** |
| -v | Output verbose description (e.g. test name, test status) |
| -p |  |
| --no-warnings | Suppress warnings in output |
| -m <marker> | Run tests with specified marker |
| -m “not <marker>” | Run all tests expect those with specific marker |
| -s | Output *print()* statements in console |
| --duration=0 | Time tracking: how long each test took to complete |
| -k “<keyword>” | Keyword flag: run specific test(s) if function name contains keyword |
| -k “not <keyword>” | Opposite of above |
|  |  |
|  |  |

Markers, Fixtures in pytest

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Feature** | **Description** | | | **Example** |
| Skip test | * Tells pytest not to run test * Also known as an *unconditional skip* * Default marker | | | @pytest.mark.skip  def test\_should\_be\_skipped() -> None:  … |
| Conditional skip | * Tells pytest to skip test if condition is true * Provide condition for skipping * Can provide reason for skipping test * Default marker | | | @pytest.mark.skipif(<condition>)  def test…  # Example  @pytest.mark.skipif(4 > 1, reason=”Skipped because 4 > 1”)  def test\_should\_be\_skipped\_if() -> None:  assert 1 == 2 |
| XFail | * Mark test functions as expected/okay to fail * E.g. test that does not pass all the time * Does not affect success of entire test run * Default marker | | | @pytest.mark.xfail  def test\_dont\_care\_if\_fails() -> None:  … |
| Custom marker | * Specify custom marker * E.g. run tests that are marked as ‘slow’ | | | @pytest.mark.<custom\_name>  def test…  # Example  @pytest.mark.slow  def test\_with\_custom\_mark():  …  # Run with command pytest . -m slow  # Not run with command pytest -m “not slow” |
| Parameterisation | * Makes it easier to debug which testcase went wrong (rather than multiple assert statements in the same test function) | | | @pytest.mark.parametrize(  <param\_name>,  [<value1>, <value2> …]  )  def test\_...(<param\_name>):  …  # Example  @pytest.mark.parametrize(  “dog\_name”,  [“Earl”, “Snoopy”, “Cherry”]  )  def test\_dog(dog\_name):  … |
| Pass in multiple arguments for each loop of parametrize   * Specified parameter names has to match that in function argument name * Use tuple to group multiple parameters | | | @pytest.mark.parametrize(  “<param1\_name>,<param2\_name>”,  [  (value11, value12),  (value21, value22), ,  …  ]  )  def test\_bar(<param1\_name>,  <param2\_name>):  … |
| Specify name of parameterized test using *ids* argument   * By default if not specified, name of test is value passed in * Order matters | | | @pytest.mark.parametrize(  <param\_name>,  [<value1>, <value2>, …],  ids=[<testname1>, <testname2>, …]  )  def test… |
| **Fixture** | * Place functions in file names conftest.py * Directory-scoped * Pass fixtures to functions as an argument * Able to run fixture before/after function | | |  |
| * Function with arguments matching fixture name will run fixture before testcase | | | @pytest.fixture  def company() -> Company:  return Company(…)  # 1) Argument name match, run fixture  # 2) Pass fixture output to argument  def test\_with\_fixture(company: Company):  … |
| Fixture scope   * *FUNCTION* - run once per function * *CLASS* - run once per class of tests * *MODULE* - run once per module * *SESSION* - run once per session | | | @pytest.fixture(scope=”session”) |
| Example of a fixture that times how long a test took to run   * *yield*: suspends a function’s execution and sends a value back to caller (use to iterate over a sequence without storing entire sequence in memory) | | | import pytest  from datetime import datetime  @pytest.fixture  def time\_tracker():  start = datetime.now()  yield  end = datetime.now()  dur = (end - start).total\_seconds()  print(f”Time taken= {dur} sec”) |
| Example of a fixture that will fail a test if it exceeds a certain run time | # fixtures.py  from datetime import datetime, timedelta  from typing import Callable  import pytest  def track\_performance(method: Callable,  runtime\_limit=timedelta(seconds=2)  ):  def run\_function\_and\_validate\_runtime(\*args, \*\*kwargs):  start = datetime.now()  result = method(\*args, \*\*kwargs)  end = datetime.now()  runtime = end - start  print(f”\n runtime= {runtime.total\_seconds()}”)  if runtime > runtime\_limit:  raise PerformanceException(runtime=runtime,  limit=runtime\_limit)  return result  return run\_function\_and\_validate\_runtime  # test.py  @track\_performance  def test\_performance():  … | | |
| Example of a fixture that can receive arguments   * Pass fixture as argument to test function * Call fixture within test function, supply values to keyword arguments * Within definition of fixture is another function, pass the values to this function * Nested function within fixture will create and return an object * Fixture will return this object * Test function receives this created object | | @pytest.fixture()  def company(\*\*kwargs):  def \_company\_factory(\*\*kwargs) -> Company:  company\_name = kwargs.pop(  “name”,  “Test Company INC”  )  return Company.objects.create(  name=company\_name,  \*\*kwargs  )  return \_company\_factory  def test\_multiple\_companies\_exist\_should\_succeed(  client,  company  ) -> None:  tiktok = company(name=”Tiktok”)  twitch = company(name=”Twitch”)  … | |
|  |  | | |  |
|  |  | | |  |
|  |  | | |  |
|  |  | | |  |
|  |  | | |  |

Features in pytest

|  |  |  |
| --- | --- | --- |
| **Feature** | **Description** | **Example** |
| Pytest configurations | * Create a *pytest.ini* file   Specify in configuration file:   * Set default flags to run when *pytest* is called * Register markers | [pytest]  addopts = -s -v  markers =  <marker\_name> |
| Environment configurations | * Create a *.env* file in the directory * Use-case: environment variables are gone when a new terminal is used * If *.env* file is present, *pipenv shell* and *pipenv run* will automatically load it | PYTHONPATH=/Users/…  DJANGO\_SETTINGS\_MODULE=… |
| Raise exception | * A context manager * Specifies that the next block of code should raise an exception * If no exception was raised or a different exception was raised, the test fails | def raise\_dog\_exception():  raise ValueError(“There is a dog!”)  def test…():  with pytest.raise(ValueError) as e:  raise\_dog\_exception()  assert str(e.value) == “There is a dog!” |
| To gain access to the actual exception information   * Use ***excinfo*** * An *ExceptionInfo* instance * A wrapper around the actual exception raised * Main attributes: *.type, .value, .traceback* | def test\_recursion\_depth():  with pytest.raise(RuntimeError) as excinfo:  def f():  f()  f()  assert “maximum recursion” in str(excinfo.value) |
| *caplog* | * A pytest fixture * Captures all logs logged during test with *WARNING* level and above by default |  |
| Change log level for captured log messages   * By default, sets level on root logger * Log levels are restored at the end of the test | def test\_foo(caplog):  caplog.set\_level(logging.INFO)  pass  # Set log level of a specific logger  def test\_foo(caplog):  caplog.set\_level(logging.CRITICAL,  logger=”root.baz”)  pass |
| For more examples, refer to “Assert logs” section below |  |
| Caching | * A dynamic solution from *functools* library * Stores previously computed results * Uses “Least Recently Used” (LRU) strategy - least recently used entries most likely to be reused, evicts the least recently used entry * Decorator *@lru\_cache* will store the result of the function for each different input * By default, maximum number of entries in cache is 128 | from functools import lru\_cache  @lru\_cache  def cached\_func(arg1):  … |
| Set maximum size of cache | @lru\_cache(maxsize=16)  # Infinite cache  @lru\_cache(maxsize=None) |
| Inspect the cache properties:   * *hits* - number of calls returned directly from memory since they exist in cache * *misses* - number of calls that didn’t come from memory and were computed * *maxsize* - size of cache as defined in *maxsize* attribute of decorator * *currsize* - current size of cache | # Code here  …  print(cached\_func.cache\_info()) |
| Remove all entries from cache | # Code here  …  cached\_func.cache\_clear() |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Assertions in pytest

|  |  |  |
| --- | --- | --- |
| **To assert** | **Description** | **Example** |
| **exceptions** | 1. Use *pytest.raise* and call the function to be tested that raises an exception 2. Verify that exception string is correct | def raise\_dog\_exception():  raise ValueError(“There is a dog!”)  def test\_raise\_exception():  with pytest.raise(ValueError) as e:  raise\_dog\_exception()  assert str(e.value) == “There is a dog!” |
| **Do not** do this   * Don’t test using try-catch | def test\_assert\_exception():  try:  x = 1/1  assert False  except Exception:  assert True |
| **logs** | Test that correct contents of a message was logged   * By default, only captures *WARNING* level and above | def test\_logged\_warning\_level(caplog):  function\_that\_logs\_something()  assert “This is logged.” in caplog.text |
| Specify log level to capture and test that correct contents of a message was logged | def test\_bar(caplog):  with caplog.at\_level(logging.INFO):  logger.info(“Beep beep”)  assert “Beep beep” in caplog.text |
| Test that correct log level was logged   * Logs logged during a test run are sent the the logger in the form of *logging.LogRecord* instances | def test\_baz(caplog):  func\_under\_test()  for record in caplog.records:  assert record.levelname != “CRITICAL”  … |
| Test that certain messages have been logged under a given logger name with a given severity and message | def test\_foo(caplog):  logging.getLogger().info(“boop boop”)  assert caplog.record\_tuples ==  [(  “root”,  logging.INFO,  “boop boop”  )] |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Other libraries to use with pytest

|  |  |  |
| --- | --- | --- |
| **Library** | **Description** | **Example** |
| **pytest-timeout** |  |  |
| **pytest-sugar** |  |  |
| **pytest-xdist** | * Run tests on multiple threads * Install by pip, start using with flag * No use if NCORE=1, overhead from creating and switching threads | pytest -n NCORE |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

fixtures, custom markers, parametrize, skip, xfail pytest-django, pytest-xdist, pytest-cov

Unit tests, integration tests, API tests, end to end tests, performance tests

intent test

- consider using classes of tests

- consider using setup and teardown fixture for database conns

- can use multiple asserts in a single test function

- indirect param (indirect=True)

- pytest request

1. https://www.udemy.com/course/elegant-automation-frameworks-with-python-and-pytest/

2. https://www.udemy.com/course/pytest-course/