



Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

AGH University of Science
and Technology

Modern Programming Techniques - Test automation with PyTest

Bartosz Konopka, Wiktor Szewczyk, Michał Partyka

AGH University of Science and Technology

Part I

Theoretical introduction to test automation and PyTest

Outline



- 1 Different areas of PyTest usage - WS
- 2 What is the test automation and why is it needed - BK
- 3 What is PyTest and why is it better than other tools - MP

Different areas of PyTest usage - WS



- 1 Different areas of PyTest usage - WS
- 2 What is the test automation and why is it needed - BK
- 3 What is PyTest and why is it better than other tools - MP

Testing software



Why do we need to test software?

Testing software ensures that our software meets all product specific requirements, does not have any bugs or errors and it is safe to release it to a critical environment. Moreover it makes it easier to add new features.

Testing software



Why do we need to test software? (Developer perspective)

- ① We owe it to our clients, ourselves and our company that we provide the best product we can.
- ② Our clients run our product on different devices, inside different operating systems, browsers or apps.
- ③ User could use our application in matter that it should not be used.
- ④ Something that works for one person, may not work for 100 people at same time.
- ⑤ Be sure that our application does what it's supposed to do.

Where Pytest is used?



Why Pytest makes testing software easier?

Pytest framework provides us with all its features that makes it easier to test our software, for example, fixtures that help us to not duplicate code or parameterization that helps us to test the same application with different parameters.

Where Pytest is used?

Pytest is used are wide from web development through mobile apps to embedded systems. Moreover Pytest is popular because of its easy entry threshold and popularity of Python language.

What is the test automation and why is it needed - BK



- 1 Different areas of PyTest usage - WS
- 2 What is the test automation and why is it needed - BK
- 3 What is PyTest and why is it better than other tools - MP

Manual and automated testing



- manual testing
 - time-consuming
 - problematic
 - costly
- automated testing
 - streamlining this process
 - reducing the time needed to perform tests
 - eliminating issues such as programmers spending additional time on tedious testing

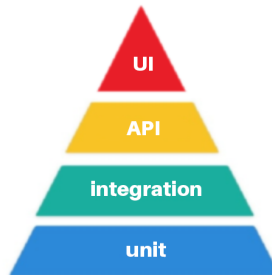
Automated testing

Automated testing is the process of using programming tools to execute a series of tests on newly developed software or updates. The goal is to identify potential errors in the code and other performance impediments.

Test automation pyramid

Test automation pyramid

Crucial element representing the hierarchy of testing.



Types of tests



- Unit tests - examination of so-called units: individual and smallest parts of code, such as functions, methods, or classes.
- Integration tests - tests that allow the verification of the operation of units integrated with each other.
- API tests - responsible for checking the interface of an application program.
 - Validation tests - checking the correctness of entered data.
 - Functional tests - focusing on verifying whether the application or system operates according to functional expectations.
 - Security tests - evaluating and identifying potential threats and vulnerabilities in the application.
 - Load tests - checking how the system handles a large amount of information and users.
- UI tests - tests that check the user interface.

Processes where automation can be beneficial



- Repetitive tests
- High-risk tests - where a small code change can have a very negative impact on the entire code.
- Time-consuming tests - where manual testing would waste a lot of employees' time.
- When an application interacts extensively with other applications or software, where conflicts may arise

What is PyTest and why is it better than other tools - MP



- 1 Different areas of PyTest usage - WS
- 2 What is the test automation and why is it needed - BK
- 3 What is PyTest and why is it better than other tools - MP

Pytest Overview



- Pytest is a software testing framework written in Python.
- Popular for its ease of writing, organizing, and running unit and integration tests.
- Offers clear and intuitive syntax, making test writing simpler.
- Extensive ecosystem of extensions for testing various technologies.
- Excellent for unit testing, allowing testing of individual code parts in isolation.
- Beyond unit tests, Pytest facilitates testing how different application parts work together.

Pytest vs. Other Tools



- Simple syntax, flexibility, and a wide range of features make Pytest a popular choice.
- Compared to other Python testing tools:
 - Surpasses traditional unittest or nose2.
 - Provides a more balanced approach to unit testing compared to the Robot Framework.

Conclusion



- Pytest is distinguished by its simplicity, flexibility, and a wide range of features.
- Popular in the programming community for unit and integration testing.

Part II

Installing PyTest and preparing the project

Outline



- 1 Installing and configuring PyTest - MP
- 2 Project structure - BK
- 3 Creating and launching the project - WS

Installing and configuring PyTest - MP



- 1 Installing and configuring PyTest - MP
- 2 Project structure - BK
- 3 Creating and launching the project - WS

Installing Pytest



- ❶ Open the terminal and update the package list and upgrade existing packages:

```
sudo apt-get update  
sudo apt-get upgrade
```
- ❷ Install Python along with the pip tool (if not already installed):

```
sudo apt-get install python3  
sudo apt-get install python3-pip
```
- ❸ Install Pytest using pip:

```
pip3 install pytest
```
- ❹ Check the Pytest version to ensure correct installation:

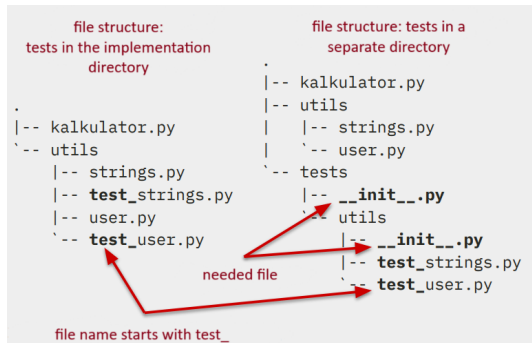
```
pytest --version
```

Project structure - BK



- 1 Installing and configuring PyTest - MP
- 2 Project structure - BK
- 3 Creating and launching the project - WS

Project structure



Testowanie aplikacji, poziomy testowania, pytest by Adam Chył

__init__.py



When we decide to place test files in a different directory, it's important to include, even if empty, a file named `init.py` in that directory. This file informs the Python interpreter that the directory in which it is located should be treated as a package. Including this file in the test directory also allows us to import functions or entire code files into our test files. Importing files to test files:

```
from directory import file_name
```

conftest.py



Conftest.py file is used for configuring and organizing tests. In this file, You can set fixture values, configure settings, or write shared test code across files. This file should be in the same directory as the test files it pertains to. Importantly, it is necessary to import into it all the test files we will use.

Creating and launching the project - WS



- 1 Installing and configuring PyTest - MP
- 2 Project structure - BK
- 3 Creating and launching the project - WS

Running first test



When we already have project structure now we can write our first test. So let's look at some dummy tests and run using the following set of commands:

Specifying set of test

- ❶ `pytest test_first_tests.py`
- ❷ `pytest tests`
(inside `video_code/first_tests`)
- ❸ `pytest -k "TestClass and not test_3"`
- ❹ `pytest test_first_tests.py::TestClass::test_2`
- ❺ `pytest -m my_mark`

Almost all examples should be run in same directory unless it was noted to change the current directory. Current directory:
[/workspaces/MPT_project_PyTest/video_code/first_tests/tests/utls](#)

Useful flags for pytest[-3] command



Let's take closer look at some of the most useful flags:

`-k <expr>`

flag will run only those tests which match `expr` (be aware that this expression is case-insensitive).

`-m <markerexpr>`

flag will run only those tests which match marked tests and its mark matches the `matchexpr`.

`-X`

flag will exit on first test failure or error.

Useful flags for pytest[-3] command



```
-v[v[v[v]]]
```

flag will set verbosity level. Highest level is four v's but it is only supported by plugins, plain pytest will show same result as triple v's.

```
--fixtures
```

flag will list available fixtures (fixtures with leading "_" are only shown with verbose flag).

```
--markers
```

flag will list all markers we have in our project, plugins and those which are builtin in Pytest.

Useful flags for pytest[-3] command



`--durations=N` and `--durations-min=N`

flag which is very useful for profiling tests executions. First one shows list of N slowest tests, second one sets minimal test duration for inclusion in slowest list.

`--pdb`

flag will start interactive python debugger on errors or KeyboardInterrupt. Also useful flag for debugging purposes is `--trace` flag which breaks immediately when running each test, for more, lookup pytest manual providing `-h` flag to pytest command.

Useful flags for pytest[-3] command



These are only few flags which are useful in most situations, for more, lookup:

```
-h
```

```
pytest -h
```

Part III

Practical examples and showcase of PyTest

Outline



- 1 Assert keyword - MP
- 2 Running multiple tests and grouping them - BK
- 3 Fixtures - WS
- 4 Tests parameterization - MP
- 5 Managing tests output and outcomes - BK
- 6 Introduction to monkeypatch/mock modules - WS

Assert keyword - MP



- 1 Assert keyword - MP
- 2 Running multiple tests and grouping them - BK
- 3 Fixtures - WS
- 4 Tests parameterization - MP
- 5 Managing tests output and outcomes - BK
- 6 Introduction to monkeypatch/mock modules - WS

Assert keyword



Assert

Assert basically means that the test will fail every value different than the assertion

```
def fun_1():  
    return 2
```

Let's begin with a simple assert:

```
def test_fun_1():  
    assert code.fun_1() == 2
```

The assertion is, of course, true, so the test will pass.

Pytest Raises



Pytest Raises

By using Raises we assert that the function will throw an error

```
def fun_2():  
    return 1/0  
  
def fun_4(a):  
    1/a  
  
def fun_3():  
    raise ValueError("Exception 123 raised")
```

Pytest Raises



```
def test_fun_2():  
    with pytest.raises(ZeroDivisionError):  
        code.fun_2()  
  
def test_fun_4():  
    pytest.raises(ZeroDivisionError, code.fun_4, 0)  
  
def test_fun_3():  
    with pytest.raises(ValueError, match=r".* 123 .*"):  
        code.fun_3()
```

Marking and Skipping



xfail and skipif

Using Pytest's `mark.xfail` we can mark tests as failed under certain conditions (or under no conditions) and still continue testing. Using Pytest's `skipif` we can skip tests under certain conditions.

```
@pytest.mark.xfail(raises=ZeroDivisionError)
def test_fun_2_2():
    code.fun_2()
```

```
run_test_condition = True
```

```
@pytest.mark.skipif(condition=run_test_condition,
reason="Skipping this test conditionally.")
def test_conditional_skip():
    assert False
```

Context-Sensitive Comparisons



Context-sensitive comparisons

Allowing for flexible and expressive assertions in test cases, lets You compare collections such as sets, lists or strings.

```
def test_list_comparison():  
    list1 = [1,2,3]  
    list2 = [1,4,3]  
    assert list1 == list2
```

Context-Sensitive Comparisons



Output feedback

These sets are different so the test will fail and Pytest will inform You that the test failed because:

```
At index 1 diff: 2 != 4.
```

A few examples about what Pytest can let You know are:

Context-Sensitive Comparisons



Output feedback

A few examples about what Pytest can let You know are:

- comparing long strings: a context diff is shown
- comparing long sequences: first failing indices
- comparing dicts: different entries

Custom Assertions Explanation



Compare function

In Pytest You can even define Your own explanations for failed assertions using:

```
def pytest_assertrepr_compare(op, left, right)
```

Custom Assertions Explanation



```
# conftest.py
def pytest_assertrepr_compare(op, left, right):
    if isinstance(left, test_code.Foo) and
        isinstance(right, test_code.Foo) and
        op == "==":
    return [
        "Comparing Foo instances:",
        f"    vals: {left.val} != {right.val}",
    ]
```

Custom Assertions Explanation



```
# test_code.py
class Foo:
    def __init__(self, val):
        self.val = val

    def __eq__(self, other):
        return self.val == other.val

def test_compare():
    f1 = Foo(1)
    f2 = Foo(2)
    assert f1 == f2
```

Running multiple tests and grouping them - BK



- 1 Assert keyword - MP
- 2 Running multiple tests and grouping them - BK
- 3 Fixtures - WS
- 4 Tests parameterization - MP
- 5 Managing tests output and outcomes - BK
- 6 Introduction to monkeypatch/mock modules - WS

Grouping tests in a class



Grouping tests in a class

When we want to group multiple tests that correspond to similar functionalities, or it is simply more convenient for us to keep them in one group, we should create a class with a chosen name and place the testing functions inside that class.

Grouping tests in a class



It is important for the testing function to take the self argument. It is always passed as the first argument and allows functions to access the class instance. Without this argument, the test functions will not work.

```
class Test_fun_1:  
    def test_sort(self)
```

Grouping tests with markers



Grouping tests with markers

Markers allow You to label functions with specific names.

```
def sum(a, b):  
    return a+b  
  
@pytest.mark.math  
def test_sum():  
    assert sum(1,2) == 3
```

Grouping tests with markers



```
def arr_sort(arr):  
    arr.sort()  
    return arr  
  
@pytest.mark.arr  
def test_sort():  
    list1 = [3,2,1]  
    assert arr_sort(list1) == [1,2,3]
```


Grouping tests with markers



With a regular command in the terminal, all tests will be executed. However, if You run the program with the command `pytest -k arr test_code.py` or `pytest -k math test_code.py`, only the tests whose markers match the specified name after `-k` will be executed.

Fixtures - WS



- 1 Assert keyword - MP
- 2 Running multiple tests and grouping them - BK
- 3 Fixtures - WS**
- 4 Tests parameterization - MP
- 5 Managing tests output and outcomes - BK
- 6 Introduction to monkeypatch/mock modules - WS

What is fixture?



What is fixture?

Fixtures provide defined, reliable and consistent context for tests. Fixtures define steps and data that constitute the phase of a test. It is powerful tool which helps developers in test automation and in designing complex tests.

What is fixture for?

Fixtures often provide initialization of services, states or other environments. They are accessed by tests through arguments named as the fixture. They offers dramatic improvement over xUnit style of setup/teardown functions.

Simple fixture



Fixutre is a function decorated by `@pytest.fixture`. Let's look at simple fixture which prepares object `Register` which will be used in test to check if user is able to register new account.

```
@pytest.fixture
def setup_register():
    return Register("admin", "admin",
                   "admin@gmail.com")
```

Simple fixture



We can access our fixture in test function providing fixture function name as argument.

```
def test_register(self, setup_register):  
    assert setup_register.register() == True
```

After running this test using:

```
cmd  
pytest requesting_fixtures/test_simple_fixture.py  
::TestSimpleFixture::test_register -v
```

we can see that our test passed with flying colours.

Fixtures requesting other fixtures



Pytest does not limit us using to use fixtures that use other fixtures e.g setup for test is so complicated that it is unreadable and unclean to do so in one fixture. Let's look how to do it:

```
@pytest.fixture
def setup_username():
    return "admin"
```

```
@pytest.fixture
def setup_password():
    return "admin"
```

Fixtures requesting other fixtures



```
@pytest.fixture
def setup_email():
    return "admin@gmail.com"

@pytest.fixture
def setup_register(setup_username, setup_password,
setup_email):
    return Register(setup_username, setup_password,
setup_email)
```

Fixtures requesting other fixtures



And again we can run our test using `pytest` command, our test is named `test_register` and it is inside class `TestFixtureRequestingAnotherFixture` in `test_fixture_requesting_fixture.py` file.

Pytest is taking care of developers



As title of this slide says, Pytest takes care about typical human mistakes and does not call fixture once again if it has been already called. Let's look:

```
def test_login(self, setup_login, try_another_login):  
    assert setup_login.login() == True
```

We have such a simple test, that tries to log in the user. We have one fixture `setup_login` and `try_another_login`, where second one does not do anything other then requesting first one but the first one has been already requested first in test.

Pytest is taking care of developers



```
@pytest.fixture
def setup_login(setup_account):
    user = User(setup_account)
    assert user.login() == True
    return user
```

```
@pytest.fixture
def try_another_login(setup_login):
    pass
```

Our test should fail because user already logged in in fixture, moreover we can see that fixture is only called once despite that in code it looks like it was called twice, if it was called again we would get an failure inside first fixture because our application is designed in such a way that logged user cannot log in another time.

Pytest is taking care of developers



To run this test we have to first register our user to get an account and then we can log in, it is simple dummy application for showing purposes, You can see how it works at: [video_code/pracitcal_examples/3_fixtures/utils/code](#). So let's run our test:

cmd

```
pytest requesting_fixtures/  
test_fixture_called_many_times.py -v
```

Why fixtures make Pytest a great tool for test automation



Another thing that makes Pytest shine compared to highly used xUnit framework is fixture, look why:

Why Pytest shine?

- Fixtures have explicit names and are activated by declaring their use from test functions, modules, classes or whole test session.
- Modularity of fixtures allow us call other fixtures from another fixture.
- Scalability of fixtures allows developers to use Pytest for simple unit tests as well as for complex and extensive tests thanks to parametrization of tests and fixtures, overloading fixtures and limiting or expanding scopes of fixtures.

Why fixtures make Pytest a great tool for test automation



Why Pytest shine? cont.

- Teardown can be easily and safely managed without any attention to how many fixtures have been used and allows us to clean at micromanagement level.
- Fixtures are reusable across functions, classes, modules or whole sessions.
- In addition, Pytest continues to support xUnit style setup so You can mix both styles to finally move from classic to new style if You want to.

Fixture assurance, starting every test from clean state



Let's look at simple set of tests:

```
def test_register_1(self, setup_register):  
    for _ in range(10):  
        assert setup_register.register() == True
```

```
def test_register_2(self, setup_register):  
    assert setup_register.register() == True
```

```
def test_register_3(self, setup_register):  
    assert setup_register.register() == True
```

Our setup_register fixture is the same one as in first example of simple fixture.

Fixture assurance, starting every test from clean state



Of course all of those tests will fail except first registration in `test_1` if we run all of them in one session because there is no teardown for registration and we are registering users with same e-mail which is not allowed, if we run second or third test solo they will pass. Thanks to those failures we are assured that each and every one of our tests took off with object `Register` with same attributes. You can look into code and find that there is a implementation of second fixture which is very similar to the previous one but in this one we remove every registered user before providing test with `Register` object so we can be sure that there is no one with same e-mail. Of course it is not very good approach but it allow us to work at least at one user level, we will find better one when we will be talking about teardowns.

Fixture decorator parameters - autouse



Autouse parameter makes fixture to be used automatically without calling it in test function. Let's how it works:

```
@pytest.fixture(autouse=True)
def login(setup_register):
    setup_register.register()
```

and test looks like this:

```
def test_login(self):
    user = User(Account("admin", "admin",
                        "admin@gmail.com"))
    assert user.login() == True
```


Fixture decorator parameters - autouse



It is very convenient way to use it when all tests or fixtures depends on one or more fixtures, it cuts redundant fixture requests and makes our code cleaner. Although fixture still can be requested by test or other fixture by passing its name as argument if it is needed.

Fixture decorator parameters - scope



Fixture scope

- function – default scope for all fixtures, destroyed at the end of the test.
- class – destroyed at teardown of last test in the class.
- module – destroyed at teardown of the last test in the module.
- package – destroyed at teardown of the last test in the package.
- session – destroyed at the end of the test session.

Fixture decorator parameters - scope



Let's look how it works:

```
@pytest.fixture(scope="session")
def setup_username():
    return "admin"
```

We are using session scope because we do not need to change username, password and e-mail across this test scenario because we are testing only registering and logging one user.

Fixture decorator parameters - scope



```
@pytest.fixture(scope="class")
def setup_register(setup_username, setup_password,
                  setup_email):
    return Register(setup_username, setup_password,
                    setup_email)
```

We are using class scope because we are logging in and registering user in one group of tests which are packed in one class. Please take Your time and analyze example `fixture_scopes` and especially `confstest.py` where all fixtures are located, it is one of the useful and one of the most used feature of Pytest.

Fixture decorator parameters - scope



If we cannot decide at code level which scope we need Pytest allows us to use dynamic scope which we will provide at run time, let's how it works:

```
def pytest_addoption(parser):  
    parser.addoption("--keep-data-between-tests",  
                    action="store",  
                    help="Keep data between tests",  
                    default=False)
```

Fixture decorator parameters - scope



Firstly we define new flag for `pytest[-3]` command which we will use to determine scope in function which will be returning scope:

```
def set_scope(fixture_name, config):  
    if config.getoption("--keep-data-between-tests",  
                        None):  
        return "session"  
    return "function"
```

In the function we return session as a string if `--keep-data-between-tests` is provided else we return function.

Fixture decorator parameters - scope



Then we define fixture where argument for scope is function name from previous slide.

```
@pytest.fixture(scope=set_scope)
def setup_db_of_users(setup_username):
    list_of_users= [setup_username + str(i)
                    for i in range(30000)]
    return list_of_users
```

This fixture will provide us 30000 usernames that we will try to register and then log in in our test set. Dynamic scope is especially useful when we are testing something that requires lots of time to setup something like huge amount of data, starting/building docker containers or starting servers.

Fixture decorator parameters - params



Pytest gives us ability to parameterize fixtures so that same fixture will be requested by test function with different parameters, let's look how it works:

```
@pytest.fixture(params=["admin1", "admin2",  
                        "admin3"])  
def setup_register(request):  
    register = Register(request.param, "admin",  
                        request.param + "@gmail.com")  
    yield register  
    register.unregister()  
del register
```


Fixture decorator parameters - params



In our fixture we return register object which will be used in test to try to register new user with different username and email. Using yield keyword we are performing teardown after test is done, we will talk about teardowns and how to perform them in next slides. Coming back to fixtures and params parameter, we are also able to run tests specifying which param we would like to test, please run below command:

```
pytest -k "test_register_2 and (admin1 or admin2)"  
--collect-only
```

--collect-only flag only collects tests, do not execute them. As You can see Pytest collected only test_register_2 with *admin1* and *admin2* parameters from fixture.

Fixture decorator parameters - params



As mentioned earlier Pytest takes care of developer, minimizing the number of active fixtures during tests, so it is grouping them by parameters. Look at code below:

```
def params():  
    return ["admin1", "admin2", "admin3"]  
  
@pytest.fixture(params=params(), scope="class")  
def setup_register(request):  
    register = Register(request.param, "admin",  
        request.param + "@example.com")  
    yield register  
    register.unregister()  
    del register
```

Fixture decorator parameters - params



```
@pytest.fixture(params=params())
def setup_login(request):
    user = User(Account(request.param, "admin",
                        request.param + "@example.com"))
    yield user
    user.logout()
    del user
```

And test looks like this:

```
class TestAutomaticGrouping:
    def test_all(self, setup_register, setup_login):
        assert setup_register.register() == True
        assert setup_login.login() == True
```

Fixture decorator parameters - params



Pytest firstly takes fixture for register with first parameter and then all permutations with second fixture. Of course for almost all parameters this test will fail but it is for showing purposes and moreover for showing how sometimes tests fails, not because of that our application is broken but because our test scenario is wrong.

Fixtures as teardowns



As You can imagine tests may often leave mark on our test system like floating data that is no longer needed or some set values that should not be there after testing. Those residues often results in unexpected tests failures or instability or even in crash of our test system. That's why we need to cleanup after each and every test and fixtures come to help with that. There are two ways of how to do that: first is by using yield keyword; second one is about using finalizers directly. First method is more often used and it is easier to maintain where second one is more complex but with that comes that it is more safe and strict. We will show You first method because there are patterns from which can get same benefits as second option gives. You have seen in previous slides how it is done but let's take a closer look:

Fixtures as teardowns



```
@pytest.fixture()
def setup_user_unsafe():
    username = "admin"
    password = "admin"
    email = "admin@gmail.com"
    account = Account(username, password, email)
    register = Register(username, password, email)
    user = User(account)
    register.register()
    user.login()
    yield user
    user.logout()
    register.unregister()
    del user
    del register
    del account
```

Fixtures as teardowns



It looks awful, isn't it? It looks that way because we would like to show how it is done wrong, if You are using yield type of teardown for our fixtures and even if You are using direct finalizers You should not have one fixture that does everything because if one thing goes wrong let's say creating register object then our teardown will never even start and we will be left with floating account object (it is dummy example, Python will destroy it at exit but what if it is a file with random name or environment variable?). To do it right we should and maybe even have to make such a setups at atomic level so firstly we create account object, then in second fixture register object and so on, so on and after that we should test if setup went well. Please lookup *conftest.py* and *test_teardown_safe.py* in:

*video_code/pracitical_examples/3_fixtures/
tests/teardowns/tests/utls*

Fixtures with markers



Markers will be described in next part of our seminar but we will talk about use of them in fixtures right now as simply as possible. Using request object we can get access to data from marked test function in fixture. Let's look at test below:

```
@pytest.mark.test_login(username="admin",
    password="admin", email="admin@gmail.com")
class TestLogin:
    def test_login(self, setup_login):
        register, user = setup_login
        assert register.register() == True
        assert user.login() == True
        assert user.is_logged() == True
        assert user.do_something() == True
```


Fixtures with markers



And fixture looks like this:

```
@pytest.fixture
def setup_login(request):
    username = request.node.get_closest_marker("test_login").
        kwargs.get("username")
    password = request.node.get_closest_marker("test_login").
        kwargs.get("password")
    email = request.node.get_closest_marker("test_login").
        kwargs.get("email")
    account = Account(username, password, email)
    register = Register(username, password, email)
    user = User(account)
    return register, user
```

Fixtures with markers



There is one interesting inbuilt marker for fixtures that we would like You to take closer look at, usefixture marker, often used for tests that do not need direct access to fixture object, let's look how it works:

```
@pytest.fixture
def open_new_dir():
    with tempfile.TemporaryDirectory() as
        tmpdirname:
            old_dir = os.getcwd()
            os.chdir(tmpdirname)
            yield
            os.chdir(old_dir)
```

Fixtures with markers



```
@pytest.fixture
def write_to_new_file(open_new_dir):
    with open("new_file.txt", "w") as f:
        f.write("Hello World!")
    yield
    os.remove("new_file.txt")
```

Fixtures with markers



And that how our tests look like:

```
@pytest.mark.usefixtures("write_to_new_file")
class TestUsefixtureMark:
    def test_file_exists(self):
        assert os.path.isfile("new_file.txt")
        == True

    def test_file_content(self):
        with open("new_file.txt", "r") as f:
            assert f.read() == "Hello World!"
```

Fixtures with markers



We are using this fixture once for whole class without scope and moreover we do not need request fixture for every test. You can specify multiple fixtures and even specify fixture usage at the test module level but for that please check documentation which will be linked in bibliography at the end of this presentation. Also we encourage You to look at this example in:

video_code/practical_examples/3_fixtures/tests/markers_with_fixtures because there is sample usage of plugin *pytest-dependency* and usage of multiple fixtures in usefixture marker

Fixtures as factories



It is useful usage of fixture if we need to get result of fixture multiple times in one test. Let's look how to do it:

```
@pytest.fixture()
def setup_register():
    registers = []
    def _setup_register(username, password, email):
        register = Register(username, password,
                             email)
        registers.append(register)
        return registers[-1]
    yield _setup_register
    for register in registers:
        register.unregister()
    del registers
```

Fixtures as factories



Then we can call it as many time as we want in our test:

```
class TestFactoryFixture:
    def test_register(self, setup_register):
        user1 = setup_register('admin1', 'admin',
                               'admin1@gmail.com')
        assert user1.register() == True
        user2 = setup_register('admin2', 'admin',
                               'admin2@gmail.com')
        assert user2.register() == True
        user3 = setup_register('admin3', 'admin',
                               'admin3@gmail.com')
        assert user3.register() == True
        assert registered_users.get_count() == 3
```

And as You can see we are able to perform teardown in such a specific fixture.

Overriding fixtures



In most of real life test environments based on Pytest, overriding fixtures is daily basis, mostly it is because it saves duplicating code. We can have main `conftest.py` file where we have all of our generic fixtures at root level of our test environment and in subdirectories where we have specified tests for certain functionalities of our application, we have another `conftest.py` with fixture/s of same name as in upper directory but those do something different. This is called overriding fixtures at directory level. We can also define fixture inside our test module because it is only for this module, this is called overriding at module level. Overriding fixture directly, with and without parametrization is left as homework for curious one's.

Fixtures summary



We highly recommend to lookup prepared samples of different fixture usage in [video_code/pracitical_examples/3_fixtures](#) and for more please check the [documentation](#).

Tests parameterization - MP



- 1 Assert keyword - MP
- 2 Running multiple tests and grouping them - BK
- 3 Fixtures - WS
- 4 Tests parameterization - MP**
- 5 Managing tests output and outcomes - BK
- 6 Introduction to monkeypatch/mock modules - WS

Test parameterization



Test parameterization

Test parameterization refers to the process of using parameters or variables in test scripts or test cases. Instead of using fixed, hardcoded values, parameters allow testers to input different values dynamically.

Let's begin with a simple case:

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

In the previous example two different tests would be run and both would pass.

Test parameterization - Class, Module



Parameterizing a class

Of course we can do the same thing with a whole test class, where every test inside of that class would be tested with the parameterized parameters.

```
@pytest.mark.parametrize("n,expected",
[(1, 2), (3, 4)])
class TestClass:
    def test_simple_case(self, n, expected):
        assert n + 1 == expected

    def test_weird_simple_case(self, n, expected):
        assert (n * 1) + 1 == expected== expected
```

Test parameterization - Class, Module



self argument

We have to remember about passing the ***self*** argument in the test class methods, otherwise the parameterization will not work.

Parameterizing a module

If You want to parametrize all of the tests in the module with the same parameters You can do it by setting the global variable ***pytestmark***, for example:

```
pytestmark = pytest.mark.parametrize("n, expected"  
    , [(1, 2), (3, 4)])
```

Test parameterization - Marking



Marking

Just like with assertions You can mark certain tests as failed so when the test will indeed fail the testing will continue. You can do it by using Pytest's param and mark.xfail.

```
@pytest.mark.parametrize(
    "test_input,expected", [("3+5", 8),
    pytest.param("6*9", 42, marks=pytest.mark.xfail),
    ("2+4", 6)],
)
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

Creating own command-line options



Command-line options

A big thing about test parameterization in Pytest is that You can create Your own command-line arguments options, that we can then call with or without arguments while running our test throught the command line. To do that we have to add two functions to the conftest.py file.

Creating own command-line options



```
def pytest_addoption(parser):
    parser.addoption(
        "--stringinput",
        action="append",
        default=[],
        help="list of stringinputs to"
        "pass to test functions")
```

pytest_addoption

This is a hook function provided by Pytest. Pytest calls this function when it starts to allow plugins to register command-line options. With ***parser.addoption*** You can add custom command-line options, in this case the options name is ***stringinput***.

Creating own command-line options



```
def pytest_addoption(parser):
    parser.addoption(
        "--stringinput",
        action="append",
        default=[],
        help="list of stringinputs to"
        "pass to test functions")
```

pytest_adoption

action specifies what the flag does, here append means that values provided for this option will be appended to a list. Default value for *~stringinput* is an empty list. Help is the string that will be displayed when You run Your test with the *~help* option.

Creating own command-line options



```
def pytest_generate_tests(metafunc):
    if "stringinput" in metafunc.fixturenames:
        metafunc.parametrize("stringinput",
                               metafunc.config.getoption("stringinput"))
```

pytest_generate_tests

It's another hook function provided by Pytest. It is called for each test function during the test collection phase. The purpose of this hook is to dynamically generate and parametrize tests based on some conditions.

Creating own command-line options



```
def pytest_generate_tests(metafunc):  
    if "stringinput" in metafunc.fixturenames:  
        metafunc.parametrize("stringinput",  
                               metafunc.config.getoption("stringinput"))
```

pytest_generate_tests

The if statement check if the test function being considered for parametrization has a fixture named "**stringinput**". The next line parametrizes the test function by assigning each value from the command line to the "**stringinput**" fixture.

Creating own command-line options



So now we define the actual test in the `test_code.py` file:

```
def test_valid_string(stringinput):
    assert stringinput.isalpha()
```

The test will check if all of the strings passed in the command-line through the `~stringinput` option are valid. You can run the tests for example like this:

```
pytest -v --stringinput="hello"
      --stringinput="world" test_code.py
```

Creating own command-line options



To demonstrate it better let's add an other option:

```
parser.addoption("--till", type=int, default=None,
                 help="run all combinations till number")
```

I added these lines to the ***pytest_adoption*** hook function. I also added these lines to the ***pytest_generate_tests*** hook function:

```
if "param1" in metafunc.fixturenames:
    till = metafunc.config.getoption("till")
    if till is not None:
        metafunc.parametrize("param1", range(till))
```

Creating own command-line options



As You can see I added a `--till` option that will test all the integers in the range from 0 to the number provided subtracted by 1. Let's go ahead and write a test:

```
def test_compute(param1):  
    assert param1 < 4
```

Creating own command-line options



Okay now to run the test let's do:

```
pytest --till=5 test_code.py
```

Output:

```

===== FAILURES =====
----- test_compute[4] -----

param1 = 4

    def test_compute(param1):
>     assert param1 < 4
E     assert 4 < 4

test_code.py:41: AssertionError

===== short test summary info =====
FAILED test_code.py::test_compute[4] - assert 4 < 4
===== 1 failed, 4 passed, 1 skipped in 0.15s =====

```

Managing tests output and outcomes - BK



- 1 Assert keyword - MP
- 2 Running multiple tests and grouping them - BK
- 3 Fixtures - WS
- 4 Tests parameterization - MP
- 5 Managing tests output and outcomes - BK**
- 6 Introduction to monkeypatch/mock modules - WS

Handling test failures - flags



Pytest allows us to execute a test program with specific flags that force the program to stop after the first failure or after a specified number of failures.

- `-x` `+-` test execution stops after the first error.
- `--maxfail=number` `+-` test execution stops after the exact number of errors.

Handling test failures - PDB



The built-in Python debugger PDB is highly useful in dealing with test failures. When running the program with the `-pdb` flag, we gain the ability to debug the program. It is also possible to combine flags that stop testing on specific errors with debugging flags.

Dealing with errors - flags



Pytest offers several flags that modify the output. In the image, You can see the most important ones along with their descriptions.

```
pytest --showlocals      # show local variables in tracebacks
pytest -l                # show local variables (shortcut)
pytest --no-showlocals  # hide local variables (if addopts enables them)

pytest --tb=auto         # (default) 'long' tracebacks for the first and last
                        # entry, but 'short' style for the other entries
pytest --tb=long         # exhaustive, informative traceback formatting
pytest --tb=short        # shorter traceback format
pytest --tb=line         # only one line per failure
pytest --tb=native       # Python standard library formatting
pytest --tb=no           # no traceback at all
```

[Pytest documentation](#)

Dealing with errors - verbosity



Pytest also allows running the program with the `-v` or `-vv` flag. This is known as verbosity, which displays much more information about the failure of the code.

After running the program with the regular `pytest test_code.py` command, we receive output informing us about the errors. However, when running the program with the `-v` or `-vv` flags, we get a much more detailed output. With these flags, each test gets its own line of output indicating success or failure, instead of consolidating all information into one line. We also receive more information about the nature of errors, such as which array indices are different from each other, which exact letters in strings differ, or the display of all elements in an array. This flag is useful when we need more information about failed tests.

Dealing with errors - -r flag



After running the program with the command `pytest -rA test_code.py`, You will receive a brief summary of how each test concluded. The letter following the `-r` flag indicates what information to include in the summary. Here, we present all the possible letters You can use:

Here is the full list of available characters that can be used:

- `f` - failed
- `E` - error
- `s` - skipped
- `x` - xfailed
- `X` - xpassed
- `p` - passed
- `P` - passed with output

Special characters for (de)selection of groups:

- `a` - all except `pP`
- `A` - all
- `N` - none, this can be used to display nothing (since `fE` is the default)

[Pytest documentation](#)

Dealing with errors - saving logs to file



Pytest allows us to redirect log output to a file. When You invoke the program with the command `pytest --report-log=path test_code.py`, where You replace the word 'path' with the path to an existing file with a `.log` extension (after installing report-log using `pip install report-log`), logs and information from the output will be saved to that file.

Handling log messages



Pytest captures logs related to warnings and other errors, displaying them automatically in a separate section of the output. Let's write a program that, in a test function, logs a message. First, we'll import the logging library, then perform basic configuration, and finally, we'll call the log at the warning level in the test function.

```
logging.basicConfig(level=logging.WARNING)
def test_1():
    logging.warning("Message")
    assert 1 + 1 == 3
```

Handling log messages



In the output, You will see information about this warning. The message is recorded in the format: module, line number, log level, and message. This format can be edited by pasting it into the `pytest.ini` file located in the main project directory (the same location as units and tests), satisfying the output format according to our preferences. In the `pytest.ini` file, we also have the option to set other configurations. For example, we can include `addopts = -v`, so that every test program invocation will include the `-v` flag.

Handling stdout/stderr messages



Pytest, just like capturing log messages, also captures messages directed to standard output and stderr. Caught messages will be shown in output. Pytest also allows access to stdout/stderr output from within the code. By capturing the `capsys` variable as an argument in a function and using `capsys.readouterr()`, we can obtain information about what messages were printed to stdout and stderr.

Handling stdout/stderr messages



```
def test_mYoutput(capsys):
    print("hello")
    sys.stderr.write("world\n")
    captured = capsys.readouterr()
    assert captured.out == "hello\n"
    assert captured.err == "world\n"
    print("next")
    captured = capsys.readouterr()
    assert captured.out == "next\n"
```

Example from [Pytest documentation](#), how to capture stdout and stderr

Handling warnings



Pytest also has the ability to capture warnings that occur in the code. In the output there is always information about every warning in code. Similarly to Python, pytest allows us to use the `-W` flag for warnings. Using this flag appropriately allows us to turn specific warnings into errors. Information about captured warnings can be disabled using the `--disable-warnings` flag.

Handling warnings



Warnings can also be captured in the code.

```
def test_fun2():
    with pytest.warns() as record:
        warnings.warn("user", UserWarning)
        warnings.warn("runtime",
                       RuntimeError)
    assert len(record) == 2
    assert str(record[0].message) == "u"
    assert str(record[1].message) == "r"
```

Example from [Pytest documentation](#), how to capture warnings

Handling warnings



```
def test_fun3(recwarn):  
    warnings.warn("hello", UserWarning)  
    assert len(recwarn) == 1  
    w = recwarn.pop(UserWarning)  
    assert isinstance(w.category,  
                      UserWarning)  
    assert str(w.message) == "hello"  
    assert w.filename  
    assert w.lineno
```

Example from [Pytest documentation](#), how to capture warnings

skip and xfail



Pytest also allows us to skip certain tests that we know will not be able to run. There are two different ways to skip test functions: using skip and xfail. We use the skip method when we expect the test to pass only in certain cases, while we use xfail when we anticipate that some tests will fail for some reason.

skip



Test functions can be skipped using the `@pytest.mark.skip` marker.

```
@pytest.mark.skip(reason="Skipping test")
def test_1():
    assert True
```

Another way to skip a test is to use the `pytest.skip()` method.

```
def test_2():
    a = 1
    if a != 2:
        pytest.skip("skip")
    assert True
```

[Pytest documentation, how to skip](#)

skipif



A similar function is performed by `@pytest.mark.skipif()`, where the first condition specifies under what circumstances the test should be skipped.

```
@pytest.mark.skipif(sys.version_info < (3, 10),  
                    reason="requires python3.10 or higher")  
def test_3():  
    assert True
```


importorskip



There is also the option to skip all functions that rely on a specific imported library if the import fails. Let's assume we are importing the `sys` library, and if the import fails, we want all tests using this library not to be executed. With the `pytest.importorskip(...)` method, we can achieve this.

```
sys = pytest.importorskip("sys")
```

xfail



By marking a function with `@pytest.mark.xfail`, we can skip the function, and it will be treated as an expected failure (xfail). We use this when we know that, for some reason, the function is expected to fail.

```
@pytest.mark.xfail
def test_4():
    assert True
```

xfail



Similarly to skip, we can also call xfail within the test code using the `pytest.xfail()` method, for example, when a certain condition in the code is met.

```
def test_5():  
    a = 1  
    if a != 2:  
        pytest.xfail("failing configuration")  
    assert True
```

xfail



Xfail also can take parameters:

- condition – first argument, condition for which a particular test will be marked as xfail
- reason – describing the reason for marking the function as an xfail
- raises – specify a particular exception. If this exception is not raised in the given function, the test will result in an xfail.
- run – when we want a test to be marked as xfail without even being executed, we use a special parameter `run=False`.
- strict – typically, xfail and xpass are treated separately in pytest output, with results displayed in yellow. However, if we want xfail and xpass to be treated as function failures, we can add the `strict=True` parameter to xfail.

--lf and --ff flags



Pytest also allows for the rerun of failed tests using the `--lf` flag, used during compilation. Similarly, You can use the `--ff` flag, which enables the rerun of first the failed tests and then the rest.

Introduction to monkeypatch/mock modules - WS



- 1 Assert keyword - MP
- 2 Running multiple tests and grouping them - BK
- 3 Fixtures - WS
- 4 Tests parameterization - MP
- 5 Managing tests output and outcomes - BK
- 6 Introduction to monkeypatch/mock modules - WS

Monkeypatch fixture



The Monkeypatch fixture allows developers to safely test those functionalities which require to use critical attributes like environment variables or system path. It also allows to mock some behaviours e.g we are expecting that our application will send *"Hello World"* at some level but currently it is under development or it will be developed in future.

Monkeypatch fixture



This fixture provides us with following methods:

Monkeypatch methods

- `monkeypatch.setattr(obj, name, value, raising=True)`
- `monkeypatch.delattr(obj, name, raising=True)`
- `monkeypatch.setitem(mapping, name, value)`
- `monkeypatch.delitem(obj, name, raising=True)`
- `monkeypatch.setenv(name, value, prepend=None)`
- `monkeypatch.delenv(name, raising=True)`
- `monkeypatch.syspath_prepend(path)`
- `monkeypatch.chdir(path)`
- `monkeypatch.context()`

Monkeypatch fixture



All modifications performed by methods from previous slide will be undone after the test or fixture has finished. Let's look at simple example:

```
class TestMonkeyPatch:
    def test_get_ssh(self, monkeypatch, tmp_path):
        def mockreturn ():
            return Path("/test")
        monkeypatch.setattr(Path, "home",
                             mockreturn)
        dummy_ssh = tmp_path / ".dummy_ssh"
        dummy_ssh.mkdir()
        assert Path.home() / ".dummy_ssh" ==
               Path("/test/.dummy_ssh")
```

Where tmp_path is another Pytest inbuilt fixture that creates temporary directory and deletes it after the test is done.

Monkeypatch fixture



Let's look where which monkeypatch method is useful:

Where to use monkeypatch methods

- If You need to modify behavior of a function of any property of a class for example there is a connection through ssh protocol to the server which you will not make in the test but you know what will be the output (for error in connection or for successful connection) use `monkeypatch.setattr` to patch the function with Your desired behaviour and then use `monkeypatch.delattr` to remove it.
- You have specific test cases in which You have to modify values of dictionaries. Use `monkeypatch.setitem` to patch the dictionary and `monkeypatch.delitem` to remove items.

Monkeypatch fixture



Where to use monkeypatch methods






- You have such a system that determines some action based on environment variables e.g you need to provide `proxy_addr` for `dhcp` but you would like to that is not left floating in system because it could crash your other connections. Use `monkeypatch.setenv` and `monkeypatch.delenv`.
- You need to change the context of directory during the test, use `monkeypatch.chdir`
- You need to change `$PATH`, u can use `monkeypatch.setenv` or `monkeypatch.syspath_prepend`.
- Use `monkeypatch.context` to apply certain patches for certain scopes, it is very helpful to control teardowns of complex fixtures.

Part IV

The End

Bibliography



-  <https://docs.pytest.org/en>
-  <https://tixware.wordpress.com/2015/05/20/why-do-we-need-software-testing/>
-  <https://www.tops-int.com/blog/7-reasons-why-software-testing-has-a-bright-future>
-  <https://www.zaptest.com/pl/czym-jest-automatyzacja-testow-bez-zargonu-prosty-przewodnik>
-  <https://blog.qalabs.pl/pytest/pytest-pierwsze-kroki/>
-  https://chyla.org/_static/files/artykuly/python/python-tutorial/Testowanie-aplikacji-Poziomy-testowania-pytest.pdf

The END

