

pytest fixtures: explicit, modular, scalable

New in version 2.0/2.3/2.4.

The purpose of test fixtures is to provide a fixed baseline upon which tests can reliably and repeatedly execute. pytest fixtures offer dramatic improvements over the classic xUnit style of setup/teardown functions:

- fixtures have explicit names and are activated by declaring their use from test functions, modules, classes or whole projects.
- fixtures are implemented in a modular manner, as each fixture name triggers a *fixture function* which can itself use other fixtures.
- fixture management scales from simple unit to complex functional testing, allowing to parametrize fixtures and tests according to configuration and component options, or to re-use fixtures across class, module or whole test session scopes.

In addition, pytest continues to support classic xunit-style setup. You can mix both styles, moving incrementally from classic to new style, as you prefer. You can also start out from existing unittest.TestCase style or nose based projects.

Fixtures as Function arguments

Test functions can receive fixture objects by naming them as an input argument. For each argument name, a fixture function with that name provides the fixture object. Fixture functions are registered by marking them with **@pytest.fixture**. Let's look at a simple self-contained test module containing a fixture and a test function using it:

```
# content of ./test_smtpsimple.py
import pytest

@pytest.fixture
def smtp():
    import smtplib
    return smtplib.SMTP("smtp.gmail.com", 587, timeout=5)

def test_ehlo(smtp):
    response, msg = smtp.ehlo()
    assert response == 250
    assert 0 # for demo purposes
```

Here, the test_ehlo needs the smtp fixture value. pytest will discover and call the **@pytest.fixture** marked smtp fixture function. Running the test looks like this:

```
$ pytest test_smtpsimple.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item

test_smtpsimple.py F

===== FAILURES =====
_____ test_ehlo _____

smtp = <smtplib.SMTP object at 0xdeadbeef>

    def test_ehlo(smtp):
        response, msg = smtp.ehlo()
        assert response == 250
>       assert 0 # for demo purposes
E       assert 0

test_smtpsimple.py:11: AssertionError
===== 1 failed in 0.12 seconds =====
```

In the failure traceback we see that the test function was called with a `smtp` argument, the `smtpplib.SMTP()` instance created by the fixture function. The test function fails on our deliberate `assert 0`. Here is the exact protocol used by pytest to call the test function this way:

1. pytest finds the `test_ehlo` because of the `test_` prefix. The test function needs a function argument named `smtp`. A matching fixture function is discovered by looking for a fixture-marked function named `smtp`.
2. `smtp()` is called to create an instance.
3. `test_ehlo(<SMTP instance>)` is called and fails in the last line of the test function.

Note that if you misspell a function argument or want to use one that isn't available, you'll see an error with a list of available function arguments.

Note:

You can always issue:

```
pytest --fixtures test_simplefactory.py
```

to see available fixtures.

Fixtures: a prime example of dependency injection

Fixtures allow test functions to easily receive and work against specific pre-initialized application objects without having to care about import/setup/cleanup details. It's a prime example of dependency injection where fixture functions take the role of the *injector* and test functions are the *consumers* of fixture objects.

Sharing a fixture across tests in a module (or class/session)

Fixtures requiring network access depend on connectivity and are usually time-expensive to create. Extending the previous example, we can add a `scope='module'` parameter to the `@pytest.fixture` invocation to cause the decorated `smtp` fixture function to only be invoked once per test module. Multiple test functions in a test module will thus each receive the same `smtp` fixture instance. The next example puts the fixture function into a separate `conftest.py` file so that tests from multiple test modules in the directory can access the fixture function:

```
# content of conftest.py
import pytest
import smtpplib

@pytest.fixture(scope="module")
def smtp():
    return smtpplib.SMTP("smtp.gmail.com", 587, timeout=5)
```

The name of the fixture again is `smtp` and you can access its result by listing the name `smtp` as an input parameter in any test or fixture function (in or below the directory where `conftest.py` is located):

```
# content of test_module.py

def test_ehlo(smtp):
    response, msg = smtp.ehlo()
    assert response == 250
    assert b"smtp.gmail.com" in msg
    assert 0 # for demo purposes

def test_noop(smtp):
    response, msg = smtp.noop()
    assert response == 250
    assert 0 # for demo purposes
```

 v: latest ▾

We deliberately insert failing `assert 0` statements in order to inspect what is going on and can now run the tests:

```
$ pytest test_module.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items

test_module.py FF

===== FAILURES =====
_____ test_ehlo _____

smtp = <smtplib.SMTP object at 0xdeadbeef>

    def test_ehlo(smtp):
        response, msg = smtp.ehlo()
        assert response == 250
        assert b"smtp.gmail.com" in msg
>       assert 0 # for demo purposes
E       assert 0

test_module.py:6: AssertionError
_____ test_noop _____

smtp = <smtplib.SMTP object at 0xdeadbeef>

    def test_noop(smtp):
        response, msg = smtp.noop()
        assert response == 250
>       assert 0 # for demo purposes
E       assert 0

test_module.py:11: AssertionError
===== 2 failed in 0.12 seconds =====
```

You see the two `assert 0` failing and more importantly you can also see that the same (module-scoped) `smtp` object was passed into the two test functions because pytest shows the incoming argument values in the traceback. As a result, the two test functions using `smtp` run as quick as a single one because they reuse the same instance.

If you decide that you rather want to have a session-scoped `smtp` instance, you can simply declare it:

```
@pytest.fixture(scope="session")
def smtp(...):
    # the returned fixture value will be shared for
    # all tests needing it
```

Fixture finalization / executing teardown code

pytest supports execution of fixture specific finalization code when the fixture goes out of scope. By using a `yield` statement instead of `return`, all the code after the `yield` statement serves as the teardown code:

```
# content of conftest.py

import smtplib
import pytest

@pytest.fixture(scope="module")
def smtp():
    smtp = smtplib.SMTP("smtp.gmail.com", 587, timeout=5)
    yield smtp # provide the fixture value
    print("teardown smtp")
    smtp.close()
```

 v: latest ▾

The `print` and `smtp.close()` statements will execute when the last test in the module has finished execution, regardless of the exception status of the tests.

Let's execute it:

```
$ pytest -s -q --tb=no
FFteardown smtp
```

```
2 failed in 0.12 seconds
```

We see that the `smtp` instance is finalized after the two tests finished execution. Note that if we decorated our fixture function with `scope='function'` then fixture setup and cleanup would occur around each single test. In either case the test module itself does not need to change or know about these details of fixture setup.

Note that we can also seamlessly use the `yield` syntax with `with` statements:

content of test_yield2.py

```
import smtplib
import pytest

@pytest.fixture(scope="module")
def smtp():
    with smtplib.SMTP("smtp.gmail.com", 587, timeout=5) as smtp:
        yield smtp # provide the fixture value
```

The `smtp` connection will be closed after the test finished execution because the `smtp` object automatically closes when the `with` statement ends.

Note that if an exception happens during the `setup` code (before the `yield` keyword), the `teardown` code (after the `yield`) will not be called.

An alternative option for executing `teardown` code is to make use of the `addfinalizer` method of the `request-context` object to register finalization functions.

Here's the `smtp` fixture changed to use `addfinalizer` for cleanup:

content of conftest.py

```
import smtplib
import pytest

@pytest.fixture(scope="module")
def smtp(request):
    smtp = smtplib.SMTP("smtp.gmail.com", 587, timeout=5)
    def fin():
        print ("teardown smtp")
        smtp.close()
    request.addfinalizer(fin)
    return smtp # provide the fixture value
```

Both `yield` and `addfinalizer` methods work similarly by calling their code after the test ends, but `addfinalizer` has two key differences over `yield`:

1. It is possible to register multiple finalizer functions.
2. Finalizers will always be called regardless if the fixture `setup` code raises an exception. This is handy to properly close all resources created by a fixture even if one of them fails to be created/acquired:

```
@pytest.fixture
def equipments(request):
    r = []
    for port in ('C1', 'C3', 'C28'):
```

 v: latest ▾

```

    equip = connect(port)
    request.addfinalizer(equip.disconnect)
    r.append(equip)
    return r

```

In the example above, if "C28" fails with an exception, "C1" and "C3" will still be properly closed. Of course, if an exception happens before the finalize function is registered then it will not be executed.

Fixtures can introspect the requesting test context

Fixture function can accept the **request** object to introspect the "requesting" test function, class or module context. Further extending the previous smtp fixture example, let's read an optional server URL from the test module which uses our fixture:

```

# content of conftest.py
import pytest
import smtplib

@pytest.fixture(scope="module")
def smtp(request):
    server = getattr(request.module, "smtpserver", "smtp.gmail.com")
    smtp = smtplib.SMTP(server, 587, timeout=5)
    yield smtp
    print ("finalizing %s (%s)" % (smtp, server))
    smtp.close()

```

We use the `request.module` attribute to optionally obtain an `smtpserver` attribute from the test module. If we just execute again, nothing much has changed:

```

$ pytest -s -q --tb=no
FFfinalizing <smtplib.SMTP object at 0xdeadbeef> (smtp.gmail.com)

2 failed in 0.12 seconds

```

Let's quickly create another test module that actually sets the server URL in its module namespace:

```

# content of test_anothersmtp.py

smtpserver = "mail.python.org" # will be read by smtp fixture

def test_showhelo(smtp):
    assert 0, smtp.helo()

```

Running it:



```

$ pytest -qq --tb=short test_anothersmtp.py
F
===== FAILURES =====
_____ test_showhelo _____
test_anothersmtp.py:5: in test_showhelo
    assert 0, smtp.helo()
E   AssertionError: (250, b'mail.python.org')
E   assert 0
----- Captured stdout teardown -----
finalizing <smtplib.SMTP object at 0xdeadbeef> (mail.python.org)

```

voila! The smtp fixture function picked up our mail server name from the module namespace.

Parametrizing fixtures

Fixture functions can be parametrized in which case they will be called multiple times, each time executing the set of dependent tests, i. e. the tests that depend on this fixture. Test functions do usually not need to be aware of their re-running.  **v: latest** 

ture parametrization helps to write exhaustive functional tests for components which themselves can be configured in multiple ways.

Extending the previous example, we can flag the fixture to create two smtp fixture instances which will cause all tests using the fixture to run twice. The fixture function gets access to each parameter through the special **request** object:

```
# content of conftest.py
import pytest
import smtplib

@pytest.fixture(scope="module",
                params=["smtp.gmail.com", "mail.python.org"])
def smtp(request):
    smtp = smtplib.SMTP(request.param, 587, timeout=5)
    yield smtp
    print("finalizing %s" % smtp)
    smtp.close()
```

The main change is the declaration of params with **@pytest.fixture**, a list of values for each of which the fixture function will execute and can access a value via `request.param`. No test function code needs to change. So let's just do another run:

```
$ pytest -q test_module.py
FFFF
===== FAILURES =====
_____ test_ehlo[smtp.gmail.com] _____

smtp = <smtplib.SMTP object at 0xdeadbeef>

    def test_ehlo(smtp):
        response, msg = smtp.ehlo()
        assert response == 250
        assert b"smtp.gmail.com" in msg
>       assert 0 # for demo purposes
E       assert 0

test_module.py:6: AssertionError
_____ test_noop[smtp.gmail.com] _____

smtp = <smtplib.SMTP object at 0xdeadbeef>

    def test_noop(smtp):
        response, msg = smtp.noop()
        assert response == 250
>       assert 0 # for demo purposes
E       assert 0

test_module.py:11: AssertionError
_____ test_ehlo[mail.python.org] _____

smtp = <smtplib.SMTP object at 0xdeadbeef>

    def test_ehlo(smtp):
        response, msg = smtp.ehlo()
        assert response == 250
>       assert b"smtp.gmail.com" in msg
E       AssertionError: assert b'smtp.gmail.com' in b'mail.python.org\nPIPELINING\nSIZE 51200000\nETRN\nSTAF

test_module.py:5: AssertionError
----- Captured stdout setup -----
finalizing <smtplib.SMTP object at 0xdeadbeef>
_____ test_noop[mail.python.org] _____

smtp = <smtplib.SMTP object at 0xdeadbeef>

    def test_noop(smtp):
```

 v: latest ▼

```

        response, msg = smtp.noop()
        assert response == 250
>       assert 0 # for demo purposes
E       assert 0

test_module.py:11: AssertionError
----- Captured stdout teardown -----
finalizing <smtp.SMTP object at 0xdeadbeef>
4 failed in 0.12 seconds

```

We see that our two test functions each ran twice, against the different smtp instances. Note also, that with the mail.python.org connection the second test fails in test_ehlo because a different server string is expected than what arrived.

pytest will build a string that is the test ID for each fixture value in a parametrized fixture, e.g. test_ehlo[smtp.gmail.com] and test_ehlo[mail.python.org] in the above examples. These IDs can be used with -k to select specific cases to run, and they will also identify the specific case when one is failing. Running pytest with --collect-only will show the generated IDs.

Numbers, strings, booleans and None will have their usual string representation used in the test ID. For other objects, pytest will make a string based on the argument name. It is possible to customise the string used in a test ID for a certain fixture value by using the ids keyword argument:

```

# content of test_ids.py
import pytest

@pytest.fixture(params=[0, 1], ids=["spam", "ham"])
def a(request):
    return request.param

def test_a(a):
    pass

def idfn(fixture_value):
    if fixture_value == 0:
        return "eggs"
    else:
        return None

@pytest.fixture(params=[0, 1], ids=idfn)
def b(request):
    return request.param

def test_b(b):
    pass

```

The above shows how ids can be either a list of strings to use or a function which will be called with the fixture value and then has to return a string to use. In the latter case if the function return None then pytest's auto-generated ID will be used.

Running the above tests results in the following test IDs being used:

```

$ pytest --collect-only
===== test session starts =====
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 10 items
<Module 'test_anothersmtp.py'>
  <Function 'test_showhelo[smtp.gmail.com]'>
  <Function 'test_showhelo[mail.python.org]'>
<Module 'test_ids.py'>
  <Function 'test_a[spam]'>
  <Function 'test_a[ham]'>
  <Function 'test_b[eggs]'>
  <Function 'test_b[1]'>
<Module 'test_module.py'>

```

 v: latest ▼

```
<Function 'test_ehlo[smtp.gmail.com]'\>
<Function 'test_noop[smtp.gmail.com]'\>
<Function 'test_ehlo[mail.python.org]'\>
<Function 'test_noop[mail.python.org]'\>
```

```
===== no tests ran in 0.12 seconds =====
```

Modularity: using fixtures from a fixture function

You can not only use fixtures in test functions but fixture functions can use other fixtures themselves. This contributes to a modular design of your fixtures and allows re-use of framework-specific fixtures across many projects. As a simple example, we can extend the previous example and instantiate an object app where we stick the already defined smtp resource into it:

content of test_appsetup.py

```
import pytest

class App(object):
    def __init__(self, smtp):
        self.smtp = smtp

@pytest.fixture(scope="module")
def app(smtp):
    return App(smtp)

def test_smtp_exists(app):
    assert app.smtp
```

Here we declare an app fixture which receives the previously defined smtp fixture and instantiates an App object with it. Let's run it:

```
$ pytest -v test_appsetup.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python3.5
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 2 items

test_appsetup.py::test_smtp_exists[smtp.gmail.com] PASSED
test_appsetup.py::test_smtp_exists[mail.python.org] PASSED

===== 2 passed in 0.12 seconds =====
```

Due to the parametrization of smtp the test will run twice with two different App instances and respective smtp servers. There is no need for the app fixture to be aware of the smtp parametrization as pytest will fully analyse the fixture dependency graph.

Note, that the app fixture has a scope of module and uses a module-scoped smtp fixture. The example would still work if smtp was cached on a session scope: it is fine for fixtures to use "broader" scoped fixtures but not the other way round: A session-scoped fixture could not use a module-scoped one in a meaningful way.

Automatic grouping of tests by fixture instances

pytest minimizes the number of active fixtures during test runs. If you have a parametrized fixture, then all the tests using it will first execute with one instance and then finalizers are called before the next fixture instance is created. Among other things, this eases testing of applications which create and use global state.

The following example uses two parametrized fixture, one of which is scoped on a per-module basis, and all the functions perform print calls to show the setup/teardown flow:

content of test_module.py

 v: latest ▼


```

import pytest

@pytest.fixture(scope="module", params=["mod1", "mod2"])
def modarg(request):
    param = request.param
    print ("  SETUP modarg %s" % param)
    yield param
    print ("  TEARDOWN modarg %s" % param)

@pytest.fixture(scope="function", params=[1,2])
def otherarg(request):
    param = request.param
    print ("  SETUP otherarg %s" % param)
    yield param
    print ("  TEARDOWN otherarg %s" % param)

def test_0(otherarg):
    print ("  RUN test0 with otherarg %s" % otherarg)
def test_1(modarg):
    print ("  RUN test1 with modarg %s" % modarg)
def test_2(otherarg, modarg):
    print ("  RUN test2 with otherarg %s and modarg %s" % (otherarg, modarg))

```

Let's run the tests in verbose mode and with looking at the print-output:

```

$ pytest -v -s test_module.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python3.5
cachedir: .cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 8 items

test_module.py::test_0[1]  SETUP otherarg 1
  RUN test0 with otherarg 1
PASSED  TEARDOWN otherarg 1

test_module.py::test_0[2]  SETUP otherarg 2
  RUN test0 with otherarg 2
PASSED  TEARDOWN otherarg 2

test_module.py::test_1[mod1]  SETUP modarg mod1
  RUN test1 with modarg mod1
PASSED
test_module.py::test_2[1-mod1]  SETUP otherarg 1
  RUN test2 with otherarg 1 and modarg mod1
PASSED  TEARDOWN otherarg 1

test_module.py::test_2[2-mod1]  SETUP otherarg 2
  RUN test2 with otherarg 2 and modarg mod1
PASSED  TEARDOWN otherarg 2

test_module.py::test_1[mod2]  TEARDOWN modarg mod1
  SETUP modarg mod2
  RUN test1 with modarg mod2
PASSED
test_module.py::test_2[1-mod2]  SETUP otherarg 1
  RUN test2 with otherarg 1 and modarg mod2
PASSED  TEARDOWN otherarg 1

test_module.py::test_2[2-mod2]  SETUP otherarg 2
  RUN test2 with otherarg 2 and modarg mod2
PASSED  TEARDOWN otherarg 2
  TEARDOWN modarg mod2

===== 8 passed in 0.12 seconds =====

```

 v:latest ▼

You can see that the parametrized module-scoped `modarg` resource caused an ordering of test execution that lead to the fewest possible “active” resources. The finalizer for the `mod1` parametrized resource was executed before the `mod2` resource was setup.

In particular notice that `test_0` is completely independent and finishes first. Then `test_1` is executed with `mod1`, then `test_2` with `mod1`, then `test_1` with `mod2` and finally `test_2` with `mod2`.

The `otherarg` parametrized resource (having function scope) was set up before and teared down after every test that used it.

Using fixtures from classes, modules or projects

Sometimes test functions do not directly need access to a fixture object. For example, tests may require to operate with an empty directory as the current working directory but otherwise do not care for the concrete directory. Here is how you can use the standard `tempfile` and pytest fixtures to achieve it. We separate the creation of the fixture into a `conftest.py` file:

```
# content of conftest.py

import pytest
import tempfile
import os

@pytest.fixture()
def cleandir():
    newpath = tempfile.mkdtemp()
    os.chdir(newpath)
```

and declare its use in a test module via a `usefixtures` marker:

```
# content of test_setenv.py
import os
import pytest

@pytest.mark.usefixtures("cleandir")
class TestDirectoryInit(object):
    def test_cwd_starts_empty(self):
        assert os.listdir(os.getcwd()) == []
        with open("myfile", "w") as f:
            f.write("hello")

    def test_cwd_again_starts_empty(self):
        assert os.listdir(os.getcwd()) == []
```

Due to the `usefixtures` marker, the `cleandir` fixture will be required for the execution of each test method, just as if you specified a “`cleandir`” function argument to each of them. Let’s run it to verify our fixture is activated and the tests pass:

```
$ pytest -q
..
2 passed in 0.12 seconds
```

You can specify multiple fixtures like this:

```
@pytest.mark.usefixtures("cleandir", "anotherfixture")
```

and you may specify fixture usage at the test module level, using a generic feature of the mark mechanism:

```
pytestmark = pytest.mark.usefixtures("cleandir")
```

Note that the assigned variable *must* be called `pytestmark`, assigning e.g. `foomark` will not activate the fixtures.

Lastly you can put fixtures required by all tests in your project into an ini-file:

 v: latest ▼

```
# content of pytest.ini
[pytest]
usefixtures = cleandir
```

Autouse fixtures (xUnit setup on steroids)

Occasionally, you may want to have fixtures get invoked automatically without declaring a function argument explicitly or a `usefixtures` decorator. As a practical example, suppose we have a database fixture which has a begin/rollback/commit architecture and we want to automatically surround each test method by a transaction and a rollback. Here is a dummy self-contained implementation of this idea:

```
# content of test_db_transact.py

import pytest

class DB(object):
    def __init__(self):
        self.intransaction = []
    def begin(self, name):
        self.intransaction.append(name)
    def rollback(self):
        self.intransaction.pop()

@pytest.fixture(scope="module")
def db():
    return DB()

class TestClass(object):
    @pytest.fixture(autouse=True)
    def transact(self, request, db):
        db.begin(request.function.__name__)
        yield
        db.rollback()

    def test_method1(self, db):
        assert db.intransaction == ["test_method1"]

    def test_method2(self, db):
        assert db.intransaction == ["test_method2"]
```

The class-level transact fixture is marked with `autouse=true` which implies that all test methods in the class will use this fixture without a need to state it in the test function signature or with a class-level `usefixtures` decorator.

If we run it, we get two passing tests:

```
$ pytest -q
..
2 passed in 0.12 seconds
```

Here is how autouse fixtures work in other scopes:

- autouse fixtures obey the `scope=` keyword-argument: if an autouse fixture has `scope='session'` it will only be run once, no matter where it is defined. `scope='class'` means it will be run once per class, etc.
- if an autouse fixture is defined in a test module, all its test functions automatically use it.
- if an autouse fixture is defined in a `conftest.py` file then all tests in all test modules below its directory will invoke the fixture.
- lastly, and please use that with care: if you define an autouse fixture in a plugin, it will be invoked for all tests in all projects where the plugin is installed. This can be useful if a fixture only anyway works in the presence of certain settings e. g. in the ini-file. Such a global fixture should always quickly determine if it should do any work and avoid otherwise expensive imports or computation.

 v: latest ▾

Note that the above `transact` fixture may very well be a fixture that you want to make available in your project without having it generally active. The canonical way to do that is to put the `transact` definition into a `conftest.py` file without using `autouse`:

```
# content of conftest.py
@pytest.fixture
def transact(self, request, db):
    db.begin()
    yield
    db.rollback()
```

and then e.g. have a `TestClass` using it by declaring the need:

```
@pytest.mark.usefixtures("transact")
class TestClass(object):
    def test_method1(self):
        ...
```

All test methods in this `TestClass` will use the transaction fixture while other test classes or functions in the module will not use it unless they also add a `transact` reference.

Shifting (visibility of) fixture functions

If during implementing your tests you realize that you want to use a fixture function from multiple test files you can move it to a `conftest.py` file or even separately installable plugins without changing test code. The discovery of fixtures functions starts at test classes, then test modules, then `conftest.py` files and finally builtin and third party plugins.

Overriding fixtures on various levels

In relatively large test suite, you most likely need to override a global or root fixture with a locally defined one, keeping the test code readable and maintainable.

Override a fixture on a folder (`conftest`) level

Given the tests file structure is:

```
tests/
  __init__.py

  conftest.py
    # content of tests/conftest.py
    import pytest

    @pytest.fixture
    def username():
        return 'username'

  test_something.py
    # content of tests/test_something.py
    def test_username(username):
        assert username == 'username'

  subfolder/
    __init__.py

    conftest.py
      # content of tests/subfolder/conftest.py
      import pytest

      @pytest.fixture
```

 v: latest ▼

```

def username(username):
    return 'overridden-' + username

test_something.py
# content of tests/subfolder/test_something.py
def test_username(username):
    assert username == 'overridden-username'

```

As you can see, a fixture with the same name can be overridden for certain test folder level. Note that the base or super fixture can be accessed from the overriding fixture easily - used in the example above.

Override a fixture on a test module level

Given the tests file structure is:

```

tests/
__init__.py

conftest.py
# content of tests/conftest.py
@pytest.fixture
def username():
    return 'username'

test_something.py
# content of tests/test_something.py
import pytest

@pytest.fixture
def username(username):
    return 'overridden-' + username

def test_username(username):
    assert username == 'overridden-username'

test_something_else.py
# content of tests/test_something_else.py
import pytest

@pytest.fixture
def username(username):
    return 'overridden-else-' + username

def test_username(username):
    assert username == 'overridden-else-username'

```

In the example above, a fixture with the same name can be overridden for certain test module.

Override a fixture with direct test parametrization

Given the tests file structure is:

```

tests/
__init__.py

conftest.py
# content of tests/conftest.py
import pytest

@pytest.fixture
def username():
    return 'username'

@pytest.fixture

```

 v: latest ▼

```

def other_username(username):
    return 'other-' + username

test_something.py
# content of tests/test_something.py
import pytest

@pytest.mark.parametrize('username', ['directly-overridden-username'])
def test_username(username):
    assert username == 'directly-overridden-username'

@pytest.mark.parametrize('username', ['directly-overridden-username-other'])
def test_username_other(other_username):
    assert other_username == 'other-directly-overridden-username-other'

```

In the example above, a fixture value is overridden by the test parameter value. Note that the value of the fixture can be overridden this way even if the test doesn't use it directly (doesn't mention it in the function prototype).

Override a parametrized fixture with non-parametrized one and vice versa

Given the tests file structure is:

```

tests/
__init__.py

conftest.py
# content of tests/conftest.py
import pytest

@pytest.fixture(params=['one', 'two', 'three'])
def parametrized_username(request):
    return request.param

@pytest.fixture
def non_parametrized_username(request):
    return 'username'

test_something.py
# content of tests/test_something.py
import pytest

@pytest.fixture
def parametrized_username():
    return 'overridden-username'

@pytest.fixture(params=['one', 'two', 'three'])
def non_parametrized_username(request):
    return request.param

def test_username(parametrized_username):
    assert parametrized_username == 'overridden-username'

def test_parametrized_username(non_parametrized_username):
    assert non_parametrized_username in ['one', 'two', 'three']

test_something_else.py
# content of tests/test_something_else.py
def test_username(parametrized_username):
    assert parametrized_username in ['one', 'two', 'three']

def test_username(non_parametrized_username):
    assert non_parametrized_username == 'username'

```

In the example above, a parametrized fixture is overridden with a non-parametrized version, and a non-parametrized fixture is overridden with a parametrized version for certain test module. The same applies for the test folder level obviously.