
pytest-dependency Documentation

Release 0.4.0

Rolf Krahl

Dec 02, 2018

Contents

1	Content of the documentation	3
1.1	About pytest-dependency	3
1.2	Installation instructions	4
1.3	Using pytest-dependency	5
1.4	Advanced usage	9
1.5	Configuring pytest-dependency	13
1.6	History of changes to pytest-dependency	14
1.7	Reference	15
	Python Module Index	17

This pytest plugin manages dependencies of tests. It allows to mark some tests as dependent from other tests. These tests will then be skipped if any of the dependencies did fail or has been skipped.

1.1 About pytest-dependency

This module is a plugin for the popular Python testing framework `pytest`. It manages dependencies of tests: you may mark some tests as dependent from other tests. These tests will then be skipped if any of the dependencies did fail or has been skipped.

1.1.1 What is the purpose?

In the theory of good test design, tests should be self-contained and independent. Each test should cover one single issue, either verify that one single feature is working or that one single bug is fixed. Tests should be designed to work in any order independent of each other.

So far the theory. The practice is often more complicated than that. Sometimes, the principle of independency of tests is simply unrealistic or impractical. Program features often depend on each other. If some feature B depends on another feature A in such a way that B cannot work without A, then it may simply be pointless to run the test for B unless the test for A has succeeded. Another case may be if the subject of the tests has an internal state that unavoidably is influenced by the tests. In this situation it may happen that test A, as a side effect, sets the system in some state that is the precondition to be able to run test B. Again, in this case it would be pointless to try running test B unless test A has been run successful.

It should be emphasized however that the principle of independency of tests is still valid. Before using `pytest-dependency`, it is still advisable to reconsider your test design and to avoid dependencies of tests whenever possible, rather than to manage these dependencies.

1.1.2 How does it work?

The `pytest-dependency` module defines a marker that can be applied to tests. The marker accepts an argument that allows to list the dependencies of the test. Both tests, the dependency and the dependent test should be decorated with the marker. Behind the scenes, the marker arranges for the result of the test to be recorded internally. If a list of dependencies has been given as argument, the marker verifies that a successful outcome of all the dependencies has been registered previously and causes a skip of the test if this was not the case.

1.1.3 Why is this useful?

The benefit of skipping dependent tests is the same as for skipping tests in general: it avoids cluttering the test report with useless and misleading failure reports from tests that have been known beforehand not to work in this particular case.

If tests depend on each other in such a way that test B cannot work unless test A has been run successfully, a failure of test A will likely result in failure messages from both tests. But the failure message from test B will not be helpful in any way. It will only distract the user from the real issue that is the failure of test A. Skipping test B in this case will help the user to concentrate on those results that really matter.

1.1.4 Copyright and License

- Copyright 2013-2015 Helmholtz-Zentrum Berlin für Materialien und Energie GmbH
- Copyright 2016-2018 Rolf Krahl

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

1.2 Installation instructions

1.2.1 System requirements

- Python 2.7 or 3.4 and newer.
- `setuptools`.
- `pytest` 3.6.0 or newer.

1.2.2 Interaction with other packages

pytest-xdist `pytest-xdist` features test run parallelization, e.g. distributing tests over separate processes that run in parallel. This is based on the assumption that the tests can be run independent of each other. Obviously, if you are using `pytest-dependency`, this assumption is not valid. Thus, `pytest-dependency` will only work if you do not enable parallelization in `pytest-xdist`.

1.2.3 Download

The latest release version of `pytest-dependency` source can be found at PyPI, see

https://pypi.python.org/pypi/pytest_dependency

1.2.4 Installation

1. Download the sources, unpack, and change into the source directory.
2. Build (optional):

```
$ python setup.py build
```

3. Test (optional):

```
$ python -m pytest
```

4. Install:

```
$ python setup.py install
```

The last step might require admin privileges in order to write into the site-packages directory of your Python installation.

For production use, it is always recommended to use the latest release version from PyPI, see above.

1.3 Using pytest-dependency

The plugin defines a new marker `pytest.mark.dependency()`.

1.3.1 Basic usage

Consider the following example test module:

```
import pytest

@pytest.mark.dependency()
@pytest.mark.xfail(reason="deliberate fail")
def test_a():
    assert False

@pytest.mark.dependency()
def test_b():
    pass

@pytest.mark.dependency(depends=["test_a"])
def test_c():
    pass

@pytest.mark.dependency(depends=["test_b"])
def test_d():
    pass

@pytest.mark.dependency(depends=["test_b", "test_c"])
def test_e():
    pass
```

All the tests are decorated with `pytest.mark.dependency()`. This will cause the test results to be registered internally and thus other tests may depend on them. The list of dependencies of a test may be set in the optional `depends` argument to the marker. Running this test, we will get the following result:

```

$ pytest -rsx basic.py
===== test session starts =====
platform linux -- Python 3.4.6, pytest-2.8.7, py-1.4.30, pluggy-0.3.1
rootdir: /home/user/pytest-dependency-0.2, inifile:
plugins: dependency-0.2
collected 5 items

basic.py x.s.s
===== short test summary info =====
SKIP [1] /usr/lib/python3.4/site-packages/pytest_dependency.py:65: test_e depends on_
↳test_c
SKIP [1] /usr/lib/python3.4/site-packages/pytest_dependency.py:65: test_c depends on_
↳test_a
XFAIL basic.py::test_a
  deliberate fail

===== 2 passed, 2 skipped, 1 xfailed in 0.02 seconds =====

```

The first test has deliberately been set to fail to illustrate the effect. We will get the following results:

test_a deliberately fails.

test_b succeeds.

test_c will be skipped because it depends on *test_a*.

test_d depends on *test_b* which did succeed. It will be run and succeed as well.

test_e depends on *test_b* and *test_c*. *test_b* did succeed, but *test_c* has been skipped. So this one will also be skipped.

1.3.2 Naming tests

Tests are referenced by their name in the *depends* argument. The default for this name is the node ID defined by pytest, that is the name of the test function, extended by the parameters if applicable. In some cases, it's not easy to predict the names of the node IDs. For this reason, the name of the tests can be overridden by an explicit *name* argument to the marker. The names must be unique in the scope, which is currently the test module. The following example works exactly as the last one, only the test names are explicitly set:

```

import pytest

@pytest.mark.dependency(name="a")
@pytest.mark.xfail(reason="deliberate fail")
def test_a():
    assert False

@pytest.mark.dependency(name="b")
def test_b():
    pass

@pytest.mark.dependency(name="c", depends=["a"])
def test_c():
    pass

@pytest.mark.dependency(name="d", depends=["b"])
def test_d():
    pass

```

(continues on next page)

(continued from previous page)

```
@pytest.mark.dependency(name="e", depends=["b", "c"])
def test_e():
    pass
```

1.3.3 Using test classes

Tests may be grouped in classes in pytest. Marking the dependencies of methods in test classes works the same way as for simple test functions. In the following example we define two test classes. Each works in the same manner as the previous examples respectively:

```
import pytest

class TestClass(object):

    @pytest.mark.dependency()
    @pytest.mark.xfail(reason="deliberate fail")
    def test_a(self):
        assert False

    @pytest.mark.dependency()
    def test_b(self):
        pass

    @pytest.mark.dependency(depends=["TestClass::test_a"])
    def test_c(self):
        pass

    @pytest.mark.dependency(depends=["TestClass::test_b"])
    def test_d(self):
        pass

    @pytest.mark.dependency(depends=["TestClass::test_b", "TestClass::test_c"])
    def test_e(self):
        pass

class TestClassNamed(object):

    @pytest.mark.dependency(name="a")
    @pytest.mark.xfail(reason="deliberate fail")
    def test_a(self):
        assert False

    @pytest.mark.dependency(name="b", depends=["a"])
    def test_b(self):
        pass

    @pytest.mark.dependency(name="c")
    def test_c(self):
        pass

    @pytest.mark.dependency(name="d", depends=["c"])
    def test_d(self):
        pass
```

(continues on next page)

```
@pytest.mark.dependency(name="e", depends=["b", "c"])
def test_e(self):
    pass
```

In *TestClass* the default names for the tests are used, which is build from the name of the class and the respective method in this case, while in *TestClassNamed* these names are overridden by an explicit name argument to the `pytest.mark.dependency()` marker.

Changed in version 0.3: The name of the class is prepended to method name to form the default name for the test.

1.3.4 Parametrized tests

In the same way as the `pytest.mark.skip()` and `pytest.mark.xfail()` markers, the `pytest.mark.dependency()` marker may be applied to individual test instances in the case of parametrized tests. Consider the following example:

```
import pytest

@pytest.mark.parametrize("x,y", [
    pytest.param(0, 0, marks=pytest.mark.dependency(name="a1")),
    pytest.param(0, 1, marks=[pytest.mark.dependency(name="a2"),
                             pytest.mark.xfail]),
    pytest.param(1, 0, marks=pytest.mark.dependency(name="a3")),
    pytest.param(1, 1, marks=pytest.mark.dependency(name="a4"))
])
def test_a(x,y):
    assert y <= x

@pytest.mark.parametrize("u,v", [
    pytest.param(1, 2, marks=pytest.mark.dependency(name="b1",
                                                    depends=["a1", "a2"])),
    pytest.param(1, 3, marks=pytest.mark.dependency(name="b2",
                                                    depends=["a1", "a3"])),
    pytest.param(1, 4, marks=pytest.mark.dependency(name="b3",
                                                    depends=["a1", "a4"])),
    pytest.param(2, 3, marks=pytest.mark.dependency(name="b4",
                                                    depends=["a2", "a3"])),
    pytest.param(2, 4, marks=pytest.mark.dependency(name="b5",
                                                    depends=["a2", "a4"])),
    pytest.param(3, 4, marks=pytest.mark.dependency(name="b6",
                                                    depends=["a3", "a4"]))
])
def test_b(u,v):
    pass

@pytest.mark.parametrize("w", [
    pytest.param(1, marks=pytest.mark.dependency(name="c1",
                                                  depends=["b1", "b2", "b6"])),
    pytest.param(2, marks=pytest.mark.dependency(name="c2",
                                                  depends=["b2", "b3", "b6"])),
    pytest.param(3, marks=pytest.mark.dependency(name="c3",
                                                  depends=["b2", "b4", "b6"]))
])
def test_c(w):
    pass
```

The test instance `test_a[0-1]`, named `a2` in the `pytest.mark.dependency()` marker, is going to fail. As a result, the dependent tests `b1`, `b4`, `b5`, and in turn `c1` and `c3` will be skipped.

1.3.5 Marking dependencies at runtime

Sometimes, dependencies of test instances are too complicated to be formulated explicitly beforehand using the `pytest.mark.dependency()` marker. It may be easier to compile the list of dependencies of a test at run time. In such cases, the function `pytest_dependency.depends()` comes handy. Consider the following example:

```
import pytest
from pytest_dependency import depends

@pytest.mark.dependency()
def test_a():
    pass

@pytest.mark.dependency()
@pytest.mark.xfail(reason="deliberate fail")
def test_b():
    assert False

@pytest.mark.dependency()
def test_c(request):
    depends(request, ["test_b"])
    pass

@pytest.mark.dependency()
def test_d(request):
    depends(request, ["test_a", "test_c"])
    pass
```

Tests `test_c` and `test_d` set their dependencies at runtime calling `pytest_dependency.depends()`. The first argument is the value of the `request` pytest fixture, the second argument is the list of dependencies. It has the same effect as passing this list as the `depends` argument to the `pytest.mark.dependency()` marker.

The present example is certainly somewhat artificial, as the use of the `pytest_dependency.depends()` function would not be needed in such a simple case. For a more involved example that can not as easily be formulated with the static the `depends` argument, see [Grouping tests using fixtures](#).

1.4 Advanced usage

This section contains some advanced examples for using `pytest-dependency`.

1.4.1 Dynamic compilation of marked parameters

Sometimes, the parameter values for parametrized tests cannot easily be typed as a simple list. It may need to be compiled at run time depending on a set of test data. This also works together with marking dependencies in the individual test instances.

Consider the following example test module:

```

import pytest

# Test data
# Consider a bunch of Nodes, some of them are parents and some are children.

class Node(object):
    NodeMap = {}
    def __init__(self, name, parent=None):
        self.name = name
        self.children = []
        self.NodeMap[self.name] = self
        if parent:
            self.parent = self.NodeMap[parent]
            self.parent.children.append(self)
        else:
            self.parent = None
    def __str__(self):
        return self.name

parents = [ Node("a"), Node("b"), Node("c"), Node("d"), ]
childs = [ Node("e", "a"), Node("f", "a"), Node("g", "a"),
          Node("h", "b"), Node("i", "c"), Node("j", "c"),
          Node("k", "d"), Node("l", "d"), Node("m", "d"), ]

# The test for the parent shall depend on the test of all its children.
# Create enriched parameter lists, decorated with the dependency marker.

childparam = [
    pytest.param(c, marks=pytest.mark.dependency(name="test_child[%s]" % c))
    for c in childs
]

parentparam = [
    pytest.param(p, marks=pytest.mark.dependency(
        name="test_parent[%s]" % p,
        depends=["test_child[%s]" % c for c in p.children]
    )) for p in parents
]

@pytest.mark.parametrize("c", childparam)
def test_child(c):
    if c.name == "l":
        pytest.xfail("deliberate fail")
        assert False

@pytest.mark.parametrize("p", parentparam)
def test_parent(p):
    pass

```

In principle, this example works the very same way as the basic example for *Parametrized tests*. The only difference is that the lists of paramters are dynamically compiled beforehand. The test for child *l* deliberately fails, just to show the effect. As a consequence, the test for its parent *d* will be skipped.

1.4.2 Grouping tests using fixtures

pytest features the *automatic grouping of tests by fixture instances*. This is particularly useful if there is a set of test cases and a series of tests shall be run for each of the test case respectively.

Consider the following example:

```
import pytest
from pytest_dependency import depends

@pytest.fixture(scope="module", params=range(1,10))
def testcase(request):
    param = request.param
    return param

@pytest.mark.dependency()
def test_a(testcase):
    if testcase % 7 == 0:
        pytest.xfail("deliberate fail")
        assert False

@pytest.mark.dependency()
def test_b(request, testcase):
    depends(request, ["test_a[%d]" % testcase])
    pass
```

The test instances of `test_b` depend on `test_a` for the same parameter value. The test `test_a[7]` deliberately fails, as a consequence `test_b[7]` will be skipped. Note that we need to call `pytest_dependency.depends()` to mark the dependencies, because there is no way to use the `pytest.mark.dependency()` marker on the parameter values here.

If many tests in the series depend on a single test, it might be an option, to move the call to `pytest_dependency.depends()` in a fixture on its own. Consider:

```
import pytest
from pytest_dependency import depends

@pytest.fixture(scope="module", params=range(1,10))
def testcase(request):
    param = request.param
    return param

@pytest.fixture(scope="module")
def dep_testcase(request, testcase):
    depends(request, ["test_a[%d]" % testcase])
    return testcase

@pytest.mark.dependency()
def test_a(testcase):
    if testcase % 7 == 0:
        pytest.xfail("deliberate fail")
        assert False

@pytest.mark.dependency()
def test_b(dep_testcase):
    pass

@pytest.mark.dependency()
def test_c(dep_testcase):
    pass
```

In this example, both `test_b[7]` and `test_c[7]` are skipped, because `test_a[7]` deliberately fails.

1.4.3 Depend on all instances of a parametrized test at once

If a test depends on a all instances of a parametrized test at once, listing all of them in the `pytest.mark.dependency()` marker explicitly might not be the best solution. But you can dynamically compile these lists from the parameter values, as in the following example:

```
import pytest

def instances(name, params):
    def vstr(val):
        if isinstance(val, (list, tuple)):
            return "-".join([str(v) for v in val])
        else:
            return str(val)
    return ["%s[%s]" % (name, vstr(v)) for v in params]

params_a = range(17)

@pytest.mark.parametrize("x", params_a)
@pytest.mark.dependency()
def test_a(x):
    if x == 13:
        pytest.xfail("deliberate fail")
        assert False
    else:
        pass

@pytest.mark.dependency(depends=instances("test_a", params_a))
def test_b():
    pass

params_c = list(zip(range(0, 8, 2), range(2, 6)))

@pytest.mark.parametrize("x,y", params_c)
@pytest.mark.dependency()
def test_c(x, y):
    if x > y:
        pytest.xfail("deliberate fail")
        assert False
    else:
        pass

@pytest.mark.dependency(depends=instances("test_c", params_c))
def test_d():
    pass

params_e = ['abc', 'def']

@pytest.mark.parametrize("s", params_e)
@pytest.mark.dependency()
def test_e(s):
    if 'e' in s:
        pytest.xfail("deliberate fail")
        assert False
    else:
        pass
```

(continues on next page)

(continued from previous page)

```
@pytest.mark.dependency(depends=instances("test_e", params_e))
def test_f():
    pass
```

Here, `test_b`, `test_d`, and `test_f` will be skipped because they depend on all instances of `test_a`, `test_c`, and `test_e` respectively, but `test_a[13]`, `test_c[6-5]`, and `test_e[def]` fail. The list of the test instances is compiled in the helper function `instances()`.

Unfortunately you need knowledge how pytest encodes parameter values in test instance names to write this helper function. Note in particular how lists of parameter values are compiled into one single string in the case of multi parameter tests. But also note that this example of the `instances()` helper will only work for simple cases. It requires the parameter values to be scalars that can easily be converted to strings. And it will fail if the same list of parameters is passed to the same test more then once, because then, pytest will add an index to the name to disambiguate the parameter values.

1.5 Configuring pytest-dependency

This section explains configuration options for pytest-dependency, but also options for pytest itself or other plugins that are recommended for the use with pytest-dependency.

1.5.1 Notes on configuration for other plugins

pytest-xdist Test run parallelization in pytest-xdist is incompatible with pytest-dependency, see [Interaction with other packages](#). By default, parallelization is disabled in pytest-xdist (`-dist=no`). You are advised to leave this default.

1.5.2 Configuration file options

Configuration file options can be set in the *ini file*.

minversion This is a builtin configuration option of pytest itself. Since pytest-dependency requires pytest 3.6.0 or newer, it is recommended to set this option accordingly, either to 3.6.0 or to a newer version, if required by your test code.

automark_dependency This is a flag. If set to *False*, the default, the outcome of a test will only be registered if the test has been decorated with the `pytest.mark.dependency()` marker. As a results, all tests, the dependencies and the dependent tests must be decorated. If set to *True*, the outcome of all tests will be registered. It has the same effect as implicitly decorating all tests with `pytest.mark.dependency()`.

New in version 0.3.

1.5.3 Command line options

The following command line options are added by `pytest.dependency`:

`-ignore-unknown-dependency` By default, a test will be skipped unless all the dependencies have been run successful. If this option is set, a test will be skipped if any of the dependencies has been skipped or failed. E.g. dependencies that have not been run at all will be ignored.

This may be useful if you run only a subset of the testsuite and some tests in the selected set are marked to depend on other tests that have not been selected.

New in version 0.3.

1.6 History of changes to pytest-dependency

0.4 (2018-12-02)

Incompatible changes

- Require pytest version 3.6.0 or newer. This implicitly drops support for Python 2.6 and for Python 3.3 and older.

Bug fixes and minor changes

- Issue #24: get_marker no longer available in pytest 4.0.0. (Thanks to Rogdham for the PR.)
- Issue #28: Applying markers directly in parametrize is no longer available in 4.0.

0.3.2 (2018-01-17)

Bug fixes and minor changes

- Issue #5: properly register the dependency marker.
- Do not add the documentation to the source distribution.

0.3.1 (2017-12-26)

Bug fixes and minor changes

- Issue #17: Move the online documentation to Read the Docs
- Some improvements in the documentation.

0.3 (2017-12-26)

New features

- Issue #7: Add a configuration switch to implicitly mark all tests.
- Issue #10: Add an option to ignore unknown dependencies.

Incompatible changes

- Prepend the class name to the default test name for test class methods. This fixes a potential name conflict, see Issue #6.

If your code uses test classes and you reference test methods by their default name, you must add the class name. E.g. if you have something like:

```
class TestClass(object):

    @pytest.mark.dependency()
    def test_a():
        pass

    @pytest.mark.dependency(depends=["test_a"])
    def test_b():
        pass
```

you need to change this to:

```
class TestClass(object):

    @pytest.mark.dependency()
    def test_a():
        pass
```

(continues on next page)

(continued from previous page)

```
@pytest.mark.dependency(depends=["TestClass::test_a"])
def test_b():
    pass
```

If you override the test name in the `pytest.mark.dependency()` marker, nothing need to be changed.

Bug fixes and minor changes

- PR #11: show the name of the skipped test (thanks asteriogonzalez).
- Issue #13: Do not import pytest in setup.py to make it compatible with pipenv.
- Issue #15: tests fail with pytest 3.3.0.
- Issue #8: document incompatibility with parallelization in pytest-xdist.
- Clarify in the documentation that Python 3.1 is not officially supported because pytest 2.8 does not support it. There is no known issue with Python 3.1 though.

0.2 (2017-05-28)

New features

- Issue #2: Add documentation.
- Issue #4: Add a `depend()` function to add a dependency to a test at runtime.

0.1 (2017-01-29)

- Initial release as an independent Python module.

This code was first developed as part of a larger package, `python-icat`, at Helmholtz-Zentrum Berlin für Materialien und Energie, see <https://icatproject.org/user-documentation/python-icat/>

1.7 Reference

`@pytest.mark.dependency` (*name=None, depends=[]*)

Mark a test to be used as a dependency for other tests or to depend on other tests.

This will cause the test results to be registered internally and thus other tests may depend on the test. The list of dependencies for the test may be set in the `depends` argument.

Parameters

- **name** (*str*) – the name of the test to be used for referencing by dependent tests. If not set, it defaults to the node ID defined by pytest, that is the name of the test function, extended by the parameters if applicable. The name must be unique in the scope, which is currently the test module.
- **depends** (*iterable of str*) – dependencies, a list of names of tests that this test depends on. The test will be skipped unless all of the dependencies have been run successfully. The dependencies must also have been decorated by the marker.

`pytest_dependency.depends` (*request, other*)

Add dependency on other test.

Call `pytest.skip()` unless a successful outcome of all of the tests in `other` has been registered previously. This has the same effect as the `depends` keyword argument to the `pytest.mark.dependency()` marker. In contrast to the marker, this function may be called at runtime during a test.

Parameters

- **request** – the value of the *request* pytest fixture related to the current test.
- **other** (iterable of `str`) – dependencies, a list of names of tests that this test depends on.

New in version 0.2.

p

`pytest_dependency`, 15

D

`depends()` (in module `pytest_dependency`), 15

P

`pytest.mark.dependency()` (built-in function), 15

`pytest_dependency` (module), 15