# pytest-dependency Documentation

*Release 0.5.2.dev61+g979f01b*

**Rolf Krahl**

**Feb 17, 2022**

# Contents

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.

CHAPTER 1

# Content of the documentation

## 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 then 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 then 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 2016–2020 Rolf Krahl

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License.

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 3.7.0 or newer.

### 1.2.2 Interaction with other packages

**pytest-order** pytest-dependency is based on the assumption that dependencies are run before the test that depends on them. If this assumption is not satisfied in the default execution order in pytest, you may want to have a look on pytest-order. It implements reordering of tests and supports taking the dependencies into account.

**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 is available on the Python Package Index (PyPI).

### 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 tests
```

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:

```python
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 3.8.1, pytest-5.3.4, py-1.8.1, pluggy-0.13.1
rootdir: /home/user/tests
plugins: dependency-0.4.0
collected 5 items

basic.py x.s.s                                                           [100%]

=========================== short test summary info ============================
SKIPPED [1] /usr/lib/python3.8/site-packages/pytest_dependency.py:87: test_c␣
↪depends on test_a
```

(continues on next page)

(continued from previous page)

```
SKIPPED [1] /usr/lib/python3.8/site-packages/pytest_dependency.py:87: test_e␣
↪depends on test_c
XFAIL basic.py::test_a
  deliberate fail
=================== 2 passed, 2 skipped, 1 xfailed in 0.06s ====================
```

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, see Section *Names* for details. 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. The following example works exactly as the last one, only the test names are explicitly set:

```python
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

@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:

```python
import pytest

```

(continues on next page)

(continued from previous page)

```python
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")
    def test_b(self):
        pass

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

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

    @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 the method name to form the default name for the test.

### 1.3.4 Applying the dependency marker to a class as a whole

The `pytest.mark.dependency()` marker may also be applied to a test class as a whole. This has the same effect as applying that marker with the same arguments to each method of the class individually. Consider:

```python
import pytest


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


@pytest.mark.dependency(depends=["test_f"])
class TestClass(object):

    def test_a(self):
        pass

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

    def test_c(self):
        pass
```

The tests *TestClass::test_a* and *TestClass::test_c* will be skipped, because they depend on *test_f*. But *TestClass::test_b* will be run, because it is individually marked. The marker on the test method overrides the marker on the class and thus effectively clears the dependency list for *TestClass::test_b*.

### 1.3.5 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:

```python
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
```

(continues on next page)

(continued from previous page)

```

@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.

### 1.3.6 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 Defining the scope of dependencies

In the previous examples, we didn't specify a scope for the dependencies. All dependencies were taken in module scope, which is the default. As a consequence, tests were constraint to depend only from other tests in the same test module.

The `pytest.mark.dependency()` marker as well as the `pytest_dependency.depends()` function take an optional *scope* argument. Possible values are *'session'*, *'package'*, *'module'*, or *'class'*.

New in version 0.5.0: the scope of dependencies has been introduced. In earlier versions, all dependencies were implicitly in module scope.

### 1.4.1 Explicitly specifying the scope

The default value for the *scope* argument is *'module'*. Thus, the very first example from Section *Basic usage* could also be written as:

```
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"], scope='module')
def test_c():
    pass

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

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

It works exactly the same. The only difference is that the default scope has been made explicit.

### 1.4.2 Dependencies in session scope

If a test depends on another test in a different test module, the dependency must either be in session or package scope. Consider the following two test modules:

```
# test_mod_01.py

import pytest

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

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

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


class TestClass(object):
```

(continues on next page)

(continued from previous page)

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

and

```
# test_mod_02.py

import pytest

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

@pytest.mark.dependency(
    depends=["tests/test_mod_01.py::test_a", "tests/test_mod_01.py::test_c"],
    scope='session'
)
def test_e():
    pass

@pytest.mark.dependency(
    depends=["tests/test_mod_01.py::test_b", "tests/test_mod_02.py::test_e"],
    scope='session'
)
def test_f():
    pass

@pytest.mark.dependency(
    depends=["tests/test_mod_01.py::TestClass::test_b"],
    scope='session'
)
def test_g():
    pass
```

Let's assume the modules to be stored as *tests/test_mod_01.py* and *tests/test_mod_02.py* relative to the current working directory respectively. The test *test_e* in *tests/test_mod_02.py* will be run and succeed. It depends on *test_a* and *test_c* in *tests/test_mod_01.py* that both succeed. It does not matter that there is another *test_a* in *tests/test_mod_02.py* that fails. Test *test_f* in *tests/test_mod_02.py* will be skipped, because it depends on *test_b* in *tests/test_mod_01.py* that fails. Test *test_g* in turn will be run and succeed. It depends on the test method *test_b* of class *TestClass* in *tests/test_mod_01.py*, not on the test function of the same name.

The *scope* argument only affects the references in the *depends* argument of the marker. It does not matter which scope is set for the dependencies: the dependency of *test_e* in *tests/test_mod_02.py* on *test_a* in *tests/test_mod_01.py* is in session scope. It is not needed to set the scope also for *test_a*.

Note that the references in session scope must use the full node id of the dependencies. This node id is composed of the module path, the name of the test class if applicable, and the name of the test, separated by a double colon "::", see Section *Names* for details. References in module scope on the other hand must omit the module path in the node id, because that is implied by the scope.

Package scope is only available if the test is in a package and then restricts the dependencies to tests within the same package. Otherwise it works the same as session scope.

### 1.4.3 The class scope

Test dependencies may also be in class scope. This is only available for methods of a test class and restricts the dependencies to other test methods of the same class.

Consider the following example:

```
import pytest

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


class TestClass1(object):

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


class TestClass2(object):

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

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

    @pytest.mark.dependency(depends=["test_a"], scope='class')
    def test_d(self):
        pass

    @pytest.mark.dependency(depends=["test_b"], scope='class')
    def test_e(self):
        pass
```

The test method *test_c* of class *TestClass2* will be skipped because it depends on *test_a*. The marker does not have a *scope* argument, so this dependency defaults to module scope. The dependency thus resolves to the function *test_a* at module level, which failed. The fact that there is also a method *test_a* in this class does not matter, because that would need to be referenced as *TestClass2::test_a* in module scope. The test method *test_d* of class *TestClass2* depends on *test_a* in class scope. This resolves to the method *test_a* of *TestClass2* which succeeds. As a result, *test_d* will be run and succeed as well. Test method *test_e* of class *TestClass2* will be skipped, because it depends on *test_b* in class scope, but there is no method by that name in this class. The fact that there is another class *TestClass1* having a method by that name is irrelevant.

## 1.5 Advanced usage

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

### 1.5.1 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
```

(continues on next page)

(continued from previous page)

```python
# 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 parameters 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.

### 1.5.2 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:

```python
import pytest
from pytest_dependency import depends
```

(continues on next page)

(continued from previous page)

```python
@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:

```python
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.

### 1.5.3 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:

```python
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

@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.

### 1.5.4 Logical combinations of dependencies

The dependencies passed as in the *depends* argument to the `pytest.mark.dependency()` marker are combined in an and-like manner: the current test is skipped unless *all* dependencies did succeed. Sometimes one may want to combine the dependencies in a different way. This is not supported by pytest-dependency out of the box, but it is not difficult to implement. Consider the following example:

```python
import pytest
from pytest_dependency import depends

def depends_or(request, other, scope='module'):
    item = request.node
    for o in other:
        try:
            depends(request, [o], scope)
        except pytest.skip.Exception:
            continue
        else:
            return
    pytest.skip("%s depends on any of %s" % (item.name, ", ".join(other)))


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

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

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

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

@pytest.mark.dependency()
def test_c(request):
    depends_or(request, ["test_ax", "test_bx"])
    pass

@pytest.mark.dependency()
def test_d(request):
    depends_or(request, ["test_ax", "test_bp"])
    pass

@pytest.mark.dependency()
def test_e(request):
    depends_or(request, ["test_ap", "test_bx"])
    pass

@pytest.mark.dependency()
def test_f(request):
```

(continues on next page)

(continued from previous page)

```
    depends_or(request, ["test_ap", "test_bp"])
    pass
```

The helper function *depends_or()* is similar to `pytest_dependency.depends()`, it takes the same arguments. The only difference is that it combines the dependencies passed in the *other* argument in an or-like manner: the current test will be run if *at least one* of the other tests did succeed.

The tests *test_c*, *test_d*, *test_e*, and *test_f* in this example all depend on two other tests. Only *test_c* will be skipped, because all tests in its dependency list fail. The other ones are run, because they have at least one succeeding test in their dependency list.

Other logical combinations of dependencies are conceivable and may be implemented in a similar way, according to the use case at hand.

**Note:** The *depends_or()* helper function above is based on pytest internals: skipping of tests works by raising an exception and the exception class is exposed as `pytest.skip.Exception`. This is not documented in pytest. It has been tested to work for pytest versions 3.7.0 through 6.2.5, but it is not guaranteed to be stable for future pytest versions.

## 1.6 Names

Dependencies of tests are referenced by name. The default name is the node id assigned to the test by pytest. This default may be overridden by an explicit *name* argument to the `pytest.mark.dependency()` marker. The references also depend on the scope.

### 1.6.1 Node ids

The node ids in pytest are built of several components, separated by a double colon "::". For test functions, these components are the relative path of the test module and the name of the function. In the case of a method of a test class the components are the module path, the name of the class, and the name of the method. If the function or method is parameterized, the parameter values, separated by minus "-", in square brackets "[]" are appended to the node id. The representation of the parameter values in the node id may be overridden using the *ids* argument to the pytest.mark.parametrize() marker.

One may check the node ids of all tests calling pytest with the *–verbose* command line option. As an example, consider the following test module:

```
import random
import pytest

def test_a():
    pass

@pytest.mark.parametrize("i,b", [
    (7, True),
    (0, False),
    pytest.param(-1, False, marks=pytest.mark.xfail(reason="nonsense"))
])
def test_b(i, b):
    assert bool(i) == b

ordered = list(range(10))
unordered = random.sample(ordered, k=len(ordered))

class TestClass:
```

(continues on next page)

(continued from previous page)

```
    def test_c(self):
        pass

    @pytest.mark.parametrize("l,ll", [(ordered, 10), (unordered, 10)],
                             ids=["order", "disorder"])
    def test_d(self, l, ll):
        assert len(l) == ll
```

If this module is stored as *tests/test_nodeid.py*, the output will look like:

```
$ pytest --verbose
============================= test session starts ==============================
platform linux -- Python 3.8.1, pytest-5.3.4, py-1.8.1, pluggy-0.13.1 -- /usr/bin/
↪python3
cachedir: .pytest_cache
rootdir: /home/user
plugins: dependency-0.4.0
collected 7 items

tests/test_nodeid.py::test_a PASSED                                      [ 14%]
tests/test_nodeid.py::test_b[7-True] PASSED                              [ 28%]
tests/test_nodeid.py::test_b[0-False] PASSED                             [ 42%]
tests/test_nodeid.py::test_b[-1-False] XFAIL                             [ 57%]
tests/test_nodeid.py::TestClass::test_c PASSED                           [ 71%]
tests/test_nodeid.py::TestClass::test_d[order] PASSED                    [ 85%]
tests/test_nodeid.py::TestClass::test_d[disorder] PASSED                 [100%]

========================= 6 passed, 1 xfailed in 0.08s =========================
```

**Note:** Old versions of pytest used to include an extra "()" component to the node ids of methods of test classes. This has been removed in pytest 4.0.0. pytest-dependency strips this if present. Thus, when referencing dependencies, the new style node ids as described above may (and must) be used, regardless of the pytest version.

## 1.6.2 References and scope

When referencing dependencies of tests, the names to be used in the *depends* argument to the `pytest.mark.dependency()` marker or the *other* argument to the `pytest_dependency.depends()` function depend on the scope as follows:

***session*** The full node id must be used.

***package*** The full node id must be used.

***module*** The node id with the leading module path including the "::" separator removed must be used.

***class*** The node id with the module path and the class name including the "::" separator removed must be used.

That is, in the example above, when referencing *test_a* as a dependency, it must be referenced as *tests/test_nodeid.py::test_a* in session scope and as *test_a* in module scope. When referencing the first invocation of *test_d* as a dependency, it must be referenced as *tests/test_nodeid.py::TestClass::test_d[order]* in session scope, as *TestClass::test_d[order]* in module scope, and as *test_d[order]* in class scope.

If the name of the dependency has been set with an explicit *name* argument to the `pytest.mark.dependency()` marker, this name must always be used as is, regardless of the scope.

**Note:** The module path in the node id is the path relative to the current working directory. This depends on the invocation of pytest. In the example above, if you change into the *tests* directory before invoking pytest, the module path in the node ids will be *test_nodeid.py*. If you use references in session scope, you'll need to make sure pytest is always invoked from the same working directory.

## 1.7 Debugging guide

This section is supposed to provide hints on what to check if pytest-dependency does not seem to behave as expected.

### 1.7.1 Example

We consider the following example in this guide:

```
from enum import Enum
import pytest


class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

    def __str__(self):
        return self.name


def get_starship(name):
    fleet = pytest.importorskip("fleet")
    return fleet.get_ship(name)


@pytest.fixture(scope="module", params=range(2))
def prepenv(request):
    pass

@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


class TestClass(object):
```

(continues on next page)

(continued from previous page)

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

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

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


@pytest.mark.dependency()
@pytest.mark.parametrize("c", [ Color.RED, Color.GREEN, Color.BLUE, ])
def test_colors(c):
    pass

@pytest.mark.dependency(depends=["test_colors"])
def test_multicolored():
    pass

@pytest.mark.dependency(depends=["test_colors[Color.RED]"])
def test_alert():
    pass

@pytest.mark.dependency(depends=["test_f"])
def test_g():
    pass

@pytest.mark.dependency(name="h")
def test_h():
    pass

@pytest.mark.dependency(depends=["test_b"])
def test_k():
    s = get_starship("NCC-1701")

@pytest.mark.dependency()
def test_l(prepenv):
    pass

@pytest.mark.dependency(depends=["test_b"], scope='session')
def test_m():
    pass

@pytest.mark.dependency(depends=["test_h"])
def test_o():
    pass

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

@pytest.mark.dependency(depends=["test_p"])
def test_q(prepenv):
    pass

@pytest.mark.dependency(depends=["test_a"])
@pytest.mark.dependency(name="r")
```

(continues on next page)

(continued from previous page)

```
def test_r():
    pass

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

This example contains several cases where the presumably intended behavior of the code differs from what happens in practice. We will show below how diagnostic tools in pytest may be used to unravel the discrepancies. The results that may (or may not) be surprising include:

- The test method *test_c* in class *TestClass* depending on *test_b* is run, although the test method *test_b* fails.
- All instances of *test_colors* succeed. Yet *test_multicolored* that only depends on *test_colors* is skipped.
- Similarly *test_alert* depending only on *test_colors[Color.RED]* is skipped, although *test_colors* with the parameter value *Color.RED* succeeds.
- *test_k* depending only on *test_b* is skipped, although *test_b* succeeds.
- Same with *test_m* depending only on *test_b* is skipped.
- *test_o* depending only on *test_h* is skipped, although *test_h* succeeds.
- *test_q* depending only on *test_p* is skipped, although *test_p* succeeds.
- *test_r* is run, although *test_a* fails.
- *test_s* depending only on *test_l* is skipped, although *test_l* succeeds.

### 1.7.2 Diagnostic tools

There are different ways to request diagnostic output from pytest. We will discuss how they may be used to better understand the behavior of pytest-dependency.

#### pytest summary

You can request a short summary from pytest including information on skipped tests using the `-rs` command line option:

```
$ pytest -rs debugging.py
============================= test session starts ==============================
platform linux -- Python 3.10.1, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/user
plugins: dependency-0.6.0
collected 25 items

debugging.py x.s.s.x....sss.s.s.sss..s                                   [100%]

=========================== short test summary info ============================
SKIPPED [1] /usr/lib/python3.10/site-packages/pytest_dependency.py:101: test_c␣
↪depends on test_a
SKIPPED [1] /usr/lib/python3.10/site-packages/pytest_dependency.py:101: test_e␣
↪depends on test_c
SKIPPED [1] /usr/lib/python3.10/site-packages/pytest_dependency.py:101: test_
↪multicolored depends on test_colors
SKIPPED [1] /usr/lib/python3.10/site-packages/pytest_dependency.py:101: test_alert␣
↪depends on test_colors[Color.RED]
SKIPPED [1] /usr/lib/python3.10/site-packages/pytest_dependency.py:101: test_g␣
↪depends on test_f
SKIPPED [1] debugging.py:15: could not import 'fleet': No module named 'fleet'
```

(continues on next page)

(continued from previous page)

```
SKIPPED [1] /usr/lib/python3.10/site-packages/pytest_dependency.py:101: test_q[0]␣
↪depends on test_p
SKIPPED [1] /usr/lib/python3.10/site-packages/pytest_dependency.py:101: test_q[1]␣
↪depends on test_p
SKIPPED [1] /usr/lib/python3.10/site-packages/pytest_dependency.py:101: test_m␣
↪depends on test_b
SKIPPED [1] /usr/lib/python3.10/site-packages/pytest_dependency.py:101: test_o␣
↪depends on test_h
SKIPPED [1] /usr/lib/python3.10/site-packages/pytest_dependency.py:101: test_s␣
↪depends on test_l
================== 12 passed, 11 skipped, 2 xfailed in 0.05s ===================
```

This summary indicates if a test has been skipped by pytest-dependency in the first place. In the present example, the summary hints that *test_k* has been skipped due to another reason, unrelated to pytest-dependency. If the test has been skipped by pytest-dependency, the summary displays the name of the missing dependency.

### Verbose pytest output

A list of all tests with their respective outcome will be displayed if you call pytest with the `--verbose` command line option:

```
$ pytest --verbose debugging.py
============================= test session starts ==============================
platform linux -- Python 3.10.1, pytest-6.2.5, py-1.11.0, pluggy-1.0.0 -- /usr/bin/
↪python3
cachedir: .pytest_cache
rootdir: /home/user
plugins: dependency-0.6.0
collecting ... collected 25 items

debugging.py::test_a XFAIL (deliberate fail)                             [  4%]
debugging.py::test_b PASSED                                              [  8%]
debugging.py::test_c SKIPPED (test_c depends on test_a)                  [ 12%]
debugging.py::test_d PASSED                                              [ 16%]
debugging.py::test_e SKIPPED (test_e depends on test_c)                  [ 20%]
debugging.py::TestClass::test_a PASSED                                   [ 24%]
debugging.py::TestClass::test_b XFAIL (deliberate fail)                  [ 28%]
debugging.py::TestClass::test_c PASSED                                   [ 32%]
debugging.py::test_colors[RED] PASSED                                    [ 36%]
debugging.py::test_colors[GREEN] PASSED                                  [ 40%]
debugging.py::test_colors[BLUE] PASSED                                   [ 44%]
debugging.py::test_multicolored SKIPPED (test_multicolored depends o...) [ 48%]
debugging.py::test_alert SKIPPED (test_alert depends on test_colors[...) [ 52%]
debugging.py::test_g SKIPPED (test_g depends on test_f)                  [ 56%]
debugging.py::test_h PASSED                                              [ 60%]
debugging.py::test_k SKIPPED (could not import 'fleet': No module na...) [ 64%]
debugging.py::test_l[0] PASSED                                           [ 68%]
debugging.py::test_q[0] SKIPPED (test_q[0] depends on test_p)            [ 72%]
debugging.py::test_l[1] PASSED                                           [ 76%]
debugging.py::test_q[1] SKIPPED (test_q[1] depends on test_p)            [ 80%]
debugging.py::test_m SKIPPED (test_m depends on test_b)                  [ 84%]
debugging.py::test_o SKIPPED (test_o depends on test_h)                  [ 88%]
debugging.py::test_p PASSED                                              [ 92%]
debugging.py::test_r PASSED                                              [ 96%]
debugging.py::test_s SKIPPED (test_s depends on test_l)                  [100%]

================== 12 passed, 11 skipped, 2 xfailed in 0.06s ===================
```

The verbose listing is particular useful, because it shows the pytest node id for each test, which is not always obvious. As explained in Section *Names*, this node id is the basis to form the default test name that need to be used to reference the test in the dependencies.

From this list we can understand why *test_multicolored* has been skipped: it depends on *test_colors*. But *test_colors* is parametrized and thus the parameter value is included in the node id. As a result, a dependency by the name *test_colors* can not be found. The same thing happens in the case of *test_s*: it depends on *test_l*, but the latter uses a parametrized fixture, so it indirectly takes a parameter value and that value must be included in the reference for the dependency.

In the case of *test_alert*, the parameter value is included in the dependency *test_colors[Color.RED]*. But in the node id as displayed in the verbose list, that test appears as *test_colors[RED]*. Note that *class Color* overrides the string representation operator and that affects how the parameter value appears in the node id in this case.

The verbose list also displays the execution order of the tests. In the present example, this order differs from the order in the source code. That is the reason why both instances of *test_q* are skipped: they are executed before the dependency *test_p*. So the outcome of the latter is yet unknown at the moment that the dependency is checked.

## Logging

pytest-dependency emits log messages when registering test results and when checking dependencies for a test. You can request these log messages to be displayed at runtime using log command line options in the pytest call. Beware that this may produce a large amount of output, even for medium size test suites. We will present only a few fragments of the output here. Consider the start of that output, covering the first test *test_a*:

```
$ pytest --log-cli-format='%(levelname)s: %(message)s' --log-cli-level=DEBUG␣
↪debugging.py
============================= test session starts ==============================
platform linux -- Python 3.10.1, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/user
plugins: dependency-0.6.0
collected 25 items

debugging.py::test_a
-------------------------------- live log setup --------------------------------
DEBUG: register setup debugging.py::test_a passed in session scope
DEBUG: register setup test_a passed in module scope
-------------------------------- live log call ---------------------------------
DEBUG: register call debugging.py::test_a skipped in session scope
DEBUG: register call test_a skipped in module scope
XFAIL (deliberate fail)                                                  [  4%]
------------------------------ live log teardown -------------------------------
DEBUG: register teardown debugging.py::test_a passed in session scope
DEBUG: register teardown test_a passed in module scope
```

It is shown how the test outcome for each of the three test phases (setup, call, and teardown) is registered in pytest-dependency. It is also shown which name is used to register the test outcome depending on the scope.

Considering the relevant fragments of the output, we can check why *TestClass::test_c* is not skipped:

```
debugging.py::test_b
-------------------------------- live log setup --------------------------------
DEBUG: register setup debugging.py::test_b passed in session scope
DEBUG: register setup test_b passed in module scope
-------------------------------- live log call ---------------------------------
DEBUG: register call debugging.py::test_b passed in session scope
DEBUG: register call test_b passed in module scope
PASSED                                                                   [  8%]
------------------------------ live log teardown -------------------------------
DEBUG: register teardown debugging.py::test_b passed in session scope
DEBUG: register teardown test_b passed in module scope

debugging.py::TestClass::test_b
```

(continues on next page)

(continued from previous page)

```
-------------------------------- live log setup --------------------------------
DEBUG: register setup debugging.py::TestClass::test_b passed in session scope
DEBUG: register setup TestClass::test_b passed in module scope
DEBUG: register setup test_b passed in class scope
-------------------------------- live log call ---------------------------------
DEBUG: register call debugging.py::TestClass::test_b skipped in session scope
DEBUG: register call TestClass::test_b skipped in module scope
DEBUG: register call test_b skipped in class scope
XFAIL (deliberate fail)                                                  [ 28%]
------------------------------ live log teardown -------------------------------
DEBUG: register teardown debugging.py::TestClass::test_b passed in session scope
DEBUG: register teardown TestClass::test_b passed in module scope
DEBUG: register teardown test_b passed in class scope

debugging.py::TestClass::test_c
-------------------------------- live log setup --------------------------------
DEBUG: check dependencies of test_c in module scope ...
DEBUG: ... test_b succeeded
DEBUG: register setup debugging.py::TestClass::test_c passed in session scope
DEBUG: register setup TestClass::test_c passed in module scope
DEBUG: register setup test_c passed in class scope
-------------------------------- live log call ---------------------------------
DEBUG: register call debugging.py::TestClass::test_c passed in session scope
DEBUG: register call TestClass::test_c passed in module scope
DEBUG: register call test_c passed in class scope
PASSED                                                                   [ 32%]
------------------------------ live log teardown -------------------------------
DEBUG: register teardown debugging.py::TestClass::test_c passed in session scope
DEBUG: register teardown TestClass::test_c passed in module scope
DEBUG: register teardown test_c passed in class scope
```

The dependency *test_b* is checked in module scope. If that dependency was meant to reference the method of the same class, it would either need to be referenced as *test_b* in class scope or as *TestClass::test_b* in module scope or as *debugging.py::TestClass::test_b* in session scope. The way it is formulated in the example, it actually references the test function *test_b*, which succeeds.

A similar case is *test_m*:

```
debugging.py::test_b
-------------------------------- live log setup --------------------------------
DEBUG: register setup debugging.py::test_b passed in session scope
DEBUG: register setup test_b passed in module scope
-------------------------------- live log call ---------------------------------
DEBUG: register call debugging.py::test_b passed in session scope
DEBUG: register call test_b passed in module scope
PASSED                                                                   [  8%]
------------------------------ live log teardown -------------------------------
DEBUG: register teardown debugging.py::test_b passed in session scope
DEBUG: register teardown test_b passed in module scope

debugging.py::test_m
-------------------------------- live log setup --------------------------------
DEBUG: check dependencies of test_m in session scope ...
DEBUG: ... test_b is unknown
INFO: skip test_m because it depends on test_b
DEBUG: register setup debugging.py::test_m skipped in session scope
DEBUG: register setup test_m skipped in module scope
SKIPPED (test_m depends on test_b)                                       [ 84%]
------------------------------ live log teardown -------------------------------
DEBUG: register teardown debugging.py::test_m passed in session scope
DEBUG: register teardown test_m passed in module scope
```

The dependency *test_b* is checked in session scope. There is no test that matches this name. If that dependency was mean to reference the test function *test_b* in the example, it would either need to be referenced as *debugging.py::test_b* in session scope or as *test_b* in module scope.

A slightly different situation is given in the case of *test_o*:

```
debugging.py::test_h
-------------------------------- live log setup --------------------------------
DEBUG: register setup h passed in session scope
DEBUG: register setup h passed in module scope
-------------------------------- live log call ---------------------------------
DEBUG: register call h passed in session scope
DEBUG: register call h passed in module scope
PASSED                                                                   [ 60%]
------------------------------ live log teardown -------------------------------
DEBUG: register teardown h passed in session scope
DEBUG: register teardown h passed in module scope

debugging.py::test_o
-------------------------------- live log setup --------------------------------
DEBUG: check dependencies of test_o in module scope ...
DEBUG: ... test_h is unknown
INFO: skip test_o because it depends on test_h
DEBUG: register setup debugging.py::test_o skipped in session scope
DEBUG: register setup test_o skipped in module scope
SKIPPED (test_o depends on test_h)                                       [ 88%]
------------------------------ live log teardown -------------------------------
DEBUG: register teardown debugging.py::test_o passed in session scope
DEBUG: register teardown test_o passed in module scope
```

In the `pytest.mark.dependency()` marker for *test_h* in the example, the name is overridden as *h*. The outcome of that test is registered using that name. It can thus not be found by the name *test_h*.

Considering the case of *test_r*:

```
debugging.py::test_r
-------------------------------- live log setup --------------------------------
DEBUG: register setup r passed in session scope
DEBUG: register setup r passed in module scope
-------------------------------- live log call ---------------------------------
DEBUG: register call r passed in session scope
DEBUG: register call r passed in module scope
PASSED                                                                   [ 96%]
------------------------------ live log teardown -------------------------------
DEBUG: register teardown r passed in session scope
DEBUG: register teardown r passed in module scope
```

That test has no dependencies. The error in the example is that the `pytest.mark.dependency()` marker is applied twice to the test. That doesn't work in pytest, only the last invocation is effective. As a result, the second invocation setting a name, effectively clears the dependency list that was set in the first invocation.

## 1.8 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.

### 1.8.1 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.

### 1.8.2 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 3.7.0 or newer, it is recommended to set this option accordingly, either to 3.7.0 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.

### 1.8.3 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 test suite and some tests in the selected set are marked to depend on other tests that have not been selected.

New in version 0.3.

## 1.9 Changelog

### 1.9.1 dev (not yet released)

#### Bug fixes and minor changes

- #40: add logging.
- #50, #51: test suite incompatibility with pytest 6.2.0.
- #58: declare the type of automark_dependency ini-option correctly as bool.

### 1.9.2 0.5.1 (2020-02-14)

#### Bug fixes and minor changes

- Fix failing documentation build.

### 1.9.3 0.5.0 (2020-02-14)

#### New features

- #3, #35: add a scope to dependencies. (Thanks to JoeSc and selenareneephillips!)

#### Incompatible changes

- Require pytest version 3.7.0 or newer.

#### Bug fixes and minor changes

- #34: failing test with pytest 4.2.0 and newer.
- Use setuptools_scm to manage the version number.

### 1.9.4 0.4.0 (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

- #24, #25: get_marker no longer available in pytest 4.0.0. (Thanks to Rogdham!)
- #28: Applying markers directly in parametrize is no longer available in 4.0.

### 1.9.5 0.3.2 (2018-01-17)

#### Bug fixes and minor changes

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

### 1.9.6 0.3.1 (2017-12-26)

#### Bug fixes and minor changes

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

### 1.9.7 0.3 (2017-12-26)

#### New features

- #7: Add a configuration switch to implicitly mark all tests.
- #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 #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

    @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

- #11: show the name of the skipped test. (Thanks asteriogonzalez!)
- #13: Do not import pytest in setup.py to make it compatible with pipenv.
- #15: tests fail with pytest 3.3.0.
- #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.

### 1.9.8 0.2 (2017-05-28)

#### New features

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

### 1.9.9 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.

## 1.10 Reference

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

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. The name must be unique.
- **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. The names of the dependencies must be adapted to the scope.
- **scope** (str) – the scope to search for the dependencies. Must be either *'session'*, *'package'*, *'module'*, or *'class'*.

See Section *Names* for details on the default name if the *name* argument is not set and on how references in the *depends* argument must be adapted to the scope.

Changed in version 0.5.0: the scope parameter has been added.

`pytest_dependency.`**`depends`**`(`*request*, *other*, *scope='module'*`)`

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. The names of the dependencies must be adapted to the scope.
- **scope** (str) – the scope to search for the dependencies. Must be either *'session'*, *'package'*, *'module'*, or *'class'*.

New in version 0.2.

Changed in version 0.5.0: the scope parameter has been added.

# Python Module Index

# Index