

**pytest Documentation**

**Release 3.6**

**holger krekel, trainer and consultant, http://merlinux.eu**

**Jul 22, 2018**

Contents

|  |  |  |  |
| --- | --- | --- | --- |
| [1 Installation and Getting Started](#page11) | | | 3 |
|  | [1.1](#page11) | [Install pytest](#page11) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 3 |
|  | [1.2](#page11) | [Create your first test](#page11) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 3 |
|  | [1.3](#page12) | [Run multiple tests](#page12) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 4 |
|  | [1.4](#page12) | [Assert that a certain exception is raised](#page12) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 4 |
|  | [1.5](#page13) | [Group multiple tests in a class](#page13) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 5 |
|  | [1.6](#page13) | [Request a unique temporary directory for functional tests](#page13) . . . . . . . . . . . . . . . . . . . . . . . . | 5 |
|  | [1.7](#page14) | [Continue reading](#page14) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 6 |
| [2](#page15) | [Usage and Invocations](#page15) | | 7 |
|  | [2.1](#page15) | [Calling pytest through python -m pytest](#page15) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 7 |
|  | [2.2](#page15) | [Possible exit codes](#page15) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 7 |
|  | [2.3](#page15) | [Getting help on version, option names, environment variables](#page15) . . . . . . . . . . . . . . . . . . . . . | 7 |
|  | [2.4](#page15) | [Stopping after the first (or N) failures](#page15) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 7 |
|  | [2.5](#page16) | [Specifying tests / selecting tests](#page16) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 8 |
|  | [2.6](#page16) | [Modifying Python traceback printing](#page16) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 8 |
|  | [2.7](#page17) | [Dropping to PDB (Python Debugger) on failures](#page17) . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 9 |
|  | [2.8](#page17) | [Setting breakpoints](#page17) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 9 |
|  | [2.9](#page18) | [Using the builtin breakpoint function](#page18) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 10 |
|  | [2.10](#page18) | [Profiling test execution duration](#page18) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 10 |
|  | [2.11](#page18) | [Creating JUnitXML format files](#page18) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 10 |
|  | [2.12](#page21) | [Creating resultlog format files](#page21) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 13 |
|  | [2.13](#page21) | [Sending test report to online pastebin service](#page21) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 13 |
|  | [2.14](#page22) | [Disabling plugins](#page22) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 14 |
|  | [2.15](#page22) | [Calling pytest from Python code](#page22) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 14 |
| [3 Using pytest with an existing test suite](#page23) | | | 15 |
|  | [3.1](#page23) | [Running an existing test suite with pytest](#page23) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 15 |
| [4 The writing and reporting of assertions in tests](#page25) | | | 17 |
|  | [4.1](#page25) | [Asserting with the assert statement](#page25) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 17 |
|  | [4.2](#page26) | [Assertions about expected exceptions](#page26) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 18 |
|  | [4.3](#page27) | [Assertions about expected warnings](#page27) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 19 |
|  | [4.4](#page27) | [Making use of context-sensitive comparisons](#page27) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 19 |
|  | [4.5](#page28) | [Defining your own assertion comparison](#page28) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 20 |
|  | [4.6](#page29) | [Advanced assertion introspection](#page29) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 21 |
| [5](#page31) | [pytest fixtures: explicit, modular, scalable](#page31) | | 23 |

**i**

|  |  |  |  |
| --- | --- | --- | --- |
|  | [5.1](#page31) | [Fixtures as Function arguments](#page31) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 23 |
|  | [5.2](#page32) | [Fixtures: a prime example of dependency injection](#page32) . . . . . . . . . . . . . . . . . . . . . . . . . . . | 24 |
|  | [5.3](#page32) | [conftest.py: sharing fixture functions](#page32) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 24 |
|  | [5.4](#page33) | [Sharing test data](#page33) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 25 |
|  | [5.5](#page33) | [Scope: sharing a fixture instance across tests in a class, module or session](#page33) . . . . . . . . . . . . . . . | 25 |
|  | [5.6](#page34) | [Higher-scoped fixtures are instantiated first](#page34) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 26 |
|  | [5.7](#page35) | [Fixture finalization / executing teardown code](#page35) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 27 |
|  | [5.8](#page37) | [Fixtures can introspect the requesting test context](#page37) . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 29 |
|  | [5.9](#page38) | [Factories as fixtures](#page38) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 30 |
|  | [5.10](#page39) | [Parametrizing fixtures](#page39) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 31 |
|  | [5.11](#page41) | [Using marks with parametrized fixtures](#page41) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 33 |
|  | [5.12](#page42) | [Modularity: using fixtures from a fixture function](#page42) . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 34 |
|  | [5.13](#page43) | [Automatic grouping of tests by fixture instances](#page43) . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 35 |
|  | [5.14](#page44) | [Using fixtures from classes, modules or projects](#page44) . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 36 |
|  | [5.15](#page45) | [Autouse fixtures (xUnit setup on steroids)](#page45) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 37 |
|  | [5.16](#page47) | [Overriding fixtures on various levels](#page47) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 39 |
| [6](#page51) | [Marking test functions with attributes](#page51) | | 43 |
|  | [6.1](#page51) | [Raising errors on unknown marks: –strict](#page51) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 43 |
|  | [6.2](#page52) | [Marker revamp and iteration](#page52) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 44 |
| [7](#page55) | [Monkeypatching/mocking modules and environments](#page55) | | 47 |
|  | [7.1](#page55) | [Simple example: monkeypatching functions](#page55) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 47 |
|  | [7.2](#page55) | [example: preventing “requests” from remote operations](#page55) . . . . . . . . . . . . . . . . . . . . . . . . | 47 |
|  | [7.3](#page56) | [API Reference](#page56) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 48 |
| [8](#page57) | [Temporary directories and files](#page57) | | 49 |
|  | [8.1](#page57) | [The ‘tmpdir’ fixture](#page57) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 49 |
|  | [8.2](#page58) | [The ‘tmpdir\_factory’ fixture](#page58) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 50 |
|  | [8.3](#page58) | [The default base temporary directory](#page58) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 50 |
| [9](#page59) | [Capturing of the stdout/stderr output](#page59) | | 51 |
|  | [9.1](#page59) | [Default stdout/stderr/stdin capturing behaviour](#page59) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 51 |
|  | [9.2](#page59) | [Setting capturing methods or disabling capturing](#page59) . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 51 |
|  | [9.3](#page59) | [Using print statements for debugging](#page59) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 51 |
|  | [9.4](#page60) | [Accessing captured output from a test function](#page60) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 52 |
| [10](#page63) | [Warnings Capture](#page63) | | 55 |
|  | [10.1](#page64) | [@pytest.mark.filterwarnings](#page64) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 56 |
|  | [10.2](#page65) | [Disabling warning capture](#page65) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 57 |
|  | [10.3](#page65) | [Asserting warnings with the warns function](#page65) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 57 |
|  | [10.4](#page66) | [Recording warnings](#page66) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 58 |
|  | [10.5](#page67) | [Ensuring a function triggers a deprecation warning](#page67) . . . . . . . . . . . . . . . . . . . . . . . . . . . | 59 |
| [11](#page69) | [Doctest integration for modules and test files](#page69) | | 61 |
|  | [11.1](#page71) | [The ‘doctest\_namespace’ fixture](#page71) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 63 |
|  | [11.2](#page71) | [Output format](#page71) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 63 |
| [12](#page73) | [Skip and xfail: dealing with tests that cannot succeed](#page73) | | 65 |
|  | [12.1](#page73) | [Skipping test functions](#page73) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 65 |
|  | [12.2](#page76) | [XFail: mark test functions as expected to fail](#page76) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 68 |
|  | [12.3](#page78) | [Skip/xfail with parametrize](#page78) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 70 |
| [13](#page81) | [Parametrizing fixtures and test functions](#page81) | | 73 |
|  | [13.1](#page81) | [@pytest.mark.parametrize: parametrizing test functions](#page81) . . . . . . . . . . . . . . . . . . . | 73 |

**ii**

[13.2](#page83) [Basic pytest\_generate\_tests example](#page83) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

[13.3](#page84) [More examples](#page84) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76

[14 Cache: working with cross-testrun state](#page85) 77

[14.1](#page85) [Usage](#page85) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

[14.2](#page85) [Rerunning only failures or failures first](#page85) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

[14.3](#page87) [Behavior when no tests failed in the last run](#page87) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

[14.4](#page87) [The new config.cache object](#page87) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

[14.5](#page88) [Inspecting Cache content](#page88) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80

[14.6](#page89) [Clearing Cache content](#page89) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

[15 unittest.TestCase Support](#page91) 83

[15.1](#page91) [Benefits out of the box](#page91) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83

[15.2](#page92) [pytest features in unittest.TestCase subclasses](#page92) . . . . . . . . . . . . . . . . . . . . . . . . . 84

[15.3](#page92) [Mixing pytest fixtures into unittest.TestCase subclasses using marks](#page92) . . . . . . . . . . . . . 84

[15.4](#page93) [Using autouse fixtures and accessing other fixtures](#page93) . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

[16 Running tests written for nose](#page95) 87

[16.1](#page95) [Usage](#page95) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87

[16.2](#page95) [Supported nose Idioms](#page95) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87

[16.3](#page95) [Unsupported idioms / known issues](#page95) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87

[17 classic xunit-style setup](#page97) 89

[17.1](#page97) [Module level setup/teardown](#page97) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

[17.2](#page97) [Class level setup/teardown](#page97) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

[17.3](#page98) [Method and function level setup/teardown](#page98) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

[18 Installing and Using plugins](#page99) 91

[18.1](#page99) [Requiring/Loading plugins in a test module or conftest file](#page99) . . . . . . . . . . . . . . . . . . . . . . . 91

[18.2](#page100) [Finding out which plugins are active](#page100) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92

[18.3](#page100) [Deactivating / unregistering a plugin by name](#page100) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92

[19 Writing plugins](#page101) 93

[19.1](#page101) [Plugin discovery order at tool startup](#page101) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93

[19.2](#page102) [conftest.py: local per-directory plugins](#page102) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

[19.3](#page102) [Writing your own plugin](#page102) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

[19.4](#page103) [Making your plugin installable by others](#page103) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95

[19.5](#page103) [Assertion Rewriting](#page103) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95

[19.6](#page104) [Requiring/Loading plugins in a test module or conftest file](#page104) . . . . . . . . . . . . . . . . . . . . . . . 96

[19.7](#page104) [Accessing another plugin by name](#page104) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96

[19.8](#page105) [Testing plugins](#page105) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97

[20 Writing hook functions](#page107) 99

[20.1](#page107) [hook function validation and execution](#page107) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99

[20.2](#page107) [firstresult: stop at first non-None result](#page107) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99

[20.3](#page107) [hookwrapper: executing around other hooks](#page107) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99

[20.4](#page108) [Hook function ordering / call example](#page108) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100

[20.5](#page109) [Declaring new hooks](#page109) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101

[20.6](#page109) [Optionally using hooks from 3rd party plugins](#page109) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101

[21 Logging](#page111) 103

[21.1](#page112) [caplog fixture](#page112) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104

[21.2](#page113) [Live Logs](#page113) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105

[21.3](#page114) [Release notes](#page114) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106

[21.4](#page114) [Incompatible changes in pytest 3.4](#page114) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106

**iii**

|  |  |  |  |
| --- | --- | --- | --- |
| [22](#page115) | [Reference](#page115) | | 107 |
|  | [22.1](#page117) | [Functions](#page117) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 109 |
|  | [22.2](#page123) | [Marks](#page123) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 115 |
|  | [22.3](#page126) | [Fixtures](#page126) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 118 |
|  | [22.4](#page136) | [Hooks](#page136) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 128 |
|  | [22.5](#page143) | [Objects](#page143) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 135 |
|  | [22.6](#page154) | [Special Variables](#page154) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 146 |
|  | [22.7](#page155) | [Environment Variables](#page155) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 147 |
|  | [22.8](#page156) | [Configuration Options](#page156) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 148 |
| [23](#page163) | [Good Integration Practices](#page163) | | 155 |
|  | [23.1](#page163) | [Conventions for Python test discovery](#page163) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 155 |
|  | [23.2](#page163) | [Choosing a test layout / import rules](#page163) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 155 |
|  | [23.3](#page165) | [tox](#page165) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 157 |
|  | [23.4](#page166) | [Integrating with setuptools / python setup.py test / pytest-runner](#page166) . . . . . . . . . . . | 158 |
| [24](#page169) | [pytest import mechanisms and **sys.path**/**PYTHONPATH**](#page169) | | 161 |
|  | [24.1](#page169) | [Test modules / conftest.py files inside packages](#page169) . . . . . . . . . . . . . . . . . . . . . . . . . . | 161 |
|  | [24.2](#page169) | [Standalone test modules / conftest.py files](#page169) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 161 |
|  | [24.3](#page170) | [Invoking pytest versus python -m pytest](#page170) . . . . . . . . . . . . . . . . . . . . . . . . . . . | 162 |
| [25](#page171) | [Configuration](#page171) | | 163 |
|  | [25.1](#page171) | [Command line options and configuration file settings](#page171) . . . . . . . . . . . . . . . . . . . . . . . . . . | 163 |
|  | [25.2](#page171) | [Initialization: determining rootdir and inifile](#page171) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 163 |
|  | [25.3](#page172) | [How to change command line options defaults](#page172) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 164 |
|  | [25.4](#page173) | [Builtin configuration file options](#page173) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 165 |
| [26](#page175) | [Examples and customization tricks](#page175) | | 167 |
|  | [26.1](#page175) | [Demo of Python failure reports with pytest](#page175) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 167 |
|  | [26.2](#page186) | [Basic patterns and examples](#page186) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 178 |
|  | [26.3](#page200) | [Parametrizing tests](#page200) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 192 |
|  | [26.4](#page210) | [Working with custom markers](#page210) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 202 |
|  | [26.5](#page221) | [A session-fixture which can look at all collected tests](#page221) . . . . . . . . . . . . . . . . . . . . . . . . . . | 213 |
|  | [26.6](#page222) | [Changing standard (Python) test discovery](#page222) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 214 |
|  | [26.7](#page226) | [Working with non-python tests](#page226) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 218 |
| [27](#page229) | [Setting up bash completion](#page229) | | 221 |
| [28](#page231) | [Backwards Compatibility Policy](#page231) | | 223 |
|  | [28.1](#page231) | [Deprecation Roadmap](#page231) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 223 |
| [29](#page233) | [Historical Notes](#page233) | | 225 |
|  | [29.1](#page233) | [cache plugin integrated into the core](#page233) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 225 |
|  | [29.2](#page233) | [funcargs and pytest\_funcarg\_\_](#page233) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 225 |
|  | [29.3](#page233) | [@pytest.yield\_fixture decorator](#page233) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 225 |
|  | [29.4](#page233) | [[pytest] header in setup.cfg](#page233) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 225 |
|  | [29.5](#page234) | [Applying marks to @pytest.mark.parametrize parameters](#page234) . . . . . . . . . . . . . . . . . . | 226 |
|  | [29.6](#page234) | [@pytest.mark.parametrize argument names as a tuple](#page234) . . . . . . . . . . . . . . . . . . . . | 226 |
|  | [29.7](#page234) | [setup: is now an “autouse fixture”](#page234) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 226 |
|  | [29.8](#page234) | [Conditions as strings instead of booleans](#page234) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 226 |
|  | [29.9](#page235) | [pytest.set\_trace()](#page235) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 227 |
| [30](#page237) | [License](#page237) | | 229 |
| [31](#page239) | [Contribution getting started](#page239) | | 231 |

**iv**

|  |  |  |  |
| --- | --- | --- | --- |
|  | [31.1](#page239) | [Feature requests and feedback](#page239) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 231 |
|  | [31.2](#page239) | [Report bugs](#page239) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 231 |
|  | [31.3](#page240) | [Fix bugs](#page240) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 232 |
|  | [31.4](#page240) | [Implement features](#page240) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 232 |
|  | [31.5](#page240) | [Write documentation](#page240) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 232 |
|  | [31.6](#page240) | [Submitting Plugins to pytest-dev](#page240) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 232 |
|  | [31.7](#page241) | [Preparing Pull Requests](#page241) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 233 |
|  | [31.8](#page243) | [Joining the Development Team](#page243) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 235 |
| [32](#page245) | [Development Guide](#page245) | | 237 |
|  | [32.1](#page245) | [Code Style](#page245) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 237 |
|  | [32.2](#page245) | [Branches](#page245) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 237 |
|  | [32.3](#page245) | [Issues](#page245) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 237 |
|  | [32.4](#page246) | [Release Procedure](#page246) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 238 |
| [33](#page247) | [Talks and Tutorials](#page247) | | 239 |
|  | [33.1](#page247) | [Books](#page247) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 239 |
|  | [33.2](#page247) | [Talks and blog postings](#page247) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 239 |
| [34](#page251) | [Project examples](#page251) | | 243 |
|  | [34.1](#page252) | [Some organisations using pytest](#page252) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 244 |
| [35](#page253) | [Some Issues and Questions](#page253) | | 245 |
|  | [35.1](#page253) | [On naming, nosetests, licensing and magic](#page253) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 245 |
|  | [35.2](#page254) | [pytest fixtures, parametrized tests](#page254) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 246 |
|  | [35.3](#page254) | [pytest interaction with other packages](#page254) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 246 |
| [36](#page255) | [Contact channels](#page255) | | 247 |

**v**

**vi**

**pytest Documentation, Release 3.6**

[Download latest version as PDF](https://media.readthedocs.org/pdf/pytest/latest/pytest.pdf)

**Contents** **1**

**pytest Documentation, Release 3.6**

**2** **Contents**

**CHAPTER 1**

**Installation and Getting Started**

Pythons: Python 2.7, 3.4, 3.5, 3.6, Jython, PyPy-2.3

Platforms: Unix/Posix and Windows

PyPI package name: [pytest](https://pypi.org/project/pytest/)

Dependencies: [py](https://pypi.org/project/py/), [colorama (Windows)](https://pypi.org/project/colorama/),

Documentation as PDF: [download latest](https://media.readthedocs.org/pdf/pytest/latest/pytest.pdf)

pytest is a framework that makes building simple and scalable tests easy. Tests are expressive and readable—no boilerplate code required. Get started in minutes with a small unit test or complex functional test for your application or library.

**Install pytest**

1. Run the following command in your command line: pip install -U pytest
2. Check that you installed the correct version:

$ pytest --version

This is pytest version 3.x.y, imported from $PYTHON\_PREFIX/lib/python3.5/site-˓→packages/pytest.py

**Create your first test**

Create a simple test function with just four lines of code:

* content of test\_sample.py **def** func(x):

**return** x+1

**def** test\_answer():

**assert** func(3)==5

That’s it. You can now execute the test function:

**3**

**pytest Documentation, Release 3.6**

$ pytest

=========================== 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\_sample.py F | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_answer \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_answer():

* assert func(3) == 5

Eassert 4 == 5

E+ where 4 = func(3)

test\_sample.py:5: AssertionError

========================= 1 failed in 0.12 seconds =========================

This test returns a failure report because func(3) does not return 5.

Note: You can use the assert statement to verify test expectations. pytest’s [Advanced assertion introspection](http://docs.python.org/reference/simple_stmts.html#the-assert-statement) will intelligently report intermediate values of the assert expression so you can avoid the many names [of JUnit legacy](http://docs.python.org/library/unittest.html#test-cases) [methods](http://docs.python.org/library/unittest.html#test-cases).

**Run multiple tests**

pytest will run all files of the form test\_\*.py or \*\_test.py in the current directory and its subdirectories. More generally, it follows [standard test discovery rules](#page163).

**Assert that a certain exception is raised**

Use the raises helper to assert that some code raises an exception:

* content of test\_sysexit.py **import pytest**

**def** f():

**raise** SystemExit(1)

**def** test\_mytest():

**with** pytest.raises(SystemExit):

f()

Execute the test function with “quiet” reporting mode:

|  |  |  |
| --- | --- | --- |
| $ pytest -q | | test\_sysexit.py |
| . |  | [100%] |
| 1 | passed in | 0.12 seconds |
|  |  |  |

**4** **Chapter 1. Installation and Getting Started**

**pytest Documentation, Release 3.6**

**Group multiple tests in a class**

Once you develop multiple tests, you may want to group them into a class. pytest makes it easy to create a class containing more than one test:

* content of test\_class.py **class TestClass**(object):

**def** test\_one(self):x = "this"

**assert** 'h' **in** x

**def** test\_two(self):

x = "hello"

**assert** hasattr(x,'check')

pytest discovers all tests following its [Conventions for Python test discovery](#page163), so it finds both test\_ prefixed functions. There is no need to subclass anything. We can simply run the module by passing its filename:

|  |  |
| --- | --- |
| $ pytest -q test\_class.py |  |
| .F | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestClass.test\_two \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <test\_class.TestClass object at 0xdeadbeef>

def test\_two(self):

x = "hello"

* assert hasattr(x, 'check')
* AssertionError: assert False
* + where False = hasattr('hello', 'check')

test\_class.py:8: AssertionError

1 failed, 1 passed in 0.12 seconds

The first test passed and the second failed. You can easily see the intermediate values in the assertion to help you understand the reason for the failure.

**Request a unique temporary directory for functional tests**

pytest provides [Builtin fixtures/function arguments](https://docs.pytest.org/en/latest/builtin.html#builtinfixtures) to request arbitrary resources, like a unique temporary directory:

* content of test\_tmpdir.py **def** test\_needsfiles(tmpdir):

print (tmpdir) **assert** 0

List the name tmpdir in the test function signature and pytest will lookup and call a fixture factory to create the resource before performing the test function call. Before the test runs, pytest creates a unique-per-test-invocation temporary directory:

$ pytest -q test\_tmpdir.py

|  |  |
| --- | --- |
| F | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_needsfiles \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**1.5. Group multiple tests in a class** **5**

**pytest Documentation, Release 3.6**

tmpdir = local('PYTEST\_TMPDIR/test\_needsfiles0')

def test\_needsfiles(tmpdir):

print (tmpdir)

* assert 0
* assert 0

test\_tmpdir.py:3: AssertionError

--------------------------- Captured stdout call ---------------------------

PYTEST\_TMPDIR/test\_needsfiles0

1 failed in 0.12 seconds

More info on tmpdir handling is available at [Temporary directories and files](#page57).

Find out what kind of builtin [pytest fixtures](#page31) exist with the command:

pytest --fixtures # shows builtin and custom fixtures

Note that this command omits fixtures with leading \_ unless the -v option is added.

**Continue reading**

Check out additional pytest resources to help you customize tests for your unique workflow:

* “[Calling pytest through python -m pytest](#page15)” for command line invocation examples
* “[Using pytest with an existing test suite](#page23)” for working with pre-existing tests
* “[Marking test functions with attributes](#page51)” for information on the pytest.mark mechanism
* “[pytest fixtures: explicit, modular, scalable](#page31)” for providing a functional baseline to your tests
* “[Writing plugins](#page101)” for managing and writing plugins
* “[Good Integration Practices](#page163)” for virtualenv and test layouts

**6** **Chapter 1. Installation and Getting Started**

**CHAPTER 2**

**Usage and Invocations**

**Calling pytest through python -m pytest**

New in version 2.0.

You can invoke testing through the Python interpreter from the command line:

python -m pytest [...]

This is almost equivalent to invoking the command line script pytest [...] directly, except that calling via python will also add the current directory to sys.path.

**Possible exit codes**

Running pytest can result in six different exit codes:

Exit code 0 All tests were collected and passed successfully

Exit code 1 Tests were collected and run but some of the tests failed

Exit code 2 Test execution was interrupted by the user

Exit code 3 Internal error happened while executing tests

Exit code 4 pytest command line usage error

Exit code 5 No tests were collected

**Getting help on version, option names, environment variables**

pytest --version # shows where pytest was imported from pytest --fixtures # show available builtin function arguments

pytest -h | --help # show help on command line and config file options

**Stopping after the first (or N) failures**

To stop the testing process after the first (N) failures:

**7**

**pytest Documentation, Release 3.6**

pytest -x # stop after first failure

pytest --maxfail=2 # stop after two failures

**Specifying tests / selecting tests**

Pytest supports several ways to run and select tests from the command-line.

Run tests in a module

pytest test\_mod.py

Run tests in a directory

pytest testing/

Run tests by keyword expressions

pytest -k "MyClass and not method"

This will run tests which contain names that match the given string expression, which can include Python operators that use filenames, class names and function names as variables. The example above will run TestMyClass.test\_something but not TestMyClass.test\_method\_simple. Run tests by node ids

Each collected test is assigned a unique nodeid which consist of the module filename followed by specifiers like class names, function names and parameters from parametrization, separated by :: characters.

To run a specific test within a module:

pytest test\_mod.py::test\_func

Another example specifying a test method in the command line:

pytest test\_mod.py::TestClass::test\_method

Run tests by marker expressions

pytest -m slow

Will run all tests which are decorated with the @pytest.mark.slow decorator.

For more information see [marks](#page51).

Run tests from packages

pytest --pyargs pkg.testing

This will import pkg.testing and use its filesystem location to find and run tests from.

**Modifying Python traceback printing**

Examples for modifying traceback printing:

**8** **Chapter 2. Usage and Invocations**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | **pytest Documentation, Release 3.6** |  |
|  |  | | | | |  |
|  | pytest --showlocals # show local variables in tracebacks | | | | |  |
|  | pytest | -l | # | show local variables (shortcut) |  |  |
|  | pytest | --tb=auto | # | (default) 'long' tracebacks for | the first and last |  |

* entry, but 'short' style for the other entries

pytest --tb=long # exhaustive, informative traceback formatting pytest --tb=short # shorter traceback format

pytest --tb=line# only one line per failure

pytest --tb=native # Python standard library formatting

pytest --tb=no# no traceback at all

The --full-trace causes very long traces to be printed on error (longer than --tb=long). It also ensures that a stack trace is printed on KeyboardInterrupt (Ctrl+C). This is very useful if the tests are taking too long and you interrupt them with Ctrl+C to find out where the tests are hanging. By default no output will be shown (because KeyboardInterrupt is caught by pytest). By using this option you make sure a trace is shown.

**Dropping to PDB (Python Debugger) on failures**

Python comes with a builtin Python debugger called [PDB](http://docs.python.org/library/pdb.html). pytest allows one to drop into the [PDB](http://docs.python.org/library/pdb.html) prompt via a command line option:

pytest --pdb

This will invoke the Python debugger on every failure (or KeyboardInterrupt). Often you might only want to do this for the first failing test to understand a certain failure situation:

pytest -x --pdb # drop to PDB on first failure, then end test session pytest --pdb --maxfail=3 # drop to PDB for first three failures

Note that on any failure the exception information is stored on sys.last\_value, sys.last\_type and sys.last\_traceback. In interactive use, this allows one to drop into postmortem debugging with any debug tool. One can also manually access the exception information, for example:

* **import sys**
* sys.last\_traceback.tb\_lineno

42

* sys.last\_value AssertionError('assert result == "ok"',)

**Setting breakpoints**

To set a breakpoint in your code use the native Python import pdb;pdb.set\_trace() call in your code and pytest automatically disables its output capture for that test:

* Output capture in other tests is not affected.
* Any prior test output that has already been captured and will be processed as such.
* Any later output produced within the same test will not be captured and will instead get sent directly to sys.stdout. Note that this holds true even for test output occurring after you exit the interactive [PDB](http://docs.python.org/library/pdb.html) tracing session and continue with the regular test run.

**2.7. Dropping to PDB (Python Debugger) on failures** **9**

**pytest Documentation, Release 3.6**

**Using the builtin breakpoint function**

Python 3.7 introduces a builtin breakpoint() function. Pytest supports the use of breakpoint() with the following behaviours:

* When breakpoint() is called and PYTHONBREAKPOINT is set to the default value, pytest will use the custom internal PDB trace UI instead of the system default Pdb.
* When tests are complete, the system will default back to the system Pdb trace UI.
* If --pdb is called on execution of pytest, the custom internal Pdb trace UI is used on bothbreakpoint() and failed tests/unhandled exceptions.
* If --pdbcls is used, the custom class debugger will be executed when a test fails (as expected within existing behaviour), but also when breakpoint() is called from within a test, the custom class debugger will be instantiated.

**Profiling test execution duration**

To get a list of the slowest 10 test durations:

pytest --durations=10

**Creating JUnitXML format files**

To create result files which can be read by [Jenkins](http://jenkins-ci.org/) or other Continuous integration servers, use this invocation:

pytest --junitxml=path

to create an XML file at path.

New in version 3.1.

To set the name of the root test suite xml item, you can configure the junit\_suite\_name option in your config file:

**[pytest]**

junit\_suite\_name = my\_suite

**record\_property**

New in version 2.8.

Changed in version 3.5: Fixture renamed from record\_xml\_property to record\_property as user proper-ties are now available to all reporters. record\_xml\_property is now deprecated.

If you want to log additional information for a test, you can use the record\_property fixture:

**def** test\_function(record\_property):

record\_property("example\_key", 1)

**assert** True

This will add an extra property example\_key="1" to the generated testcase tag:

**10** **Chapter 2. Usage and Invocations**

**pytest Documentation, Release 3.6**

**<testcase** classname="test\_function" file="test\_function.py" line="0" name="test\_˓→function" time="0.0009"**>**

**<properties>**

**<property** name="example\_key" value="1" **/>**

**</properties>**

**</testcase>**

Alternatively, you can integrate this functionality with custom markers:

# content of conftest.py

**def** pytest\_collection\_modifyitems(session, config, items):

**for** item **in** items:

**for** marker **in** item.iter\_markers(name="test\_id"):

test\_id = marker.args[0]

item.user\_properties.append(("test\_id", test\_id))

And in your tests:

* content of test\_function.py **import pytest**

**@pytest.mark.test\_id**(1501)

**def** test\_function():

**assert** True

Will result in:

**<testcase** classname="test\_function" file="test\_function.py" line="0" name="test\_˓→function" time="0.0009"**>**

**<properties>**

**<property** name="test\_id" value="1501" **/>**

**</properties>**

**</testcase>**

Warning: record\_property is an experimental feature and may change in the future.

Also please note that using this feature will break any schema verification. This might be a problem when used with some CI servers.

**record\_xml\_attribute**

New in version 3.4.

To add an additional xml attribute to a testcase element, you can use record\_xml\_attribute fixture. This can also be used to override existing values:

**def** test\_function(record\_xml\_attribute):

record\_xml\_attribute("assertions", "REQ-1234")

record\_xml\_attribute("classname", "custom\_classname")

**print**("hello world")

**assert** True

**2.11. Creating JUnitXML format files** **11**

**pytest Documentation, Release 3.6**

Unlike record\_property, this will not add a new child element. Instead, this will add an attribute assertions="REQ-1234" inside the generated testcase tag and override the default classname with

"classname=custom\_classname":

**<testcase** classname="custom\_classname" file="test\_function.py" line="0" name="test\_˓→function" time="0.003" assertions="REQ-1234"**>**

**<system-out>**

hello world

**</system-out>**

**</testcase>**

Warning: record\_xml\_attribute is an experimental feature, and its interface might be replaced by some-thing more powerful and general in future versions. The functionality per-se will be kept, however.

Using this over record\_xml\_property can help when using ci tools to parse the xml report. However, some parsers are quite strict about the elements and attributes that are allowed. Many tools use an xsd schema (like the example below) to validate incoming xml. Make sure you are using attribute names that are allowed by your parser.

Below is the Scheme used by Jenkins to validate the XML report:

**<xs:element** name="testcase"**>**

**<xs:complexType>**

**<xs:sequence>**

**<xs:element** ref="skipped" minOccurs="0" maxOccurs="1"**/> <xs:element** ref="error" minOccurs="0" maxOccurs="unbounded"**/> <xs:element** ref="failure" minOccurs="0" maxOccurs="unbounded"**/> <xs:element** ref="system-out" minOccurs="0" maxOccurs="unbounded"**/> <xs:element** ref="system-err" minOccurs="0" maxOccurs="unbounded"**/>**

**</xs:sequence>**

**<xs:attribute** name="name" type="xs:string" use="required"**/> <xs:attribute** name="assertions" type="xs:string" use="optional"**/> <xs:attribute** name="time" type="xs:string" use="optional"**/> <xs:attribute** name="classname" type="xs:string" use="optional"**/> <xs:attribute** name="status" type="xs:string" use="optional"**/>**

**</xs:complexType>**

**</xs:element>**

**LogXML: add\_global\_property**

New in version 3.0.

If you want to add a properties node in the testsuite level, which may contains properties that are relevant to all testcases you can use LogXML.add\_global\_properties

**import pytest**

**@pytest.fixture**(scope="session")

**def** log\_global\_env\_facts(f):

**if** pytest.config.pluginmanager.hasplugin("junitxml"):

my\_junit = getattr(pytest.config, "\_xml", None)

my\_junit.add\_global\_property("ARCH", "PPC")

my\_junit.add\_global\_property("STORAGE\_TYPE", "CEPH")

**12** **Chapter 2. Usage and Invocations**

**pytest Documentation, Release 3.6**

**@pytest.mark.usefixtures**(log\_global\_env\_facts.\_\_name\_\_)

**def** start\_and\_prepare\_env():

**pass**

**class TestMe**(object):

**def** test\_foo(self):

**assert** True

This will add a property node below the testsuite node to the generated xml:

**<testsuite** errors="0" failures="0" name="pytest" skips="0" tests="1" time="0.006"**>**

**<properties>**

**<property** name="ARCH" value="PPC"**/> <property** name="STORAGE\_TYPE" value="CEPH"**/>**

**</properties>**

**<testcase** classname="test\_me.TestMe" file="test\_me.py" line="16" name="test\_foo"˓→time="0.000243663787842"**/>**

**</testsuite>**

Warning: This is an experimental feature, and its interface might be replaced by something more powerful and general in future versions. The functionality per-se will be kept.

**Creating resultlog format files**

Deprecated since version 3.0: This option is rarely used and is scheduled for removal in 4.0.

An alternative for users which still need similar functionality is to use the [pytest-tap](https://pypi.org/project/pytest-tap/) plugin which provides a stream of test data.

If you have any concerns, please don’t hesitate to [open an issue](https://github.com/pytest-dev/pytest/issues).

To create plain-text machine-readable result files you can issue:

pytest --resultlog=path

and look at the content at the path location. Such files are used e.g. by the [PyPy-test](http://buildbot.pypy.org/summary) web page to show test results over several revisions.

**Sending test report to online pastebin service**

Creating a URL for each test failure:

pytest --pastebin=failed

This will submit test run information to a remote Paste service and provide a URL for each failure. You may select tests as usual or add for example -x if you only want to send one particular failure.

Creating a URL for a whole test session log:

**2.12. Creating resultlog format files** **13**

**pytest Documentation, Release 3.6**

pytest --pastebin=all

Currently only pasting to the <http://bpaste.net>service is implemented.

**Disabling plugins**

To disable loading specific plugins at invocation time, use the -p option together with the prefix no:.

Example: to disable loading the plugin doctest, which is responsible for executing doctest tests from text files, invoke pytest like this:

pytest -p no:doctest

**Calling pytest from Python code**

New in version 2.0.

You can invoke pytest from Python code directly:

pytest.main()

this acts as if you would call “pytest” from the command line. It will not raise SystemExit but return the exitcode instead. You can pass in options and arguments:

pytest.main(['-x', 'mytestdir'])

You can specify additional plugins to pytest.main:

* content of myinvoke.py **import pytest**

**class MyPlugin**(object):

**def** pytest\_sessionfinish(self):

print("\*\*\* test run reporting finishing")

pytest.main(["-qq"], plugins=[MyPlugin()])

Running it will show that MyPlugin was added and its hook was invoked:

$ python myinvoke.py

. [100%]\*\*\* test ˓→run reporting finishing

Note: Calling pytest.main() will result in importing your tests and any modules that they import. Due to the caching mechanism of python’s import system, making subsequent calls to pytest.main() from the same process will not reflect changes to those files between the calls. For this reason, making multiple calls to pytest.main() from the same process (in order to re-run tests, for example) is not recommended.

**14** **Chapter 2. Usage and Invocations**

**CHAPTER 3**

**Using pytest with an existing test suite**

Pytest can be used with most existing test suites, but its behavior differs from other test runners such as [nose](#page95) or Python’s default unittest framework.

Before using this section you will want to [install pytest](#page11).

**Running an existing test suite with pytest**

Say you want to contribute to an existing repository somewhere. After pulling the code into your development space using some flavor of version control and (optionally) setting up a virtualenv you will want to run:

cd <repository>

pip install -e . # Environment dependent alternatives include

# 'python setup.py develop' and 'conda develop'

in your project root. This will set up a symlink to your code in site-packages, allowing you to edit your code while your tests run against it as if it were installed.

Setting up your project in development mode lets you avoid having to reinstall every time you want to run your tests, and is less brittle than mucking about with sys.path to point your tests at local code.

Also consider using [tox](#page165).

**15**

**pytest Documentation, Release 3.6**

**16** **Chapter 3. Using pytest with an existing test suite**

**CHAPTER 4**

**The writing and reporting of assertions in tests**

**Asserting with the assert statement**

pytest allows you to use the standard python assert for verifying expectations and values in Python tests. For example, you can write the following:

* content of test\_assert1.py **def** f():

**return** 3

**def** test\_function():

**assert** f()==4

to assert that your function returns a certain value. If this assertion fails you will see the return value of the function call:

$ pytest test\_assert1.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\_assert1.py F | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_function \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_function():

* assert f() == 4

Eassert 3 == 4

E+ where 3 = f()

test\_assert1.py:5: AssertionError

========================= 1 failed in 0.12 seconds =========================

pytest has support for showing the values of the most common subexpressions including calls, attributes, compar-isons, and binary and unary operators. (See [Demo of Python failure reports with pytest](#page175)). This allows you to use the idiomatic python constructs without boilerplate code while not losing introspection information.

However, if you specify a message with the assertion like this:

**17**

**pytest Documentation, Release 3.6**

**assert** a%2==0,"value was odd, should be even"

then no assertion introspection takes places at all and the message will be simply shown in the traceback.

See [Advanced assertion introspection](#page29) for more information on assertion introspection.

**Assertions about expected exceptions**

In order to write assertions about raised exceptions, you can use pytest.raises as a context manager like this:

**import pytest**

**def** test\_zero\_division():

**with** pytest.raises(ZeroDivisionError):

1 / 0

and if you need to have access to the actual exception info you may use:

**def** test\_recursion\_depth():

**with** pytest.raises(RuntimeError) **as** excinfo:

**def** f():

f()

f()

**assert** 'maximum recursion' **in** str(excinfo.value)

excinfo is a ExceptionInfo instance, which is a wrapper around the actual exception raised. The main attributes of interest are .type, .value and .traceback.

Changed in version 3.0.

In the context manager form you may use the keyword argument message to specify a custom failure message:

* **with** raises(ZeroDivisionError, message="Expecting ZeroDivisionError"):

**...pass**

**...** Failed: ExpectingZeroDivisionError

If you want to write test code that works on Python 2.4 as well, you may also use two other ways to test for an expected exception:

pytest.raises(ExpectedException, func, \*args, \*\*kwargs)

pytest.raises(ExpectedException, "func(\*args, \*\*kwargs)")

both of which execute the specified function with args and kwargs and asserts that the given ExpectedException is raised. The reporter will provide you with helpful output in case of failures such as no exception or wrong exception.

Note that it is also possible to specify a “raises” argument to pytest.mark.xfail, which checks that the test is failing in a more specific way than just having any exception raised:

**@pytest**.mark.xfail(raises=IndexError)

**def** test\_f():

f()

Using pytest.raises is likely to be better for cases where you are testing exceptions your own code is deliber-ately raising, whereas using @pytest.mark.xfail with a check function is probably better for something like documenting unfixed bugs (where the test describes what “should” happen) or bugs in dependencies.

**18** **Chapter 4. The writing and reporting of assertions in tests**

**pytest Documentation, Release 3.6**

Also, the context manager form accepts a match keyword parameter to test that a regular expression matches on the string representation of an exception (like the TestCase.assertRaisesRegexp method from unittest):

**import pytest**

**def** myfunc():

**raise** ValueError("Exception 123 raised")

**def** test\_match():

**with** pytest.raises(ValueError, match=r'.\*123 .\*'):

myfunc()

The regexp parameter of the match method is matched with the re.search function. So in the above example match='123' would have worked as well.

**Assertions about expected warnings**

New in version 2.8.

You can check that code raises a particular warning using [pytest.warns](#page65).

**Making use of context-sensitive comparisons**

New in version 2.0.

pytest has rich support for providing context-sensitive information when it encounters comparisons. For example:

# content of test\_assert2.py

**def** test\_set\_comparison():

set1 = set("1308")

set2 = set("8035")

**assert** set1==set2

if you run this module:

$ pytest test\_assert2.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\_assert2.py F | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_set\_comparison \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_set\_comparison():

set1 = set("1308")

set2 = set("8035")

* assert set1 == set2
* AssertionError: assert {'0', '1', '3', '8'} == {'0', '3', '5', '8'}
* Extra items in the left set:
* '1'
* Extra items in the right set:

**4.3. Assertions about expected warnings** **19**

**pytest Documentation, Release 3.6**

* '5'
* Use -v to get the full diff

test\_assert2.py:5: AssertionError

========================= 1 failed in 0.12 seconds =========================

Special comparisons are done for a number of cases:

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

See the [reporting demo](#page175) for many more examples.

**Defining your own assertion comparison**

It is possible to add your own detailed explanations by implementing the pytest\_assertrepr\_compare hook.

**pytest\_assertrepr\_compare**(config,op,left,right)

return explanation for comparisons in failing assert expressions.

Return None for no custom explanation, otherwise return a list of strings. The strings will be joined by newlines but any newlines in a string will be escaped. Note that all but the first line will be indented slightly, the intention is for the first line to be a summary.

Parameters **config** ([\_pytest.config.Config](#page144)) – pytest config object

As an example consider adding the following hook in a [conftest.py](#page32) file which provides an alternative explanation for

Foo objects:

# content of conftest.py

**from test\_foocompare import** Foo

**def** pytest\_assertrepr\_compare(op, left, right):

**if** isinstance(left, Foo) **and** isinstance(right, Foo) **and** op=="==":

**return** ['Comparing Foo instances:',

* vals: %s != %s' % (left.val, right.val)]

now, given this test module:

* content of test\_foocompare.py **class Foo**(object):

**def** \_\_init\_\_(self, val):self.val = val

**def** \_\_eq\_\_(self, other):

**return** self.val==other.val

**def** test\_compare():

f1 = Foo(1)

f2 = Foo(2)

**assert** f1==f2

you can run the test module and get the custom output defined in the conftest file:

$ pytest -q test\_foocompare.py

|  |  |
| --- | --- |
| F | [100%] |

**20** **Chapter 4. The writing and reporting of assertions in tests**

**pytest Documentation, Release 3.6**

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_compare \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_compare():

f1 = Foo(1)

f2 = Foo(2)

* assert f1 == f2
* assert Comparing Foo instances:
* vals: 1 != 2

test\_foocompare.py:11: AssertionError

1 failed in 0.12 seconds

**Advanced assertion introspection**

New in version 2.1.

Reporting details about a failing assertion is achieved by rewriting assert statements before they are run. Rewritten assert statements put introspection information into the assertion failure message. pytest only rewrites test modules directly discovered by its test collection process, so asserts in supporting modules which are not themselves test modules will not be rewritten.

Note: pytest rewrites test modules on import by using an import hook to write new pyc files. Most of the time this works transparently. However, if you are messing with import yourself, the import hook may interfere.

If this is the case you have two options:

* Disable rewriting for a specific module by adding the string PYTEST\_DONT\_REWRITE to its docstring.
* Disable rewriting for all modules by using --assert=plain.

Additionally, rewriting will fail silently if it cannot write new .pyc files, i.e. in a read-only filesystem or a zipfile.

For further information, Benjamin Peterson wrote up [Behind the scenes of pytest’s new assertion rewriting](http://pybites.blogspot.com/2011/07/behind-scenes-of-pytests-new-assertion.html).

New in version 2.1: Add assert rewriting as an alternate introspection technique.

Changed in version 2.1: Introduce the --assert option. Deprecate --no-assert and --nomagic.

Changed in version 3.0: Removes the --no-assert and --nomagic options. Removes the --assert=reinterp option.

**4.6. Advanced assertion introspection** **21**

**pytest Documentation, Release 3.6**

**22** **Chapter 4. The writing and reporting of assertions in tests**

**CHAPTER 5**

**pytest fixtures: explicit, modular, scalable**

New in version 2.0/2.3/2.4.

The [purpose of test fixtures](http://en.wikipedia.org/wiki/Test_fixture#Software) 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 function, class, module or whole test session scopes.

In addition, pytest continues to support [classic xunit-style setup](#page97). 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](#page91) or [nose based](#page95) projects.

**Fixtures as Function arguments**

Test functions can receive fixture objects by naming them as an input argument. For each argument name, a fix-ture 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\_connection():

**import smtplib**

**return** smtplib.SMTP("smtp.gmail.com",587, timeout=5)

**def** test\_ehlo(smtp\_connection):

response, msg = smtp\_connection.ehlo()

**assert** response==250

**assert** 0# for demo purposes

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

**23**

**pytest Documentation, Release 3.6**

$ 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 | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_ehlo \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

smtp\_connection = <smtplib.SMTP object at 0xdeadbeef>

def test\_ehlo(smtp\_connection):

response, msg = smtp\_connection.ehlo()

assert response == 250

* assert 0 # for demo purposes
* 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\_connection argument, the smtplib.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](#page163) the test\_ehlo because of the test\_ prefix. The test function needs a function argument named smtp\_connection. A matching fixture function is discovered by looking for a fixture-marked function named smtp\_connection.
2. smtp\_connection() is called to create an instance.
3. test\_ehlo(<smtp\_connection 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 with leading \_ are only shown if you add the -v option).

**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](http://en.wikipedia.org/wiki/Dependency_injection) where fixture functions take the role of the injector and test functions are the consumers of fixture objects.

**conftest.py: sharing 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. You don’t need to import the fixture you want to use in a test, it automatically gets

**24** **Chapter 5. pytest fixtures: explicit, modular, scalable**

**pytest Documentation, Release 3.6**

discovered by pytest. The discovery of fixture functions starts at test classes, then test modules, then conftest.py files and finally builtin and third party plugins.

You can also use the conftest.py file to implement [local per-directory plugins](#page102).

**Sharing test data**

If you want to make test data from files available to your tests, a good way to do this is by loading these data in a fixture for use by your tests. This makes use of the automatic caching mechanisms of pytest.

Another good approach is by adding the data files in the tests folder. There are also community plugins available to help managing this aspect of testing, e.g. [pytest-datadir](https://github.com/gabrielcnr/pytest-datadir) and [pytest-datafiles](https://pypi.org/project/pytest-datafiles/).

**Scope: sharing a fixture instance across tests in a class, module or 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\_connection fixture function to only be invoked once per test module (the default is to invoke once per test function). Multiple test functions in a test module will thus each receive the same smtp\_connection fixture instance, thus saving time.

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 smtplib**

**@pytest**.fixture(scope="module")

**def** smtp\_connection():

**return** smtplib.SMTP("smtp.gmail.com",587, timeout=5)

The name of the fixture again is smtp\_connection and you can access its result by listing the name smtp\_connection 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\_connection):

response, msg = smtp\_connection.ehlo()

**assert** response==250

**assert** b"smtp.gmail.com" **in** msg

**assert** 0 # for demo purposes

**def** test\_noop(smtp\_connection):

response, msg = smtp\_connection.noop()

**assert** response==250

**assert** 0 # for demo purposes

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

**5.4. Sharing test data** **25**

**pytest Documentation, Release 3.6**

$ 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 | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_ehlo \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

smtp\_connection = <smtplib.SMTP object at 0xdeadbeef>

def test\_ehlo(smtp\_connection):

response, msg = smtp\_connection.ehlo()

assert response == 250

assert b"smtp.gmail.com" in msg

* assert 0 # for demo purposes
* assert 0

test\_module.py:6: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_noop \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

smtp\_connection = <smtplib.SMTP object at 0xdeadbeef>

def test\_noop(smtp\_connection):

response, msg = smtp\_connection.noop()

assert response == 250

* assert 0 # for demo purposes
* 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\_connection 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\_connection 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\_connection instance, you can simply declare it:

**@pytest.fixture**(scope="session")

**def** smtp\_connection():

* the returned fixture value will be shared for
* all tests needing it

...

Finally, the class scope will invoke the fixture once per test class.

**Higher-scoped fixtures are instantiated first**

New in version 3.5.

Within a function request for features, fixture of higher-scopes (such as session) are instantiated first than lower-scoped fixtures (such as function or class). The relative order of fixtures of same scope follows the declared order in the test function and honours dependencies between fixtures.

**26** **Chapter 5. pytest fixtures: explicit, modular, scalable**

**pytest Documentation, Release 3.6**

Consider the code below:

**@pytest.fixture**(scope="session")

**def** s1():

**pass**

**@pytest.fixture**(scope="module")

**def** m1():

**pass**

**@pytest.fixture**

**def** f1(tmpdir):

**pass**

**@pytest.fixture**

**def** f2():

**pass**

**def** test\_foo(f1, m1, f2, s1):

...

The fixtures requested by test\_foo will be instantiated in the following order:

1. s1: is the highest-scoped fixture (session).
2. m1: is the second highest-scoped fixture (module).
3. tmpdir: is a function-scoped fixture, required by f1: it needs to be instantiated at this point because it is a dependency of f1.
4. f1: is the first function-scoped fixture in test\_foo parameter list.
5. f2: is the last function-scoped fixture in test\_foo parameter list.

**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\_connection():

smtp\_connection = smtplib.SMTP("smtp.gmail.com", 587, timeout=5)

**yield** smtp\_connection # provide the fixture value

**print**("teardown smtp")

smtp\_connection.close()

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.

**5.7. Fixture finalization / executing teardown code** **27**

**pytest Documentation, Release 3.6**

Let’s execute it:

* pytest -s -q --tb=no FFteardown smtp

2 failed in 0.12 seconds

We see that the smtp\_connection 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\_connection():

**with** smtplib.SMTP("smtp.gmail.com",587, timeout=5) **as** smtp\_connection:

**yield** smtp\_connection # provide the fixture value

The smtp\_connection connection will be closed after the test finished execution because the smtp\_connection 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](#page37) object to register finalization functions.

Here’s the smtp\_connection fixture changed to use addfinalizer for cleanup:

* content of conftest.py **import smtplib**

**import pytest**

**@pytest.fixture**(scope="module")

**def** smtp\_connection(request):

smtp\_connection = smtplib.SMTP("smtp.gmail.com", 587, timeout=5)

**def** fin():

**print**("teardown smtp\_connection")

smtp\_connection.close()

request.addfinalizer(fin)

**return** smtp\_connection # 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:

**28** **Chapter 5. pytest fixtures: explicit, modular, scalable**

**pytest Documentation, Release 3.6**

**@pytest**.fixture

**def** equipments(request):

r = []

**for** port **in** ('C1','C3','C28'):

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 functions can accept the request object to introspect the “requesting” test function, class or module context. Further extending the previous smtp\_connection 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\_connection(request):

server = getattr(request.module, "smtpserver", "smtp.gmail.com")

smtp\_connection = smtplib.SMTP(server, 587, timeout=5)

**yield** smtp\_connection

print ("finalizing %s (%s)" % (smtp\_connection, server))

smtp\_connection.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\_connection):

**assert** 0, smtp\_connection.helo()

Running it:

$ pytest -qq --tb=short test\_anothersmtp.py

F [100%]

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_showhelo \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

test\_anothersmtp.py:5: in test\_showhelo

assert 0, smtp\_connection.helo()

**5.8. Fixtures can introspect the requesting test context** **29**

**pytest Documentation, Release 3.6**

* AssertionError: (250, b'mail.python.org')
* assert 0

------------------------- Captured stdout teardown ------------------------- finalizing <smtplib.SMTP object at 0xdeadbeef> (mail.python.org)

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

**Factories as fixtures**

The “factory as fixture” pattern can help in situations where the result of a fixture is needed multiple times in a single test. Instead of returning data directly, the fixture instead returns a function which generates the data. This function can then be called multiple times in the test.

Factories can have have parameters as needed:

**@pytest**.fixture

**def** make\_customer\_record():

**def** \_make\_customer\_record(name):

**return** {

"name": name,

"orders": []

}

**return** \_make\_customer\_record

**def** test\_customer\_records(make\_customer\_record): customer\_1 = make\_customer\_record("Lisa") customer\_2 = make\_customer\_record("Mike") customer\_3 = make\_customer\_record("Meredith")

If the data created by the factory requires managing, the fixture can take care of that:

**@pytest**.fixture

**def** make\_customer\_record():

created\_records = []

**def** \_make\_customer\_record(name):

record = models.Customer(name=name, orders=[])

created\_records.append(record)

**return** record

**yield** \_make\_customer\_record

**for** record **in** created\_records:

record.destroy()

**def** test\_customer\_records(make\_customer\_record): customer\_1 = make\_customer\_record("Lisa") customer\_2 = make\_customer\_record("Mike") customer\_3 = make\_customer\_record("Meredith")

**30** **Chapter 5. pytest fixtures: explicit, modular, scalable**

**pytest Documentation, Release 3.6**

**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 usually do not need to be aware of their re-running. Fixture 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\_connection 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\_connection(request):

smtp\_connection = smtplib.SMTP(request.param, 587, timeout=5)

**yield** smtp\_connection

print("finalizing %s" % smtp\_connection)

smtp\_connection.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 [100%]

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_ehlo[smtp.gmail.com] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

smtp\_connection = <smtplib.SMTP object at 0xdeadbeef>

def test\_ehlo(smtp\_connection):

response, msg = smtp\_connection.ehlo()

assert response == 250

assert b"smtp.gmail.com" in msg

* assert 0 # for demo purposes
* assert 0

test\_module.py:6: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_noop[smtp.gmail.com] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

smtp\_connection = <smtplib.SMTP object at 0xdeadbeef>

def test\_noop(smtp\_connection):

response, msg = smtp\_connection.noop()

assert response == 250

* assert 0 # for demo purposes
* assert 0

test\_module.py:11: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_ehlo[mail.python.org] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

smtp\_connection = <smtplib.SMTP object at 0xdeadbeef>

def test\_ehlo(smtp\_connection):

**5.10. Parametrizing fixtures** **31**

**pytest Documentation, Release 3.6**

response, msg = smtp\_connection.ehlo()

assert response == 250

* assert b"smtp.gmail.com" in msg
* AssertionError: assert b'smtp.gmail.com' in b'mail.python.

˓→org\nPIPELINING\nSIZE 51200000\nETRN\nSTARTTLS\nAUTH DIGEST-MD5 NTLM CRAM-˓→MD5\nENHANCEDSTATUSCODES\n8BITMIME\nDSN\nSMTPUTF8'

test\_module.py:5: AssertionError

-------------------------- Captured stdout setup ---------------------------

finalizing <smtplib.SMTP object at 0xdeadbeef>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_noop[mail.python.org] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

smtp\_connection = <smtplib.SMTP object at 0xdeadbeef>

def test\_noop(smtp\_connection):

response, msg = smtp\_connection.noop()

assert response == 250

* assert 0 # for demo purposes
* assert 0

test\_module.py:11: AssertionError

------------------------- Captured stdout teardown -------------------------

finalizing <smtplib.SMTP object at 0xdeadbeef> 4 failed in 0.12 seconds

We see that our two test functions each ran twice, against the different smtp\_connection 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

**32** **Chapter 5. pytest fixtures: explicit, modular, scalable**

**pytest Documentation, Release 3.6**

**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'>

<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 =======================

**Using marks with parametrized fixtures**

[pytest.param()](#page120) can be used to apply marks in values sets of parametrized fixtures in the same way that they can be used with [@pytest.mark.parametrize](#page81).

Example:

* content of test\_fixture\_marks.py **import pytest**

**@pytest**.fixture(params=[0,1, pytest.param(2, marks=pytest.mark.skip)]) **def** data\_set(request):

**return** request.param

**def** test\_data(data\_set):

**pass**

Running this test will skip the invocation of data\_set with value 2:

$ pytest test\_fixture\_marks.py -v

=========================== 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: .pytest\_cache

rootdir: $REGENDOC\_TMPDIR, inifile:

collecting ... collected 3 items

**5.11. Using marks with parametrized fixtures** **33**

**pytest Documentation, Release 3.6**

|  |  |  |  |
| --- | --- | --- | --- |
| test\_fixture\_marks.py::test\_data[0] PASSED | | [ | 33%] |
| test\_fixture\_marks.py::test\_data[1] | PASSED | [ | 66%] |
| test\_fixture\_marks.py::test\_data[2] | SKIPPED | [100%] | |

=================== 2 passed, 1 skipped 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\_connection resource into it:

# content of test\_appsetup.py

**import pytest**

**class App**(object):

**def** \_\_init\_\_(self, smtp\_connection):

self.smtp\_connection = smtp\_connection

**@pytest**.fixture(scope="module")

**def** app(smtp\_connection):

**return** App(smtp\_connection)

**def** test\_smtp\_connection\_exists(app):

**assert** app.smtp\_connection

Here we declare an app fixture which receives the previously defined smtp\_connection 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: .pytest\_cache

rootdir: $REGENDOC\_TMPDIR, inifile:

collecting ... collected 2 items

test\_appsetup.py::test\_smtp\_connection\_exists[smtp.gmail.com] PASSED [ 50%] test\_appsetup.py::test\_smtp\_connection\_exists[mail.python.org] PASSED [100%]

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

Due to the parametrization of smtp\_connection, 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\_connection parametrization because pytest will fully analyse the fixture dependency graph.

Note, that the app fixture has a scope of module and uses a module-scoped smtp\_connection fixture. The example would still work if smtp\_connection 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.

**34** **Chapter 5. pytest fixtures: explicit, modular, scalable**

**pytest Documentation, Release 3.6**

**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 fixtures, 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 **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: .pytest\_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[mod1-1] SETUP otherarg 1

RUN test2 with otherarg 1 and modarg mod1

PASSED TEARDOWN otherarg 1

**5.13. Automatic grouping of tests by fixture instances** **35**

**pytest Documentation, Release 3.6**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| test\_module.py::test\_2[mod1-2] 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[mod2-1] SETUP | | | otherarg 1 |
| RUN test2 with | | otherarg 1 and modarg | mod2 |
| PASSED | TEARDOWN | otherarg 1 |  |
| test\_module.py::test\_2[mod2-2] SETUP | | | otherarg 2 |
| RUN test2 with | | otherarg 2 and modarg | mod2 |
| PASSED | TEARDOWN | otherarg 2 |  |

TEARDOWN modarg mod2

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

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](http://docs.python.org/library/tempfile.html) 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):

**36** **Chapter 5. pytest fixtures: explicit, modular, scalable**

**pytest Documentation, Release 3.6**

**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 |  |
| .. | | [100%] |
| 2 | passed in 0.12 seconds |  |
|  |  |  |

You can specify multiple fixtures like this:

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

**def** test():

...

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.

It is also possible to put fixtures required by all tests in your project into an ini-file:

* content of pytest.ini

**[pytest]**

usefixtures = cleandir

Warning: Note this mark has no effect in fixture functions. For example, this will not work as expected:

**@pytest.mark.usefixtures**("my\_other\_fixture")

**@pytest.fixture**

**def** my\_fixture\_that\_sadly\_wont\_use\_my\_other\_fixture():

...

Currently this will not generate any error or warning, but this is intended to be handled by [#3664](https://github.com/pytest-dev/pytest/issues/3664).

**Autouse fixtures (xUnit setup on steroids)**

Occasionally, you may want to have fixtures get invoked automatically without declaring a function argument ex-plicitly or a [usefixtures](#page44) decorator. As a practical example, suppose we have a database fixture which has a be-gin/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**

**5.15. Autouse fixtures (xUnit setup on steroids)** **37**

**pytest Documentation, Release 3.6**

**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 |  |
| .. | | [100%] |
| 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.

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(request, db):db.begin()

**38** **Chapter 5. pytest fixtures: explicit, modular, scalable**

**pytest Documentation, Release 3.6**

**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.

**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

**def** username(username):

**return** 'overridden-'+username

test\_something.py

* content of tests/subfolder/test\_something.py **def** test\_username(username):

**assert** username=='overridden-username'

**5.16. Overriding fixtures on various levels** **39**

**pytest Documentation, Release 3.6**

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

**40** **Chapter 5. pytest fixtures: explicit, modular, scalable**

**pytest Documentation, Release 3.6**

**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

**5.16. Overriding fixtures on various levels** **41**

**pytest Documentation, Release 3.6**

**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.

**42** **Chapter 5. pytest fixtures: explicit, modular, scalable**

**CHAPTER 6**

**Marking test functions with attributes**

By using the pytest.mark helper you can easily set metadata on your test functions. There are some builtin markers, for example:

* [skip](#page73) - always skip a test function
* [skipif](#page73) - skip a test function if a certain condition is met
* [xfail](#page76) - produce an “expected failure” outcome if a certain condition is met
* [parametrize](#page81) to perform multiple calls to the same test function.

It’s easy to create custom markers or to apply markers to whole test classes or modules. See [Working with custom](#page210) [markers](#page210) for examples which also serve as documentation.

Note: Marks can only be applied to tests, having no effect on [fixtures](#page31).

**Raising errors on unknown marks: –strict**

When the --strict command-line flag is passed, any marks not registered in the pytest.ini file will trigger an error.

Marks can be registered like this:

**[pytest]**

markers =

slow

serial

This can be used to prevent users mistyping mark names by accident. Test suites that want to enforce this should add --strict to addopts:

**[pytest]**

addopts = --strict

markers =

slow

serial

**43**

**pytest Documentation, Release 3.6**

**Marker revamp and iteration**

New in version 3.6.

pytest’s marker implementation traditionally worked by simply updating the \_\_dict\_\_ attribute of functions to cu-mulatively add markers. As a result, markers would unintentionally be passed along class hierarchies in surprising ways. Further, the API for retrieving them was inconsistent, as markers from parameterization would be stored differ-ently than markers applied using the @pytest.mark decorator and markers added via node.add\_marker.

This state of things made it technically next to impossible to use data from markers correctly without having a deep understanding of the internals, leading to subtle and hard to understand bugs in more advanced usages.

Depending on how a marker got declared/changed one would get either a MarkerInfo which might contain markers from sibling classes, MarkDecorators when marks came from parameterization or from a node.add\_marker call, discarding prior marks. Also MarkerInfo acts like a single mark, when it in fact represents a merged view on multiple marks with the same name.

On top of that markers were not accessible the same way for modules, classes, and functions/methods. In fact, markers were only accessible in functions, even if they were declared on classes/modules.

A new API to access markers has been introduced in pytest 3.6 in order to solve the problems with the initial design, providing [\_pytest.nodes.Node.iter\_markers()](#page150) method to iterate over markers in a consistent manner and reworking the internals, which solved great deal of problems with the initial design.

**Updating code**

The old Node.get\_marker(name) function is considered deprecated because it returns an internal MarkerInfo object which contains the merged name, \*args and \*\*kwargs of all the markers which apply to that node.

In general there are two scenarios on how markers should be handled:

1. Marks overwrite each other. Order matters but you only want to think of your mark as a single item. E.g. log\_level('info') at a module level can be overwritten by log\_level('debug') for a specific test.

In this case, use Node.get\_closest\_marker(name):

# replace this:

marker = item.get\_marker("log\_level")

**if** marker:

level = marker.args[0]

# by this:

marker = item.get\_closest\_marker("log\_level")

**if** marker:

level = marker.args[0]

1. Marks compose in an additive manner. E.g. skipif(condition) marks mean you just want to evaluate all of them, order doesn’t even matter. You probably want to think of your marks as a set here.

In this case iterate over each mark and handle their \*args and \*\*kwargs individually.

# replace this

skipif = item.get\_marker("skipif")

**if** skipif:

**for** condition **in** skipif.args:

* eval condition

...

**44** **Chapter 6. Marking test functions with attributes**

**pytest Documentation, Release 3.6**

# by this:

**for** skipif **in** item.iter\_markers("skipif"):

condition = skipif.args[0]

# eval condition

If you are unsure or have any questions, please consider opening [an issue](https://github.com/pytest-dev/pytest/issues).

**Related issues**

Here is a non-exhaustive list of issues fixed by the new implementation:

* Marks don’t pick up nested classes ([#199](https://github.com/pytest-dev/pytest/issues/199)).
* Markers stain on all related classes ([#568](https://github.com/pytest-dev/pytest/issues/568)).
* Combining marks - args and kwargs calculation ([#2897](https://github.com/pytest-dev/pytest/issues/2897)).
* request.node.get\_marker('name') returns None for markers applied in classes ([#902](https://github.com/pytest-dev/pytest/issues/902)).
* Marks applied in parametrize are stored as markdecorator ([#2400](https://github.com/pytest-dev/pytest/issues/2400)).
* Fix marker interaction in a backward incompatible way ([#1670](https://github.com/pytest-dev/pytest/issues/1670)).
* Refactor marks to get rid of the current “marks transfer” mechanism ([#2363](https://github.com/pytest-dev/pytest/issues/2363)).
* Introduce FunctionDefinition node, use it in generate\_tests ([#2522](https://github.com/pytest-dev/pytest/issues/2522)).
* Remove named marker attributes and collect markers in items ([#891](https://github.com/pytest-dev/pytest/issues/891)).
* skipif mark from parametrize hides module level skipif mark ([#1540](https://github.com/pytest-dev/pytest/issues/1540)).
* skipif + parametrize not skipping tests ([#1296](https://github.com/pytest-dev/pytest/issues/1296)).
* Marker transfer incompatible with inheritance ([#535](https://github.com/pytest-dev/pytest/issues/535)).

More details can be found in the [original PR](https://github.com/pytest-dev/pytest/pull/3317).

Note: in a future major relase of pytest we will introduce class based markers, at which points markers will no longer be limited to instances of Mark

**6.2. Marker revamp and iteration** **45**

**pytest Documentation, Release 3.6**

**46** **Chapter 6. Marking test functions with attributes**

**CHAPTER 7**

**Monkeypatching/mocking modules and environments**

Sometimes tests need to invoke functionality which depends on global settings or which invokes code which cannot be easily tested such as network access. The monkeypatch fixture helps you to safely set/delete an attribute, dictionary item or environment variable or to modify sys.path for importing. See the [monkeypatch blog post](http://tetamap.wordpress.com/2009/03/03/monkeypatching-in-unit-tests-done-right/) for some introduction material and a discussion of its motivation.

**Simple example: monkeypatching functions**

If you want to pretend that os.expanduser returns a certain directory, you can use the monkeypatch.setattr() method to patch this function before calling into a function which uses it:

* content of test\_module.py **import os.path**

**def** getssh():# pseudo application code

**return** os.path.join(os.path.expanduser("~admin"),'.ssh')

**def** test\_mytest(monkeypatch):

**def** mockreturn(path):

**return** '/abc'

monkeypatch.setattr(os.path, 'expanduser', mockreturn)

x = getssh()

**assert** x=='/abc/.ssh'

Here our test function monkeypatches os.path.expanduser and then calls into a function that calls it. After the test function finishes the os.path.expanduser modification will be undone.

**example: preventing “requests” from remote operations**

If you want to prevent the “requests” library from performing http requests in all your tests, you can do:

* content of conftest.py **import pytest @pytest**.fixture(autouse=**True**) **def** no\_requests(monkeypatch):

monkeypatch.delattr("requests.sessions.Session.request")

This autouse fixture will be executed for each test function and it will delete the method request.session.Session.request so that any attempts within tests to create http requests will fail.

**47**

**pytest Documentation, Release 3.6**

Note: Be advised that it is not recommended to patch builtin functions such as open, compile, etc., because it might break pytest’s internals. If that’s unavoidable, passing --tb=native, --assert=plain and --capture=no might help although there’s no guarantee.

Note: Mind that patching stdlib functions and some third-party libraries used by pytest might break pytest itself, therefore in those cases it is recommended to use [MonkeyPatch.context()](#page133) to limit the patching to the block you want tested:

**import functools**

**def** test\_partial(monkeypatch):

**with** monkeypatch.context() **as** m:

m.setattr(functools, "partial", 3)

**assert** functools.partial==3

See issue [#3290](https://github.com/pytest-dev/pytest/issues/3290) for details.

**API Reference**

Consult the docs for the [MonkeyPatch](#page132) class.

**48** **Chapter 7. Monkeypatching/mocking modules and environments**

**CHAPTER 8**

**Temporary directories and files**

**The ‘tmpdir’ fixture**

You can use the tmpdir fixture which will provide a temporary directory unique to the test invocation, created in the [base temporary directory](#page58).

tmpdir is a [py.path.local](https://py.readthedocs.io/en/latest/path.html) object which offers os.path methods and more. Here is an example test usage:

* content of test\_tmpdir.py **import os**

**def** test\_create\_file(tmpdir):

p = tmpdir.mkdir("sub").join("hello.txt")

p.write("content")

**assert** p.read()=="content" **assert** len(tmpdir.listdir())==1 **assert** 0

Running this would result in a passed test except for the last assert 0 line which we use to look at values:

$ pytest test\_tmpdir.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\_tmpdir.py F | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_create\_file \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

tmpdir = local('PYTEST\_TMPDIR/test\_create\_file0')

def test\_create\_file(tmpdir):

p = tmpdir.mkdir("sub").join("hello.txt")

p.write("content")

assert p.read() == "content"

assert len(tmpdir.listdir()) == 1

* assert 0
* assert 0

test\_tmpdir.py:7: AssertionError

========================= 1 failed in 0.12 seconds =========================

**49**

**pytest Documentation, Release 3.6**

**The ‘tmpdir\_factory’ fixture**

New in version 2.8.

The tmpdir\_factory is a session-scoped fixture which can be used to create arbitrary temporary directories from any other fixture or test.

For example, suppose your test suite needs a large image on disk, which is generated procedurally. Instead of com-puting the same image for each test that uses it into its own tmpdir, you can generate it once per-session to save time:

* contents of conftest.py **import pytest**

**@pytest.fixture**(scope="session")

**def** image\_file(tmpdir\_factory):

img = compute\_expensive\_image()

fn = tmpdir\_factory.mktemp("data").join("img.png")

img.save(str(fn))

**return** fn

* contents of test\_image.py **def** test\_histogram(image\_file):

img = load\_image(image\_file)

* + compute and test histogram

See [tmpdir\_factory API](#page136) for details.

**The default base temporary directory**

Temporary directories are by default created as sub-directories of the system temporary directory. The base name will be pytest-NUM where NUM will be incremented with each test run. Moreover, entries older than 3 temporary directories will be removed.

You can override the default temporary directory setting like this:

pytest --basetemp=mydir

When distributing tests on the local machine, pytest takes care to configure a basetemp directory for the sub pro-cesses such that all temporary data lands below a single per-test run basetemp directory.

**50** **Chapter 8. Temporary directories and files**

**CHAPTER 9**

**Capturing of the stdout/stderr output**

**Default stdout/stderr/stdin capturing behaviour**

During test execution any output sent to stdout and stderr is captured. If a test or a setup method fails its according captured output will usually be shown along with the failure traceback. (this behavior can be configured by the --show-capture command-line option).

In addition, stdin is set to a “null” object which will fail on attempts to read from it because it is rarely desired to wait for interactive input when running automated tests.

By default capturing is done by intercepting writes to low level file descriptors. This allows to capture output from simple print statements as well as output from a subprocess started by a test.

**Setting capturing methods or disabling capturing**

There are two ways in which pytest can perform capturing:

* file descriptor (FD) level capturing (default): All writes going to the operating system file descriptors 1 and 2 will be captured.
* sys level capturing: Only writes to Python files sys.stdout and sys.stderr will be captured. No capturing of writes to filedescriptors is performed.

You can influence output capturing mechanisms from the command line:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| pytest -s | | # disable | | all capturing |  |
| pytest | --capture=sys | # | replace | sys.stdout/stderr with in-mem | files |
| pytest | --capture=fd | # | also point filedescriptors 1 and 2 to | | temp file |
|  |  |  |  |  |  |

**Using print statements for debugging**

One primary benefit of the default capturing of stdout/stderr output is that you can use print statements for debugging:

# content of test\_module.py

**def** setup\_function(function):

print ("setting up %s" % function)

**def** test\_func1():

**51**

**pytest Documentation, Release 3.6**

**assert True**

**def** test\_func2():

**assert False**

and running this module will show you precisely the output of the failing function and hide the other one:

$ pytest

=========================== 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 .F | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_func2 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_func2():

* assert False

Eassert False

test\_module.py:9: AssertionError

-------------------------- Captured stdout setup ---------------------------

setting up <function test\_func2 at 0xdeadbeef>

==================== 1 failed, 1 passed in 0.12 seconds ====================

**Accessing captured output from a test function**

The capsys, capsysbinary, capfd, and capfdbinary fixtures allow access to stdout/stderr output created during test execution. Here is an example test function that performs some output related checks:

**def** test\_myoutput(capsys): # or use "capfd" for fd-level **print**("hello")

sys.stderr.write("world**\n**")

captured = capsys.readouterr()

**assert** captured.out=="hello**\n**"

**assert** captured.err=="world**\n**"

**print**("next")

captured = capsys.readouterr()

**assert** captured.out=="next**\n**"

The readouterr() call snapshots the output so far - and capturing will be continued. After the test function finishes the original streams will be restored. Using capsys this way frees your test from having to care about setting/resetting output streams and also interacts well with pytest’s own per-test capturing.

If you want to capture on filedescriptor level you can use the capfd fixture which offers the exact same interface but allows to also capture output from libraries or subprocesses that directly write to operating system level output streams (FD1 and FD2).

New in version 3.3.

The return value from readouterr changed to a namedtuple with two attributes, out and err.

New in version 3.3.

**52** **Chapter 9. Capturing of the stdout/stderr output**

**pytest Documentation, Release 3.6**

If the code under test writes non-textual data, you can capture this using the capsysbinary fixture which instead returns bytes from the readouterr method. The capfsysbinary fixture is currently only available in python 3.

New in version 3.3.

If the code under test writes non-textual data, you can capture this using the capfdbinary fixture which instead returns bytes from the readouterr method. The capfdbinary fixture operates on the filedescriptor level.

New in version 3.0.

To temporarily disable capture within a test, both capsys and capfd have a disabled() method that can be used as a context manager, disabling capture inside the with block:

**def** test\_disabling\_capturing(capsys):

**print**("this output is captured")

**with** capsys.disabled():

**print**("output not captured, going directly to sys.stdout")

**print**("this output is also captured")

**9.4. Accessing captured output from a test function** **53**

**pytest Documentation, Release 3.6**

**54** **Chapter 9. Capturing of the stdout/stderr output**

**CHAPTER 10**

**Warnings Capture**

New in version 3.1.

Starting from version 3.1, pytest now automatically catches warnings during test execution and displays them at the end of the session:

* content of test\_show\_warnings.py **import warnings**

**def** api\_v1():

warnings.warn(UserWarning("api v1, should use functions from v2"))

**return** 1

**def** test\_one():

**assert** api\_v1()==1

Running pytest now produces this output:

$ pytest test\_show\_warnings.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\_show\_warnings.py . | [100%] |

============================= warnings summary ============================= test\_show\_warnings.py::test\_one

$REGENDOC\_TMPDIR/test\_show\_warnings.py:4: UserWarning: api v1, should use functions ˓→from v2

warnings.warn(UserWarning("api v1, should use functions from v2"))

-- Docs: http://doc.pytest.org/en/latest/warnings.html

=================== 1 passed, 1 warnings in 0.12 seconds ===================

Pytest by default catches all warnings except for DeprecationWarning and PendingDeprecationWarning.

The -W flag can be passed to control which warnings will be displayed or even turn them into errors:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | $ pytest -q test\_show\_warnings.py | -W error::UserWarning | | |
|  | F |  | [100%] |  |
| ================================= | | FAILURES | ================================= |  |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | test\_one | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
|  |  |  |  |  |
|  |  |  |  |  |

**55**

**pytest Documentation, Release 3.6**

def test\_one():

* assert api\_v1() == 1

test\_show\_warnings.py:8:

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

def api\_v1():

* warnings.warn(UserWarning("api v1, should use functions from v2"))

EUserWarning: api v1, should use functions from v2

test\_show\_warnings.py:4: UserWarning

1 failed in 0.12 seconds

The same option can be set in the pytest.ini file using the filterwarnings ini option. For example, the configuration below will ignore all user warnings, but will transform all other warnings into errors.

**[pytest]**

filterwarnings =

error

ignore::UserWarning

When a warning matches more than one option in the list, the action for the last matching option is performed.

Both -W command-line option and filterwarnings ini option are based on Python’s own [-W option](https://docs.python.org/3/using/cmdline.html?highlight=#cmdoption-W) and [warn-ings.simplefilter](https://docs.python.org/3/library/warnings.html#warnings.simplefilter), so please refer to those sections in the Python documentation for other examples and advanced usage.

**@pytest.mark.filterwarnings**

New in version 3.2.

You can use the @pytest.mark.filterwarnings to add warning filters to specific test items, allowing you to have finer control of which warnings should be captured at test, class or even module level:

**import warnings**

**def** api\_v1():

warnings.warn(UserWarning("api v1, should use functions from v2"))

**return** 1

**@pytest.mark.filterwarnings**("ignore:api v1")

**def** test\_one():

**assert** api\_v1()==1

Filters applied using a mark take precedence over filters passed on the command line or configured by the filterwarnings ini option.

You may apply a filter to all tests of a class by using the filterwarnings mark as a class decorator or to all tests in a module by setting the pytestmark variable:

* turns all warnings into errors for this module pytestmark = pytest.mark.filterwarnings("error")

Note: Except for these features, pytest does not change the python warning filter; it only captures and displays the

**56** **Chapter 10. Warnings Capture**

**pytest Documentation, Release 3.6**

warnings which are issued with respect to the currently configured filter, including changes to the filter made by test functions or by the system under test.

Note: DeprecationWarning and PendingDeprecationWarning are hidden by the standard library by default so you have to explicitly configure them to be displayed in your pytest.ini:

**[pytest]**

filterwarnings =

once::DeprecationWarning

once::PendingDeprecationWarning

Credits go to Florian Schulze for the reference implementation in the [pytest-warnings](https://github.com/fschulze/pytest-warnings) plugin.

**Disabling warning capture**

This feature is enabled by default but can be disabled entirely in your pytest.ini file with:

**[pytest]**

addopts = -p no:warnings

Or passing -p no:warnings in the command-line.

**Asserting warnings with the warns function**

New in version 2.8.

You can check that code raises a particular warning using pytest.warns, which works in a similar manner to [raises](#page26):

**import warnings**

**import pytest**

**def** test\_warning():

**with** pytest.warns(UserWarning):

warnings.warn("my warning", UserWarning)

The test will fail if the warning in question is not raised. The keyword argument match to assert that the exception matches a text or regex:

* **with** warns(UserWarning, match='must be 0 or None'):

**...** warnings.warn("value must be 0 or None", UserWarning)

* **with** warns(UserWarning, match=r'must be \d+$'):

**...**warnings.warn("value must be 42", UserWarning)

* **with** warns(UserWarning, match=r'must be \d+$'):

**...** warnings.warn("this is not here",UserWarning)Traceback (most recent call last):

...

Failed: DID NOT WARN. No warnings of type ...UserWarning... was emitted...

**10.2. Disabling warning capture** **57**

**pytest Documentation, Release 3.6**

You can also call pytest.warns on a function or code string:

pytest.warns(expected\_warning, func, \*args, \*\*kwargs)

pytest.warns(expected\_warning, "func(\*args, \*\*kwargs)")

The function also returns a list of all raised warnings (as warnings.WarningMessage objects), which you can query for additional information:

**with** pytest.warns(RuntimeWarning) **as** record:

warnings.warn("another warning", RuntimeWarning)

* check that only one warning was raised **assert** len(record)==1
* check that the message matches

**assert** record[0].message.args[0]=="another warning"

Alternatively, you can examine raised warnings in detail using the [recwarn](#page66) fixture (see below).

Note: DeprecationWarning and PendingDeprecationWarning are treated differently; see [Ensuring a](#page67) [function triggers a deprecation warning](#page67).

**Recording warnings**

You can record raised warnings either using pytest.warns or with the recwarn fixture.

To record with pytest.warns without asserting anything about the warnings, pass None as the expected warning type:

**with** pytest.warns(**None**) **as** record:warnings.warn("user", UserWarning) warnings.warn("runtime", RuntimeWarning)

**assert** len(record)==2

**assert** str(record[0].message)=="user"

**assert** str(record[1].message)=="runtime"

The recwarn fixture will record warnings for the whole function:

**import warnings**

**def** test\_hello(recwarn):

warnings.warn("hello", UserWarning)

**assert** len(recwarn)==1

w = recwarn.pop(UserWarning)

**assert** issubclass(w.category,UserWarning)

**assert** str(w.message)=="hello"

**assert** w.filename

**assert** w.lineno

Both recwarn and pytest.warns return the same interface for recorded warnings: a WarningsRecorder instance. To view the recorded warnings, you can iterate over this instance, call len on it to get the number of recorded warnings, or index into it to get a particular recorded warning.

Full API: WarningsRecorder.

**58** **Chapter 10. Warnings Capture**

**pytest Documentation, Release 3.6**

**Ensuring a function triggers a deprecation warning**

You can also call a global helper for checking that a certain function call triggers a DeprecationWarning or

PendingDeprecationWarning:

**import pytest**

**def** test\_global():

pytest.deprecated\_call(myfunction, 17)

By default, DeprecationWarning and PendingDeprecationWarning will not be caught when using pytest.warns or recwarn because default Python warnings filters hide them. If you wish to record them in your own code, use the command warnings.simplefilter('always'):

**import warnings**

**import pytest**

**def** test\_deprecation(recwarn):

warnings.simplefilter('always')

warnings.warn("deprecated", DeprecationWarning)

**assert** len(recwarn)==1

**assert** recwarn.pop(DeprecationWarning)

You can also use it as a contextmanager:

**def** test\_global():

**with** pytest.deprecated\_call():

myobject.deprecated\_method()

**10.5. Ensuring a function triggers a deprecation warning** **59**

**pytest Documentation, Release 3.6**

**60** **Chapter 10. Warnings Capture**

**CHAPTER 11**

**Doctest integration for modules and test files**

By default all files matching the test\*.txt pattern will be run through the python standard doctest module. You can change the pattern by issuing:

pytest --doctest-glob='\*.rst'

on the command line. Since version 2.9, --doctest-glob can be given multiple times in the command-line.

New in version 3.1: You can specify the encoding that will be used for those doctest files using the doctest\_encoding ini option:

* content of pytest.ini

**[pytest]**

doctest\_encoding = latin1

The default encoding is UTF-8.

You can also trigger running of doctests from docstrings in all python modules (including regular python test modules):

pytest --doctest-modules

You can make these changes permanent in your project by putting them into a pytest.ini file like this:

* content of pytest.ini

**[pytest]**

addopts = --doctest-modules

If you then have a text file like this:

# content of example.rst

hello this **is** a doctest

* x = 3
* x

3

and another like this:

* content of mymodule.py **def** something():
  + a doctest in a docstring

>>> something() 42

**61**

**pytest Documentation, Release 3.6**

"""

**return** 42

then you can just invoke pytest without command line options:

$ pytest

=========================== 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: pytest.ini collected 1 item

|  |  |
| --- | --- |
| mymodule.py . | [100%] |

========================= 1 passed in 0.12 seconds =========================

It is possible to use fixtures using the getfixture helper:

# content of example.rst

* tmp = getfixture('tmpdir')
* ...

>>>

Also, [Using fixtures from classes, modules or projects](#page44) and [Autouse fixtures (xUnit setup on steroids)](#page45) fixtures are supported when executing text doctest files.

The standard doctest module provides some setting flags to configure the strictness of doctest tests. In pytest, you can enable those flags using the configuration file. To make pytest ignore trailing whitespaces and ignore lengthy exception stack traces you can just write:

**[pytest]**

doctest\_optionflags= NORMALIZE\_WHITESPACE IGNORE\_EXCEPTION\_DETAIL

pytest also introduces new options to allow doctests to run in Python 2 and Python 3 unchanged:

* ALLOW\_UNICODE: when enabled, the u prefix is stripped from unicode strings in expected doctest output.
* ALLOW\_BYTES: when enabled, the b prefix is stripped from byte strings in expected doctest output.

As with any other option flag, these flags can be enabled in pytest.ini using the doctest\_optionflags ini option:

**[pytest]**

doctest\_optionflags = ALLOW\_UNICODE ALLOW\_BYTES

Alternatively, it can be enabled by an inline comment in the doc test itself:

# content of example.rst

* get\_unicode\_greeting() # doctest: +ALLOW\_UNICODE 'Hello'

By default, pytest would report only the first failure for a given doctest. If you want to continue the test even when you have failures, do:

pytest --doctest-modules --doctest-**continue**-on-failure

**62** **Chapter 11. Doctest integration for modules and test files**

**pytest Documentation, Release 3.6**

**The ‘doctest\_namespace’ fixture**

New in version 3.0.

The doctest\_namespace fixture can be used to inject items into the namespace in which your doctests run. It is intended to be used within your own fixtures to provide the tests that use them with context.

doctest\_namespace is a standard dict object into which you place the objects you want to appear in the doctest namespace:

* content of conftest.py **import numpy @pytest**.fixture(autouse=**True**) **def** add\_np(doctest\_namespace):

doctest\_namespace['np'] = numpy

which can then be used in your doctests directly:

* content of numpy.py **def** arange():

"""

* + a = np.arange(10)
  + len(a)

10

"""

**pass**

**Output format**

New in version 3.0.

You can change the diff output format on failure for your doctests by using one of standard doctest modules format in options (see [doctest.REPORT\_UDIFF](https://docs.python.org/3/library/doctest.html#doctest.REPORT_UDIFF), [doctest.REPORT\_CDIFF](https://docs.python.org/3/library/doctest.html#doctest.REPORT_CDIFF), [doctest.REPORT\_NDIFF](https://docs.python.org/3/library/doctest.html#doctest.REPORT_NDIFF), [doctest.REPORT\_ONLY\_FIRST\_FAILURE](https://docs.python.org/3/library/doctest.html#doctest.REPORT_ONLY_FIRST_FAILURE)):

pytest --doctest-modules --doctest-report none pytest --doctest-modules --doctest-report udiff pytest --doctest-modules --doctest-report cdiff pytest --doctest-modules --doctest-report ndiff

pytest --doctest-modules --doctest-report only\_first\_failure

**11.1. The ‘doctest\_namespace’ fixture** **63**

**pytest Documentation, Release 3.6**

**64** **Chapter 11. Doctest integration for modules and test files**

**CHAPTER 12**

**Skip and xfail: dealing with tests that cannot succeed**

You can mark test functions that cannot be run on certain platforms or that you expect to fail so pytest can deal with them accordingly and present a summary of the test session, while keeping the test suite green.

A skip means that you expect your test to pass only if some conditions are met, otherwise pytest should skip running the test altogether. Common examples are skipping windows-only tests on non-windows platforms, or skipping tests that depend on an external resource which is not available at the moment (for example a database).

A xfail means that you expect a test to fail for some reason. A common example is a test for a feature not yet implemented, or a bug not yet fixed. When a test passes despite being expected to fail (marked with pytest.mark.xfail), it’s an xpass and will be reported in the test summary.

pytest counts and lists skip and xfail tests separately. Detailed information about skipped/xfailed tests is not shown by default to avoid cluttering the output. You can use the -r option to see details corresponding to the “short” letters shown in the test progress:

pytest -rxXs # show extra info on xfailed, xpassed, and skipped tests

More details on the -r option can be found by running pytest -h.

(See [How to change command line options defaults](#page172))

**Skipping test functions**

New in version 2.9.

The simplest way to skip a test function is to mark it with the skip decorator which may be passed an optional reason:

**@pytest.mark.skip**(reason="no way of currently testing this")

**def** test\_the\_unknown():

...

Alternatively, it is also possible to skip imperatively during test execution or setup by calling the pytest.skip(reason) function:

**def** test\_function():

**if not** valid\_config():

pytest.skip("unsupported configuration")

It is also possible to skip the whole module using pytest.skip(reason,allow\_module\_level=True) at the module level:

**65**

**pytest Documentation, Release 3.6**

**import pytest**

**if not** pytest.config.getoption("--custom-flag"):

pytest.skip("--custom-flag is missing, skipping tests", allow\_module\_level=True)

The imperative method is useful when it is not possible to evaluate the skip condition during import time.

Reference: [pytest.mark.skip](#page124)

**skipif**

New in version 2.0.

If you wish to skip something conditionally then you can use skipif instead. Here is an example of marking a test function to be skipped when run on an interpreter earlier than Python3.6

**import sys**

**@pytest**.mark.skipif(sys.version\_info<(3,6),

reason="requires python3.6 or higher")

**def** test\_function():

...

If the condition evaluates to True during collection, the test function will be skipped, with the specified reason appearing in the summary when using -rs.

You can share skipif markers between modules. Consider this test module:

* content of test\_mymodule.py **import mymodule**

minversion = pytest.mark.skipif(mymodule.\_\_versioninfo\_\_ < (1,1),

reason="at least mymodule-1.1 required")

**@minversion**

**def** test\_function():

...

You can import the marker and reuse it in another test module:

# test\_myothermodule.py

**from test\_mymodule import** minversion

**@minversion**

**def** test\_anotherfunction():

...

For larger test suites it’s usually a good idea to have one file where you define the markers which you then consistently apply throughout your test suite.

Alternatively, you can use [condition strings](#page234) instead of booleans, but they can’t be shared between modules easily so they are supported mainly for backward compatibility reasons.

Reference: [pytest.mark.skipif](#page125)

**Skip all test functions of a class or module**

You can use the skipif marker (as any other marker) on classes:

**66** **Chapter 12. Skip and xfail: dealing with tests that cannot succeed**

**pytest Documentation, Release 3.6**

**@pytest**.mark.skipif(sys.platform=='win32',

reason="does not run on windows")

**class TestPosixCalls**(object):

**def** test\_function(self):

"will not be setup or run under 'win32' platform"

If the condition is True, this marker will produce a skip result for each of the test methods of that class.

Warning: The use of skipif on classes that use inheritance is strongly discouraged. [A Known bug](https://github.com/pytest-dev/pytest/issues/568) in pytest’s markers may cause unexpected behavior in super classes.

If you want to skip all test functions of a module, you may use the pytestmark name on the global level:

# test\_module.py

pytestmark = pytest.mark.skipif(...)

If multiple skipif decorators are applied to a test function, it will be skipped if any of the skip conditions is true.

**Skipping files or directories**

Sometimes you may need to skip an entire file or directory, for example if the tests rely on Python version-specific features or contain code that you do not wish pytest to run. In this case, you must exclude the files and directories from collection. Refer to [Customizing test collection](#page225) for more information.

**Skipping on a missing import dependency**

You can use the following helper at module level or within a test or test setup function:

docutils = pytest.importorskip("docutils")

If docutils cannot be imported here, this will lead to a skip outcome of the test. You can also skip based on the version number of a library:

docutils = pytest.importorskip("docutils", minversion="0.3")

The version will be read from the specified module’s \_\_version\_\_ attribute.

**Summary**

Here’s a quick guide on how to skip tests in a module in different situations:

1. Skip all tests in a module unconditionally:

pytestmark = pytest.mark.skip("all tests still WIP")

1. Skip all tests in a module based on some condition:

pytestmark = pytest.mark.skipif(sys.platform == "win32", "tests for linux ˓→only")

1. Skip all tests in a module if some import is missing:

**12.1. Skipping test functions** **67**

**pytest Documentation, Release 3.6**

pexpect = pytest.importorskip("pexpect")

**XFail: mark test functions as expected to fail**

You can use the xfail marker to indicate that you expect a test to fail:

**@pytest**.mark.xfail

**def** test\_function():

...

This test will be run but no traceback will be reported when it fails. Instead terminal reporting will list it in the “expected to fail” (XFAIL) or “unexpectedly passing” (XPASS) sections.

Alternatively, you can also mark a test as XFAIL from within a test or setup function imperatively:

**def** test\_function():

**if not** valid\_config():

pytest.xfail("failing configuration (but should work)")

This will unconditionally make test\_function XFAIL. Note that no other code is executed after pytest.xfail call, differently from the marker. That’s because it is implemented internally by raising a known exception.

Reference: [pytest.mark.xfail](#page125)

**strict parameter**

New in version 2.9.

Both XFAIL and XPASS don’t fail the test suite, unless the strict keyword-only parameter is passed as True:

**@pytest.mark.xfail**(strict=True)

**def** test\_function():

...

This will make XPASS (“unexpectedly passing”) results from this test to fail the test suite.

You can change the default value of the strict parameter using the xfail\_strict ini option:

**[pytest]**

xfail\_strict=true

**reason parameter**

As with [skipif](#page73) you can also mark your expectation of a failure on a particular platform:

**@pytest**.mark.xfail(sys.version\_info>=(3,6),reason="python3.6 api changes")

**def** test\_function():

...

**68** **Chapter 12. Skip and xfail: dealing with tests that cannot succeed**

**pytest Documentation, Release 3.6**

**raises parameter**

If you want to be more specific as to why the test is failing, you can specify a single exception, or a list of exceptions, in the raises argument.

**@pytest.mark.xfail**(raises=RuntimeError)

**def** test\_function():

...

Then the test will be reported as a regular failure if it fails with an exception not mentioned in raises.

**run parameter**

If a test should be marked as xfail and reported as such but should not be even executed, use the run parameter as

False:

**@pytest.mark.xfail**(run=False)

**def** test\_function():

...

This is specially useful for xfailing tests that are crashing the interpreter and should be investigated later.

**Ignoring xfail**

By specifying on the commandline:

pytest --runxfail

you can force the running and reporting of an xfail marked test as if it weren’t marked at all. This also causes pytest.xfail to produce no effect.

**Examples**

Here is a simple test file with the several usages:

**import pytest**

xfail = pytest.mark.xfail

**@xfail**

**def** test\_hello():

**assert** 0

**@xfail**(run=**False**)

**def** test\_hello2():

**assert** 0

**@xfail**("hasattr(os, 'sep')")

**def** test\_hello3():

**assert** 0

**12.2. XFail: mark test functions as expected to fail** **69**

**pytest Documentation, Release 3.6**

**@xfail**(reason="bug 110")

**def** test\_hello4():

**assert** 0

**@xfail**('pytest.\_\_version\_\_[0] != "17"')

**def** test\_hello5():

**assert** 0

**def** test\_hello6():

pytest.xfail("reason")

**@xfail**(raises=IndexError)

**def** test\_hello7():

x = []

x[1] = 1

Running it with the report-on-xfail option gives this output:

example $ pytest -rx xfail\_demo.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/example, inifile: collected 7 items

|  |  |
| --- | --- |
| xfail\_demo.py xxxxxxx | [100%] |

========================= short test summary info ==========================

XFAIL xfail\_demo.py::test\_hello

XFAIL xfail\_demo.py::test\_hello2

reason: [NOTRUN]

XFAIL xfail\_demo.py::test\_hello3

condition: hasattr(os, 'sep')

XFAIL xfail\_demo.py::test\_hello4

bug 110

XFAIL xfail\_demo.py::test\_hello5

condition: pytest.\_\_version\_\_[0] != "17"

XFAIL xfail\_demo.py::test\_hello6

reason: reason

XFAIL xfail\_demo.py::test\_hello7

======================== 7 xfailed in 0.12 seconds =========================

**Skip/xfail with parametrize**

It is possible to apply markers like skip and xfail to individual test instances when using parametrize:

**import pytest**

**@pytest.mark.parametrize**(

("n", "expected"),

[

**70** **Chapter 12. Skip and xfail: dealing with tests that cannot succeed**

**pytest Documentation, Release 3.6**

(1, 2),

pytest.param(1, 0, marks=pytest.mark.xfail),

pytest.param(1, 3, marks=pytest.mark.xfail(reason="some bug")),

(2, 3),

(3, 4),

(4, 5),

pytest.param(

10, 11, marks=pytest.mark.skipif(sys.version\_info >= (3, 0), reason="py2k

˓→")

),

],

)

**def** test\_increment(n, expected):

**assert** n+1==expected

**12.3. Skip/xfail with parametrize** **71**

**pytest Documentation, Release 3.6**

**72** **Chapter 12. Skip and xfail: dealing with tests that cannot succeed**

**CHAPTER 13**

**Parametrizing fixtures and test functions**

pytest enables test parametrization at several levels:

* [pytest.fixture()](#page126) allows one to [parametrize fixture functions](#page39).
* [@pytest.mark.parametrize](#page81) allows one to define multiple sets of arguments and fixtures at the test function or class.
* [pytest\_generate\_tests](#page83) allows one to define custom parametrization schemes or extensions.

**@pytest.mark.parametrize: parametrizing test functions**

New in version 2.2.

Changed in version 2.4: Several improvements.

The builtin [pytest.mark.parametrize](#page124) decorator enables parametrization of arguments for a test function. Here is a typical example of a test function that implements checking that a certain input leads to an expected output:

* content of test\_expectation.py **import pytest**

**@pytest**.mark.parametrize("test\_input,expected", [("3+5", 8),

("2+4", 6), ("6\*9", 42),

])

**def** test\_eval(test\_input, expected): **assert** eval(test\_input)==expected

Here, the @parametrize decorator defines three different (test\_input,expected) tuples so that the test\_eval function will run three times using them in turn:

$ pytest

=========================== 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 3 items

|  |  |
| --- | --- |
| test\_expectation.py ..F | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_eval[6\*9-42] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**73**

**pytest Documentation, Release 3.6**

test\_input = '6\*9', expected = 42

@pytest.mark.parametrize("test\_input,expected", [

("3+5", 8),

("2+4", 6),

("6\*9", 42),

])

def test\_eval(test\_input, expected):

* assert eval(test\_input) == expected

EAssertionError: assert 54 == 42

E+ where 54 = eval('6\*9')

test\_expectation.py:8: AssertionError

==================== 1 failed, 2 passed in 0.12 seconds ====================

As designed in this example, only one pair of input/output values fails the simple test function. And as usual with test function arguments, you can see the input and output values in the traceback.

Note that you could also use the parametrize marker on a class or a module (see [Marking test functions with attributes](#page51)) which would invoke several functions with the argument sets.

It is also possible to mark individual test instances within parametrize, for example with the builtin mark.xfail:

* content of test\_expectation.py **import pytest**

**@pytest**.mark.parametrize("test\_input,expected", [("3+5", 8), ("2+4", 6),

pytest.param("6\*9", 42,

marks=pytest.mark.xfail),

])

**def** test\_eval(test\_input, expected):

**assert** eval(test\_input)==expected

Let’s run this:

$ pytest

=========================== 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 3 items

|  |  |
| --- | --- |
| test\_expectation.py ..x | [100%] |

=================== 2 passed, 1 xfailed in 0.12 seconds ====================

The one parameter set which caused a failure previously now shows up as an “xfailed (expected to fail)” test.

To get all combinations of multiple parametrized arguments you can stack parametrize decorators:

**import pytest**

**@pytest**.mark.parametrize("x", [0,1])

**@pytest**.mark.parametrize("y", [2,3])

**def** test\_foo(x, y):

**pass**

This will run the test with the arguments set to x=0/y=2, x=1/y=2, x=0/y=3, and x=1/y=3 exhausting parame-ters in the order of the decorators.

**74** **Chapter 13. Parametrizing fixtures and test functions**

**pytest Documentation, Release 3.6**

**Basic pytest\_generate\_tests example**

Sometimes you may want to implement your own parametrization scheme or implement some dynamism for deter-mining the parameters or scope of a fixture. For this, you can use the pytest\_generate\_tests hook which is called when collecting a test function. Through the passed in metafunc object you can inspect the requesting test context and, most importantly, you can call metafunc.parametrize() to cause parametrization.

For example, let’s say we want to run a test taking string inputs which we want to set via a new pytest command line option. Let’s first write a simple test accepting a stringinput fixture function argument:

# content of test\_strings.py

**def** test\_valid\_string(stringinput):

**assert** stringinput.isalpha()

Now we add a conftest.py file containing the addition of a command line option and the parametrization of our test function:

# content of conftest.py

**def** pytest\_addoption(parser):

parser.addoption("--stringinput", action="append", default=[],

help="list of stringinputs to pass to test functions")

**def** pytest\_generate\_tests(metafunc):

**if** 'stringinput' **in** metafunc.fixturenames:

metafunc.parametrize("stringinput",

metafunc.config.getoption('stringinput'))

If we now pass two stringinput values, our test will run twice:

|  |  |  |
| --- | --- | --- |
| $ | pytest -q --stringinput="hello" --stringinput="world" test\_strings.py | |
| .. | | [100%] |
| 2 | passed in 0.12 seconds |  |
|  |  |  |

Let’s also run with a stringinput that will lead to a failing test:

$ pytest -q --stringinput="!" test\_strings.py

F [100%]

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_valid\_string[!] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

stringinput = '!'

def test\_valid\_string(stringinput):

* assert stringinput.isalpha()

EAssertionError: assert False

E+ where False = <built-in method isalpha of str object at 0xdeadbeef>()

* + where <built-in method isalpha of str object at 0xdeadbeef> = '!'. ˓→isalpha

test\_strings.py:3: AssertionError

1 failed in 0.12 seconds

As expected our test function fails.

If you don’t specify a stringinput it will be skipped because metafunc.parametrize() will be called with an empty parameter list:

**13.2. Basic pytest\_generate\_tests example** **75**

**pytest Documentation, Release 3.6**

$ pytest -q -rs test\_strings.py

s [100%]

========================= short test summary info ==========================

SKIP [1] test\_strings.py: got empty parameter set ['stringinput'], function test\_

˓→valid\_string at $REGENDOC\_TMPDIR/test\_strings.py:1 1 skipped in 0.12 seconds

Note that when calling metafunc.parametrize multiple times with different parameter sets, all parameter names across those sets cannot be duplicated, otherwise an error will be raised.

**More examples**

For further examples, you might want to look at [more parametrization examples](#page200).

**76** **Chapter 13. Parametrizing fixtures and test functions**

**CHAPTER 14**

**Cache: working with cross-testrun state**

New in version 2.8.

**Usage**

The plugin provides two command line options to rerun failures from the last pytest invocation:

* --lf, --last-failed - to only re-run the failures.
* --ff, --failed-first - to run the failures first and then the rest of the tests.

For cleanup (usually not needed), a --cache-clear option allows to remove all cross-session cache contents ahead of a test run.

Other plugins may access the [config.cache](#page87) object to set/get json encodable values between pytest invocations.

Note: This plugin is enabled by default, but can be disabled if needed: see [Deactivating / unregistering a plugin by](#page100) [name](#page100) (the internal name for this plugin is cacheprovider).

**Rerunning only failures or failures first**

First, let’s create 50 test invocation of which only 2 fail:

* content of test\_50.py **import pytest**

**@pytest**.mark.parametrize("i",range(50))

**def** test\_num(i):

**if** i **in** (17,25):

pytest.fail("bad luck")

If you run this for the first time you will see two failures:

|  |  |  |
| --- | --- | --- |
| $ pytest -q |  |  |
| .................F....... | F........................ | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_num[17] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

i = 17

**77**

**pytest Documentation, Release 3.6**

@pytest.mark.parametrize("i", range(50))

def test\_num(i):

if i in (17, 25):

* pytest.fail("bad luck")

EFailed: bad luck

test\_50.py:6: Failed

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_num[25] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

i = 25

@pytest.mark.parametrize("i", range(50))

def test\_num(i):

if i in (17, 25):

* pytest.fail("bad luck")

EFailed: bad luck

test\_50.py:6: Failed

2 failed, 48 passed in 0.12 seconds

If you then run it with --lf:

$ pytest --lf

=========================== 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 50 items / 48 deselected

run-last-failure: rerun previous 2 failures

|  |  |
| --- | --- |
| test\_50.py FF | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_num[17] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

i = 17

@pytest.mark.parametrize("i", range(50))

def test\_num(i):

if i in (17, 25):

* pytest.fail("bad luck")

EFailed: bad luck

test\_50.py:6: Failed

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_num[25] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

i = 25

@pytest.mark.parametrize("i", range(50))

def test\_num(i):

if i in (17, 25):

* pytest.fail("bad luck")

EFailed: bad luck

test\_50.py:6: Failed

================= 2 failed, 48 deselected in 0.12 seconds ==================

You have run only the two failing test from the last run, while 48 tests have not been run (“deselected”).

**78** **Chapter 14. Cache: working with cross-testrun state**

**pytest Documentation, Release 3.6**

Now, if you run with the --ff option, all tests will be run but the first previous failures will be executed first (as can be seen from the series of FF and dots):

$ pytest --ff

=========================== 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 50 items

run-last-failure: rerun previous 2 failures first

|  |  |
| --- | --- |
| test\_50.py FF................................................ | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_num[17] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

i = 17

@pytest.mark.parametrize("i", range(50))

def test\_num(i):

if i in (17, 25):

* pytest.fail("bad luck")

EFailed: bad luck

test\_50.py:6: Failed

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_num[25] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

i = 25

@pytest.mark.parametrize("i", range(50))

def test\_num(i):

if i in (17, 25):

* pytest.fail("bad luck")

EFailed: bad luck

test\_50.py:6: Failed

=================== 2 failed, 48 passed in 0.12 seconds ====================

New --nf, --new-first options: run new tests first followed by the rest of the tests, in both cases tests are also sorted by the file modified time, with more recent files coming first.

**Behavior when no tests failed in the last run**

When no tests failed in the last run, or when no cached lastfailed data was found, pytest can be configured either to run all of the tests or no tests, using the --last-failed-no-failures option, which takes one of the following values:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| pytest | --last-failed-no-failures | all | # | run | all tests (default behavior) |
| pytest | --last-failed-no-failures | none | # | run | no tests and exit |
|  |  |  |  |  |  |

**The new config.cache object**

Plugins or conftest.py support code can get a cached value using the pytest config object. Here is a basic example plugin which implements a [pytest fixtures: explicit, modular, scalable](#page31) which re-uses previously created state across pytest invocations:

**14.3. Behavior when no tests failed in the last run** **79**

**pytest Documentation, Release 3.6**

* content of test\_caching.py **import pytest**

**import time**

**@pytest**.fixture

**def** mydata(request):

val = request.config.cache.get("example/value", **None**)

**if** val **is None**:

time.sleep(9\*0.6) # expensive computation :)

val = 42

request.config.cache.set("example/value", val)

**return** val

**def** test\_function(mydata):

**assert** mydata==23

If you run this command once, it will take a while because of the sleep:

$ pytest -q

F [100%]

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_function \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

mydata = 42

def test\_function(mydata):

* assert mydata == 23

Eassert 42 == 23

test\_caching.py:14: AssertionError

1 failed in 0.12 seconds

If you run it a second time the value will be retrieved from the cache and this will be quick:

$ pytest -q

F [100%]

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_function \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

mydata = 42

def test\_function(mydata):

* assert mydata == 23

Eassert 42 == 23

test\_caching.py:14: AssertionError

1 failed in 0.12 seconds

See the [config.cache](#page127) for more details.

**Inspecting Cache content**

You can always peek at the content of the cache using the --cache-show command line option:

$ pytest --cache-show

=========================== test session starts ============================

**80** **Chapter 14. Cache: working with cross-testrun state**

**pytest Documentation, Release 3.6**

platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y rootdir: $REGENDOC\_TMPDIR, inifile: cachedir: $REGENDOC\_TMPDIR/.pytest\_cache

------------------------------- cache values -------------------------------

cache/lastfailed contains:

{'test\_caching.py::test\_function': True}

cache/nodeids contains:

['test\_caching.py::test\_function']

example/value contains:

42

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

**Clearing Cache content**

You can instruct pytest to clear all cache files and values by adding the --cache-clear option like this:

pytest --cache-clear

This is recommended for invocations from Continuous Integration servers where isolation and correctness is more important than speed.

**14.6. Clearing Cache content** **81**

**pytest Documentation, Release 3.6**

**82** **Chapter 14. Cache: working with cross-testrun state**

**CHAPTER 15**

**unittest.TestCase Support**

pytest supports running Python unittest-based tests out of the box. It’s meant for leveraging existing unittest-based test suites to use pytest as a test runner and also allow to incrementally adapt the test suite to take full advantage of pytest’s features.

To run an existing unittest-style test suite using pytest, type:

pytest tests

pytest will automatically collect unittest.TestCase subclasses and their test methods in test\_\*.py or \*\_test.py files.

Almost all unittest features are supported:

* @unittest.skip style decorators;
* setUp/tearDown;
* setUpClass/tearDownClass();

Up to this point pytest does not have support for the following features:

* [load\_tests protocol](https://docs.python.org/3/library/unittest.html#load-tests-protocol);
* [setUpModule/tearDownModule](https://docs.python.org/3/library/unittest.html#setupmodule-and-teardownmodule);
* [subtests](https://docs.python.org/3/library/unittest.html#distinguishing-test-iterations-using-subtests);

**Benefits out of the box**

By running your test suite with pytest you can make use of several features, in most cases without having to modify existing code:

* Obtain [more informative tracebacks](#page175);
* [stdout and stderr](#page59) capturing;
* [Test selection options](#page16) using -k and -m flags;
* [Stopping after the first (or N) failures](#page15);
* [–pdb](#page17) command-line option for debugging on test failures (see [note](#page94) below);
* Distribute tests to multiple CPUs using the [pytest-xdist](https://pypi.org/project/pytest-xdist/) plugin;
* Use [plain assert-statements](#page25) instead of self.assert\* functions ([unittest2pytest](https://pypi.org/project/unittest2pytest/) is immensely helpful in this);

**83**

**pytest Documentation, Release 3.6**

**pytest features in unittest.TestCase subclasses**

The following pytest features work in unittest.TestCase subclasses:

* [Marks](#page51): [skip](#page73), [skipif](#page73) , [xfail](#page76);
* [Auto-use fixtures](#page92);

The following pytest features do not work, and probably never will due to different design philosophies:

* [Fixtures](#page31) (except for autouse fixtures, see [below](#page92));
* [Parametrization](#page81);
* [Custom hooks](#page101);

Third party plugins may or may not work well, depending on the plugin and the test suite.

**Mixing pytest fixtures into unittest.TestCase subclasses using marks**

Running your unittest with pytest allows you to use its [fixture mechanism](#page31) with unittest.TestCase style tests. Assuming you have at least skimmed the pytest fixture features, let’s jump-start into an example that integrates a pytest db\_class fixture, setting up a class-cached database object, and then reference it from a unittest-style test:

* content of conftest.py
* we define a fixture function below and it will be "used" by
* referencing its name from tests

**import pytest**

**@pytest**.fixture(scope="class")

**def** db\_class(request):

**class DummyDB**(object):

**pass**

* set a class attribute on the invoking test context request.cls.db = DummyDB()

This defines a fixture function db\_class which - if used - is called once for each test class and which sets the class-level db attribute to a DummyDB instance. The fixture function achieves this by receiving a special request object which gives access to [the requesting test context](#page37) such as the cls attribute, denoting the class from which the fixture is used. This architecture de-couples fixture writing from actual test code and allows re-use of the fixture by a minimal reference, the fixture name. So let’s write an actual unittest.TestCase class using our fixture definition:

# content of test\_unittest\_db.py

**import unittest**

**import pytest**

**@pytest**.mark.usefixtures("db\_class")

**class MyTest**(unittest.TestCase):

**def** test\_method1(self):

**assert** hasattr(self,"db")

**assert** 0,self.db # fail for demo purposes

**84** **Chapter 15. unittest.TestCase Support**

**pytest Documentation, Release 3.6**

**def** test\_method2(self):

**assert** 0,self.db # fail for demo purposes

The @pytest.mark.usefixtures("db\_class") class-decorator makes sure that the pytest fixture function db\_class is called once per class. Due to the deliberately failing assert statements, we can take a look at the self.db values in the traceback:

$ pytest test\_unittest\_db.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\_unittest\_db.py FF | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ MyTest.test\_method1 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <test\_unittest\_db.MyTest testMethod=test\_method1>

def test\_method1(self):

assert hasattr(self, "db")

> assert 0, self.db # fail for demo purposes

* AssertionError: <conftest.db\_class.<locals>.DummyDB object at 0xdeadbeef>
* assert 0

test\_unittest\_db.py:9: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ MyTest.test\_method2 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <test\_unittest\_db.MyTest testMethod=test\_method2>

def test\_method2(self):

> assert 0, self.db # fail for demo purposes

* AssertionError: <conftest.db\_class.<locals>.DummyDB object at 0xdeadbeef>
* assert 0

test\_unittest\_db.py:12: AssertionError

========================= 2 failed in 0.12 seconds =========================

This default pytest traceback shows that the two test methods share the same self.db instance which was our intention when writing the class-scoped fixture function above.

**Using autouse fixtures and accessing other fixtures**

Although it’s usually better to explicitly declare use of fixtures you need for a given test, you may sometimes want to have fixtures that are automatically used in a given context. After all, the traditional style of unittest-setup mandates the use of this implicit fixture writing and chances are, you are used to it or like it.

You can flag fixture functions with @pytest.fixture(autouse=True) and define the fixture function in the context where you want it used. Let’s look at an initdir fixture which makes all test methods of a TestCase class execute in a temporary directory with a pre-initialized samplefile.ini. Our initdir fixture itself uses the pytest builtin [tmpdir](#page57) fixture to delegate the creation of a per-test temporary directory:

* content of test\_unittest\_cleandir.py **import pytest**

**15.4. Using autouse fixtures and accessing other fixtures** **85**

**pytest Documentation, Release 3.6**

**import unittest**

**class MyTest**(unittest.TestCase):

**@pytest**.fixture(autouse=**True**)

**def** initdir(self, tmpdir):

tmpdir.chdir() # change to pytest-provided temporary directory tmpdir.join("samplefile.ini").write("# testdata")

**def** test\_method(self):

**with** open("samplefile.ini") **as** f:

s = f.read()

**assert** "testdata" **in** s

Due to the autouse flag the initdir fixture function will be used for all methods of the class where it is de-fined. This is a shortcut for using a @pytest.mark.usefixtures("initdir") marker on the class like in the previous example.

Running this test module ...:

|  |  |  |
| --- | --- | --- |
| $ pytest -q | | test\_unittest\_cleandir.py |
| . |  | [100%] |
| 1 | passed in | 0.12 seconds |
|  |  |  |

... gives us one passed test because the initdir fixture function was executed ahead of the test\_method.

Note: unittest.TestCase methods cannot directly receive fixture arguments as implementing that is likely to inflict on the ability to run general unittest.TestCase test suites.

The above usefixtures and autouse examples should help to mix in pytest fixtures into unittest suites.

You can also gradually move away from subclassing from unittest.TestCase to plain asserts and then start to benefit from the full pytest feature set step by step.

Note: Running tests from unittest.TestCase subclasses with --pdb will disable tearDown and cleanup methods for the case that an Exception occurs. This allows proper post mortem debugging for all applications which have significant logic in their tearDown machinery. However, supporting this feature has the following side effect: If people overwrite unittest.TestCase \_\_call\_\_ or run, they need to to overwrite debug in the same way (this is also true for standard unittest).

Note: Due to architectural differences between the two frameworks, setup and teardown for unittest-based tests is performed during the call phase of testing instead of in pytest‘s standard setup and teardown stages. This can be important to understand in some situations, particularly when reasoning about errors. For example, if a unittest-based suite exhibits errors during setup, pytest will report no errors during its setup phase and will instead raise the error during call.

**86** **Chapter 15. unittest.TestCase Support**

**CHAPTER 16**

**Running tests written for nose**

pytest has basic support for running tests written for [nose](https://nose.readthedocs.io/en/latest/).

**Usage**

After [Install pytest](#page11) type:

python setup.py develop # make sure tests can import our package pytest # instead of 'nosetests'

and you should be able to run your nose style tests and make use of pytest’s capabilities.

**Supported nose Idioms**

* setup and teardown at module/class/method level
* SkipTest exceptions and markers
* setup/teardown decorators
* yield-based tests and their setup (considered deprecated as of pytest 3.0)
* \_\_test\_\_ attribute on modules/classes/functions
* general usage of nose utilities

**Unsupported idioms / known issues**

* unittest-style setUp,tearDown,setUpClass,tearDownClass are recognized only on

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| unittest.TestCase classes but not on plain | classes. | nose | supports | these | methods also | |
| on plain classes but pytest deliberately does not. | As | nose and | pytest | already | both | support |
| setup\_class,teardown\_class,setup\_method,teardown\_method it | | | | doesn’t | seem | useful |

to duplicate the unittest-API like nose does. If you however rather think pytest should support the unittest-spelling on plain classes please post [to this issue](https://github.com/pytest-dev/pytest/issues/377/).

* nose imports test modules with the same import path (e.g. tests.test\_mod) but different file system paths (e.g. tests/test\_mode.py and other/tests/test\_mode.py) by extending sys.path/import seman-tics. pytest does not do that but there is discussion in [#268](https://github.com/pytest-dev/pytest/issues/268) for adding some support. Note that [nose2 choose to](https://nose2.readthedocs.io/en/latest/differences.html#test-discovery-and-loading) [avoid this sys.path/import hackery](https://nose2.readthedocs.io/en/latest/differences.html#test-discovery-and-loading).

**87**

**pytest Documentation, Release 3.6**

If you place a conftest.py file in the root directory of your project (as determined by pytest) pytest will run tests “nose style” against the code below that directory by adding it to your sys.path instead of running against your installed code.

You may find yourself wanting to do this if you ran python setup.py install to set up your project, as opposed to python setup.py develop or any of the package manager equivalents. Installing with develop in a virtual environment like tox is recommended over this pattern.

* nose-style doctests are not collected and executed correctly, also doctest fixtures don’t work.
* no nose-configuration is recognized.
* yield-based methods don’t support setup properly because the setup method is always called in the same class instance. There are no plans to fix this currently because yield-tests are deprecated in pytest 3.0, with pytest.mark.parametrize being the recommended alternative.

**88** **Chapter 16. Running tests written for nose**

**CHAPTER 17**

**classic xunit-style setup**

This section describes a classic and popular way how you can implement fixtures (setup and teardown test state) on a per-module/class/function basis.

Note: While these setup/teardown methods are simple and familiar to those coming from a unittest or nose background, you may also consider using pytest’s more powerful [fixture mechanism](#page31) which leverages the concept of dependency injection, allowing for a more modular and more scalable approach for managing test state, especially for larger projects and for functional testing. You can mix both fixture mechanisms in the same file but test methods of unittest.TestCase subclasses cannot receive fixture arguments.

**Module level setup/teardown**

If you have multiple test functions and test classes in a single module you can optionally implement the following fixture methods which will usually be called once for all the functions:

**def** setup\_module(module):

""" setup any state specific to the execution of the given module."""

**def** teardown\_module(module):

* teardown any state that was previously setup with a setup\_module method.

As of pytest-3.0, the module parameter is optional.

**Class level setup/teardown**

Similarly, the following methods are called at class level before and after all test methods of the class are called:

**@classmethod**

**def** setup\_class(cls):

* setup any state specific to the execution of the given class (which usually contains tests).

**@classmethod**

**def** teardown\_class(cls):

**89**

**pytest Documentation, Release 3.6**

* teardown any state that was previously setup with a call to setup\_class.

**Method and function level setup/teardown**

Similarly, the following methods are called around each method invocation:

**def** setup\_method(self, method):

* setup any state tied to the execution of the given method in a class. setup\_method is invoked for every test method of a class.

**def** teardown\_method(self, method):

* teardown any state that was previously setup with a setup\_method call.

As of pytest-3.0, the method parameter is optional.

If you would rather define test functions directly at module level you can also use the following functions to implement fixtures:

**def** setup\_function(function):

* setup any state tied to the execution of the given function. Invoked for every test function in the module.

**def** teardown\_function(function):

* teardown any state that was previously setup with a setup\_function call.

As of pytest-3.0, the function parameter is optional.

Remarks:

* It is possible for setup/teardown pairs to be invoked multiple times per testing process.
* teardown functions are not called if the corresponding setup function existed and failed/was skipped.

**90** **Chapter 17. classic xunit-style setup**

**CHAPTER 18**

**Installing and Using plugins**

This section talks about installing and using third party plugins. For writing your own plugins, please refer to [Writing](#page101) [plugins](#page101).

Installing a third party plugin can be easily done with pip:

pip install pytest-NAME

pip uninstall pytest-NAME

If a plugin is installed, pytest automatically finds and integrates it, there is no need to activate it.

Here is a little annotated list for some popular plugins:

* [pytest-django](https://pypi.org/project/pytest-django/): write tests for [django](https://www.djangoproject.com/) apps, using pytest integration.
* [pytest-twisted](https://pypi.org/project/pytest-twisted/): write tests for [twisted](http://twistedmatrix.com) apps, starting a reactor and processing deferreds from test functions.
* [pytest-cov](https://pypi.org/project/pytest-cov/): coverage reporting, compatible with distributed testing
* [pytest-xdist](https://pypi.org/project/pytest-xdist/): to distribute tests to CPUs and remote hosts, to run in boxed mode which allows to survive seg-mentation faults, to run in looponfailing mode, automatically re-running failing tests on file changes.
* [pytest-instafail](https://pypi.org/project/pytest-instafail/): to report failures while the test run is happening.
* [pytest-bdd](https://pypi.org/project/pytest-bdd/) and [pytest-konira](https://pypi.org/project/pytest-konira/) to write tests using behaviour-driven testing.
* [pytest-timeout](https://pypi.org/project/pytest-timeout/): to timeout tests based on function marks or global definitions.
* [pytest-pep8](https://pypi.org/project/pytest-pep8/): a --pep8 option to enable PEP8 compliance checking.
* [pytest-flakes](https://pypi.org/project/pytest-flakes/): check source code with pyflakes.
* [oejskit](https://pypi.org/project/oejskit/): a plugin to run javascript unittests in live browsers.

To see a complete list of all plugins with their latest testing status against different pytest and Python versions, please visit [plugincompat](http://plugincompat.herokuapp.com/).

You may also discover more plugins through a [pytest- pypi.python.org search](https://pypi.org/search/?q=pytest-).

**Requiring/Loading plugins in a test module or conftest file**

You can require plugins in a test module or a conftest file like this:

pytest\_plugins = "myapp.testsupport.myplugin",

When the test module or conftest plugin is loaded the specified plugins will be loaded as well.

**91**

**pytest Documentation, Release 3.6**

pytest\_plugins = “myapp.testsupport.myplugin”

which will import the specified module as a pytest plugin.

Note: Requiring plugins using a pytest\_plugins variable in non-root conftest.py files is deprecated. See [full explanation](#page104) in the Writing plugins section.

**Finding out which plugins are active**

If you want to find out which plugins are active in your environment you can type:

pytest --trace-config

and will get an extended test header which shows activated plugins and their names. It will also print local plugins aka [conftest.py](#page102) files when they are loaded.

**Deactivating / unregistering a plugin by name**

You can prevent plugins from loading or unregister them:

pytest -p no:NAME

This means that any subsequent try to activate/load the named plugin will not work.

If you want to unconditionally disable a plugin for a project, you can add this option to your pytest.ini file:

**[pytest]**

addopts = -p no:NAME

Alternatively to disable it only in certain environments (for example in a CI server), you can set PYTEST\_ADDOPTS environment variable to -p no:name.

See [Finding out which plugins are active](#page100) for how to obtain the name of a plugin.

**92** **Chapter 18. Installing and Using plugins**

**CHAPTER 19**

**Writing plugins**

It is easy to implement [local conftest plugins](#page102) for your own project or [pip-installable plugins](#page103) that can be used throughout many projects, including third party projects. Please refer to [Installing and Using plugins](#page99) if you only want to use but not write plugins.

A plugin contains one or multiple hook functions. [Writing hooks](#page107) explains the basics and details of how you can write a hook function yourself. pytest implements all aspects of configuration, collection, running and reporting by calling [well specified hooks](#page136) of the following plugins:

* builtin plugins: loaded from pytest’s internal \_pytest directory.
* [external plugins](#page99): modules discovered through [setuptools entry points](#page103)
* [conftest.py plugins](#page102): modules auto-discovered in test directories

In principle, each hook call is a 1:N Python function call where N is the number of registered implementation functions for a given specification. All specifications and implementations follow the pytest\_ prefix naming convention, making them easy to distinguish and find.

**Plugin discovery order at tool startup**

pytest loads plugin modules at tool startup in the following way:

* by loading all builtin plugins
* by loading all plugins registered through [setuptools entry points](#page103).
* by pre-scanning the command line for the -p name option and loading the specified plugin before actual command line parsing.
* by loading all conftest.py files as inferred by the command line invocation:

– if no test paths are specified use current dir as a test path

– if exists, load conftest.py and test\*/conftest.py relative to the directory part of the first test path.

Note that pytest does not find conftest.py files in deeper nested sub directories at tool startup. It is usually a good idea to keep your conftest.py file in the top level test or project root directory.

• by recursively loading all plugins specified by the pytest\_plugins variable in conftest.py files

**93**

**pytest Documentation, Release 3.6**

**conftest.py: local per-directory plugins**

Local conftest.py plugins contain directory-specific hook implementations. Hook Session and test running activi-ties will invoke all hooks defined in conftest.py files closer to the root of the filesystem. Example of implementing the pytest\_runtest\_setup hook so that is called for tests in the a sub directory but not for other directories:

a/conftest.py:

**def** pytest\_runtest\_setup(item):

* called for running each test in 'a' directory print ("setting up", item)

a/test\_sub.py:

**def** test\_sub():

**pass**

test\_flat.py:

**def** test\_flat():

**pass**

Here is how you might run it:

pytest test\_flat.py --capture=no # will not show "setting up"

pytest a/test\_sub.py --capture=no # will show "setting up"

Note: If you have conftest.py files which do not reside in a python package directory (i.e. one containing an \_\_init\_\_.py) then “import conftest” can be ambiguous because there might be other conftest.py files as well on your PYTHONPATH or sys.path. It is thus good practice for projects to either put conftest.py under a package scope or to never import anything from a conftest.py file.

See also: [pytest import mechanisms and sys.path/PYTHONPATH](#page169).

**Writing your own plugin**

If you want to write a plugin, there are many real-life examples you can copy from:

* a custom collection example plugin: [A basic example for specifying tests in Yaml files](#page226)
* builtin plugins which provide pytest’s own functionality
* many [external plugins](http://plugincompat.herokuapp.com) providing additional features

All of these plugins implement [hooks](#page136) and/or [fixtures](#page31) to extend and add functionality.

Note: Make sure to check out the excellent [cookiecutter-pytest-plugin](https://github.com/pytest-dev/cookiecutter-pytest-plugin) project, which is a [cookiecutter template](https://github.com/audreyr/cookiecutter) for authoring plugins.

The template provides an excellent starting point with a working plugin, tests running with tox, a comprehensive README file as well as a pre-configured entry-point.

Also consider [contributing your plugin to pytest-dev](#page240) once it has some happy users other than yourself.

**94** **Chapter 19. Writing plugins**

**pytest Documentation, Release 3.6**

**Making your plugin installable by others**

If you want to make your plugin externally available, you may define a so-called entry point for your distribution so that pytest finds your plugin module. Entry points are a feature that is provided by [setuptools](https://pypi.org/project/setuptools/). pytest looks up the pytest11 entrypoint to discover its plugins and you can thus make your plugin available by defining it in your setuptools-invocation:

* sample ./setup.py file **from setuptools import** setup

setup(

name="myproject",

packages=["myproject"],

# the following makes a plugin available to pytest

entry\_points={"pytest11": ["name\_of\_plugin = myproject.pluginmodule"]},

* custom PyPI classifier for pytest plugins classifiers=["Framework :: Pytest"],

)

If a package is installed this way, pytest will load myproject.pluginmodule as a plugin which can define [hooks](#page136).

Note: Make sure to include Framework :: Pytest in your list of [PyPI classifiers](https://python-packaging-user-guide.readthedocs.io/distributing/#classifiers) to make it easy for users to find your plugin.

**Assertion Rewriting**

One of the main features of pytest is the use of plain assert statements and the detailed introspection of expressions upon assertion failures. This is provided by “assertion rewriting” which modifies the parsed AST before it gets com-piled to bytecode. This is done via a [PEP 302](https://www.python.org/dev/peps/pep-0302) import hook which gets installed early on when pytest starts up and will perform this rewriting when modules get imported. However since we do not want to test different bytecode then you will run in production this hook only rewrites test modules themselves as well as any modules which are part of plugins. Any other imported module will not be rewritten and normal assertion behaviour will happen.

If you have assertion helpers in other modules where you would need assertion rewriting to be enabled you need to ask pytest explicitly to rewrite this module before it gets imported.

**register\_assert\_rewrite**(\*names)

Register one or more module names to be rewritten on import.

This function will make sure that this module or all modules inside the package will get their assert statements rewritten. Thus you should make sure to call this before the module is actually imported, usually in your \_\_init\_\_.py if you are a plugin using a package.

Raises [**TypeError**](https://docs.python.org/3/library/exceptions.html#TypeError) – if the given module names are not strings.

This is especially important when you write a pytest plugin which is created using a package. The import hook only treats conftest.py files and any modules which are listed in the pytest11 entrypoint as plugins. As an example consider the following package:

pytest\_foo/\_\_init\_\_.py

pytest\_foo/plugin.py

pytest\_foo/helper.py

**19.4. Making your plugin installable by others** **95**

**pytest Documentation, Release 3.6**

With the following typical setup.py extract:

setup(..., entry\_points={"pytest11": ["foo = pytest\_foo.plugin"]}, ...)

In this case only pytest\_foo/plugin.py will be rewritten. If the helper module also contains assert state-ments which need to be rewritten it needs to be marked as such, before it gets imported. This is easiest by marking it for rewriting inside the \_\_init\_\_.py module, which will always be imported first when a module inside a package is imported. This way plugin.py can still import helper.py normally. The contents of pytest\_foo/\_\_init\_\_.py will then need to look like this:

**import pytest**

pytest.register\_assert\_rewrite("pytest\_foo.helper")

**Requiring/Loading plugins in a test module or conftest file**

You can require plugins in a test module or a conftest.py file like this:

pytest\_plugins = ["name1", "name2"]

When the test module or conftest plugin is loaded the specified plugins will be loaded as well. Any module can be blessed as a plugin, including internal application modules:

pytest\_plugins = "myapp.testsupport.myplugin"

pytest\_plugins variables are processed recursively, so note that in the example above if myapp.testsupport.myplugin also declares pytest\_plugins, the contents of the variable will also be loaded as plugins, and so on.

Note: Requiring plugins using a pytest\_plugins variable in non-root conftest.py files is deprecated.

This is important because conftest.py files implement per-directory hook implementations, but once a plugin is imported, it will affect the entire directory tree. In order to avoid confusion, defining pytest\_plugins in any conftest.py file which is not located in the tests root directory is deprecated, and will raise a warning.

This mechanism makes it easy to share fixtures within applications or even external applications without the need to create external plugins using the setuptools‘s entry point technique.

Plugins imported by pytest\_plugins will also automatically be marked for assertion rewriting (see [pytest.register\_assert\_rewrite()](#page122)). However for this to have any effect the module must not be imported already; if it was already imported at the time the pytest\_plugins statement is processed, a warning will result and assertions inside the plugin will not be rewritten. To fix this you can either call [pytest.register\_assert\_rewrite()](#page122) yourself before the module is imported, or you can arrange the code to delay the importing until after the plugin is registered.

**Accessing another plugin by name**

If a plugin wants to collaborate with code from another plugin it can obtain a reference through the plugin manager like this:

**96** **Chapter 19. Writing plugins**

**pytest Documentation, Release 3.6**

plugin = config.pluginmanager.get\_plugin("name\_of\_plugin")

If you want to look at the names of existing plugins, use the --trace-config option.

**Testing plugins**

pytest comes with a plugin named pytester that helps you write tests for your plugin code. The plugin is disabled by default, so you will have to enable it before you can use it.

You can do so by adding the following line to a conftest.py file in your testing directory:

# content of conftest.py

pytest\_plugins = ["pytester"]

Alternatively you can invoke pytest with the -p pytester command line option.

This will allow you to use the [testdir](#page134) fixture for testing your plugin code.

Let’s demonstrate what you can do with the plugin with an example. Imagine we developed a plugin that provides a fixture hello which yields a function and we can invoke this function with one optional parameter. It will return a string value of Hello World! if we do not supply a value or Hello {value}! if we do supply a string value.

# -\*- coding: utf-8 -\*-

**import pytest**

**def** pytest\_addoption(parser):

group = parser.getgroup("helloworld")

group.addoption(

"--name",

action="store",

dest="name",

default="World",

help='Default "name" for hello().',

)

**@pytest.fixture**

**def** hello(request):

name = request.config.getoption("name")

**def** \_hello(name=None):

**if not** name:

name = request.config.getoption("name")

**return** "Hello {name}!".format(name=name)

**return** \_hello

Now the testdir fixture provides a convenient API for creating temporary conftest.py files and test files. It also allows us to run the tests and return a result object, with which we can assert the tests’ outcomes.

**def** test\_hello(testdir):

"""Make sure that our plugin works."""

**19.8. Testing plugins** **97**

**pytest Documentation, Release 3.6**

* create a temporary conftest.py file testdir.makeconftest(

"""

import pytest

@pytest.fixture(params=[

"Brianna",

"Andreas",

"Floris",

])

def name(request):

return request.param

"""

)

* create a temporary pytest test file testdir.makepyfile(

"""

def test\_hello\_default(hello): assert hello() == "Hello World!"

def test\_hello\_name(hello, name):

assert hello(name) == "Hello {0}!".format(name)

"""

)

* run all tests with pytest result = testdir.runpytest()
* check that all 4 tests passed result.assert\_outcomes(passed=4)

For more information about the result object that runpytest() returns, and the methods that it provides please check out the [RunResult](#page134) documentation.

**98** **Chapter 19. Writing plugins**

**CHAPTER 20**

**Writing hook functions**

**hook function validation and execution**

pytest calls hook functions from registered plugins for any given hook specification. Let’s look at a typical hook function for the pytest\_collection\_modifyitems(session,config,items) hook which pytest calls after collection of all test items is completed.

When we implement a pytest\_collection\_modifyitems function in our plugin pytest will during registra-tion verify that you use argument names which match the specification and bail out if not.

Let’s look at a possible implementation:

**def** pytest\_collection\_modifyitems(config, items):

* called after collection is completed
* you can modify the ``items`` list

...

Here, pytest will pass in config (the pytest config object) and items (the list of collected test items) but will not pass in the session argument because we didn’t list it in the function signature. This dynamic “pruning” of arguments allows pytest to be “future-compatible”: we can introduce new hook named parameters without breaking the signatures of existing hook implementations. It is one of the reasons for the general long-lived compatibility of pytest plugins.

Note that hook functions other than pytest\_runtest\_\* are not allowed to raise exceptions. Doing so will break the pytest run.

**firstresult: stop at first non-None result**

Most calls to pytest hooks result in a list of results which contains all non-None results of the called hook functions.

Some hook specifications use the firstresult=True option so that the hook call only executes until the first of N registered functions returns a non-None result which is then taken as result of the overall hook call. The remaining hook functions will not be called in this case.

**hookwrapper: executing around other hooks**

New in version 2.7.

**99**

**pytest Documentation, Release 3.6**

pytest plugins can implement hook wrappers which wrap the execution of other hook implementations. A hook wrapper is a generator function which yields exactly once. When pytest invokes hooks it first executes hook wrappers and passes the same arguments as to the regular hooks.

At the yield point of the hook wrapper pytest will execute the next hook implementations and return their result to the yield point in the form of a [Result](#page154) instance which encapsulates a result or exception info. The yield point itself will thus typically not raise exceptions (unless there are bugs).

Here is an example definition of a hook wrapper:

**import pytest**

**@pytest**.hookimpl(hookwrapper=**True**)

**def** pytest\_pyfunc\_call(pyfuncitem):

do\_something\_before\_next\_hook\_executes()

outcome = **yield**

# outcome.excinfo may be None or a (cls, val, tb) tuple

res = outcome.get\_result() # will raise if outcome was exception

post\_process\_result(res)

outcome.force\_result(new\_res) # to override the return value to the plugin system

Note that hook wrappers don’t return results themselves, they merely perform tracing or other side effects around the actual hook implementations. If the result of the underlying hook is a mutable object, they may modify that result but it’s probably better to avoid it.

For more information, consult the [pluggy documentation](http://pluggy.readthedocs.io/en/latest/#wrappers).

**Hook function ordering / call example**

For any given hook specification there may be more than one implementation and we thus generally view hook execution as a 1:N function call where N is the number of registered functions. There are ways to influence if a hook implementation comes before or after others, i.e. the position in the N-sized list of functions:

# Plugin 1

**@pytest.hookimpl**(tryfirst=True)

**def** pytest\_collection\_modifyitems(items):

* + will execute as early as possible

...

* Plugin 2

**@pytest.hookimpl**(trylast=True)

**def** pytest\_collection\_modifyitems(items):

* + will execute as late as possible

...

* Plugin 3

**@pytest.hookimpl**(hookwrapper=True)

**def** pytest\_collection\_modifyitems(items):

* will execute even before the tryfirst one above! outcome = **yield**
* will execute after all non-hookwrappers executed

**100** **Chapter 20. Writing hook functions**

**pytest\_addhooks**(pluginmanager)called at plugin registration

**pytest Documentation, Release 3.6**

Here is the order of execution:

1. Plugin3’s pytest\_collection\_modifyitems called until the yield point because it is a hook wrapper.
2. Plugin1’s pytest\_collection\_modifyitems is called because it is marked with tryfirst=True.
3. Plugin2’s pytest\_collection\_modifyitems is called because it is marked with trylast=True (but even without this mark it would come after Plugin1).
4. Plugin3’s pytest\_collection\_modifyitems then executing the code after the yield point. The yield receives a [Result](#page154) instance which encapsulates the result from calling the non-wrappers. Wrappers shall not modify the result.

It’s possible to use tryfirst and trylast also in conjunction with hookwrapper=True in which case it will influence the ordering of hookwrappers among each other.

**Declaring new hooks**

Plugins and conftest.py files may declare new hooks that can then be implemented by other plugins in order to alter behaviour or interact with the new plugin:

time to allow adding new hooks via a call to pluginmanager.add\_hookspecs(module\_or\_class,prefix).

Parameters **pluginmanager** ([\_pytest.config.PytestPluginManager](#page152)) – pytest plu-gin manager

Note: This hook is incompatible with hookwrapper=True.

Hooks are usually declared as do-nothing functions that contain only documentation describing when the hook will be called and what return values are expected.

For an example, see [newhooks.py](https://github.com/pytest-dev/pytest-xdist/blob/974bd566c599dc6a9ea291838c6f226197208b46/xdist/newhooks.py) from [xdist](https://github.com/pytest-dev/pytest-xdist).

**Optionally using hooks from 3rd party plugins**

Using new hooks from plugins as explained above might be a little tricky because of the standard [validation mecha-nism](#page107): if you depend on a plugin that is not installed, validation will fail and the error message will not make much sense to your users.

One approach is to defer the hook implementation to a new plugin instead of declaring the hook functions directly in your plugin module, for example:

# contents of myplugin.py

**class DeferPlugin**(object):

"""Simple plugin to defer pytest-xdist hook functions."""

**def** pytest\_testnodedown(self, node, error):

"""standard xdist hook function.

"""

**def** pytest\_configure(config):

**20.5. Declaring new hooks** **101**

**pytest Documentation, Release 3.6**

**if** config.pluginmanager.hasplugin('xdist'):

config.pluginmanager.register(DeferPlugin())

This has the added benefit of allowing you to conditionally install hooks depending on which plugins are installed.

**102** **Chapter 20. Writing hook functions**

**CHAPTER 21**

**Logging**

New in version 3.3.

Changed in version 3.4.

pytest captures log messages of level WARNING or above automatically and displays them in their own section for each failed test in the same manner as captured stdout and stderr.

Running without options:

pytest

Shows failed tests like so:

|  |  |  |  |
| --- | --- | --- | --- |
| ----------------------- | Captured | stdlog call ---------------------- |  |
|  |  |
| test\_reporting.py | 26 WARNING | text going to logger |  |
| ----------------------- | Captured | stdout call ---------------------- |  |
| text going to stdout |  |  |  |
| ----------------------- | Captured | stderr call ---------------------- |  |
| text going to stderr |  |  |  |
| ==================== | 2 failed **in** | 0.02 seconds ===================== |  |
|  |  |  |  |

By default each captured log message shows the module, line number, log level and message.

If desired the log and date format can be specified to anything that the logging module supports by passing specific formatting options:

pytest --log-format="%(asctime)s %(levelname)s %(message)s" \ --log-date-format="%Y-%m-%d %H:%M:%S"

Shows failed tests like so:

|  |  |
| --- | --- |
| ----------------------- Captured | stdlog call ---------------------- |
| 2010-04-10 14:48:44 WARNING text | going to logger |
| ----------------------- Captured | stdout call ---------------------- |
| text going to stdout |  |
| ----------------------- Captured | stderr call ---------------------- |
| text going to stderr |  |
| ==================== 2 failed **in** | 0.02 seconds ===================== |
|  |  |

These options can also be customized through pytest.ini file:

**[pytest]**

log\_format = %(asctime)s %(levelname)s %(message)s log\_date\_format = %Y-%m-%d %H:%M:%S

**103**

**pytest Documentation, Release 3.6**

Further it is possible to disable reporting of captured content (stdout, stderr and logs) on failed tests completely with:

pytest --show-capture=no

**caplog fixture**

Inside tests it is possible to change the log level for the captured log messages. This is supported by the caplog fixture:

**def** test\_foo(caplog):

caplog.set\_level(logging.INFO)

**pass**

By default the level is set on the root logger, however as a convenience it is also possible to set the log level of any logger:

**def** test\_foo(caplog):

caplog.set\_level(logging.CRITICAL, logger='root.baz')

**pass**

The log levels set are restored automatically at the end of the test.

It is also possible to use a context manager to temporarily change the log level inside a with block:

**def** test\_bar(caplog):

**with** caplog.at\_level(logging.INFO):

**pass**

Again, by default the level of the root logger is affected but the level of any logger can be changed instead with:

**def** test\_bar(caplog):

**with** caplog.at\_level(logging.CRITICAL, logger='root.baz'):

**pass**

Lastly all the logs sent to the logger during the test run are made available on the fixture in the form of both the logging.LogRecord instances and the final log text. This is useful for when you want to assert on the contents of a message:

**def** test\_baz(caplog):

func\_under\_test()

**for** record **in** caplog.records:

**assert** record.levelname!='CRITICAL'

**assert** 'wally' **not in** caplog.text

For all the available attributes of the log records see the logging.LogRecord class.

You can also resort to record\_tuples if all you want to do is to ensure, that certain messages have been logged under a given logger name with a given severity and message:

**def** test\_foo(caplog):

logging.getLogger().info('boo %s', 'arg')

**assert** caplog.record\_tuples==[

('root', logging.INFO, 'boo arg'),

]

**104** **Chapter 21. Logging**

**pytest Documentation, Release 3.6**

You can call caplog.clear() to reset the captured log records in a test:

**def** test\_something\_with\_clearing\_records(caplog): some\_method\_that\_creates\_log\_records() caplog.clear()

your\_test\_method()

**assert** ['Foo']==[rec.message **for** rec **in** caplog.records]

The caplog.records attribute contains records from the current stage only, so inside the setup phase it contains only setup logs, same with the call and teardown phases.

To access logs from other stages, use the caplog.get\_records(when) method. As an example, if you want to make sure that tests which use a certain fixture never log any warnings, you can inspect the records for the setup and call stages during teardown like so:

**@pytest.fixture**

**def** window(caplog):

window = create\_window()

**yield** window

**for** when **in** ("setup","call"):

messages = [

x.message **for** x **in** caplog.get\_records(when) **if** x.level == logging.WARNING

]

**if** messages:

pytest.fail(

"warning messages encountered during testing: {}".format(messages)

)

The full API is available at [\_pytest.logging.LogCaptureFixture](#page131).

**Live Logs**

By setting the log\_cli configuration option to true, pytest will output logging records as they are emitted directly into the console.

You can specify the logging level for which log records with equal or higher level are printed to the console by passing --log-cli-level. This setting accepts the logging level names as seen in python’s documentation or an integer as the logging level num.

Additionally, you can also specify --log-cli-format and --log-cli-date-format which mirror and de-fault to --log-format and --log-date-format if not provided, but are applied only to the console logging handler.

All of the CLI log options can also be set in the configuration INI file. The option names are:

* log\_cli\_level
* log\_cli\_format
* log\_cli\_date\_format

If you need to record the whole test suite logging calls to a file, you can pass --log-file=/path/to/log/file.

This log file is opened in write mode which means that it will be overwritten at each run tests session.

You can also specify the logging level for the log file by passing --log-file-level. This setting accepts the logging level names as seen in python’s documentation(ie, uppercased level names) or an integer as the logging level num.

**21.2. Live Logs** **105**

**pytest Documentation, Release 3.6**

Additionally, you can also specify --log-file-format and --log-file-date-format which are equal to --log-format and --log-date-format but are applied to the log file logging handler.

All of the log file options can also be set in the configuration INI file. The option names are:

* log\_file
* log\_file\_level
* log\_file\_format
* log\_file\_date\_format

**Release notes**

This feature was introduced as a drop-in replacement for the [pytest-catchlog](https://pypi.org/project/pytest-catchlog/) plugin and they conflict with each other. The backward compatibility API with pytest-capturelog has been dropped when this feature was introduced, so if for that reason you still need pytest-catchlog you can disable the internal feature by adding to your pytest.ini:

**[pytest]**

addopts=-p no:logging

**Incompatible changes in pytest 3.4**

This feature was introduced in 3.3 and some incompatible changes have been made in 3.4 after community feed-back:

* Log levels are no longer changed unless explicitly requested by the [log\_level](#page159) configuration or --log-level command-line options. This allows users to configure logger objects themselves.
* [Live Logs](#page113) is now disabled by default and can be enabled setting the log\_cli configuration option to true. When enabled, the verbosity is increased so logging for each test is visible.
* [Live Logs](#page113) are now sent to sys.stdout and no longer require the -s command-line option to work.

If you want to partially restore the logging behavior of version 3.3, you can add this options to your ini file:

**[pytest]**

log\_cli=true

log\_level=NOTSET

More details about the discussion that lead to this changes can be read in issue [#3013](https://github.com/pytest-dev/pytest/issues/3013).

**106** **Chapter 21. Logging**

**CHAPTER 22**

**Reference**

This page contains the full reference to pytest’s API.

* [Functions](#page117)

– [pytest.approx](#page117)

– [pytest.fail](#page119)

– [pytest.skip](#page119)

– [pytest.importorskip](#page120)

– [pytest.xfail](#page120)

– [pytest.exit](#page120)

– [pytest.main](#page120)

– [pytest.param](#page120)

– [pytest.raises](#page120)

– [pytest.deprecated\_call](#page122)

– [pytest.register\_assert\_rewrite](#page122)

– [pytest.warns](#page123)

* [Marks](#page123)

– [pytest.mark.filterwarnings](#page123)

– [pytest.mark.parametrize](#page124)

– [pytest.mark.skip](#page124)

– [pytest.mark.skipif](#page125)

– [pytest.mark.usefixtures](#page125)

– [pytest.mark.xfail](#page125)

– [custom marks](#page126)

* [Fixtures](#page126)

– [@pytest.fixture](#page126)

– [config.cache](#page127)

**107**

**pytest Documentation, Release 3.6**

– [capsys](#page127)

– [capsysbinary](#page128)

– [capfd](#page128)

– [capfdbinary](#page129)

– [doctest\_namespace](#page129)

– [request](#page129)

– [pytestconfig](#page131)

– [record\_property](#page131)

– [caplog](#page131)

– [monkeypatch](#page132)

– [testdir](#page134)

– [recwarn](#page135)

– [tmpdir](#page136)

– [tmpdir\_factory](#page136)

* [Hooks](#page136)

– [Bootstrapping hooks](#page136)

– [Initialization hooks](#page138)

– [Test running hooks](#page139)

– [Collection hooks](#page140)

– [Reporting hooks](#page141)

– [Debugging/Interaction hooks](#page142)

* [Objects](#page143)

– [CallInfo](#page143)

– [Class](#page143)

– [Collector](#page143)

– [Config](#page144)

– [ExceptionInfo](#page144)

– [FixtureDef](#page145)

– [FSCollector](#page145)

– [Function](#page145)

– [Item](#page146)

– [MarkDecorator](#page146)

– [MarkGenerator](#page147)

– [MarkInfo](#page147)

– [Mark](#page147)

– [Metafunc](#page148)

**108** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

– [Module](#page149)

– [Node](#page149)

– [Parser](#page150)

– [PluginManager](#page151)

– [PytestPluginManager](#page152)

– [Session](#page153)

– [TestReport](#page153)

– [\_Result](#page154)

* [Special Variables](#page154)

– [pytest\_plugins](#page155)

– [pytest\_mark](#page155)

– [PYTEST\_DONT\_REWRITE (module docstring)](#page155)

* [Environment Variables](#page155)

– [PYTEST\_ADDOPTS](#page155)

– [PYTEST\_DEBUG](#page155)

– [PYTEST\_PLUGINS](#page155)

– [PYTEST\_CURRENT\_TEST](#page156)

* [Configuration Options](#page156)

**Functions**

**pytest.approx**

**approx**(expected,rel=None,abs=None,nan\_ok=False)

Assert that two numbers (or two sets of numbers) are equal to each other within some tolerance.

Due to the [intricacies of floating-point arithmetic](https://docs.python.org/3/tutorial/floatingpoint.html), numbers that we would intuitively expect to be equal are not always so:

* 0.1 + 0.2 == 0.3 False

This problem is commonly encountered when writing tests, e.g. when making sure that floating-point values are what you expect them to be. One way to deal with this problem is to assert that two floating-point numbers are equal to within some appropriate tolerance:

* abs((0.1 + 0.2) - 0.3) < 1e-6

True

However, comparisons like this are tedious to write and difficult to understand. Furthermore, absolute compar-isons like the one above are usually discouraged because there’s no tolerance that works well for all situations. 1e-6 is good for numbers around 1, but too small for very big numbers and too big for very small ones. It’s better to express the tolerance as a fraction of the expected value, but relative comparisons like that are even more difficult to write correctly and concisely.

**22.1. Functions** **109**

**pytest Documentation, Release 3.6**

The approx class performs floating-point comparisons using a syntax that’s as intuitive as possible:

* **from pytest import** approx
* 0.1 + 0.2 == approx(0.3)

True

The same syntax also works for sequences of numbers:

* (0.1 + 0.2, 0.2 + 0.4) == approx((0.3, 0.6))

True

Dictionary values:

* {'a': 0.1 + 0.2, 'b': 0.2 + 0.4} == approx({'a': 0.3, 'b': 0.6})

True

numpy arrays:

* **import numpy as np**
* np.array([0.1, 0.2]) + np.array([0.2, 0.4]) == approx(np.array([0.3, 0.6]))

True

And for a numpy array against a scalar:

* **import numpy as np**
* np.array([0.1, 0.2]) + np.array([0.2, 0.1]) == approx(0.3)

True

By default, approx considers numbers within a relative tolerance of 1e-6 (i.e. one part in a million) of its expected value to be equal. This treatment would lead to surprising results if the expected value was 0.0, because nothing but 0.0 itself is relatively close to 0.0. To handle this case less surprisingly, approx also considers numbers within an absolute tolerance of 1e-12 of its expected value to be equal. Infinity and NaN are special cases. Infinity is only considered equal to itself, regardless of the relative tolerance. NaN is not considered equal to anything by default, but you can make it be equal to itself by setting the nan\_ok argument to True. (This is meant to facilitate comparing arrays that use NaN to mean “no data”.)

Both the relative and absolute tolerances can be changed by passing arguments to the approx constructor:

* 1.0001 == approx(1) False
* 1.0001 == approx(1, rel=1e-3)

True

* 1.0001 == approx(1, abs=1e-3)

True

If you specify abs but not rel, the comparison will not consider the relative tolerance at all. In other words, two numbers that are within the default relative tolerance of 1e-6 will still be considered unequal if they exceed the specified absolute tolerance. If you specify both abs and rel, the numbers will be considered equal if either tolerance is met:

* 1 + 1e-8 == approx(1)

True

* 1 + 1e-8 == approx(1, abs=1e-12) False
* 1 + 1e-8 == approx(1, rel=1e-6, abs=1e-12)

True

**110** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

If you’re thinking about using approx, then you might want to know how it compares to other good ways of comparing floating-point numbers. All of these algorithms are based on relative and absolute tolerances and should agree for the most part, but they do have meaningful differences:

•math.isclose(a,b,rel\_tol=1e-9,abs\_tol=0.0): True if the relative tolerance is met w.r.t. either a or b or if the absolute tolerance is met. Because the relative tolerance is calculated w.r.t. both a and b, this test is symmetric (i.e. neither a nor b is a “reference value”). You have to specify an absolute tolerance if you want to compare to 0.0 because there is no tolerance by default. Only available in python>=3.5. [More information...](https://docs.python.org/3/library/math.html#math.isclose)

•numpy.isclose(a,b,rtol=1e-5,atol=1e-8): True if the difference between a and b is less that the sum of the relative tolerance w.r.t. b and the absolute tolerance. Because the relative tolerance is only calculated w.r.t. b, this test is asymmetric and you can think of b as the reference value. Support for comparing sequences is provided by numpy.allclose. [More information...](http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.isclose.html)

•unittest.TestCase.assertAlmostEqual(a,b): True if a and b are within an absolute toler-ance of 1e-7. No relative tolerance is considered and the absolute tolerance cannot be changed, so this function is not appropriate for very large or very small numbers. Also, it’s only available in subclasses of unittest.TestCase and it’s ugly because it doesn’t follow PEP8. [More information...](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertAlmostEqual)

•a == pytest.approx(b,rel=1e-6,abs=1e-12): True if the relative tolerance is met w.r.t. b or if the absolute tolerance is met. Because the relative tolerance is only calculated w.r.t. b, this test is asymmetric and you can think of b as the reference value. In the special case that you explicitly specify an absolute tolerance but not a relative tolerance, only the absolute tolerance is considered.

Warning: Changed in version 3.2.

In order to avoid inconsistent behavior, TypeError is raised for >, >=, < and <= comparisons. The example below illustrates the problem:

**assert** approx(0.1)>0.1+1e-10 # calls approx(0.1).\_\_gt\_\_(0.1 + 1e-10)

**assert** 0.1+1e-10>approx(0.1) # calls approx(0.1).\_\_lt\_\_(0.1 + 1e-10)

In the second example one expects approx(0.1).\_\_le\_\_(0.1 + 1e-10) to be called. But instead, approx(0.1).\_\_lt\_\_(0.1 + 1e-10) is used to comparison. This is because the call hierarchy of rich comparisons follows a fixed behavior. [More information...](https://docs.python.org/3/reference/datamodel.html#object.__ge__)

**pytest.fail**

Tutorial: [Skip and xfail: dealing with tests that cannot succeed](#page73)

**fail**(msg=’‘,pytrace=True)

explicitly fail a currently-executing test with the given Message.

Parameters **pytrace** – if false the msg represents the full failure information and no python trace-back will be reported.

**pytest.skip**

**skip**(msg[,allow\_module\_level=False])

skip an executing test with the given message. Note: it’s usually better to use the pytest.mark.skipif marker to declare a test to be skipped under certain conditions like mismatching platforms or dependencies. See the pytest\_skipping plugin for details.

Parameters **allow\_module\_level** ([bool](https://docs.python.org/3/library/functions.html#bool)) – allows this function to be called at module level, skipping the rest of the module. Default to False.

**22.1. Functions** **111**

**pytest Documentation, Release 3.6**

**pytest.importorskip**

**importorskip**(modname,minversion=None)

return imported module if it has at least “minversion” as its \_\_version\_\_ attribute. If no minversion is specified the a skip is only triggered if the module can not be imported.

**pytest.xfail**

**xfail**(reason=’‘)

xfail an executing test or setup functions with the given reason.

**pytest.exit**

**exit**(msg)

exit testing process as if KeyboardInterrupt was triggered.

**pytest.main**

**main**(args=None,plugins=None)

return exit code, after performing an in-process test run.

Parameters

* **args** – list of command line arguments.
* **plugins** – list of plugin objects to be auto-registered during initialization.

**pytest.param**

**param**(\*values[,id][,marks])

Specify a parameter in [pytest.mark.parametrize](#page124) calls or [parametrized fixtures](#page41).

**@pytest.mark.parametrize**("test\_input,expected", [

("3+5", 8),

pytest.param("6\*9", 42, marks=pytest.mark.xfail),

])

**def** test\_eval(test\_input, expected):

**assert** eval(test\_input)==expected

Parameters

* **values** – variable args of the values of the parameter set, in order.
* **marks** – a single mark or a list of marks to be applied to this parameter set.
* **id** ([str](#page135))– the id to attribute to this parameter set.

**pytest.raises**

Tutorial: [Assertions about expected exceptions](#page26).

with **raises**(expected\_exception: Exception[, match ][, message ]) as excinfo

Assert that a code block/function call raises expected\_exception and raise a failure exception otherwise.

**112** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

Parameters

* **message** – if specified, provides a custom failure message if the exception is not raised
* **match** – if specified, asserts that the exception matches a text or regex

This helper produces a ExceptionInfo() object (see below).

You may use this function as a context manager:

* **with** raises(ZeroDivisionError):

**...**1/0

Changed in version 2.10.

In the context manager form you may use the keyword argument message to specify a custom failure message:

* **with** raises(ZeroDivisionError, message="Expecting ZeroDivisionError"):

**...pass**

Traceback (most recent call last):

...

Failed: Expecting ZeroDivisionError

Note: When using pytest.raises as a context manager, it’s worthwhile to note that normal context manager rules apply and that the exception raised must be the final line in the scope of the context manager. Lines of code after that, within the scope of the context manager will not be executed. For example:

* value = 15
* **with** raises(ValueError) **as** exc\_info:

**...if** value>10:

**...** **raise** ValueError("value must be <= 10")

**...** **assert** exc\_info.type==ValueError # this will not execute

Instead, the following approach must be taken (note the difference in scope):

* **with** raises(ValueError) **as** exc\_info:

**...if** value>10:

**...** **raise** ValueError("value must be <= 10")

**...**

* **assert** exc\_info.type==ValueError

Since version 3.1 you can use the keyword argument match to assert that the exception matches a text or regex:

* **with** raises(ValueError, match='must be 0 or None'):

**...** **raise** ValueError("value must be 0 or None")

* **with** raises(ValueError, match=r'must be \d+$'):

**...raise** ValueError("value must be 42")

Legacy forms

The forms below are fully supported but are discouraged for new code because the context manager form is regarded as more readable and less error-prone.

It is possible to specify a callable by passing a to-be-called lambda:

**22.1. Functions** **113**

**pytest Documentation, Release 3.6**

* raises(ZeroDivisionError, **lambda**: 1/0) <ExceptionInfo ...>

or you can specify an arbitrary callable with arguments:

* **def** f(x): **return** 1/x

**...**

* raises(ZeroDivisionError, f, 0) <ExceptionInfo ...>
* raises(ZeroDivisionError, f, x=0) <ExceptionInfo ...>

It is also possible to pass a string to be evaluated at runtime:

* raises(ZeroDivisionError, "f(0)") <ExceptionInfo ...>

The string will be evaluated using the same locals() and globals() at the moment of the raises call.

Consult the API of excinfo objects: [ExceptionInfo](#page144).

Note: Similar to caught exception objects in Python, explicitly clearing local references to returned ExceptionInfo objects can help the Python interpreter speed up its garbage collection.

Clearing those references breaks a reference cycle (ExceptionInfo –> caught exception –> frame stack raising the exception –> current frame stack –> local variables –> ExceptionInfo) which makes Python keep all objects referenced from that cycle (including all local variables in the current frame) alive until the next cyclic garbage collection run. See the official Python try statement documentation for more detailed information.

**pytest.deprecated\_call**

Tutorial: [Ensuring a function triggers a deprecation warning](#page67).

with **deprecated\_call**()

context manager that can be used to ensure a block of code triggers a DeprecationWarning or

PendingDeprecationWarning:

* **import warnings**
* **def** api\_call\_v2():

**...** warnings.warn('use v3 of this api', DeprecationWarning)

**...** **return** 200

* **with** deprecated\_call():

**...** **assert** api\_call\_v2()==200

deprecated\_call can also be used by passing a function and \*args and \*kwargs, in which case it will ensure calling func(\*args,\*\*kwargs) produces one of the warnings types above.

**pytest.register\_assert\_rewrite**

Tutorial: [Assertion Rewriting](#page103).

**114** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

**register\_assert\_rewrite**(\*names)

Register one or more module names to be rewritten on import.

This function will make sure that this module or all modules inside the package will get their assert statements rewritten. Thus you should make sure to call this before the module is actually imported, usually in your \_\_init\_\_.py if you are a plugin using a package.

Raises [**TypeError**](https://docs.python.org/3/library/exceptions.html#TypeError) – if the given module names are not strings.

**pytest.warns**

Tutorial: [Asserting warnings with the warns function](#page65)

with **warns**(expected\_warning: Exception[, match ])

Assert that code raises a particular class of warning.

Specifically, the parameter expected\_warning can be a warning class or sequence of warning classes, and the inside the with block must issue a warning of that class or classes.

This helper produces a list of warnings.WarningMessage objects, one for each warning raised.

This function can be used as a context manager, or any of the other ways pytest.raises can be used:

* **with** warns(RuntimeWarning):

**...** warnings.warn("my warning", RuntimeWarning)

In the context manager form you may use the keyword argument match to assert that the exception matches a text or regex:

* **with** warns(UserWarning, match='must be 0 or None'):

**...** warnings.warn("value must be 0 or None", UserWarning)

* **with** warns(UserWarning, match=r'must be \d+$'):

**...**warnings.warn("value must be 42", UserWarning)

* **with** warns(UserWarning, match=r'must be \d+$'):

**...** warnings.warn("this is not here",UserWarning)Traceback (most recent call last):

...

Failed: DID NOT WARN. No warnings of type ...UserWarning... was emitted...

**Marks**

Marks can be used apply meta data to test functions (but not fixtures), which can then be accessed by fixtures or plugins.

**pytest.mark.filterwarnings**

Tutorial: [@pytest.mark.filterwarnings](#page64).

Add warning filters to marked test items.

pytest.mark.**filterwarnings**(filter)

**22.2. Marks** **115**

**pytest Documentation, Release 3.6**

Parameters **filter** ([str](#page135)) – A warning specification string, which is composed of contents of the tuple (action,message,category,module,lineno) as specified in [The Warnings](https://docs.python.org/3/library/warnings.html#warning-filter) [filter](https://docs.python.org/3/library/warnings.html#warning-filter) section of the Python documentation, separated by ":". Optional fields can be omitted.

For example:

**@pytest.mark.warnings**("ignore:.\*usage will be deprecated.\*:

˓→DeprecationWarning")

**def** test\_foo():

...

**pytest.mark.parametrize**

Tutorial: [Parametrizing fixtures and test functions](#page81).

Metafunc.**parametrize**(argnames, argvalues, indirect=False, ids=None, scope=None)

Add new invocations to the underlying test function using the list of argvalues for the given argnames. Parametrization is performed during the collection phase. If you need to setup expensive resources see about setting indirect to do it rather at test setup time.

Parameters

* **argnames** – a comma-separated string denoting one or more argument names, or alist/tuple of argument strings.
* **argvalues** – The list of argvalues determines how often a test is invoked with differentargument values. If only one argname was specified argvalues is a list of values. If N argnames were specified, argvalues must be a list of N-tuples, where each tuple-element specifies a value for its respective argname.
* **indirect** – The list of argnames or boolean. A list of arguments’ names (subset ofargnames). If True the list contains all names from the argnames. Each argvalue corre-sponding to an argname in this list will be passed as request.param to its respective argname fixture function so that it can perform more expensive setups during the setup phase of a test rather than at collection time.
* **ids** – list of string ids, or a callable. If strings, each is corresponding to the argvalues so thatthey are part of the test id. If None is given as id of specific test, the automatically generated id for that argument will be used. If callable, it should take one argument (a single argvalue) and return a string or return None. If None, the automatically generated id for that argument will be used. If no ids are provided they will be generated automatically from the argvalues.
* **scope** – if specified it denotes the scope of the parameters. The scope is used for group-ing tests by parameter instances. It will also override any fixture-function defined scope, allowing to set a dynamic scope using test context or configuration.

**pytest.mark.skip**

Tutorial: [Skipping test functions](#page73).

Unconditionally skip a test function.

pytest.mark.**skip**(\*, reason=None)

Parameters **reason** ([str](#page135)) – Reason why the test function is being skipped.

**116** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

**pytest.mark.skipif**

Tutorial: [Skipping test functions](#page73).

Skip a test function if a condition is True.

pytest.mark.**skipif**(condition, \*, reason=None)

Parameters

* **condition** (bool or str) –True/Falseif the condition should be skipped or a[condition string](#page234).
* **reason** ([str](#page135))– Reason why the test function is being skipped.

**pytest.mark.usefixtures**

Tutorial: [Using fixtures from classes, modules or projects](#page44).

Mark a test function as using the given fixture names.

Warning: This mark can be used with test functions only, having no affect when applied to a fixture function.

pytest.mark.**usefixtures**(\*names)

Parameters **args** – the names of the fixture to use, as strings

**pytest.mark.xfail**

Tutorial: [XFail: mark test functions as expected to fail](#page76).

Marks a test function as expected to fail.

pytest.mark.**xfail**(condition=None, \*, reason=None, raises=None, run=True, strict=False)

Parameters

* **condition** (bool or str) –True/Falseif the condition should be marked as xfailor a [condition string](#page234).
* **reason** ([str](#page135))– Reason why the test function is marked as xfail.
* **raises** ([Exception](https://docs.python.org/3/library/exceptions.html#Exception))– Exception subclass expected to be raised by the test function;other exceptions will fail the test.
* **run** ([bool](https://docs.python.org/3/library/functions.html#bool))– If the test function should actually be executed. IfFalse, the function willalways xfail and will not be executed (useful a function is segfaulting).
* **strict** ([bool](https://docs.python.org/3/library/functions.html#bool))–

– If False (the default) the function will be shown in the terminal output as xfailed if it fails and as xpass if it passes. In both cases this will not cause the test suite to fail as a whole. This is particularly useful to mark flaky tests (tests that random at fail) to be tackled later.

– If True, the function will be shown in the terminal output as xfailed if it fails, but if it unexpectedly passes then it will fail the test suite. This is particularly useful to mark func-tions that are always failing and there should be a clear indication if they unexpectedly start to pass (for example a new release of a library fixes a known bug).

**22.2. Marks** **117**

**pytest Documentation, Release 3.6**

**custom marks**

Marks are created dynamically using the factory object pytest.mark and applied as a decorator.

For example:

**@pytest.mark.timeout**(10,"slow", method="thread")

**def** test\_function():

...

Will create and attach a [Mark](#page147) object to the collected [Item](#page146), which can then be accessed by fixtures or hooks with

[Node.iter\_markers](#page150). The mark object will have the following attributes:

mark.args == (10, "slow")

mark.kwargs == {"method": "thread"}

**Fixtures**

Tutorial: [pytest fixtures: explicit, modular, scalable](#page31).

Fixtures are requested by test functions or other fixtures by declaring them as argument names.

Example of a test requiring a fixture:

**def** test\_output(capsys):

**print**("hello")

out, err = capsys.readouterr()

**assert** out=="hello**\n**"

Example of a fixture requiring another fixture:

**@pytest.fixture**

**def** db\_session(tmpdir):

fn = tmpdir / "db.file"

**return** connect(str(fn))

For more details, consult the full [fixtures docs](#page31).

**@pytest.fixture**

@**fixture**(scope=’function’, params=None, autouse=False, ids=None, name=None)

Decorator to mark a fixture factory function.

This decorator can be used (with or without parameters) to define a fixture function. The name of the fixture function can later be referenced to cause its invocation ahead of running tests: test modules or classes can use the pytest.mark.usefixtures(fixturename) marker. Test functions can directly use fixture names as input arguments in which case the fixture instance returned from the fixture function will be injected.

Parameters

* **scope** – the scope for which this fixture is shared, one of “function” (default), “class”,“module” or “session”.
* **params** – an optional list of parameters which will cause multiple invocations of the fixturefunction and all of the tests using it.

**118** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

* **autouse** – if True, the fixture func is activated for all tests that can see it. If False (thedefault) then an explicit reference is needed to activate the fixture.
* **ids** – list of string ids each corresponding to the params so that they are part of the test id.If no ids are provided they will be generated automatically from the params.
* **name** – the name of the fixture. This defaults to the name of the decorated function.If a fixture is used in the same module in which it is defined, the function name of the fixture will be shadowed by the function arg that requests the fixture; one way to re-solve this is to name the decorated function fixture\_<fixturename> and then use @pytest.fixture(name='<fixturename>').

Fixtures can optionally provide their values to test functions using a yield statement, instead of return. In this case, the code block after the yield statement is executed as teardown code regardless of the test outcome. A fixture function must yield exactly once.

**config.cache**

Tutorial: [Cache: working with cross-testrun state](#page85).

The config.cache object allows other plugins and fixtures to store and retrieve values across test runs. To access it from fixtures request pytestconfig into your fixture and get it with pytestconfig.cache.

Under the hood, the cache plugin uses the simple dumps/loads API of the [json](https://docs.python.org/3/library/json.html#module-json) stdlib module.

Cache.**get**(key, default)

return cached value for the given key. If no value was yet cached or the value cannot be read, the specified default is returned.

Parameters

* **key** – must be a/separated value. Usually the first name is the name of your plugin oryour application.
* **default** – must be provided in case of a cache-miss or invalid cache values.

Cache.**set**(key, value)

save value for the given key.

Parameters

* **key** – must be a/separated value. Usually the first name is the name of your plugin oryour application.
* **value** – must be of any combination of basic python types, including nested types like e.g. lists of dictionaries.

Cache.**makedir**(name)

return a directory path object with the given name. If the directory does not yet exist, it will be created. You can use it to manage files likes e. g. store/retrieve database dumps across test sessions.

Parameters **name** – must be a string not containing a / separator. Make sure the name contains your plugin or application identifiers to prevent clashes with other cache users.

**capsys**

Tutorial: [Capturing of the stdout/stderr output](#page59).

**22.3. Fixtures** **119**

**pytest Documentation, Release 3.6**

**capsys**()

Enable capturing of writes to sys.stdout and sys.stderr and make captured output available via capsys.readouterr() method calls which return a (out,err) namedtuple. out and err will be text objects.

Returns an instance of [CaptureFixture](#page128).

Example:

**def** test\_output(capsys):

**print**("hello")

captured = capsys.readouterr()

**assert** captured.out=="hello**\n**"

class **CaptureFixture**

Object returned by [capsys()](#page127), [capsysbinary()](#page128), [capfd()](#page128) and [capfdbinary()](#page129) fixtures.

**readouterr**()

Read and return the captured output so far, resetting the internal buffer.

Returns captured content as a namedtuple with out and err string attributes

with **disabled**()

Temporarily disables capture while inside the ‘with’ block.

**capsysbinary**

Tutorial: [Capturing of the stdout/stderr output](#page59).

**capsysbinary**()

Enable capturing of writes to sys.stdout and sys.stderr and make captured output available via capsys.readouterr() method calls which return a (out,err) tuple. out and err will be bytes objects.

Returns an instance of [CaptureFixture](#page128).

Example:

**def** test\_output(capsysbinary):

**print**("hello")

captured = capsysbinary.readouterr()

**assert** captured.out==b"hello**\n**"

**capfd**

Tutorial: [Capturing of the stdout/stderr output](#page59).

**capfd**()

Enable capturing of writes to file descriptors 1 and 2 and make captured output available via capfd.readouterr() method calls which return a (out,err) tuple. out and err will be text objects.

Returns an instance of [CaptureFixture](#page128).

Example:

**def** test\_system\_echo(capfd):

os.system('echo "hello"')

captured = capsys.readouterr()

**assert** captured.out=="hello**\n**"

**120** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

**capfdbinary**

Tutorial: [Capturing of the stdout/stderr output](#page59).

**capfdbinary**()

Enable capturing of write to file descriptors 1 and 2 and make captured output available via capfdbinary.readouterr method calls which return a (out,err) tuple. out and err will be bytes objects.

Returns an instance of [CaptureFixture](#page128).

Example:

**def** test\_system\_echo(capfdbinary):

os.system('echo "hello"')

captured = capfdbinary.readouterr()

**assert** captured.out==b"hello**\n**"

**doctest\_namespace**

Tutorial: [Doctest integration for modules and test files](#page69).

**doctest\_namespace**()

Fixture that returns a [dict](https://docs.python.org/3/library/stdtypes.html#dict) that will be injected into the namespace of doctests.

Usually this fixture is used in conjunction with another autouse fixture:

**@pytest.fixture**(autouse=True)

**def** add\_np(doctest\_namespace):

doctest\_namespace["np"] = numpy

For more details: [The ‘doctest\_namespace’ fixture](#page71).

**request**

Tutorial: [Pass different values to a test function, depending on command line options](#page186).

The request fixture is a special fixture providing information of the requesting test function.

class **FixtureRequest**

A request for a fixture from a test or fixture function.

A request object gives access to the requesting test context and has an optional param attribute in case the fixture is parametrized indirectly.

**fixturename** = None

fixture for which this request is being performed

**scope** = None

Scope string, one of “function”, “class”, “module”, “session”

**node**

underlying collection node (depends on current request scope)

**config**

the pytest config object associated with this request.

**22.3. Fixtures** **121**

**pytest Documentation, Release 3.6**

**function**

test function object if the request has a per-function scope.

**cls**

class (can be None) where the test function was collected.

**instance**

instance (can be None) on which test function was collected.

**module**

python module object where the test function was collected.

**fspath**

the file system path of the test module which collected this test.

**keywords**

keywords/markers dictionary for the underlying node.

**session**

pytest session object.

**addfinalizer**(finalizer)

add finalizer/teardown function to be called after the last test within the requesting test context finished execution.

**applymarker**(marker)

Apply a marker to a single test function invocation. This method is useful if you don’t want to have a keyword/marker on all function invocations.

Parameters **marker** – a [\_pytest.mark.MarkDecorator](#page146) object created by a call to pytest.mark.NAME(...).

**raiseerror**(msg)

raise a FixtureLookupError with the given message.

**cached\_setup**(setup,teardown=None,scope=’module’,extrakey=None)

(deprecated) Return a testing resource managed by setup & teardown calls. scope and extrakey determine when the teardown function will be called so that subsequent calls to setup would recreate the resource. With pytest-2.3 you often do not need cached\_setup() as you can directly declare a scope on a fixture function and register a finalizer through request.addfinalizer().

Parameters

* **teardown** – function receiving a previously setup resource.
* **setup** – a no-argument function creating a resource.
* **scope** – a string value out offunction,class,moduleorsessionindicating thecaching lifecycle of the resource.
* **extrakey** – added to internal caching key of (funcargname, scope).

**getfixturevalue**(argname)

Dynamically run a named fixture function.

Declaring fixtures via function argument is recommended where possible. But if you can only decide whether to use another fixture at test setup time, you may use this function to retrieve it inside a fixture or test function body.

**getfuncargvalue**(argname)

Deprecated, use getfixturevalue.

**122** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

**pytestconfig**

**pytestconfig**()

Session-scoped fixture that returns the [\_pytest.config.Config](#page144) object.

Example:

**def** test\_foo(pytestconfig):

**if** pytestconfig.getoption("verbose"):

...

**record\_property**

Tutorial: [record\_property](#page18).

**record\_property**()

Add an extra properties the calling test. User properties become part of the test report and are available to the configured reporters, like JUnit XML. The fixture is callable with (name,value), with value being automat-ically xml-encoded.

Example:

**def** test\_function(record\_property):

record\_property("example\_key", 1)

**caplog**

Tutorial: [Logging](#page111).

**caplog**()

Access and control log capturing.

Captured logs are available through the following methods:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| \* caplog.text | | -> string containing formatted log output | | | |
| \* caplog.records | | -> list | | of | logging.LogRecord instances |
| \* | caplog.record\_tuples | -> | list | of | (logger\_name, level, message) tuples |
| \* | caplog.clear() | -> | clear captured records **and** formatted log output string | | |
|  |  |  |  |  |  |

This returns a [\_pytest.logging.LogCaptureFixture](#page131) instance.

class **LogCaptureFixture**(item)

Provides access and control of log capturing.

**handler**

Return type LogCaptureHandler

**get\_records**(when)

Get the logging records for one of the possible test phases.

Parameters **when** ([str](#page135)) – Which test phase to obtain the records from. Valid values are:

“setup”, “call” and “teardown”.

Return type List[logging.LogRecord]

Returns the list of captured records at the given stage

New in version 3.4.

**22.3. Fixtures** **123**

**pytest Documentation, Release 3.6**

**text**

Returns the log text.

**records**

Returns the list of log records.

**record\_tuples**

Returns a list of a striped down version of log records intended for use in assertion comparison.

The format of the tuple is:

(logger\_name, log\_level, message)

**clear**()

Reset the list of log records and the captured log text.

**set\_level**(level,logger=None)

Sets the level for capturing of logs. The level will be restored to its previous value at the end of the test.

Parameters

* **level** ([int](https://docs.python.org/3/library/functions.html#int))– the logger to level.
* **logger** ([str](#page135))– the logger to update the level. If not given, the root logger level isupdated.

Changed in version 3.4: The levels of the loggers changed by this function will be restored to their initial values at the end of the test.

with **at\_level**(level, logger=None)

Context manager that sets the level for capturing of logs. After the end of the ‘with’ statement the level is restored to its original value.

Parameters

* **level** ([int](https://docs.python.org/3/library/functions.html#int))– the logger to level.
* **logger** ([str](#page135))– the logger to update the level. If not given, the root logger level isupdated.

**monkeypatch**

Tutorial: [Monkeypatching/mocking modules and environments](#page55).

**monkeypatch**()

The returned monkeypatch fixture provides these helper methods to modify objects, dictionaries or os.environ:

monkeypatch.setattr(obj, name, value, raising=**True**)

monkeypatch.delattr(obj, name, raising=**True**)

monkeypatch.setitem(mapping, name, value)

monkeypatch.delitem(obj, name, raising=**True**)

monkeypatch.setenv(name, value, prepend=**False**)

monkeypatch.delenv(name, raising=**True**)

monkeypatch.syspath\_prepend(path)

monkeypatch.chdir(path)

All modifications will be undone after the requesting test function or fixture has finished. The raising pa-rameter determines if a KeyError or AttributeError will be raised if the set/deletion operation has no target.

This returns a [MonkeyPatch](#page132) instance.

**124** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

class **MonkeyPatch**

Object returned by the monkeypatch fixture keeping a record of setattr/item/env/syspath changes.

with **context**()

Context manager that returns a new [MonkeyPatch](#page132) object which undoes any patching done inside the with block upon exit:

**import functools**

**def** test\_partial(monkeypatch):

**with** monkeypatch.context() **as** m:

m.setattr(functools, "partial", 3)

Useful in situations where it is desired to undo some patches before the test ends, such as mocking stdlib functions that might break pytest itself if mocked (for examples of this see [#3290](https://github.com/pytest-dev/pytest/issues/3290).

**setattr**(target,name,value=<notset>,raising=True)

Set attribute value on target, memorizing the old value. By default raise AttributeError if the attribute did not exist.

|  |  |  |  |
| --- | --- | --- | --- |
| For convenience you can | specify a string as target which | will be | interpreted |
| as a dotted import path, | with the last part being the attribute name. | | Example: |
| monkeypatch.setattr("os.getcwd",lambda: "/") would set | | the getcwd function | |
| of the os module. |  |  |  |

The raising value determines if the setattr should fail if the attribute is not already present (defaults to True which means it will raise).

**delattr**(target,name=<notset>,raising=True)

Delete attribute name from target, by default raise AttributeError it the attribute did not previously exist.

If no name is specified and target is a string it will be interpreted as a dotted import path with the last part being the attribute name.

If raising is set to False, no exception will be raised if the attribute is missing.

**setitem**(dic,name,value)

Set dictionary entry name to value.

**delitem**(dic,name,raising=True)

Delete name from dict. Raise KeyError if it doesn’t exist.

If raising is set to False, no exception will be raised if the key is missing.

**setenv**(name,value,prepend=None)

Set environment variable name to value. If prepend is a character, read the current environment variable value and prepend the value adjoined with the prepend character.

**delenv**(name,raising=True)

Delete name from the environment. Raise KeyError it does not exist.

If raising is set to False, no exception will be raised if the environment variable is missing.

**syspath\_prepend**(path)

Prepend path to sys.path list of import locations.

**chdir**(path)

Change the current working directory to the specified path. Path can be a string or a py.path.local object.

**undo**()

Undo previous changes. This call consumes the undo stack. Calling it a second time has no effect unless you do more monkeypatching after the undo call.

**22.3. Fixtures** **125**

**pytest Documentation, Release 3.6**

There is generally no need to call undo(), since it is called automatically during tear-down.

Note that the same monkeypatch fixture is used across a single test function invocation. If monkeypatch is used both by the test function itself and one of the test fixtures, calling undo() will undo all of the changes made in both functions.

**testdir**

This fixture provides a [Testdir](#page134) instance useful for black-box testing of test files, making it ideal to test plugins.

To use it, include in your top-most conftest.py file:

pytest\_plugins = 'pytester'

class **Testdir**

Temporary test directory with tools to test/run pytest itself.

This is based on the tmpdir fixture but provides a number of methods which aid with testing pytest itself.

Unless chdir() is used all methods will use tmpdir as their current working directory.

Attributes:

Tmpdir The py.path.local instance of the temporary directory.

Plugins A list of plugins to use with parseconfig() and [runpytest()](#page134). Initially this is an empty list but plugins can be added to the list. The type of items to add to the list depends on the method using them so refer to them for details.

**makeconftest**(source)

Write a contest.py file with ‘source’ as contents.

**makepyfile**(\*args,\*\*kwargs)

Shortcut for .makefile() with a .py extension.

**runpytest\_inprocess**(\*args,\*\*kwargs)

Return result of running pytest in-process, providing a similar interface to what self.runpytest() provides.

**runpytest**(\*args,\*\*kwargs)

Run pytest inline or in a subprocess, depending on the command line option “–runpytest” and return a [RunResult](#page134).

**runpytest\_subprocess**(\*args,\*\*kwargs)

Run pytest as a subprocess with given arguments.

Any plugins added to the plugins list will added using the -p command line option. Additionally --basetemp is used put any temporary files and directories in a numbered directory prefixed with “runpytest-” so they do not conflict with the normal numbered pytest location for temporary files and directories.

Returns a [RunResult](#page134).

class **RunResult**

The result of running a command.

Attributes:

Ret the return value

Outlines list of lines captured from stdout

Errlines list of lines captures from stderr

**126** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

Stdout [LineMatcher](#page135) of stdout, use stdout.str() to reconstruct stdout or the commonly used stdout.fnmatch\_lines() method

Stderr [LineMatcher](#page135) of stderr

Duration duration in seconds

**parseoutcomes**()

Return a dictionary of outcomestring->num from parsing the terminal output that the test process produced.

**assert\_outcomes**(passed=0,skipped=0,failed=0,error=0)

Assert that the specified outcomes appear with the respective numbers (0 means it didn’t occur) in the text output from a test run.

class **LineMatcher**

Flexible matching of text.

This is a convenience class to test large texts like the output of commands.

The constructor takes a list of lines without their trailing newlines, i.e. text.splitlines().

**str**()

Return the entire original text.

**fnmatch\_lines\_random**(lines2)

Check lines exist in the output using in any order.

Lines are checked using fnmatch.fnmatch. The argument is a list of lines which have to occur in the output, in any order.

**re\_match\_lines\_random**(lines2)

Check lines exist in the output using re.match, in any order.

The argument is a list of lines which have to occur in the output, in any order.

**get\_lines\_after**(fnline)

Return all lines following the given line in the text.

The given line can contain glob wildcards.

**fnmatch\_lines**(lines2)

Search captured text for matching lines using fnmatch.fnmatch.

The argument is a list of lines which have to match and can use glob wildcards. If they do not match a pytest.fail() is called. The matches and non-matches are also printed on stdout.

**re\_match\_lines**(lines2)

Search captured text for matching lines using re.match.

The argument is a list of lines which have to match using re.match. If they do not match a pytest.fail() is called.

The matches and non-matches are also printed on stdout.

**recwarn**

Tutorial: [Asserting warnings with the warns function](#page65)

**recwarn**()

Return a [WarningsRecorder](#page135) instance that records all warnings emitted by test functions.

See <http://docs.python.org/library/warnings.html>for information on warning categories.

**22.3. Fixtures** **127**

**pytest Documentation, Release 3.6**

class **WarningsRecorder**

A context manager to record raised warnings.

Adapted from warnings.catch\_warnings.

**list**

The list of recorded warnings.

**pop**(cls=<class ‘Warning’>)

Pop the first recorded warning, raise exception if not exists.

**clear**()

Clear the list of recorded warnings.

Each recorded warning is an instance of warnings.WarningMessage.

Note: RecordedWarning was changed from a plain class to a namedtuple in pytest 3.1

Note: DeprecationWarning and PendingDeprecationWarning are treated differently; see [Ensuring a](#page67) [function triggers a deprecation warning](#page67).

**tmpdir**

Tutorial: [Temporary directories and files](#page57)

**tmpdir**()

Return a temporary directory path object which is unique to each test function invocation, created as a sub directory of the base temporary directory. The returned object is a [py.path.local](https://py.readthedocs.io/en/latest/path.html) path object.

**tmpdir\_factory**

Tutorial: [The ‘tmpdir\_factory’ fixture](#page58) tmpdir\_factory instances have the following methods:

TempdirFactory.**mktemp**(basename, numbered=True)

Create a subdirectory of the base temporary directory and return it. If numbered, ensure the directory is unique by adding a number prefix greater than any existing one.

TempdirFactory.**getbasetemp**()

return base temporary directory.

**Hooks**

Tutorial: [Writing plugins](#page101).

Reference to all hooks which can be implemented by [conftest.py files](#page102) and [plugins](#page101).

**Bootstrapping hooks**

Bootstrapping hooks called for plugins registered early enough (internal and setuptools plugins).

**128** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

**pytest\_load\_initial\_conftests**(early\_config,parser,args)

implements the loading of initial conftest files ahead of command line option parsing.

Note: This hook will not be called for conftest.py files, only for setuptools plugins.

Parameters

* **early\_config** ([\_pytest.config.Config](#page144))– pytest config object
* **args** (list[str]) – list of arguments passed on the command line
* **parser** (\_pytest.config.Parser) – to add command line options

**pytest\_cmdline\_preparse**(config,args)

(Deprecated) modify command line arguments before option parsing.

This hook is considered deprecated and will be removed in a future pytest version. Consider using [pytest\_load\_initial\_conftests()](#page136) instead.

Note: This hook will not be called for conftest.py files, only for setuptools plugins.

Parameters

* **config** ([\_pytest.config.Config](#page144))– pytest config object
* **args** (list[str]) – list of arguments passed on the command line

**pytest\_cmdline\_parse**(pluginmanager,args)

return initialized config object, parsing the specified args.

Stops at first non-None result, see [firstresult: stop at first non-None result](#page107)

Note: This hook will not be called for conftest.py files, only for setuptools plugins.

Parameters

* **pluginmanager** ([\_pytest.config.PytestPluginManager](#page152))– pytest pluginmanager
* **args** (list[str]) – list of arguments passed on the command line

**pytest\_cmdline\_main**(config)

called for performing the main command line action. The default implementation will invoke the configure hooks and runtest\_mainloop.

Note: This hook will not be called for conftest.py files, only for setuptools plugins.

Stops at first non-None result, see [firstresult: stop at first non-None result](#page107)

Parameters **config** ([\_pytest.config.Config](#page144)) – pytest config object

**22.4. Hooks** **129**

**pytest Documentation, Release 3.6**

**Initialization hooks**

Initialization hooks called for plugins and conftest.py files.

**pytest\_addoption**(parser)

register argparse-style options and ini-style config values, called once at the beginning of a test run.

Note: This function should be implemented only in plugins or conftest.py files situated at the tests root directory due to how pytest [discovers plugins during startup](#page101).

Parameters **parser** (\_pytest.config.Parser) – To add command line options, call parser.addoption(...). To add ini-file values call parser.addini(...).

Options can later be accessed through the [config](#page144) object, respectively:

•[config.getoption(name)](#page144) to retrieve the value of a command line option.

•[config.getini(name)](#page144) to retrieve a value read from an ini-style file.

The config object is passed around on many internal objects via the .config attribute or can be retrieved as the pytestconfig fixture.

Note: This hook is incompatible with hookwrapper=True.

**pytest\_addhooks**(pluginmanager)

called at plugin registration time to allow adding new hooks via a call to pluginmanager.add\_hookspecs(module\_or\_class,prefix).

Parameters **pluginmanager** ([\_pytest.config.PytestPluginManager](#page152)) – pytest plu-gin manager

Note: This hook is incompatible with hookwrapper=True.

**pytest\_configure**(config)

Allows plugins and conftest files to perform initial configuration.

This hook is called for every plugin and initial conftest file after command line options have been parsed.

After that, the hook is called for other conftest files as they are imported.

Note: This hook is incompatible with hookwrapper=True.

Parameters **config** ([\_pytest.config.Config](#page144)) – pytest config object

**pytest\_unconfigure**(config)

called before test process is exited.

Parameters **config** ([\_pytest.config.Config](#page144)) – pytest config object

**pytest\_sessionstart**(session)

called after the Session object has been created and before performing collection and entering the run test loop.

Parameters **session** ([\_pytest.main.Session](#page153)) – the pytest session object

**130** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

**pytest\_sessionfinish**(session,exitstatus)

called after whole test run finished, right before returning the exit status to the system.

Parameters

* **session** ([\_pytest.main.Session](#page153))– the pytest session object
* **exitstatus** ([int](https://docs.python.org/3/library/functions.html#int))– the status which pytest will return to the system

**Test running hooks**

All runtest related hooks receive a pytest.Item object.

**pytest\_runtestloop**(session)

called for performing the main runtest loop (after collection finished).

Stops at first non-None result, see [firstresult: stop at first non-None result](#page107)

Parameters **session** ([\_pytest.main.Session](#page153)) – the pytest session object

**pytest\_runtest\_protocol**(item,nextitem)

implements the runtest\_setup/call/teardown protocol for the given test item, including capturing exceptions and calling reporting hooks.

Parameters

* **item** – test item for which the runtest protocol is performed.
* **nextitem** – the scheduled-to-be-next test item (or None if this is the end my friend). Thisargument is passed on to [pytest\_runtest\_teardown()](#page139).

Return boolean True if no further hook implementations should be invoked.

Stops at first non-None result, see [firstresult: stop at first non-None result](#page107)

**pytest\_runtest\_logstart**(nodeid,location)

signal the start of running a single test item.

This hook will be called before [pytest\_runtest\_setup()](#page139), [pytest\_runtest\_call()](#page139) and [pytest\_runtest\_teardown()](#page139) hooks.

Parameters

* **nodeid** ([str](#page135))– full id of the item
* **location** – a triple of(filename,linenum,testname)

**pytest\_runtest\_logfinish**(nodeid,location)signal the complete finish of running a single test item.

This hook will be called after [pytest\_runtest\_setup()](#page139), [pytest\_runtest\_call()](#page139) and [pytest\_runtest\_teardown()](#page139) hooks.

Parameters

* **nodeid** ([str](#page135))– full id of the item
* **location** – a triple of(filename,linenum,testname)

**pytest\_runtest\_setup**(item)

called before pytest\_runtest\_call(item).

**pytest\_runtest\_call**(item)

called to execute the test item.

**22.4. Hooks** **131**

**pytest Documentation, Release 3.6**

**pytest\_runtest\_teardown**(item,nextitem)

called after pytest\_runtest\_call.

Parameters **nextitem** – the scheduled-to-be-next test item (None if no further test item is sched-uled). This argument can be used to perform exact teardowns, i.e. calling just enough finalizers so that nextitem only needs to call setup-functions.

**pytest\_runtest\_makereport**(item,call)

return a [\_pytest.runner.TestReport](#page153) object for the given pytest.Item and [\_pytest.runner.CallInfo](#page143).

Stops at first non-None result, see [firstresult: stop at first non-None result](#page107)

For deeper understanding you may look at the default implementation of these hooks in \_pytest.runner and maybe also in \_pytest.pdb which interacts with \_pytest.capture and its input/output capturing in order to immediately drop into interactive debugging when a test failure occurs.

The \_pytest.terminal reported specifically uses the reporting hook to print information about a test run.

**Collection hooks**

pytest calls the following hooks for collecting files and directories:

**pytest\_collection**(session)

Perform the collection protocol for the given session.

Stops at first non-None result, see [firstresult: stop at first non-None result](#page107).

Parameters **session** ([\_pytest.main.Session](#page153)) – the pytest session object

**pytest\_ignore\_collect**(path,config)

return True to prevent considering this path for collection. This hook is consulted for all files and directories prior to calling more specific hooks.

Stops at first non-None result, see [firstresult: stop at first non-None result](#page107)

Parameters

* **path** ([str](#page135))– the path to analyze
* **config** ([\_pytest.config.Config](#page144))– pytest config object

**pytest\_collect\_directory**(path,parent)

called before traversing a directory for collection files.

Stops at first non-None result, see [firstresult: stop at first non-None result](#page107)

Parameters **path** ([str](#page135)) – the path to analyze

**pytest\_collect\_file**(path,parent)

return collection Node or None for the given path. Any new node needs to have the specified parent as a parent.

Parameters **path** ([str](#page135)) – the path to collect

For influencing the collection of objects in Python modules you can use the following hook:

**pytest\_pycollect\_makeitem**(collector,name,obj)

return custom item/collector for a python object in a module, or None.

Stops at first non-None result, see [firstresult: stop at first non-None result](#page107)

**pytest\_generate\_tests**(metafunc)

generate (multiple) parametrized calls to a test function.

**132** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

**pytest\_make\_parametrize\_id**(config,val,argname)

Return a user-friendly string representation of the given val that will be used by @pytest.mark.parametrize calls. Return None if the hook doesn’t know about val. The parameter name is available as argname, if required.

Stops at first non-None result, see [firstresult: stop at first non-None result](#page107)

Parameters

* **config** ([\_pytest.config.Config](#page144))– pytest config object
* **val** – the parametrized value
* **argname** ([str](#page135))– the automatic parameter name produced by pytest

After collection is complete, you can modify the order of items, delete or otherwise amend the test items:

**pytest\_collection\_modifyitems**(session,config,items)

called after collection has been performed, may filter or re-order the items in-place.

Parameters

* **session** ([\_pytest.main.Session](#page153))– the pytest session object
* **config** ([\_pytest.config.Config](#page144))– pytest config object
* **items** (List[\_pytest.nodes.Item]) – list of item objects

**Reporting hooks**

Session related reporting hooks:

**pytest\_collectstart**(collector)

collector starts collecting.

**pytest\_itemcollected**(item)

we just collected a test item.

**pytest\_collectreport**(report)

collector finished collecting.

**pytest\_deselected**(items)

called for test items deselected by keyword.

**pytest\_report\_header**(config,startdir)

return a string or list of strings to be displayed as header info for terminal reporting.

Parameters

* **config** ([\_pytest.config.Config](#page144))– pytest config object
* **startdir** – py.path object with the starting dir

Note: This function should be implemented only in plugins or conftest.py files situated at the tests root directory due to how pytest [discovers plugins during startup](#page101).

**pytest\_report\_collectionfinish**(config,startdir,items)

New in version 3.2.

return a string or list of strings to be displayed after collection has finished successfully.

This strings will be displayed after the standard “collected X items” message.

**22.4. Hooks** **133**

**pytest Documentation, Release 3.6**

Parameters

* **config** ([\_pytest.config.Config](#page144))– pytest config object
* **startdir** – py.path object with the starting dir
* **items** – list of pytest items that are going to be executed; this list should not be modified.

**pytest\_report\_teststatus**(report)

return result-category, shortletter and verbose word for reporting.

Stops at first non-None result, see [firstresult: stop at first non-None result](#page107)

**pytest\_terminal\_summary**(terminalreporter,exitstatus)

Add a section to terminal summary reporting.

Parameters

* **terminalreporter** (\_pytest.terminal.TerminalReporter) – the internalterminal reporter object
* **exitstatus** ([int](https://docs.python.org/3/library/functions.html#int))– the exit status that will be reported back to the OS

New in version 3.5: The config parameter.

**pytest\_fixture\_setup**(fixturedef,request)

performs fixture setup execution.

Returns The return value of the call to the fixture function

Stops at first non-None result, see [firstresult: stop at first non-None result](#page107)

Note: If the fixture function returns None, other implementations of this hook function will continue to be

called, according to the behavior of the [firstresult: stop at first non-None result](#page107) option.

**pytest\_fixture\_post\_finalizer**(fixturedef,request)

called after fixture teardown, but before the cache is cleared so the fixture result cache fixturedef.cached\_result can still be accessed.

And here is the central hook for reporting about test execution:

**pytest\_runtest\_logreport**(report)

process a test setup/call/teardown report relating to the respective phase of executing a test.

You can also use this hook to customize assertion representation for some types:

**pytest\_assertrepr\_compare**(config,op,left,right)

return explanation for comparisons in failing assert expressions.

Return None for no custom explanation, otherwise return a list of strings. The strings will be joined by newlines but any newlines in a string will be escaped. Note that all but the first line will be indented slightly, the intention is for the first line to be a summary.

Parameters **config** ([\_pytest.config.Config](#page144)) – pytest config object

**Debugging/Interaction hooks**

There are few hooks which can be used for special reporting or interaction with exceptions:

**pytest\_internalerror**(excrepr,excinfo)

called for internal errors.

**134** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

**pytest\_keyboard\_interrupt**(excinfo)

called for keyboard interrupt.

**pytest\_exception\_interact**(node,call,report)

called when an exception was raised which can potentially be interactively handled.

This hook is only called if an exception was raised that is not an internal exception like skip.Exception.

**pytest\_enter\_pdb**(config)

called upon pdb.set\_trace(), can be used by plugins to take special action just before the python debugger enters in interactive mode.

Parameters **config** ([\_pytest.config.Config](#page144)) – pytest config object

**Objects**

Full reference to objects accessible from [fixtures](#page31) or [hooks](#page136).

**CallInfo**

class **CallInfo**

Result/Exception info a function invocation.

**when** = None

context of invocation: one of “setup”, “call”, “teardown”, “memocollect”

**excinfo** = None

None or ExceptionInfo object.

**Class**

class **Class**

Bases: \_pytest.python.PyCollector

Collector for test methods.

**Collector**

class **Collector**

Bases: [\_pytest.nodes.Node](#page149)

Collector instances create children through collect() and thus iteratively build a tree.

exception **CollectError**

Bases: [Exception](https://docs.python.org/3/library/exceptions.html#Exception)

an error during collection, contains a custom message.

Collector.**collect**()

returns a list of children (items and collectors) for this collection node.

Collector.**repr\_failure**(excinfo)

represent a collection failure.

**22.5. Objects** **135**

**pytest Documentation, Release 3.6**

**Config**

class **Config**

access to configuration values, pluginmanager and plugin hooks.

**option** = None

access to command line option as attributes. (deprecated), use [getoption()](#page144) instead

**pluginmanager** = None

a pluginmanager instance

**add\_cleanup**(func)

Add a function to be called when the config object gets out of use (usually coninciding with pytest\_unconfigure).

**warn**(code,message,fslocation=None,nodeid=None)

generate a warning for this test session.

classmethod **fromdictargs**(option\_dict, args)

constructor useable for subprocesses.

**addinivalue\_line**(name,line)

add a line to an ini-file option. The option must have been declared but might not yet be set in which case the line becomes the the first line in its value.

**getini**(name)

return configuration value from an [ini file](#page171). If the specified name hasn’t been registered through a prior parser.addini call (usually from a plugin), a ValueError is raised.

**getoption**(name,default=<NOTSET>,skip=False)

return command line option value.

Parameters

* **name** – name of the option. You may also specify the literal--OPToption instead of the“dest” option name.
* **default** – default value if no option of that name exists.
* **skip** – if True raise pytest.skip if option does not exists or has a None value.

**getvalue**(name,path=None)

(deprecated, use getoption())

**getvalueorskip**(name,path=None)

(deprecated, use getoption(skip=True))

**ExceptionInfo**

class **ExceptionInfo**(tup=None, exprinfo=None)

wraps sys.exc\_info() objects and offers help for navigating the traceback.

**type** = None

the exception class

**value** = None

the exception instance

**tb** = None

the exception raw traceback

**136** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

**typename** = None

the exception type name

**traceback** = None

the exception traceback (\_pytest.\_code.Traceback instance)

**exconly**(tryshort=False)

return the exception as a string

when ‘tryshort’ resolves to True, and the exception is a \_pytest.\_code.\_AssertionError, only the actual exception part of the exception representation is returned (so ‘AssertionError: ‘ is removed from the be-ginning)

**errisinstance**(exc)

return True if the exception is an instance of exc

**getrepr**(showlocals=False,style=’long’,abspath=False,tbfilter=True,funcargs=False)

return str()able representation of this exception info. showlocals: show locals per traceback entry style:

long|short|no|native traceback style tbfilter: hide entries (where \_\_tracebackhide\_\_ is true)

in case of style==native, tbfilter and showlocals is ignored.

**match**(regexp)

Match the regular expression ‘regexp’ on the string representation of the exception. If it matches then True is returned (so that it is possible to write ‘assert excinfo.match()’). If it doesn’t match an AssertionError is raised.

**FixtureDef**

class **FixtureDef**

Bases: [object](https://docs.python.org/3/library/functions.html#object)

A container for a factory definition.

**FSCollector**

class **FSCollector**

Bases: [\_pytest.nodes.Collector](#page143)

**Function**

class **Function**

Bases: \_pytest.python.FunctionMixin, [\_pytest.nodes.Item](#page146), \_pytest.compat.FuncargnamesCompatAttr

a Function Item is responsible for setting up and executing a Python test function.

**originalname** = None

original function name, without any decorations (for example parametrization adds a "[...]" suffix to function names).

New in version 3.0.

**function**

underlying python ‘function’ object

**runtest**()

execute the underlying test function.

**22.5. Objects** **137**

**pytest Documentation, Release 3.6**

**Item**

class **Item**

Bases: [\_pytest.nodes.Node](#page149)

a basic test invocation item. Note that for a single function there might be multiple test invocation items.

**user\_properties** = None

user properties is a list of tuples (name, value) that holds user defined properties for this test.

**add\_report\_section**(when,key,content)

Adds a new report section, similar to what’s done internally to add stdout and stderr captured output:

item.add\_report\_section("call", "stdout", "report section contents")

Parameters

* **when** ([str](#page135))– One of the possible capture states,"setup","call","teardown".
* **key** ([str](#page135))– Name of the section, can be customized at will. Pytest uses"stdout"and"stderr" internally.
* **content** ([str](#page135))– The full contents as a string.

**MarkDecorator**

class **MarkDecorator**(mark) → None

A decorator for test functions and test classes. When applied it will create MarkInfo objects which may be [retrieved by hooks as item keywords](#page188). MarkDecorator instances are often created like this:

mark1 = pytest.mark.NAME # simple MarkDecorator

mark2 = pytest.mark.NAME(name1=value) # parametrized MarkDecorator

and can then be applied as decorators to test functions:

**@mark2**

**def** test\_function():

**pass**

When a MarkDecorator instance is called it does the following:

1. If called with a single class as its only positional argument and no additional keyword arguments, it attaches itself to the class so it gets applied automatically to all test cases found in that class.
2. If called with a single function as its only positional argument and no additional keyword arguments, it attaches a MarkInfo object to the function, containing all the arguments already stored internally in the MarkDecorator.
3. When called in any other case, it performs a ‘fake construction’ call, i.e. it returns a new MarkDec-orator instance with the original MarkDecorator’s content updated with the arguments passed to this call.

Note: The rules above prevent MarkDecorator objects from storing only a single function or class reference as their positional argument with no additional keyword or positional arguments.

**name**

alias for mark.name

**138** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

**args**

alias for mark.args

**kwargs**

alias for mark.kwargs

**with\_args**(\*args,\*\*kwargs)

return a MarkDecorator with extra arguments added

unlike call this can be used even if the sole argument is a callable/class

Returns MarkDecorator

**MarkGenerator**

class **MarkGenerator**

Factory for MarkDecorator objects - exposed as a pytest.mark singleton instance. Example:

**import pytest**

**@pytest**.mark.slowtest

**def** test\_function():

**pass**

will set a ‘slowtest’ MarkInfo object on the test\_function object.

**MarkInfo**

class **MarkInfo**(marks, combined=NOTHING) → None Marking object created by MarkDecorator instances.

**name**

alias for combined.name

**args**

alias for combined.args

**kwargs**

alias for combined.kwargs

**add\_mark**(mark)

add a MarkInfo with the given args and kwargs.

**Mark**

class **Mark**(name: str, args, kwargs) → None

**name** = None

name of the mark

**args** = None

positional arguments of the mark decorator

**kwargs** = None

keyword arguments of the mark decorator

**combined\_with**(other)

**22.5. Objects** **139**

**pytest Documentation, Release 3.6**

Parameters **other** ([Mark](#page147)) – the mark to combine with

Return type [Mark](#page147)

combines by appending aargs and merging the mappings

**Metafunc**

class **Metafunc**(definition, fixtureinfo, config, cls=None, module=None)

Metafunc objects are passed to the [pytest\_generate\_tests](#page140) hook. They help to inspect a test function and to generate tests according to test configuration or values specified in the class or module where a test function is defined.

**module** = None

the module object where the test function is defined in.

**function** = None

underlying python test function

**fixturenames** = None

set of fixture names required by the test function

**cls** = None

class object where the test function is defined in or None.

**parametrize**(argnames,argvalues,indirect=False,ids=None,scope=None)

Add new invocations to the underlying test function using the list of argvalues for the given argnames. Parametrization is performed during the collection phase. If you need to setup expensive resources see about setting indirect to do it rather at test setup time.

Parameters

* **argnames** – a comma-separated string denoting one or more argument names, or alist/tuple of argument strings.
* **argvalues** – The list of argvalues determines how often a test is invoked with differentargument values. If only one argname was specified argvalues is a list of values. If N argnames were specified, argvalues must be a list of N-tuples, where each tuple-element specifies a value for its respective argname.
* **indirect** – The list of argnames or boolean. A list of arguments’ names (subset ofargnames). If True the list contains all names from the argnames. Each argvalue cor-responding to an argname in this list will be passed as request.param to its respective argname fixture function so that it can perform more expensive setups during the setup phase of a test rather than at collection time.
* **ids** – list of string ids, or a callable. If strings, each is corresponding to the argvalues sothat they are part of the test id. If None is given as id of specific test, the automatically generated id for that argument will be used. If callable, it should take one argument (a single argvalue) and return a string or return None. If None, the automatically generated id for that argument will be used. If no ids are provided they will be generated automatically from the argvalues.
* **scope** – if specified it denotes the scope of the parameters. The scope is used for group-ing tests by parameter instances. It will also override any fixture-function defined scope, allowing to set a dynamic scope using test context or configuration.

**addcall**(funcargs=None,id=<object object>,param=<object object>)

Add a new call to the underlying test function during the collection phase of a test run.

Deprecated since version 3.3: Use parametrize() instead.

**140** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

Note that request.addcall() is called during the test collection phase prior and independently to actual test execution. You should only use addcall() if you need to specify multiple arguments of a test function.

Parameters

* **funcargs** – argument keyword dictionary used when invoking the test function.
* **id** – used for reporting and identification purposes. If you don’t supply anidan automaticunique id will be generated.
* **param** – a parameter which will be exposed to a later fixture function invocation throughthe request.param attribute.

**Module**

class **Module**

Bases: \_pytest.nodes.File, \_pytest.python.PyCollector

Collector for test classes and functions.

**Node**

class **Node**

base class for Collector and Item the test collection tree. Collector subclasses have children, Items are terminal nodes.

**name** = None

a unique name within the scope of the parent node

**parent** = None

the parent collector node.

**config** = None

the pytest config object

**session** = None

the session this node is part of

**fspath** = None

filesystem path where this node was collected from (can be None)

**keywords** = None

keywords/markers collected from all scopes

**own\_markers** = None

the marker objects belonging to this node

**extra\_keyword\_matches** = None

allow adding of extra keywords to use for matching

**ihook**

fspath sensitive hook proxy used to call pytest hooks

**warn**(code,message)

generate a warning with the given code and message for this item.

**nodeid**

a ::-separated string denoting its collection tree address.

**listchain**()

return list of all parent collectors up to self, starting from root of collection tree.

**22.5. Objects** **141**

**pytest Documentation, Release 3.6**

**add\_marker**(marker)

dynamically add a marker object to the node.

**iter\_markers**(name=None)

Parameters **name** – if given, filter the results by the name attribute

iterate over all markers of the node

for ... in **iter\_markers\_with\_node**(name=None)

Parameters **name** – if given, filter the results by the name attribute

iterate over all markers of the node returns sequence of tuples (node, mark)

**get\_closest\_marker**(name,default=None)

return the first marker matching the name, from closest (for example function) to farther level (for example module level).

Parameters

* **default** – fallback return value of no marker was found
* **name** – name to filter by

**get\_marker**(name)

get a marker object from this node or None if the node doesn’t have a marker with that name.

Deprecated since version 3.6: This function has been deprecated in favor of [Node.get\_closest\_marker](#page150) and [Node.iter\_markers](#page150), see [Updating code](#page52) for more de-tails.

**listextrakeywords**()

Return a set of all extra keywords in self and any parents.

**addfinalizer**(fin)

register a function to be called when this node is finalized.

This method can only be called when this node is active in a setup chain, for example during self.setup().

**getparent**(cls)

get the next parent node (including ourself) which is an instance of the given class

**Parser**

class **Parser**

Parser for command line arguments and ini-file values.

Variables **extra\_info** – dict of generic param -> value to display in case there’s an error pro-cessing the command line arguments.

**getgroup**(name,description=’‘,after=None)

get (or create) a named option Group.

Name name of the option group.

Description long description for –help output.

After name of other group, used for ordering –help output.

The returned group object has an addoption method with the same signature as parser.addoption but will be shown in the respective group in the output of pytest. --help.

**142** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

**addoption**(\*opts,\*\*attrs)

register a command line option.

Opts option names, can be short or long options.

Attrs same attributes which the add\_option() function of the [argparse library](http://docs.python.org/2/library/argparse.html) accepts.

After command line parsing options are available on the pytest config object via config.option.NAME where NAME is usually set by passing a dest attribute, for example addoption("--long",dest="NAME",...).

**parse\_known\_args**(args,namespace=None)

parses and returns a namespace object with known arguments at this point.

**parse\_known\_and\_unknown\_args**(args,namespace=None)

parses and returns a namespace object with known arguments, and the remaining arguments unknown at this point.

**addini**(name,help,type=None,default=None)

register an ini-file option.

Name name of the ini-variable

Type type of the variable, can be pathlist, args, linelist or bool.

Default default value if no ini-file option exists but is queried.

The value of ini-variables can be retrieved via a call to [config.getini(name)](#page144).

**PluginManager**

class **PluginManager**

Core Pluginmanager class which manages registration of plugin objects and 1:N hook calling.

You can register new hooks by calling add\_hookspec(module\_or\_class). You can register plugin objects (which contain hooks) by calling register(plugin). The Pluginmanager is initialized with a prefix that is searched for in the names of the dict of registered plugin objects. An optional excludefunc allows to blacklist names which are not considered as hooks despite a matching prefix.

For debugging purposes you can call enable\_tracing() which will subsequently send debug information to the trace helper.

**register**(plugin,name=None)

Register a plugin and return its canonical name or None if the name is blocked from registering. Raise a ValueError if the plugin is already registered.

**unregister**(plugin=None,name=None)

unregister a plugin object and all its contained hook implementations from internal data structures.

**set\_blocked**(name)

block registrations of the given name, unregister if already registered.

**is\_blocked**(name)

return True if the name blogs registering plugins of that name.

**add\_hookspecs**(module\_or\_class)

add new hook specifications defined in the given module\_or\_class. Functions are recognized if they have been decorated accordingly.

**get\_plugins**()

return the set of registered plugins.

**22.5. Objects** **143**

**pytest Documentation, Release 3.6**

**is\_registered**(plugin)

Return True if the plugin is already registered.

**get\_canonical\_name**(plugin)

Return canonical name for a plugin object. Note that a plugin may be registered under a different name which was specified by the caller of register(plugin, name). To obtain the name of an registered plugin use get\_name(plugin) instead.

**get\_plugin**(name)

Return a plugin or None for the given name.

**has\_plugin**(name)

Return True if a plugin with the given name is registered.

**get\_name**(plugin)

Return name for registered plugin or None if not registered.

**check\_pending**()

Verify that all hooks which have not been verified against a hook specification are optional, otherwise raise PluginValidationError

**load\_setuptools\_entrypoints**(entrypoint\_name)

Load modules from querying the specified setuptools entrypoint name. Return the number of loaded plugins.

**list\_plugin\_distinfo**()

return list of distinfo/plugin tuples for all setuptools registered plugins.

**list\_name\_plugin**()

return list of name/plugin pairs.

**get\_hookcallers**(plugin)

get all hook callers for the specified plugin.

**add\_hookcall\_monitoring**(before,after)

add before/after tracing functions for all hooks and return an undo function which, when called, will remove the added tracers.

before(hook\_name,hook\_impls,kwargs) will be called ahead of all hook calls and receive a hookcaller instance, a list of HookImpl instances and the keyword arguments for the hook call.

after(outcome,hook\_name,hook\_impls,kwargs) receives the same arguments as before but also a \_Result` object which represents the result of the overall hook call.

**enable\_tracing**()

enable tracing of hook calls and return an undo function.

**subset\_hook\_caller**(name,remove\_plugins)

Return a new \_HookCaller instance for the named method which manages calls to all registered plugins except the ones from remove\_plugins.

**PytestPluginManager**

class **PytestPluginManager**

Bases: [pluggy.PluginManager](#page151)

Overwrites [pluggy.PluginManager](#page151) to add pytest-specific functionality:

•loading plugins from the command line, PYTEST\_PLUGINS env variable and pytest\_plugins global variables found in plugins being loaded;

•conftest.py loading during start-up;

**144** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

**addhooks**(module\_or\_class)

Deprecated since version 2.8.

Use pluggy.PluginManager.add\_hookspecs instead.

**parse\_hookimpl\_opts**(plugin,name)

**parse\_hookspec\_opts**(module\_or\_class,name)

**register**(plugin,name=None)

**getplugin**(name)

**hasplugin**(name)

Return True if the plugin with the given name is registered.

**pytest\_configure**(config)

**consider\_preparse**(args)

**consider\_pluginarg**(arg)

**consider\_conftest**(conftestmodule)

**consider\_env**()

**consider\_module**(mod)

**import\_plugin**(modname)

**Session**

class **Session**

Bases: [\_pytest.nodes.FSCollector](#page145)

exception **Interrupted**

Bases: [KeyboardInterrupt](https://docs.python.org/3/library/exceptions.html#KeyboardInterrupt)

signals an interrupted test run.

exception Session.**Failed**

Bases: [Exception](https://docs.python.org/3/library/exceptions.html#Exception)

signals a stop as failed test run.

**TestReport**

class **TestReport**

Basic test report object (also used for setup and teardown calls if they fail).

**nodeid** = None

normalized collection node id

**location** = None

a (filesystempath, lineno, domaininfo) tuple indicating the actual location of a test item - it might be different from the collected one e.g. if a method is inherited from a different module.

**keywords** = None

a name -> value dictionary containing all keywords and markers associated with a test invocation.

**outcome** = None

test outcome, always one of “passed”, “failed”, “skipped”.

**22.5. Objects** **145**

**pytest Documentation, Release 3.6**

**longrepr** = None

None or a failure representation.

**when** = None

one of ‘setup’, ‘call’, ‘teardown’ to indicate runtest phase.

**user\_properties** = None

user properties is a list of tuples (name, value) that holds user defined properties of the test

**sections** = None

list of pairs (str,str) of extra information which needs to marshallable. Used by pytest to add captured text from stdout and stderr, but may be used by other plugins to add arbitrary information to reports.

**duration** = None

time it took to run just the test

**caplog**

Return captured log lines, if log capturing is enabled

New in version 3.5.

**capstderr**

Return captured text from stderr, if capturing is enabled

New in version 3.0.

**capstdout**

Return captured text from stdout, if capturing is enabled

New in version 3.0.

**longreprtext**

Read-only property that returns the full string representation of longrepr.

New in version 3.0.

**\_Result**

class **\_Result**(result, excinfo)

**result**

Get the result(s) for this hook call (DEPRECATED in favor of get\_result()).

**force\_result**(result)

Force the result(s) to result.

If the hook was marked as a firstresult a single value should be set otherwise set a (modified) list of results. Any exceptions found during invocation will be deleted.

**get\_result**()

Get the result(s) for this hook call.

If the hook was marked as a firstresult only a single value will be returned otherwise a list of results.

**Special Variables**

pytest treats some global variables in a special manner when defined in a test module.

**146** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

**pytest\_plugins**

Tutorial: [Requiring/Loading plugins in a test module or conftest file](#page99)

Can be declared at the global level in test modules and conftest.py files to register additional plugins. Can be either a str or Sequence[str].

pytest\_plugins = "myapp.testsupport.myplugin"

pytest\_plugins = ("myapp.testsupport.tools", "myapp.testsupport.regression")

**pytest\_mark**

Tutorial: [Marking whole classes or modules](#page214)

Can be declared at the global level in test modules to apply one or more [marks](#page123) to all test functions and methods. Can be either a single mark or a sequence of marks.

**import pytest**

pytestmark = pytest.mark.webtest

**import pytest**

pytestmark = (pytest.mark.integration, pytest.mark.slow)

**PYTEST\_DONT\_REWRITE (module docstring)**

The text PYTEST\_DONT\_REWRITE can be add to any module docstring to disable [assertion rewriting](#page29) for that module.

**Environment Variables**

Environment variables that can be used to change pytest’s behavior.

**PYTEST\_ADDOPTS**

This contains a command-line (parsed by the py:mod:shlex module) that will be prepended to the command line given by the user, see [How to change command line options defaults](#page172) for more information.

**PYTEST\_DEBUG**

When set, pytest will print tracing and debug information.

**PYTEST\_PLUGINS**

Contains comma-separated list of modules that should be loaded as plugins:

**22.7. Environment Variables** **147**

**pytest Documentation, Release 3.6**

export PYTEST\_PLUGINS=mymodule.plugin,xdist

**PYTEST\_CURRENT\_TEST**

This is not meant to be set by users, but is set by pytest internally with the name of the current test so other processes can inspect it, see [PYTEST\_CURRENT\_TEST environment variable](#page199) for more information.

**Configuration Options**

Here is a list of builtin configuration options that may be written in a pytest.ini, tox.ini or setup.cfg file, usually located at the root of your repository. All options must be under a [pytest] section ([tool:pytest] for setup.cfg files).

Configuration file options may be overwritten in the command-line by using -o/--override, which can also be passed multiple times. The expected format is name=value. For example:

pytest -o console\_output\_style=classic -o cache\_dir=/tmp/mycache

**addopts**

Add the specified OPTS to the set of command line arguments as if they had been specified by the user. Example:

if you have this ini file content:

* content of pytest.ini

**[pytest]**

addopts = --maxfail=2 -rf # exit after 2 failures, report fail info

issuing pytest test\_hello.py actually means:

pytest --maxfail=2 -rf test\_hello.py

Default is to add no options.

**cache\_dir**

New in version 3.2.

Sets a directory where stores content of cache plugin. Default directory is .pytest\_cache which is created in [rootdir](#page171). Directory may be relative or absolute path. If setting relative path, then directory is created relative to [rootdir](#page171). Additionally path may contain environment variables, that will be expanded. For more information about cache plugin please refer to [Cache: working with cross-testrun state](#page85).

**confcutdir**

Sets a directory where search upwards for conftest.py files stops. By default, pytest will stop searching for conftest.py files upwards from pytest.ini/tox.ini/setup.cfg of the project if any, or up to the file-system root.

**console\_output\_style**

New in version 3.3.

Sets the console output style while running tests:

•classic: classic pytest output.

•progress: like classic pytest output, but with a progress indicator.

The default is progress, but you can fallback to classic if you prefer or the new mode is causing unex-pected problems:

**148** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

* content of pytest.ini

**[pytest]**

console\_output\_style = classic

**doctest\_encoding**

New in version 3.1.

Default encoding to use to decode text files with docstrings. [See how pytest handles doctests](#page69).

**doctest\_optionflags**

One or more doctest flag names from the standard doctest module. [See how pytest handles doctests](#page69).

**empty\_parameter\_set\_mark**

New in version 3.4.

Allows to pick the action for empty parametersets in parameterization

•skip skips tests with an empty parameterset (default)

•xfail marks tests with an empty parameterset as xfail(run=False)

* content of pytest.ini

**[pytest]**

empty\_parameter\_set\_mark = xfail

Note: The default value of this option is planned to change to xfail in future releases as this is considered less error prone, see [#3155](https://github.com/pytest-dev/pytest/issues/3155) for more details.

**filterwarnings**

New in version 3.1.

Sets a list of filters and actions that should be taken for matched warnings. By default all warnings emitted during the test session will be displayed in a summary at the end of the test session.

* content of pytest.ini

**[pytest]** filterwarnings =

error ignore::DeprecationWarning

This tells pytest to ignore deprecation warnings and turn all other warnings into errors. For more information please refer to [Warnings Capture](#page63).

**junit\_suite\_name**

New in version 3.1.

To set the name of the root test suite xml item, you can configure the junit\_suite\_name option in your config file:

**[pytest]**

junit\_suite\_name = my\_suite

**log\_cli\_date\_format**

New in version 3.3.

Sets a [time.strftime()](https://docs.python.org/3/library/time.html#time.strftime)-compatible string that will be used when formatting dates for live logging.

**[pytest]**

log\_cli\_date\_format = %Y-%m-%d %H:%M:%S

**22.8. Configuration Options** **149**

**pytest Documentation, Release 3.6**

For more information, see [Live Logs](#page113).

**log\_cli\_format**

New in version 3.3.

Sets a [logging](https://docs.python.org/3/library/logging.html#module-logging)-compatible string used to format live logging messages.

**[pytest]**

log\_cli\_format = %(asctime)s %(levelname)s %(message)s

For more information, see [Live Logs](#page113).

**log\_cli\_level**

New in version 3.3.

Sets the minimum log message level that should be captured for live logging. The integer value or the names of the levels can be used.

**[pytest]**

log\_cli\_level = INFO

For more information, see [Live Logs](#page113).

**log\_date\_format**

New in version 3.3.

Sets a [time.strftime()](https://docs.python.org/3/library/time.html#time.strftime)-compatible string that will be used when formatting dates for logging capture.

**[pytest]**

log\_date\_format = %Y-%m-%d %H:%M:%S

For more information, see [Logging](#page111).

**log\_file**

New in version 3.3.

Sets a file name relative to the pytest.ini file where log messages should be written to, in addition to the other logging facilities that are active.

**[pytest]**

log\_file = logs/pytest-logs.txt

For more information, see [Logging](#page111).

**log\_file\_date\_format**

New in version 3.3.

Sets a [time.strftime()](https://docs.python.org/3/library/time.html#time.strftime)-compatible string that will be used when formatting dates for the logging file.

**[pytest]**

log\_file\_date\_format = %Y-%m-%d %H:%M:%S

For more information, see [Logging](#page111).

**log\_file\_format**

New in version 3.3.

Sets a [logging](https://docs.python.org/3/library/logging.html#module-logging)-compatible string used to format logging messages redirected to the logging file.

**[pytest]**

log\_file\_format = %(asctime)s %(levelname)s %(message)s

**150** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

For more information, see [Logging](#page111).

**log\_file\_level**

New in version 3.3.

Sets the minimum log message level that should be captured for the logging file. The integer value or the names of the levels can be used.

**[pytest]**

log\_file\_level = INFO

For more information, see [Logging](#page111).

**log\_format**

New in version 3.3.

Sets a [logging](https://docs.python.org/3/library/logging.html#module-logging)-compatible string used to format captured logging messages.

**[pytest]**

log\_format = %(asctime)s %(levelname)s %(message)s

For more information, see [Logging](#page111).

**log\_level**

New in version 3.3.

Sets the minimum log message level that should be captured for logging capture. The integer value or the names of the levels can be used.

**[pytest]**

log\_level = INFO

For more information, see [Logging](#page111).

**log\_print**

New in version 3.3.

If set to False, will disable displaying captured logging messages for failed tests.

**[pytest]**

log\_print = False

For more information, see [Logging](#page111).

**markers**

List of markers that are allowed in test functions, enforced when --strict command-line argument is used.

You can use a marker name per line, indented from the option name.

**[pytest]**

markers =

slow

serial

**minversion**

Specifies a minimal pytest version required for running tests.

* content of pytest.ini

**[pytest]**

minversion = 3.0 # will fail if we run with pytest-2.8

**22.8. Configuration Options** **151**

**pytest Documentation, Release 3.6**

**norecursedirs**

Set the directory basename patterns to avoid when recursing for test discovery. The individual (fnmatch-style)

patterns are applied to the basename of a directory to decide if to recurse into it. Pattern matching characters:

* matches everything
* matches any single character

[seq] matches any character in seq

[!seq] matches any char not in seq

Default patterns are '.\*','build','dist','CVS','\_darcs','{arch}','\*.egg','venv'.

Setting a norecursedirs replaces the default. Here is an example of how to avoid certain directories:

**[pytest]**

norecursedirs = .svn \_build tmp\*

This would tell pytest to not look into typical subversion or sphinx-build directories or into any tmp prefixed directory.

Additionally, pytest will attempt to intelligently identify and ignore a virtualenv by the presence of an acti-vation script. Any directory deemed to be the root of a virtual environment will not be considered during test collection unless collectinvirtualenv is given. Note also that norecursedirs takes precedence over collectinvirtualenv; e.g. if you intend to run tests in a virtualenv with a base directory that matches '.\*' you must override norecursedirs in addition to using the collectinvirtualenv flag.

**python\_classes**

One or more name prefixes or glob-style patterns determining which classes are considered for test collection. By default, pytest will consider any class prefixed with Test as a test collection. Here is an example of how to collect tests from classes that end in Suite:

**[pytest]**

python\_classes = \*Suite

Note that unittest.TestCase derived classes are always collected regardless of this option, as unittest‘s own collection framework is used to collect those tests.

**python\_files**

One or more Glob-style file patterns determining which python files are considered as test modules. By default, pytest will consider any file matching with test\_\*.py and \*\_test.py globs as a test module.

**python\_functions**

One or more name prefixes or glob-patterns determining which test functions and methods are considered tests. By default, pytest will consider any function prefixed with test as a test. Here is an example of how to collect test functions and methods that end in \_test:

**[pytest]**

python\_functions = \*\_test

Note that this has no effect on methods that live on a unittest .TestCase derived class, as unittest‘s own collection framework is used to collect those tests.

See [Changing naming conventions](#page223) for more detailed examples.

**testpaths**

New in version 2.8.

Sets list of directories that should be searched for tests when no specific directories, files or test ids are given in the command line when executing pytest from the [rootdir](#page171) directory. Useful when all project tests are in a known location to speed up test collection and to avoid picking up undesired tests by accident.

**152** **Chapter 22. Reference**

**pytest Documentation, Release 3.6**

**[pytest]**

testpaths = testing doc

This tells pytest to only look for tests in testing and doc directories when executing from the root directory.

**usefixtures**

List of fixtures that will be applied to all test functions; this is semantically the same to apply the @pytest.mark.usefixtures marker to all test functions.

**[pytest]**

usefixtures =

clean\_db

**xfail\_strict**

If set to True, tests marked with @pytest.mark.xfail that actually succeed will by default fail the test suite. For more information, see [strict parameter](#page76).

**[pytest]**

xfail\_strict = True

**22.8. Configuration Options** **153**

**pytest Documentation, Release 3.6**

**154** **Chapter 22. Reference**

**CHAPTER 23**

**Good Integration Practices**

**Conventions for Python test discovery**

pytest implements the following standard test discovery:

* If no arguments are specified then collection starts from [testpaths](#page160) (if configured) or the current directory. Alternatively, command line arguments can be used in any combination of directories, file names or node ids.
* Recurse into directories, unless they match [norecursedirs](#page159).
* In those directories, search for test\_\*.py or \*\_test.py files, imported by their [test package name](#page165).
* From those files, collect test items:

– test\_ prefixed test functions or methods outside of class

– test\_ prefixed test functions or methods inside Test prefixed test classes (without an \_\_init\_\_ method)

For examples of how to customize your test discovery [Changing standard (Python) test discovery](#page222).

Within Python modules, pytest also discovers tests using the standard [unittest.TestCase](#page91) subclassing technique.

**Choosing a test layout / import rules**

pytest supports two common test layouts:

**Tests outside application code**

Putting tests into an extra directory outside your actual application code might be useful if you have many functional tests or for other reasons want to keep tests separate from actual application code (often a good idea):

setup.py

mypkg/

\_\_init\_\_.py

app.py

view.py

tests/

test\_app.py

test\_view.py

...

**155**

**pytest Documentation, Release 3.6**

This way your tests can run easily against an installed version of mypkg.

Note that using this scheme your test files must have unique names, because pytest will import them as top-level modules since there are no packages to derive a full package name from. In other words, the test files in the example above will be imported as test\_app and test\_view top-level modules by adding tests/ to sys.path.

If you need to have test modules with the same name, you might add \_\_init\_\_.py files to your tests folder and subfolders, changing them to packages:

setup.py

mypkg/

...

tests/

\_\_init\_\_.py

foo/

\_\_init\_\_.py

test\_view.py

bar/

\_\_init\_\_.py

test\_view.py

Now pytest will load the modules as tests.foo.test\_view and tests.bar.test\_view, allowing you to have modules with the same name. But now this introduces a subtle problem: in order to load the test modules from the tests directory, pytest prepends the root of the repository to sys.path, which adds the side-effect that now mypkg is also importable. This is problematic if you are using a tool like [tox](http://testrun.org/tox) to test your package in a virtual environment, because you want to test the installed version of your package, not the local code from the repository.

In this situation, it is strongly suggested to use a src layout where application root package resides in a sub-directory of your root:

setup.py

src/

mypkg/

\_\_init\_\_.py

app.py

view.py

tests/

\_\_init\_\_.py

foo/

\_\_init\_\_.py

test\_view.py

bar/

\_\_init\_\_.py

test\_view.py

This layout prevents a lot of common pitfalls and has many benefits, which are better explained in this excellent [blog](https://blog.ionelmc.ro/2014/05/25/python-packaging/#the-structure) [post by Ionel Cristian Maries˘,](https://blog.ionelmc.ro/2014/05/25/python-packaging/#the-structure).

**Tests as part of application code**

Inlining test directories into your application package is useful if you have direct relation between tests and application modules and want to distribute them along with your application:

setup.py

mypkg/

\_\_init\_\_.py

app.py

**156** **Chapter 23. Good Integration Practices**

**pytest Documentation, Release 3.6**

view.py

test/

\_\_init\_\_.py

test\_app.py

test\_view.py

...

In this scheme, it is easy to run your tests using the --pyargs option:

pytest --pyargs mypkg

pytest will discover where mypkg is installed and collect tests from there.

Note that this layout also works in conjunction with the src layout mentioned in the previous section.

Note: You can use Python3 namespace packages (PEP420) for your application but pytest will still perform [test](#page165) [package name](#page165) discovery based on the presence of \_\_init\_\_.py files. If you use one of the two recommended file system layouts above but leave away the \_\_init\_\_.py files from your directories it should just work on Python3.3 and above. From “inlined tests”, however, you will need to use absolute imports for getting at your application code.

Note: If pytest finds an “a/b/test\_module.py” test file while recursing into the filesystem it determines the import name as follows:

* determine basedir: this is the first “upward” (towards the root) directory not containing an \_\_init\_\_.py. If e.g. both a and b contain an \_\_init\_\_.py file then the parent directory of a will become the basedir.
* perform sys.path.insert(0,basedir) to make the test module importable under the fully qualified import name.
* import a.b.test\_module where the path is determined by converting path separators / into ”.” charac-ters. This means you must follow the convention of having directory and file names map directly to the import names.

The reason for this somewhat evolved importing technique is that in larger projects multiple test modules might import from each other and thus deriving a canonical import name helps to avoid surprises such as a test module getting imported twice.

**tox**

For development, we recommend to use [virtualenv](https://pypi.org/project/virtualenv/) environments and [pip](https://pypi.org/project/pip/) for installing your application and any de-pendencies as well as the pytest package itself. This ensures your code and dependencies are isolated from the system Python installation.

You can then install your package in “editable” mode:

pip install -e .

which lets you change your source code (both tests and application) and rerun tests at will. This is similar to run-ning python setup.py develop or conda develop in that it installs your package using a symlink to your development code.

Once you are done with your work and want to make sure that your actual package passes all tests you may want to look into [tox](http://testrun.org/tox), the virtualenv test automation tool and its [pytest support](https://tox.readthedocs.io/en/latest/example/pytest.html). tox helps you to setup virtualenv environments

**23.3. tox** **157**

**pytest Documentation, Release 3.6**

with pre-defined dependencies and then executing a pre-configured test command with options. It will run tests against the installed package and not against your source code checkout, helping to detect packaging glitches.

**Integrating with setuptools /** **python setup.py test** **/**

**pytest-runner**

You can integrate test runs into your setuptools based project with the [pytest-runner](https://pypi.org/project/pytest-runner/) plugin.

Add this to setup.py file:

**from setuptools import** setup

setup(

# ...,

setup\_requires=["pytest-runner", ...],

tests\_require=["pytest", ...],

# ...,

)

And create an alias into setup.cfg file:

**[aliases]**

test=pytest

If you now type:

python setup.py test

this will execute your tests using pytest-runner. As this is a standalone version of pytest no prior installation whatsoever is required for calling the test command. You can also pass additional arguments to pytest such as your test directory or other options using --addopts.

You can also specify other pytest-ini options in your setup.cfg file by putting them into a [tool:pytest] section:

**[tool:pytest]**

addopts = --verbose

python\_files = testing/\*/\*.py

**Manual Integration**

If for some reason you don’t want/can’t use pytest-runner, you can write your own setuptools Test command for invoking pytest.

**import sys**

**from setuptools.command.test import** test **as** TestCommand

**class PyTest**(TestCommand):

user\_options = [("pytest-args=", "a", "Arguments to pass to pytest")]

**def** initialize\_options(self):

TestCommand.initialize\_options(self)

**158** **Chapter 23. Good Integration Practices**

**pytest Documentation, Release 3.6**

self.pytest\_args = ""

**def** run\_tests(self):

**import shlex**

* import here, cause outside the eggs aren't loaded **import pytest**

errno = pytest.main(shlex.split(self.pytest\_args))

sys.exit(errno)

setup(

# ...,

tests\_require=["pytest"],

cmdclass={"test": PyTest},

)

Now if you run:

python setup.py test

this will download pytest if needed and then run your tests as you would expect it to. You can pass a single string of arguments using the --pytest-args or -a command-line option. For example:

python setup.py test -a "--durations=5"

is equivalent to running pytest --durations=5.

**23.4. Integrating with setuptools / python setup.py test / pytest-runner** **159**

**pytest Documentation, Release 3.6**

**160** **Chapter 23. Good Integration Practices**

**CHAPTER 24**

**pytest import mechanisms and sys.path/PYTHONPATH**

Here’s a list of scenarios where pytest may need to change sys.path in order to import test modules or conftest.py files.

**Test modules / conftest.py files inside packages**

Consider this file and directory layout:

root/

|- foo/

|- \_\_init\_\_.py

|- conftest.py

|- bar/

|- \_\_init\_\_.py

|- tests/

|- \_\_init\_\_.py

|- test\_foo.py

When executing:

pytest root/

pytest will find foo/bar/tests/test\_foo.py and realize it is part of a package given that there’s an \_\_init\_\_.py file in the same folder. It will then search upwards until it can find the last folder which still contains an \_\_init\_\_.py file in order to find the package root (in this case foo/). To load the module, it will insert root/ to the front of sys.path (if not there already) in order to load test\_foo.py as the module foo.bar.tests.test\_foo.

The same logic applies to the conftest.py file: it will be imported as foo.conftest module.

Preserving the full package name is important when tests live in a package to avoid problems and allow test modules to have duplicated names. This is also discussed in details in [Conventions for Python test discovery](#page163).

**Standalone test modules / conftest.py files**

Consider this file and directory layout:

root/

|- foo/

**161**

**pytest Documentation, Release 3.6**

|- conftest.py

|- bar/

|- tests/

|- test\_foo.py

When executing:

pytest root/

pytest will find foo/bar/tests/test\_foo.py and realize it is NOT part of a package given that there’s no \_\_init\_\_.py file in the same folder. It will then add root/foo/bar/tests to sys.path in order to import test\_foo.py as the module test\_foo. The same is done with the conftest.py file by adding root/foo to sys.path to import it as conftest.

For this reason this layout cannot have test modules with the same name, as they all will be imported in the global import namespace.

This is also discussed in details in [Conventions for Python test discovery](#page163).

**Invoking pytest versus python -m pytest**

Running pytest with python -m pytest [...] instead of pytest [...] yields nearly equivalent behaviour, except that the former call will add the current directory to sys.path. See also [Calling pytest through python -m](#page15) [pytest](#page15).

**162** **Chapter 24. pytest import mechanisms and sys.path/PYTHONPATH**

**CHAPTER 25**

**Configuration**

**Command line options and configuration file settings**

You can get help on command line options and values in INI-style configurations files by using the general help option:

pytest -h # prints options \_and\_ config file settings

This will display command line and configuration file settings which were registered by installed plugins.

**Initialization: determining rootdir and inifile**

New in version 2.7.

pytest determines a rootdir for each test run which depends on the command line arguments (specified test files, paths) and on the existence of ini-files. The determined rootdir and ini-file are printed as part of the pytest header during startup.

Here’s a summary what pytest uses rootdir for:

* Construct nodeids during collection; each test is assigned a unique nodeid which is rooted at the rootdir and takes in account full path, class name, function name and parametrization (if any).
* Is used by plugins as a stable location to store project/test run specific information; for example, the internal [cache](#page85) plugin creates a .cache subdirectory in rootdir to store its cross-test run state.

Important to emphasize that rootdir is NOT used to modify sys.path/PYTHONPATH or influence how modules are imported. See [pytest import mechanisms and sys.path/PYTHONPATH](#page169) for more details.

--rootdir=path command-line option can be used to force a specific directory. The directory passed may contain environment variables when it is used in conjunction with addopts in a pytest.ini file.

**Finding the rootdir**

Here is the algorithm which finds the rootdir from args:

* determine the common ancestor directory for the specified args that are recognised as paths that exist in the file system. If no such paths are found, the common ancestor directory is set to the current working directory.
* look for pytest.ini, tox.ini and setup.cfg files in the ancestor directory and upwards. If one is matched, it becomes the ini-file and its directory becomes the rootdir.

**163**

**pytest Documentation, Release 3.6**

* if no ini-file was found, look for setup.py upwards from the common ancestor directory to determine the rootdir.
* if no setup.py was found, look for pytest.ini, tox.ini and setup.cfg in each of the specified args and upwards. If one is matched, it becomes the ini-file and its directory becomes the rootdir.
* if no ini-file was found, use the already determined common ancestor as root directory. This allows the use of pytest in structures that are not part of a package and don’t have any particular ini-file configuration.

If no args are given, pytest collects test below the current working directory and also starts determining the rootdir from there.

warning custom pytest plugin commandline arguments may include a path, as in pytest --log-output ../../test.log args. Then args is mandatory, otherwise pytest uses the folder of test.log for rootdir determination (see also [issue 1435](https://github.com/pytest-dev/pytest/issues/1435)). A dot . for referencing to the current working directory is also possible.

Note that an existing pytest.ini file will always be considered a match, whereas tox.ini and setup.cfg will only match if they contain a [pytest] or [tool:pytest] section, respectively. Options from multiple ini-files candidates are never merged - the first one wins (pytest.ini always wins, even if it does not contain a [pytest] section).

The config object will subsequently carry these attributes:

* config.rootdir: the determined root directory, guaranteed to exist.
* config.inifile: the determined ini-file, may be None.

The rootdir is used a reference directory for constructing test addresses (“nodeids”) and can be used also by plugins for storing per-testrun information.

Example:

pytest path/to/testdir path/other/

will determine the common ancestor as path and then check for ini-files as follows:

* first look for pytest.ini files path/pytest.ini

path/setup.cfg # must also contain [tool:pytest] section to match

path/tox.ini # must also contain [pytest] section to match pytest.ini

... # all the way down to the root

* now look for setup.py

path/setup.py

setup.py

... # all the way down to the root

**How to change command line options defaults**

It can be tedious to type the same series of command line options every time you use pytest. For example, if you always want to see detailed info on skipped and xfailed tests, as well as have terser “dot” progress output, you can write it into a configuration file:

* content of pytest.ini
* (or tox.ini or setup.cfg)

**[pytest]**

addopts = -ra -q

**164** **Chapter 25. Configuration**

**pytest Documentation, Release 3.6**

Alternatively, you can set a PYTEST\_ADDOPTS environment variable to add command line options while the envi-ronment is in use:

export PYTEST\_ADDOPTS="-v"

Here’s how the command-line is built in the presence of addopts or the environment variable:

<pytest.ini:addopts> $PYTEST\_ADDOPTS <extra command-line arguments>

So if the user executes in the command-line:

pytest -m slow

The actual command line executed is:

pytest -ra -q -v -m slow

Note that as usual for other command-line applications, in case of conflicting options the last one wins, so the example above will show verbose output because -v overwrites -q.

**Builtin configuration file options**

For the full list of options consult the [reference documentation](#page156).

**25.4. Builtin configuration file options** **165**

**pytest Documentation, Release 3.6**

**166** **Chapter 25. Configuration**

**CHAPTER 26**

**Examples and customization tricks**

Here is a (growing) list of examples. [Contact](#page255) us if you need more examples or have questions. Also take a look at the comprehensive documentation which contains many example snippets as well. Also, [pytest on stackoverflow.com](http://stackoverflow.com/search?q=pytest) often comes with example answers.

For basic examples, see

* [Installation and Getting Started](#page11) for basic introductory examples
* [Asserting with the assert statement](#page25) for basic assertion examples
* [pytest fixtures: explicit, modular, scalable](#page31) for basic fixture/setup examples
* [Parametrizing fixtures and test functions](#page81) for basic test function parametrization
* [unittest.TestCase Support](#page91) for basic unittest integration
* [Running tests written for nose](#page95) for basic nosetests integration

The following examples aim at various use cases you might encounter.

**Demo of Python failure reports with pytest**

Here is a nice run of several tens of failures and how pytest presents things (unfortunately not showing the nice colors here in the HTML that you get on the terminal - we are working on that):

assertion $ pytest failure\_demo.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/assertion, inifile: collected 42 items

|  |  |
| --- | --- |
| failure\_demo.py FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_generative[0] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

param1 = 3, param2 = 6

def test\_generative(param1, param2):

* assert param1 \* 2 < param2

Eassert (3 \* 2) < 6

failure\_demo.py:19: AssertionError

**167**

**pytest Documentation, Release 3.6**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestFailing.test\_simple \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestFailing object at 0xdeadbeef>

def test\_simple(self):

def f():

return 42

def g():

return 43

* assert f() == g()

Eassert 42 == 43

E+ where 42 = <function TestFailing.test\_simple.<locals>.f at 0xdeadbeef>()

E + and 43 = <function TestFailing.test\_simple.<locals>.g at 0xdeadbeef>()

failure\_demo.py:35: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestFailing.test\_simple\_multiline \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestFailing object at 0xdeadbeef>

def test\_simple\_multiline(self):

* otherfunc\_multi(42, 6 \* 9)

failure\_demo.py:38:

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

a = 42, b = 54

def otherfunc\_multi(a, b):

* assert a == b
* assert 42 == 54

failure\_demo.py:15: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestFailing.test\_not \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestFailing object at 0xdeadbeef>

def test\_not(self):

def f():

return 42

* assert not f()

Eassert not 42

E+ where 42 = <function TestFailing.test\_not.<locals>.f at 0xdeadbeef>()

failure\_demo.py:44: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestSpecialisedExplanations.test\_eq\_text \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_eq\_text(self):

* assert "spam" == "eggs"
* AssertionError: assert 'spam' == 'eggs'
* - spam
* + eggs

failure\_demo.py:49: AssertionError

**168** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

\_\_\_\_\_\_\_\_\_\_\_\_\_ TestSpecialisedExplanations.test\_eq\_similar\_text \_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_eq\_similar\_text(self):

* assert "foo 1 bar" == "foo 2 bar"
* AssertionError: assert 'foo 1 bar' == 'foo 2 bar'
* - foo 1 bar
* ?^
* + foo 2 bar
* ?^

failure\_demo.py:52: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_ TestSpecialisedExplanations.test\_eq\_multiline\_text \_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_eq\_multiline\_text(self):

* assert "foo\nspam\nbar" == "foo\neggs\nbar"
* AssertionError: assert 'foo\nspam\nbar' == 'foo\neggs\nbar'
* foo
* - spam
* + eggs
* bar

failure\_demo.py:55: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestSpecialisedExplanations.test\_eq\_long\_text \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_eq\_long\_text(self):

a = "1" \* 100 + "a" + "2" \* 100

b = "1" \* 100 + "b" + "2" \* 100

* assert a == b
* AssertionError: assert '111111111111...2222222222222' == '1111111111111...

˓→2222222222222'

* Skipping 90 identical leading characters in diff, use -v to show
* Skipping 91 identical trailing characters in diff, use -v to show
* - 1111111111a222222222

E ? ^

* + 1111111111b222222222
* ?^

failure\_demo.py:60: AssertionError

\_\_\_\_\_\_\_\_\_ TestSpecialisedExplanations.test\_eq\_long\_text\_multiline \_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_eq\_long\_text\_multiline(self):

a = "1\n" \* 100 + "a" + "2\n" \* 100

b = "1\n" \* 100 + "b" + "2\n" \* 100

* assert a == b
* AssertionError: assert '1\n1\n1\n1\n...n2\n2\n2\n2\n' == '1\n1\n1\n1\n1...

˓→n2\n2\n2\n2\n'

* Skipping 190 identical leading characters in diff, use -v to show
* Skipping 191 identical trailing characters in diff, use -v to show
* 1
* 1

**26.1. Demo of Python failure reports with pytest** **169**

**pytest Documentation, Release 3.6**

* 1
* 1
* 1...
* ...Full output truncated (7 lines hidden), use '-vv' to show

failure\_demo.py:65: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestSpecialisedExplanations.test\_eq\_list \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_eq\_list(self):

* assert [0, 1, 2] == [0, 1, 3]

Eassert [0, 1, 2] == [0, 1, 3]

EAt index 2 diff: 2 != 3

EUse -v to get the full diff

failure\_demo.py:68: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestSpecialisedExplanations.test\_eq\_list\_long \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_eq\_list\_long(self):

a = [0] \* 100 + [1] + [3] \* 100

b = [0] \* 100 + [2] + [3] \* 100

* assert a == b
* assert [0, 0, 0, 0, 0, 0, ...] == [0, 0, 0, 0, 0, 0, ...]
* At index 100 diff: 1 != 2
* Use -v to get the full diff

failure\_demo.py:73: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestSpecialisedExplanations.test\_eq\_dict \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_eq\_dict(self):

* assert {"a": 0, "b": 1, "c": 0} == {"a": 0, "b": 2, "d": 0}
* AssertionError: assert {'a': 0, 'b': 1, 'c': 0} == {'a': 0, 'b': 2, 'd': 0}
* Omitting 1 identical items, use -vv to show
* Differing items:
* {'b': 1} != {'b': 2}
* Left contains more items:
* {'c': 0}
* Right contains more items:
* {'d': 0}...

E

* ...Full output truncated (2 lines hidden), use '-vv' to show

failure\_demo.py:76: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestSpecialisedExplanations.test\_eq\_set \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_eq\_set(self):

* assert {0, 10, 11, 12} == {0, 20, 21}
* AssertionError: assert {0, 10, 11, 12} == {0, 20, 21}
* Extra items in the left set:
* 10

**170** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

* 11
* 12
* Extra items in the right set:
* 20
* 21...

E

* ...Full output truncated (2 lines hidden), use '-vv' to show

failure\_demo.py:79: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_ TestSpecialisedExplanations.test\_eq\_longer\_list \_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_eq\_longer\_list(self):

* assert [1, 2] == [1, 2, 3]

Eassert [1, 2] == [1, 2, 3]

ERight contains more items, first extra item: 3

EUse -v to get the full diff

failure\_demo.py:82: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestSpecialisedExplanations.test\_in\_list \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_in\_list(self):

* assert 1 in [0, 2, 3, 4, 5]

Eassert 1 in [0, 2, 3, 4, 5]

failure\_demo.py:85: AssertionError

\_\_\_\_\_\_\_\_\_\_ TestSpecialisedExplanations.test\_not\_in\_text\_multiline \_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_not\_in\_text\_multiline(self):

text = "some multiline\ntext\nwhich\nincludes foo\nand a\ntail"

* assert "foo" not in text
* AssertionError: assert 'foo' not in 'some multiline\ntext\nw...ncludes ˓→foo\nand a\ntail'
* 'foo' is contained here:
* some multiline
* text
* which
* includes foo

E ? +++

* and a...
* ...Full output truncated (2 lines hidden), use '-vv' to show

failure\_demo.py:89: AssertionError

\_\_\_\_\_\_\_\_\_\_\_ TestSpecialisedExplanations.test\_not\_in\_text\_single \_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_not\_in\_text\_single(self):

text = "single foo line"

* assert "foo" not in text
* AssertionError: assert 'foo' not in 'single foo line'
* 'foo' is contained here:

**26.1. Demo of Python failure reports with pytest** **171**

**pytest Documentation, Release 3.6**

* single foo line
* ?+++

failure\_demo.py:93: AssertionError

\_\_\_\_\_\_\_\_\_ TestSpecialisedExplanations.test\_not\_in\_text\_single\_long \_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_not\_in\_text\_single\_long(self):

text = "head " \* 50 + "foo " + "tail " \* 20

* assert "foo" not in text
* AssertionError: assert 'foo' not in 'head head head head hea...ail tail tail ˓→tail tail '
* 'foo' is contained here:
* head head foo tail tail tail tail tail tail tail tail tail tail tail tail ˓→tail tail tail tail tail tail tail tail
* ?+++

failure\_demo.py:97: AssertionError

\_\_\_\_\_\_ TestSpecialisedExplanations.test\_not\_in\_text\_single\_long\_term \_\_\_\_\_\_\_

self = <failure\_demo.TestSpecialisedExplanations object at 0xdeadbeef>

def test\_not\_in\_text\_single\_long\_term(self):

text = "head " \* 50 + "f" \* 70 + "tail " \* 20

* assert "f" \* 70 not in text
* AssertionError: assert 'fffffffffff...ffffffffffff' not in 'head head he...l ˓→tail tail '
* 'ffffffffffffffffff...fffffffffffffffffff' is contained here:
* head head

˓→fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffftail tail ˓→tail tail tail tail tail tail tail tail tail tail tail tail tail tail tail tail ˓→tail tail

* ?

˓→++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

failure\_demo.py:101: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_attribute \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_attribute():

class Foo(object):

b = 1

i = Foo()

* assert i.b == 2

Eassert 1 == 2

E+ where 1 = <failure\_demo.test\_attribute.<locals>.Foo object at 0xdeadbeef>.

˓→b

failure\_demo.py:109: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_attribute\_instance \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_attribute\_instance():

class Foo(object):

b = 1

* assert Foo().b == 2

E AssertionError: assert 1 == 2

**172** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

* + where 1 = <failure\_demo.test\_attribute\_instance.<locals>.Foo object at ˓→0xdeadbeef>.b
* +where <failure\_demo.test\_attribute\_instance.<locals>.Foo object at

˓→0xdeadbeef> = <class 'failure\_demo.test\_attribute\_instance.<locals>.Foo'>()

failure\_demo.py:116: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_attribute\_failure \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_attribute\_failure():

class Foo(object):

def \_get\_b(self):

raise Exception("Failed to get attrib")

b = property(\_get\_b)

i = Foo()

* assert i.b == 2

failure\_demo.py:127:

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

self = <failure\_demo.test\_attribute\_failure.<locals>.Foo object at 0xdeadbeef>

def \_get\_b(self):

* raise Exception("Failed to get attrib")

EException: Failed to get attrib

failure\_demo.py:122: Exception

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_attribute\_multiple \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_attribute\_multiple():

class Foo(object):

b = 1

class Bar(object):

b = 2

* assert Foo().b == Bar().b
* AssertionError: assert 1 == 2
* + where 1 = <failure\_demo.test\_attribute\_multiple.<locals>.Foo object at ˓→0xdeadbeef>.b
* +where <failure\_demo.test\_attribute\_multiple.<locals>.Foo object at

˓→0xdeadbeef> = <class 'failure\_demo.test\_attribute\_multiple.<locals>.Foo'>()

E + and 2 = <failure\_demo.test\_attribute\_multiple.<locals>.Bar object at ˓→0xdeadbeef>.b

* + where <failure\_demo.test\_attribute\_multiple.<locals>.Bar object at ˓→0xdeadbeef> = <class 'failure\_demo.test\_attribute\_multiple.<locals>.Bar'>()

failure\_demo.py:137: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestRaises.test\_raises \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestRaises object at 0xdeadbeef>

def test\_raises(self):

s = "qwe" # NOQA

* raises(TypeError, "int(s)") failure\_demo.py:147:

**26.1. Demo of Python failure reports with pytest** **173**

**pytest Documentation, Release 3.6**

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

* int(s)
* ValueError: invalid literal for int() with base 10: 'qwe'

<0-codegen $PYTHON\_PREFIX/lib/python3.5/site-packages/\_pytest/python\_api.py:635>:1:

˓→ValueError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestRaises.test\_raises\_doesnt \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestRaises object at 0xdeadbeef>

def test\_raises\_doesnt(self):

* raises(IOError, "int('3')")
* Failed: DID NOT RAISE <class 'OSError'>

failure\_demo.py:150: Failed

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestRaises.test\_raise \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestRaises object at 0xdeadbeef>

def test\_raise(self):

* raise ValueError("demo error")

EValueError: demo error

failure\_demo.py:153: ValueError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestRaises.test\_tupleerror \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestRaises object at 0xdeadbeef>

def test\_tupleerror(self):

* a, b = [1] # NOQA
* ValueError: not enough values to unpack (expected 2, got 1)

failure\_demo.py:156: ValueError

\_\_\_\_\_\_ TestRaises.test\_reinterpret\_fails\_with\_print\_for\_the\_fun\_of\_it \_\_\_\_\_\_

self = <failure\_demo.TestRaises object at 0xdeadbeef>

def test\_reinterpret\_fails\_with\_print\_for\_the\_fun\_of\_it(self):

items = [1, 2, 3]

print("items is %r" % items)

* a, b = items.pop()
* TypeError: 'int' object is not iterable

failure\_demo.py:161: TypeError

--------------------------- Captured stdout call ---------------------------

items is [1, 2, 3]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestRaises.test\_some\_error \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestRaises object at 0xdeadbeef>

def test\_some\_error(self):

* if namenotexi: # NOQA
* NameError: name 'namenotexi' is not defined

failure\_demo.py:164: NameError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_dynamic\_compile\_shows\_nicely \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**174** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

def test\_dynamic\_compile\_shows\_nicely():

import imp

import sys

src = "def foo():\n assert 1 == 0\n"

name = "abc-123"

module = imp.new\_module(name)

code = \_pytest.\_code.compile(src, name, "exec") py.builtin.exec\_(code, module.\_\_dict\_\_) sys.modules[name] = module

* module.foo()

failure\_demo.py:182:

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

def foo():

* assert 1 == 0

EAssertionError

<2-codegen 'abc-123' $REGENDOC\_TMPDIR/assertion/failure\_demo.py:179>:2: AssertionError \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestMoreErrors.test\_complex\_error \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestMoreErrors object at 0xdeadbeef>

def test\_complex\_error(self):

def f():

return 44

def g():

return 43

* somefunc(f(), g())

failure\_demo.py:193:

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

failure\_demo.py:11: in somefunc

otherfunc(x, y)

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

a = 44, b = 43

def otherfunc(a, b):

* assert a == b
* assert 44 == 43

failure\_demo.py:7: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestMoreErrors.test\_z1\_unpack\_error \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestMoreErrors object at 0xdeadbeef>

def test\_z1\_unpack\_error(self):

items = []

* a, b = items
* ValueError: not enough values to unpack (expected 2, got 0)

failure\_demo.py:197: ValueError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestMoreErrors.test\_z2\_type\_error \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**26.1. Demo of Python failure reports with pytest** **175**

**pytest Documentation, Release 3.6**

self = <failure\_demo.TestMoreErrors object at 0xdeadbeef>

def test\_z2\_type\_error(self):

items = 3

* a, b = items
* TypeError: 'int' object is not iterable

failure\_demo.py:201: TypeError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestMoreErrors.test\_startswith \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestMoreErrors object at 0xdeadbeef>

def test\_startswith(self):

s = "123"

g = "456"

* assert s.startswith(g)
* AssertionError: assert False
* + where False = <built-in method startswith of str object at 0xdeadbeef>(

˓→'456')

* + where <built-in method startswith of str object at 0xdeadbeef> = '123'. ˓→startswith

failure\_demo.py:206: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestMoreErrors.test\_startswith\_nested \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestMoreErrors object at 0xdeadbeef>

def test\_startswith\_nested(self):

def f():

return "123"

def g():

return "456"

* assert f().startswith(g())
* AssertionError: assert False
* + where False = <built-in method startswith of str object at 0xdeadbeef>(

˓→'456')

* + where <built-in method startswith of str object at 0xdeadbeef> = '123'. ˓→startswith
* + where '123' = <function TestMoreErrors.test\_startswith\_nested.<locals> ˓→.f at 0xdeadbeef>()

E + and '456' = <function TestMoreErrors.test\_startswith\_nested.<locals>. ˓→g at 0xdeadbeef>()

failure\_demo.py:215: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestMoreErrors.test\_global\_func \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestMoreErrors object at 0xdeadbeef>

def test\_global\_func(self):

* assert isinstance(globf(42), float)

Eassert False

E+ where False = isinstance(43, float)

* +where 43 = globf(42)

failure\_demo.py:218: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestMoreErrors.test\_instance \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**176** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

self = <failure\_demo.TestMoreErrors object at 0xdeadbeef>

def test\_instance(self):

self.x = 6 \* 7

* assert self.x != 42

Eassert 42 != 42

E+ where 42 = <failure\_demo.TestMoreErrors object at 0xdeadbeef>.x

failure\_demo.py:222: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestMoreErrors.test\_compare \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestMoreErrors object at 0xdeadbeef>

def test\_compare(self):

* assert globf(10) < 5

Eassert 11 < 5

E+ where 11 = globf(10)

failure\_demo.py:225: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestMoreErrors.test\_try\_finally \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestMoreErrors object at 0xdeadbeef>

def test\_try\_finally(self):

x = 1

try:

* assert x == 0
* assert 1 == 0

failure\_demo.py:230: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestCustomAssertMsg.test\_single\_line \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestCustomAssertMsg object at 0xdeadbeef>

def test\_single\_line(self):

class A(object):

a = 1

b = 2

* assert A.a == b, "A.a appears not to be b"

EAssertionError: A.a appears not to be b

Eassert 1 == 2

E+ where 1 = <class 'failure\_demo.TestCustomAssertMsg.test\_single\_line. ˓→<locals>.A'>.a

failure\_demo.py:241: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestCustomAssertMsg.test\_multiline \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestCustomAssertMsg object at 0xdeadbeef>

def test\_multiline(self):

class A(object):

a = 1

b = 2

* assert ( A.a == b

**26.1. Demo of Python failure reports with pytest** **177**

**pytest Documentation, Release 3.6**

), "A.a appears not to be b\n" "or does not appear to be b\none of those"

* AssertionError: A.a appears not to be b
* or does not appear to be b
* one of those
* assert 1 == 2
* + where 1 = <class 'failure\_demo.TestCustomAssertMsg.test\_multiline.<locals>

˓→.A'>.a

failure\_demo.py:248: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestCustomAssertMsg.test\_custom\_repr \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <failure\_demo.TestCustomAssertMsg object at 0xdeadbeef>

def test\_custom\_repr(self):

class JSON(object):

a = 1

def \_\_repr\_\_(self):

return "This is JSON\n{\n 'foo': 'bar'\n}"

a = JSON()

b = 2

* assert a.a == b, a
* AssertionError: This is JSON
* {
* 'foo': 'bar'
* }
* assert 1 == 2
* + where 1 = This is JSON\n{\n 'foo': 'bar'\n}.a

failure\_demo.py:261: AssertionError

============================= warnings summary =============================

<undetermined location>

Metafunc.addcall is deprecated and scheduled to be removed in pytest 4.0.

Please use Metafunc.parametrize instead.

-- Docs: http://doc.pytest.org/en/latest/warnings.html

================== 42 failed, 1 warnings in 0.12 seconds ===================

**Basic patterns and examples**

**Pass different values to a test function, depending on command line options**

Suppose we want to write a test that depends on a command line option. Here is a basic pattern to achieve this:

* content of test\_sample.py **def** test\_answer(cmdopt):

**if** cmdopt=="type1": **print**("first")

**elif** cmdopt=="type2":

**print**("second")

**assert** 0# to see what was printed

For this to work we need to add a command line option and provide the cmdopt through a [fixture function](#page31):

**178** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

* content of conftest.py **import pytest**

**def** pytest\_addoption(parser):

parser.addoption(

"--cmdopt", action="store", default="type1", help="my option: type1 or type2"

)

**@pytest.fixture**

**def** cmdopt(request):

**return** request.config.getoption("--cmdopt")

Let’s run this without supplying our new option:

$ pytest -q test\_sample.py

F [100%]

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_answer \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

cmdopt = 'type1'

def test\_answer(cmdopt):

if cmdopt == "type1":

print("first")

elif cmdopt == "type2":

print("second")

* assert 0 # to see what was printed
* assert 0

test\_sample.py:6: AssertionError

--------------------------- Captured stdout call ---------------------------

first

1 failed in 0.12 seconds

And now with supplying a command line option:

$ pytest -q --cmdopt=type2

F [100%]

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_answer \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

cmdopt = 'type2'

def test\_answer(cmdopt):

if cmdopt == "type1":

print("first")

elif cmdopt == "type2":

print("second")

* assert 0 # to see what was printed
* assert 0

test\_sample.py:6: AssertionError

--------------------------- Captured stdout call ---------------------------

second

1 failed in 0.12 seconds

**26.2. Basic patterns and examples** **179**

**pytest Documentation, Release 3.6**

You can see that the command line option arrived in our test. This completes the basic pattern. However, one often rather wants to process command line options outside of the test and rather pass in different or more complex objects.

**Dynamically adding command line options**

Through [addopts](#page156) you can statically add command line options for your project. You can also dynamically modify the command line arguments before they get processed:

* content of conftest.py **import sys**

**def** pytest\_load\_initial\_conftests(args):

**if** "xdist" **in** sys.modules: # pytest-xdist plugin **import multiprocessing**

num = max(multiprocessing.cpu\_count() / 2, 1)

args[:] = ["-n", str(num)] + args

If you have the [xdist plugin](https://pypi.org/project/pytest-xdist/) installed you will now always perform test runs using a number of subprocesses close to your CPU. Running in an empty directory with the above conftest.py:

$ pytest

=========================== 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 0 items

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

**Control skipping of tests according to command line option**

Here is a conftest.py file adding a --runslow command line option to control skipping of pytest.mark.slow marked tests:

# content of conftest.py

**import pytest**

**def** pytest\_addoption(parser):

parser.addoption(

"--runslow", action="store\_true", default=False, help="run slow tests"

)

**def** pytest\_collection\_modifyitems(config, items):

**if** config.getoption("--runslow"):

* --runslow given in cli: do not skip slow tests **return**

skip\_slow = pytest.mark.skip(reason="need --runslow option to run")

**for** item **in** items:

**if** "slow" **in** item.keywords:

item.add\_marker(skip\_slow)

**180** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

We can now write a test module like this:

* content of test\_module.py **import pytest**

**def** test\_func\_fast():

**pass**

**@pytest.mark.slow**

**def** test\_func\_slow():

**pass**

and when running it will see a skipped “slow” test:

$ pytest -rs # "-rs" means report details on the little 's'

=========================== 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 .s | [100%] |

========================= short test summary info ========================== SKIP [1] test\_module.py:8: need --runslow option to run

=================== 1 passed, 1 skipped in 0.12 seconds ====================

Or run it including the slow marked test:

$ pytest --runslow

=========================== 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 .. | [100%] |

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

**Writing well integrated assertion helpers**

If you have a test helper function called from a test you can use the pytest.fail marker to fail a test with a certain message. The test support function will not show up in the traceback if you set the \_\_tracebackhide\_\_ option somewhere in the helper function. Example:

* content of test\_checkconfig.py **import pytest**

**def** checkconfig(x):

\_\_tracebackhide\_\_ = True

**if not** hasattr(x,"config"):

pytest.fail("not configured: %s" % (x,))

**26.2. Basic patterns and examples** **181**

**pytest Documentation, Release 3.6**

**def** test\_something():

checkconfig(42)

The \_\_tracebackhide\_\_ setting influences pytest showing of tracebacks: the checkconfig function will not be shown unless the --full-trace command line option is specified. Let’s run our little function:

$ pytest -q test\_checkconfig.py

F [100%]

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_something \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_something():

* checkconfig(42)
* Failed: not configured: 42

test\_checkconfig.py:11: Failed

1 failed in 0.12 seconds

If you only want to hide certain exceptions, you can set \_\_tracebackhide\_\_ to a callable which gets the

ExceptionInfo object. You can for example use this to make sure unexpected exception types aren’t hidden:

**import operator**

**import pytest**

**class ConfigException**(Exception):

**pass**

**def** checkconfig(x):

\_\_tracebackhide\_\_ = operator.methodcaller("errisinstance", ConfigException)

**if not** hasattr(x,"config"):

**raise** ConfigException("not configured:%s"%(x,))

**def** test\_something():

checkconfig(42)

This will avoid hiding the exception traceback on unrelated exceptions (i.e. bugs in assertion helpers).

**Detect if running from within a pytest run**

Usually it is a bad idea to make application code behave differently if called from a test. But if you absolutely must find out if your application code is running from a test you can do something like this:

# content of conftest.py

**def** pytest\_configure(config):

**import sys**

sys.\_called\_from\_test = True

**def** pytest\_unconfigure(config):

**import sys**

**182** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

**del** sys.\_called\_from\_test

and then check for the sys.\_called\_from\_test flag:

**if** hasattr(sys,"\_called\_from\_test"):

* called from within a test run

...

**else**:

* called "normally"

...

accordingly in your application. It’s also a good idea to use your own application module rather than sys for handling flag.

**Adding info to test report header**

It’s easy to present extra information in a pytest run:

# content of conftest.py

**def** pytest\_report\_header(config):

**return** "project deps: mylib-1.1"

which will add the string to the test header accordingly:

$ pytest

=========================== test session starts ============================

platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y project deps: mylib-1.1

rootdir: $REGENDOC\_TMPDIR, inifile:

collected 0 items

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

It is also possible to return a list of strings which will be considered as several lines of information. You may consider config.getoption('verbose') in order to display more information if applicable:

# content of conftest.py

**def** pytest\_report\_header(config):

**if** config.getoption("verbose")>0:

**return** ["info1: did you know that ...","did you?"]

which will add info only when run with “–v”:

$ pytest -v

=========================== 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: .pytest\_cache

info1: did you know that ...

did you?

rootdir: $REGENDOC\_TMPDIR, inifile:

**26.2. Basic patterns and examples** **183**

**pytest Documentation, Release 3.6**

collecting ... collected 0 items

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

and nothing when run plainly:

$ pytest

=========================== 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 0 items

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

**profiling test duration**

If you have a slow running large test suite you might want to find out which tests are the slowest. Let’s make an artificial test suite:

* content of test\_some\_are\_slow.py **import time**

**def** test\_funcfast():

time.sleep(0.1)

**def** test\_funcslow1():

time.sleep(0.2)

**def** test\_funcslow2():

time.sleep(0.3)

Now we can profile which test functions execute the slowest:

$ pytest --durations=3

=========================== 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 3 items

|  |  |  |  |
| --- | --- | --- | --- |
| test\_some\_are\_slow.py ... | | | [100%] |
| ========================= | | | slowest 3 test durations ========================= |
| 0.30s | call | test\_some\_are\_slow.py::test\_funcslow2 | |
| 0.20s | call | test\_some\_are\_slow.py::test\_funcslow1 | |
| 0.13s | call | test\_some\_are\_slow.py::test\_funcfast | |
| ========================= | | | 3 passed in 0.12 seconds ========================= |

**incremental testing - test steps**

Sometimes you may have a testing situation which consists of a series of test steps. If one step fails it makes no sense to execute further steps as they are all expected to fail anyway and their tracebacks add no insight. Here is a simple conftest.py file which introduces an incremental marker which is to be used on classes:

**184** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

# content of conftest.py

**import pytest**

**def** pytest\_runtest\_makereport(item, call):

**if** "incremental" **in** item.keywords:

**if** call.excinfo **is not** None:

parent = item.parent

parent.\_previousfailed = item

**def** pytest\_runtest\_setup(item):

**if** "incremental" **in** item.keywords:

previousfailed = getattr(item.parent, "\_previousfailed", None)

**if** previousfailed **is not** None:

pytest.xfail("previous test failed (%s)" % previousfailed.name)

These two hook implementations work together to abort incremental-marked tests in a class. Here is a test module example:

# content of test\_step.py

**import pytest**

**@pytest.mark.incremental**

**class TestUserHandling**(object):

**def** test\_login(self):

**pass**

**def** test\_modification(self):

**assert** 0

**def** test\_deletion(self):

**pass**

**def** test\_normal():

**pass**

If we run this:

$ pytest -rx

=========================== 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 4 items

|  |  |
| --- | --- |
| test\_step.py .Fx. | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestUserHandling.test\_modification \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <test\_step.TestUserHandling object at 0xdeadbeef>

def test\_modification(self):

* assert 0

**26.2. Basic patterns and examples** **185**

**pytest Documentation, Release 3.6**

* assert 0

test\_step.py:11: AssertionError

========================= short test summary info ==========================

XFAIL test\_step.py::TestUserHandling::()::test\_deletion reason: previous test failed (test\_modification)

============== 1 failed, 2 passed, 1 xfailed in 0.12 seconds ===============

We’ll see that test\_deletion was not executed because test\_modification failed. It is reported as an “expected failure”.

**Package/Directory-level fixtures (setups)**

If you have nested test directories, you can have per-directory fixture scopes by placing fixture functions in a conftest.py file in that directory You can use all types of fixtures including [autouse fixtures](#page45) which are the equiv-alent of xUnit’s setup/teardown concept. It’s however recommended to have explicit fixture references in your tests or test classes rather than relying on implicitly executing setup/teardown functions, especially if they are far away from the actual tests.

Here is an example for making a db fixture available in a directory:

* content of a/conftest.py **import pytest**

**class DB**(object):

**pass**

**@pytest.fixture**(scope="session")

**def** db():

**return** DB()

and then a test module in that directory:

* content of a/test\_db.py **def** test\_a1(db):

**assert** 0, db# to show value

another test module:

* content of a/test\_db2.py **def** test\_a2(db):

**assert** 0, db# to show value

and then a module in a sister directory which will not see the db fixture:

# content of b/test\_error.py

**def** test\_root(db): # no db here, will error out **pass**

We can run this:

$ pytest

=========================== test session starts ============================

platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y

**186** **Chapter 26. Examples and customization tricks**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **pytest Documentation, Release 3.6** |  |
|  |  |  |  |
|  | rootdir: $REGENDOC\_TMPDIR, inifile: |  |  |
|  | collected 7 items |  |  |
|  | test\_step.py .Fx. | [ 57%] |  |
|  | a/test\_db.py F | [ 71%] |  |
|  | a/test\_db2.py F | [ 85%] |  |
|  | b/test\_error.py E | [100%] |  |

================================== ERRORS ==================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ERROR at setup of test\_root \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

file $REGENDOC\_TMPDIR/b/test\_error.py, line 1

def test\_root(db): # no db here, will error out

* fixture 'db' not found
* available fixtures: cache, capfd, capfdbinary, caplog, capsys, capsysbinary,

˓→doctest\_namespace, monkeypatch, pytestconfig, record\_property, record\_xml\_attribute, ˓→ record\_xml\_property, recwarn, tmpdir, tmpdir\_factory

* use 'pytest --fixtures [testpath]' for help on them.

$REGENDOC\_TMPDIR/b/test\_error.py:1

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestUserHandling.test\_modification \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <test\_step.TestUserHandling object at 0xdeadbeef>

def test\_modification(self):

* assert 0
* assert 0

test\_step.py:11: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_a1 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

db = <conftest.DB object at 0xdeadbeef>

def test\_a1(db):

* assert 0, db # to show value
* AssertionError: <conftest.DB object at 0xdeadbeef>
* assert 0

a/test\_db.py:2: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_a2 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

db = <conftest.DB object at 0xdeadbeef>

def test\_a2(db):

* assert 0, db # to show value
* AssertionError: <conftest.DB object at 0xdeadbeef>
* assert 0

a/test\_db2.py:2: AssertionError

========== 3 failed, 2 passed, 1 xfailed, 1 error in 0.12 seconds ==========

The two test modules in the a directory see the same db fixture instance while the one test in the sister-directory b doesn’t see it. We could of course also define a db fixture in that sister directory’s conftest.py file. Note that each fixture is only instantiated if there is a test actually needing it (unless you use “autouse” fixture which are always executed ahead of the first test executing).

**26.2. Basic patterns and examples** **187**

**pytest Documentation, Release 3.6**

**post-process test reports / failures**

If you want to postprocess test reports and need access to the executing environment you can implement a hook that gets called when the test “report” object is about to be created. Here we write out all failing test calls and also access a fixture (if it was used by the test) in case you want to query/look at it during your post processing. In our case we just write some information out to a failures file:

# content of conftest.py

**import pytest**

**import os.path**

**@pytest.hookimpl**(tryfirst=True, hookwrapper=True)

**def** pytest\_runtest\_makereport(item, call):

* execute all other hooks to obtain the report object outcome = **yield**

rep = outcome.get\_result()

* we only look at actual failing test calls, not setup/teardown **if** rep.when=="call" **and** rep.failed:

mode = "a" **if** os.path.exists("failures") **else** "w" **with** open("failures", mode) **as** f:

* let's also access a fixture for the fun of it **if** "tmpdir" **in** item.fixturenames:

extra = " (%s)" % item.funcargs["tmpdir"] **else**:

extra = ""

f.write(rep.nodeid + extra + "**\n**")

if you then have failing tests:

* content of test\_module.py **def** test\_fail1(tmpdir):

**assert** 0

**def** test\_fail2():

**assert** 0

and run them:

$ 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 | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_fail1 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

tmpdir = local('PYTEST\_TMPDIR/test\_fail10')

def test\_fail1(tmpdir):

* assert 0

**188** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

* assert 0

test\_module.py:2: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_fail2 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_fail2():

* assert 0
* assert 0

test\_module.py:6: AssertionError

========================= 2 failed in 0.12 seconds =========================

you will have a “failures” file which contains the failing test ids:

$ cat failures

test\_module.py::test\_fail1 (PYTEST\_TMPDIR/test\_fail10)

test\_module.py::test\_fail2

**Making test result information available in fixtures**

If you want to make test result reports available in fixture finalizers here is a little example implemented via a local plugin:

# content of conftest.py

**import pytest**

**@pytest.hookimpl**(tryfirst=True, hookwrapper=True)

**def** pytest\_runtest\_makereport(item, call):

* execute all other hooks to obtain the report object outcome = **yield**

rep = outcome.get\_result()

* set a report attribute for each phase of a call, which can
* be "setup", "call", "teardown"

setattr(item, "rep\_" + rep.when, rep)

**@pytest.fixture**

**def** something(request):

**yield**

* request.node is an "item" because we use the default
* "function" scope

**if** request.node.rep\_setup.failed:

**print**("setting up a test failed!", request.node.nodeid)

**elif** request.node.rep\_setup.passed:

**if** request.node.rep\_call.failed:

**print**("executing test failed", request.node.nodeid)

if you then have failing tests:

# content of test\_module.py

**import pytest**

**26.2. Basic patterns and examples** **189**

**pytest Documentation, Release 3.6**

**@pytest.fixture**

**def** other():

**assert** 0

**def** test\_setup\_fails(something, other):

**pass**

**def** test\_call\_fails(something):

**assert** 0

**def** test\_fail2():

**assert** 0

and run it:

$ pytest -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 rootdir: $REGENDOC\_TMPDIR, inifile: collected 3 items

test\_module.py Esetting up a test failed! test\_module.py::test\_setup\_fails Fexecuting test failed test\_module.py::test\_call\_fails F

================================== ERRORS ==================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ERROR at setup of test\_setup\_fails \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

@pytest.fixture

def other():

* assert 0
* assert 0

test\_module.py:7: AssertionError

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_call\_fails \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

something = None

def test\_call\_fails(something):

* assert 0
* assert 0

test\_module.py:15: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_fail2 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_fail2():

* assert 0
* assert 0

test\_module.py:19: AssertionError

==================== 2 failed, 1 error in 0.12 seconds =====================

**190** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

You’ll see that the fixture finalizers could use the precise reporting information.

**PYTEST\_CURRENT\_TEST environment variable**

New in version 3.2.

Sometimes a test session might get stuck and there might be no easy way to figure out which test got stuck, for example if pytest was run in quiet mode (-q) or you don’t have access to the console output. This is particularly a problem if the problem helps only sporadically, the famous “flaky” kind of tests.

pytest sets a PYTEST\_CURRENT\_TEST environment variable when running tests, which can be inspected by process monitoring utilities or libraries like [psutil](https://pypi.org/project/psutil/) to discover which test got stuck if necessary:

**import psutil**

**for** pid **in** psutil.pids():

environ = psutil.Process(pid).environ()

**if** "PYTEST\_CURRENT\_TEST" **in** environ:

**print**(f'pytest process {pid} running: {environ["PYTEST\_CURRENT\_TEST"]}')

During the test session pytest will set PYTEST\_CURRENT\_TEST to the current test [nodeid](#page16) and the current stage, which can be setup, call and teardown.

For example, when running a single test function named test\_foo from foo\_module.py,

PYTEST\_CURRENT\_TEST will be set to:

1. foo\_module.py::test\_foo (setup)
2. foo\_module.py::test\_foo (call)
3. foo\_module.py::test\_foo (teardown)

In that order.

Note: The contents of PYTEST\_CURRENT\_TEST is meant to be human readable and the actual format can be changed between releases (even bug fixes) so it shouldn’t be relied on for scripting or automation.

**Freezing pytest**

If you freeze your application using a tool like [PyInstaller](https://pyinstaller.readthedocs.io) in order to distribute it to your end-users, it is a good idea to also package your test runner and run your tests using the frozen application. This way packaging errors such as dependencies not being included into the executable can be detected early while also allowing you to send test files to users so they can run them in their machines, which can be useful to obtain more information about a hard to reproduce bug.

Fortunately recent PyInstaller releases already have a custom hook for pytest, but if you are using another tool to freeze executables such as cx\_freeze or py2exe, you can use pytest.freeze\_includes() to obtain the full list of internal pytest modules. How to configure the tools to find the internal modules varies from tool to tool, however.

Instead of freezing the pytest runner as a separate executable, you can make your frozen program work as the pytest runner by some clever argument handling during program startup. This allows you to have a single executable, which is usually more convenient. Please note that the mechanism for plugin discovery used by pytest (setupttools entry points) doesn’t work with frozen executables so pytest can’t find any third party plugins automatically. To include third party plugins like pytest-timeout they must be imported explicitly and passed on to pytest.main.

**26.2. Basic patterns and examples** **191**

**pytest Documentation, Release 3.6**

* contents of app\_main.py **import sys**

**import pytest\_timeout** # Third party plugin

**if** len(sys.argv)>1 **and** sys.argv[1]=="--pytest":

**import pytest**

sys.exit(pytest.main(sys.argv[2:], plugins=[pytest\_timeout]))

**else**:

* normal application execution: at this point argv can be parsed
* by your argument-parsing library of choice as usual

...

This allows you to execute tests using the frozen application with standard pytest command-line options:

./app\_main --pytest --verbose --tb=long --junitxml=results.xml test-suite/

**Parametrizing tests**

pytest allows to easily parametrize test functions. For basic docs, see [Parametrizing fixtures and test functions](#page81).

In the following we provide some examples using the builtin mechanisms.

**Generating parameters combinations, depending on command line**

Let’s say we want to execute a test with different computation parameters and the parameter range shall be determined by a command line argument. Let’s first write a simple (do-nothing) computation test:

# content of test\_compute.py

**def** test\_compute(param1):

**assert** param1<4

Now we add a test configuration like this:

# content of conftest.py

**def** pytest\_addoption(parser):

parser.addoption("--all", action="store\_true",

help="run all combinations")

**def** pytest\_generate\_tests(metafunc):

**if** 'param1' **in** metafunc.fixturenames:

**if** metafunc.config.getoption('all'):

end = 5

**else**:

end = 2

metafunc.parametrize("param1", range(end))

This means that we only run 2 tests if we do not pass --all:

|  |  |  |
| --- | --- | --- |
| $ | pytest -q test\_compute.py |  |
| .. | | [100%] |
| 2 | passed in 0.12 seconds |  |
|  |  |  |

**192** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

We run only two computations, so we see two dots. let’s run the full monty:

$ pytest -q --all

....F [100%]

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_compute[4] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

param1 = 4

def test\_compute(param1):

* assert param1 < 4

Eassert 4 < 4

test\_compute.py:3: AssertionError

1 failed, 4 passed in 0.12 seconds

As expected when running the full range of param1 values we’ll get an error on the last one.

**Different options for test IDs**

pytest will build a string that is the test ID for each set of values in a parametrized test. 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:

# content of test\_time.py

**import pytest**

**from datetime import** datetime, timedelta

testdata = [

(datetime(2001, 12, 12), datetime(2001, 12, 11), timedelta(1)), (datetime(2001, 12, 11), datetime(2001, 12, 12), timedelta(-1)),

]

**@pytest**.mark.parametrize("a,b,expected", testdata)

**def** test\_timedistance\_v0(a, b, expected):

diff = a - b

**assert** diff==expected

**@pytest**.mark.parametrize("a,b,expected", testdata, ids=["forward","backward"])

**def** test\_timedistance\_v1(a, b, expected):

diff = a - b

**assert** diff==expected

**def** idfn(val):

**if** isinstance(val, (datetime,)):

* note this wouldn't show any hours/minutes/seconds **return** val.strftime('%Y%m%d')

**26.3. Parametrizing tests** **193**

**pytest Documentation, Release 3.6**

**@pytest**.mark.parametrize("a,b,expected", testdata, ids=idfn)

**def** test\_timedistance\_v2(a, b, expected):

diff = a - b

**assert** diff==expected

**@pytest**.mark.parametrize("a,b,expected", [

pytest.param(datetime(2001, 12, 12), datetime(2001, 12, 11),

timedelta(1), id='forward'),

pytest.param(datetime(2001, 12, 11), datetime(2001, 12, 12),

timedelta(-1), id='backward'),

])

**def** test\_timedistance\_v3(a, b, expected):

diff = a - b

**assert** diff==expected

In test\_timedistance\_v0, we let pytest generate the test IDs.

In test\_timedistance\_v1, we specified ids as a list of strings which were used as the test IDs. These are succinct, but can be a pain to maintain.

In test\_timedistance\_v2, we specified ids as a function that can generate a string representation to make part of the test ID. So our datetime values use the label generated by idfn, but because we didn’t generate a label for timedelta objects, they are still using the default pytest representation:

$ pytest test\_time.py --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 8 | items |
| <Module 'test\_time.py'> | |
| <Function | 'test\_timedistance\_v0[a0-b0-expected0]'> |
| <Function | 'test\_timedistance\_v0[a1-b1-expected1]'> |
| <Function | 'test\_timedistance\_v1[forward]'> |
| <Function | 'test\_timedistance\_v1[backward]'> |
| <Function | 'test\_timedistance\_v2[20011212-20011211-expected0]'> |
| <Function | 'test\_timedistance\_v2[20011211-20011212-expected1]'> |
| <Function | 'test\_timedistance\_v3[forward]'> |
| <Function | 'test\_timedistance\_v3[backward]'> |

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

In test\_timedistance\_v3, we used pytest.param to specify the test IDs together with the actual data, instead of listing them separately.

**A quick port of “testscenarios”**

Here is a quick port to run tests configured with [test scenarios](https://pypi.org/project/testscenarios/), an add-on from Robert Collins for the standard unittest framework. We only have to work a bit to construct the correct arguments for pytest’s [Metafunc.parametrize()](#page124):

# content of test\_scenarios.py

**def** pytest\_generate\_tests(metafunc):

idlist = []

argvalues = []

**for** scenario **in** metafunc.cls.scenarios:

idlist.append(scenario[0])

**194** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

items = scenario[1].items()

argnames = [x[0] **for** x **in** items]

argvalues.append(([x[1] **for** x **in** items]))

metafunc.parametrize(argnames, argvalues, ids=idlist, scope="class")

scenario1 = ('basic', {'attribute': 'value'})

scenario2 = ('advanced', {'attribute': 'value2'})

**class TestSampleWithScenarios**(object):

scenarios = [scenario1, scenario2]

**def** test\_demo1(self, attribute):

**assert** isinstance(attribute,str)

**def** test\_demo2(self, attribute):

**assert** isinstance(attribute,str)

this is a fully self-contained example which you can run with:

$ pytest test\_scenarios.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 4 items

|  |  |
| --- | --- |
| test\_scenarios.py .... | [100%] |

========================= 4 passed in 0.12 seconds =========================

If you just collect tests you’ll also nicely see ‘advanced’ and ‘basic’ as variants for the test function:

$ pytest --collect-only test\_scenarios.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 4 items

<Module 'test\_scenarios.py'>

<Class 'TestSampleWithScenarios'>

<Instance '()'>

<Function 'test\_demo1[basic]'>

<Function 'test\_demo2[basic]'>

<Function 'test\_demo1[advanced]'>

<Function 'test\_demo2[advanced]'>

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

Note that we told metafunc.parametrize() that your scenario values should be considered class-scoped. With pytest-2.3 this leads to a resource-based ordering.

**Deferring the setup of parametrized resources**

The parametrization of test functions happens at collection time. It is a good idea to setup expensive resources like DB connections or subprocess only when the actual test is run. Here is a simple example how you can achieve that, first the actual test requiring a db object:

**26.3. Parametrizing tests** **195**

**pytest Documentation, Release 3.6**

# content of test\_backends.py

**import pytest**

**def** test\_db\_initialized(db):

# a dummy test

**if** db.\_\_class\_\_.\_\_name\_\_=="DB2":

pytest.fail("deliberately failing for demo purposes")

We can now add a test configuration that generates two invocations of the test\_db\_initialized function and also implements a factory that creates a database object for the actual test invocations:

* content of conftest.py **import pytest**

**def** pytest\_generate\_tests(metafunc):

**if** 'db' **in** metafunc.fixturenames:

metafunc.parametrize("db", ['d1', 'd2'], indirect=**True**)

**class DB1**(object):

"one database object"

**class DB2**(object):

"alternative database object"

**@pytest**.fixture

**def** db(request):

**if** request.param=="d1":

**return** DB1()

**elif** request.param=="d2":

**return** DB2()

**else**:

**raise** ValueError("invalid internal test config")

Let’s first see how it looks like at collection time:

$ pytest test\_backends.py --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 2 items

<Module 'test\_backends.py'>

<Function 'test\_db\_initialized[d1]'>

<Function 'test\_db\_initialized[d2]'>

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

And then when we run the test:

|  |  |
| --- | --- |
| $ pytest -q test\_backends.py |  |
| .F | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_db\_initialized[d2] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

db = <conftest.DB2 object at 0xdeadbeef>

def test\_db\_initialized(db):

# a dummy test

if db.\_\_class\_\_.\_\_name\_\_ == "DB2":

> pytest.fail("deliberately failing for demo purposes")

**196** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

* Failed: deliberately failing for demo purposes

test\_backends.py:6: Failed

1 failed, 1 passed in 0.12 seconds

The first invocation with db == "DB1" passed while the second with db == "DB2" failed. Our db fixture func-tion has instantiated each of the DB values during the setup phase while the pytest\_generate\_tests generated two according calls to the test\_db\_initialized during the collection phase.

**Apply indirect on particular arguments**

Very often parametrization uses more than one argument name. There is opportunity to apply indirect parameter on particular arguments. It can be done by passing list or tuple of arguments’ names to indirect. In the example below there is a function test\_indirect which uses two fixtures: x and y. Here we give to indirect the list, which contains the name of the fixture x. The indirect parameter will be applied to this argument only, and the value a will be passed to respective fixture function:

# content of test\_indirect\_list.py

**import pytest**

**@pytest**.fixture(scope='function')

**def** x(request):

**return** request.param\*3

**@pytest**.fixture(scope='function')

**def** y(request):

**return** request.param\*2

**@pytest**.mark.parametrize('x, y', [('a','b')], indirect=['x'])

**def** test\_indirect(x,y):

**assert** x=='aaa'

**assert** y=='b'

The result of this test will be successful:

$ pytest test\_indirect\_list.py --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 1 item

<Module 'test\_indirect\_list.py'>

<Function 'test\_indirect[a-b]'>

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

**Parametrizing test methods through per-class configuration**

Here is an example pytest\_generate\_tests function implementing a parametrization scheme similar to

Michael Foord’s [unittest parametrizer](https://github.com/testing-cabal/unittest-ext/blob/master/params.py) but in a lot less code:

* content of ./test\_parametrize.py **import pytest**

**def** pytest\_generate\_tests(metafunc):

**26.3. Parametrizing tests** **197**

**pytest Documentation, Release 3.6**

# called once per each test function

funcarglist = metafunc.cls.params[metafunc.function.\_\_name\_\_]

argnames = sorted(funcarglist[0])

metafunc.parametrize(argnames, [[funcargs[name] **for** name **in** argnames]

**for** funcargs **in** funcarglist])

**class TestClass**(object):

* a map specifying multiple argument sets for a test method params = {

'test\_equals': [dict(a=1, b=2), dict(a=3, b=3), ],

'test\_zerodivision': [dict(a=1, b=0), ],

}

**def** test\_equals(self, a, b):

**assert** a==b

**def** test\_zerodivision(self, a, b):

pytest.raises(ZeroDivisionError, "a/b")

Our test generator looks up a class-level definition which specifies which argument sets to use for each test function.

Let’s run it:

$ pytest -q

F.. [100%]

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestClass.test\_equals[1-2] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <test\_parametrize.TestClass object at 0xdeadbeef>, a = 1, b = 2

def test\_equals(self, a, b):

* assert a == b

Eassert 1 == 2

test\_parametrize.py:18: AssertionError

1 failed, 2 passed in 0.12 seconds

**Indirect parametrization with multiple fixtures**

Here is a stripped down real-life example of using parametrized testing for testing serialization of objects between different python interpreters. We define a test\_basic\_objects function which is to be run with different sets of arguments for its three arguments:

* python1: first python interpreter, run to pickle-dump an object to a file
* python2: second interpreter, run to pickle-load an object from a file
* obj: object to be dumped/loaded

"""

module containing a parametrized tests testing cross-python serialization via the pickle module. """

**import py**

**import pytest**

**import \_pytest.\_code**

pythonlist = ["python2.7", "python3.4", "python3.5"]

**198** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

**@pytest**.fixture(params=pythonlist)

**def** python1(request, tmpdir):

picklefile = tmpdir.join("data.pickle")

**return** Python(request.param, picklefile)

**@pytest**.fixture(params=pythonlist)

**def** python2(request, python1):

**return** Python(request.param, python1.picklefile)

**class Python**(object):

**def** \_\_init\_\_(self, version, picklefile):

self.pythonpath = py.path.local.sysfind(version)

**if not** self.pythonpath:

pytest.skip("%r not found" % (version,))

self.picklefile = picklefile

**def** dumps(self, obj):

dumpfile = self.picklefile.dirpath("dump.py")

dumpfile.write(

\_pytest.\_code.Source(

"""

import pickle

f = open(%r, 'wb')

s = pickle.dump(%r, f, protocol=2)

f.close()

"""

% (str(self.picklefile), obj)

)

)

py.process.cmdexec("%s %s" % (self.pythonpath, dumpfile))

**def** load\_and\_is\_true(self, expression):

loadfile = self.picklefile.dirpath("load.py")

loadfile.write(

\_pytest.\_code.Source(

"""

import pickle

f = open(%r, 'rb')

obj = pickle.load(f)

f.close()

res = eval(%r)

if not res:

raise SystemExit(1)

"""

% (str(self.picklefile), expression)

)

)

print(loadfile)

py.process.cmdexec("%s %s" % (self.pythonpath, loadfile))

**@pytest**.mark.parametrize("obj", [42, {}, {1:3}])

**def** test\_basic\_objects(python1, python2, obj):

python1.dumps(obj)

**26.3. Parametrizing tests** **199**

**pytest Documentation, Release 3.6**

python2.load\_and\_is\_true("obj == %s" % obj)

Running it results in some skips if we don’t have all the python interpreters installed and otherwise runs all combina-tions (5 interpreters times 5 interpreters times 3 objects to serialize/deserialize):

. $ pytest -rs -q multipython.py

........................... [100%]

27 passed in 0.12 seconds

**Indirect parametrization of optional implementations/imports**

If you want to compare the outcomes of several implementations of a given API, you can write test functions that receive the already imported implementations and get skipped in case the implementation is not importable/available. Let’s say we have a “base” implementation and the other (possibly optimized ones) need to provide similar results:

# content of conftest.py

**import pytest**

**@pytest**.fixture(scope="session")

**def** basemod(request):

**return** pytest.importorskip("base")

**@pytest**.fixture(scope="session", params=["opt1","opt2"])

**def** optmod(request):

**return** pytest.importorskip(request.param)

And then a base implementation of a simple function:

* content of base.py **def** func1():

**return** 1

And an optimized version:

* content of opt1.py **def** func1():

**return** 1.0001

And finally a little test module:

# content of test\_module.py

**def** test\_func1(basemod, optmod):

**assert** round(basemod.func1(),3)==round(optmod.func1(),3)

If you run this with reporting for skips enabled:

$ pytest -rs 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 .s | [100%] |

========================= short test summary info ==========================

**200** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

SKIP [1] $REGENDOC\_TMPDIR/conftest.py:11: could not import 'opt2'

=================== 1 passed, 1 skipped in 0.12 seconds ====================

You’ll see that we don’t have an opt2 module and thus the second test run of our test\_func1 was skipped. A few notes:

* the fixture functions in the conftest.py file are “session-scoped” because we don’t need to import more than once
* if you have multiple test functions and a skipped import, you will see the [1] count increasing in the report
* you can put [@pytest.mark.parametrize](#page81) style parametrization on the test functions to parametrize input/output values as well.

**Set marks or test ID for individual parametrized test**

Use pytest.param to apply marks or set test ID to individual parametrized test. For example:

* content of test\_pytest\_param\_example.py **import pytest @pytest**.mark.parametrize('test\_input,expected', [

('3+5', 8),

pytest.param('1+7', 8,

marks=pytest.mark.basic),

pytest.param('2+4', 6,

marks=pytest.mark.basic,

id='basic\_2+4'),

pytest.param('6\*9', 42,

marks=[pytest.mark.basic, pytest.mark.xfail], id='basic\_6\*9'),

])

**def** test\_eval(test\_input, expected):

**assert** eval(test\_input)==expected

In this example, we have 4 parametrized tests. Except for the first test, we mark the rest three parametrized tests with the custom marker basic, and for the fourth test we also use the built-in mark xfail to indicate this test is expected to fail. For explicitness, we set test ids for some tests.

Then run pytest with verbose mode and with only the basic marker:

pytest -v -m basic

============================================ 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 4 items

test\_pytest\_param\_example.py::test\_eval[1+7-8] PASSED test\_pytest\_param\_example.py::test\_eval[basic\_2+4] PASSED test\_pytest\_param\_example.py::test\_eval[basic\_6\*9] xfail

========================================== short test summary info ˓→===========================================

XFAIL test\_pytest\_param\_example.py::test\_eval[basic\_6\*9]

============================================= 1 tests deselected ˓→=============================================

**26.3. Parametrizing tests** **201**

**pytest Documentation, Release 3.6**

As the result:

* Four tests were collected
* One test was deselected because it doesn’t have the basic mark.
* Three tests with the basic mark was selected.
* The test test\_eval[1+7-8] passed, but the name is autogenerated and confusing.
* The test test\_eval[basic\_2+4] passed.
* The test test\_eval[basic\_6\*9] was expected to fail and did fail.

**Working with custom markers**

Here are some example using the [Marking test functions with attributes](#page51) mechanism.

**Marking test functions and selecting them for a run**

You can “mark” a test function with custom metadata like this:

# content of test\_server.py

**import pytest**

**@pytest**.mark.webtest

**def** test\_send\_http():

**pass** # perform some webtest test for your app **def** test\_something\_quick():

**pass**

**def** test\_another():

**pass**

**class TestClass**(object):

**def** test\_method(self):

**pass**

New in version 2.2.

You can then restrict a test run to only run tests marked with webtest:

$ pytest -v -m webtest

=========================== 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: .pytest\_cache

|  |  |  |
| --- | --- | --- |
| rootdir: $REGENDOC\_TMPDIR, | inifile: |  |
| collecting ... collected 4 | items / 3 deselected |  |
| test\_server.py::test\_send\_http PASSED | | [100%] |

================== 1 passed, 3 deselected in 0.12 seconds ==================

Or the inverse, running all tests except the webtest ones:

$ pytest -v -m "not webtest"

=========================== 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

**202** **Chapter 26. Examples and customization tricks**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **pytest Documentation, Release 3.6** |  |
|  |  |  |  |
|  | cachedir: .pytest\_cache |  |  |
|  | rootdir: $REGENDOC\_TMPDIR, inifile: |  |  |
|  | collecting ... collected 4 items / 1 deselected |  |  |
|  | test\_server.py::test\_something\_quick PASSED | [ 33%] |  |
|  | test\_server.py::test\_another PASSED | [ 66%] |  |
|  | test\_server.py::TestClass::test\_method PASSED | [100%] |  |

================== 3 passed, 1 deselected in 0.12 seconds ==================

**Selecting tests based on their node ID**

You can provide one or more [node IDs](#page212) as positional arguments to select only specified tests. This makes it easy to select tests based on their module, class, method, or function name:

* pytest -v test\_server.py::TestClass::test\_method

=========================== 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: .pytest\_cache

rootdir: $REGENDOC\_TMPDIR, inifile: collecting ... collected 1 item

|  |  |
| --- | --- |
| test\_server.py::TestClass::test\_method PASSED | [100%] |

========================= 1 passed in 0.12 seconds =========================

You can also select on the class:

$ pytest -v test\_server.py::TestClass

=========================== 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: .pytest\_cache |  |  |
| rootdir: $REGENDOC\_TMPDIR, | inifile: |  |
| collecting ... collected 1 | item |  |
| test\_server.py::TestClass::test\_method PASSED | | [100%] |

========================= 1 passed in 0.12 seconds =========================

Or select multiple nodes:

* pytest -v test\_server.py::TestClass test\_server.py::test\_send\_http

=========================== 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: .pytest\_cache

rootdir: $REGENDOC\_TMPDIR, inifile: collecting ... collected 2 items

|  |  |
| --- | --- |
| test\_server.py::TestClass::test\_method PASSED | [ 50%] |
| test\_server.py::test\_send\_http PASSED | [100%] |

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

**26.4. Working with custom markers** **203**

**pytest Documentation, Release 3.6**

Note: Node IDs are of the form module.py::class::method or module.py::function. Node IDs control which tests are collected, so module.py::class will select all test methods on the class. Nodes are also created for each parameter of a parametrized fixture or test, so selecting a parametrized test must include the parameter value, e.g. module.py::function[param].

Node IDs for failing tests are displayed in the test summary info when running pytest with the -rf option. You can also construct Node IDs from the output of pytest --collectonly.

**Using -k expr to select tests based on their name**

You can use the -k command line option to specify an expression which implements a substring match on the test names instead of the exact match on markers that -m provides. This makes it easy to select tests based on their names:

* pytest -v -k http # running with the above defined example module

=========================== 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: .pytest\_cache

rootdir: $REGENDOC\_TMPDIR, inifile:

collecting ... collected 4 items / 3 deselected

|  |  |
| --- | --- |
| test\_server.py::test\_send\_http PASSED | [100%] |

================== 1 passed, 3 deselected in 0.12 seconds ==================

And you can also run all tests except the ones that match the keyword:

$ pytest -k "not send\_http" -v

=========================== 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: .pytest\_cache

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| rootdir: $REGENDOC\_TMPDIR, | inifile: |  |  |  |
| collecting ... collected 4 | items / 1 | deselected |  |  |
| test\_server.py::test\_something\_quick | | PASSED | [ | 33%] |
| test\_server.py::test\_another PASSED | |  | [ | 66%] |
| test\_server.py::TestClass::test\_method PASSED | | | [100%] | |

================== 3 passed, 1 deselected in 0.12 seconds ==================

Or to select “http” and “quick” tests:

$ pytest -k "http or quick" -v

=========================== 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: .pytest\_cache

|  |  |  |  |
| --- | --- | --- | --- |
| rootdir: $REGENDOC\_TMPDIR, | inifile: |  |  |
| collecting ... collected 4 | items / 2 | deselected |  |
| test\_server.py::test\_send\_http PASSED | |  | [ 50%] |
| test\_server.py::test\_something\_quick | | PASSED | [100%] |

================== 2 passed, 2 deselected in 0.12 seconds ==================

**204** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

Note: If you are using expressions such as "X and Y" then both X and Y need to be simple non-keyword names. For example, "pass" or "from" will result in SyntaxErrors because "-k" evaluates the expression using Python’s [eval](https://docs.python.org/3.6/library/functions.html#eval) function.

However, if the "-k" argument is a simple string, no such restrictions apply. Also "-k 'not STRING'" has no restrictions. You can also specify numbers like "-k 1.3" to match tests which are parametrized with the float "1.3".

**Registering markers**

New in version 2.2.

Registering markers for your test suite is simple:

* content of pytest.ini [pytest]

markers =

webtest: mark a test **as** a webtest.

You can ask which markers exist for your test suite - the list includes our just defined webtest markers:

$ pytest --markers

@pytest.mark.webtest: mark a test as a webtest.

@pytest.mark.skip(reason=None): skip the given test function with an optional reason.

˓→Example: skip(reason="no way of currently testing this") skips the test.

@pytest.mark.skipif(condition): skip the given test function if eval(condition)

˓→results in a True value. Evaluation happens within the module global context.

˓→Example: skipif('sys.platform == "win32"') skips the test if we are on the win32 ˓→platform. see http://pytest.org/latest/skipping.html

@pytest.mark.xfail(condition, reason=None, run=True, raises=None, strict=False): mark

˓→the test function as an expected failure if eval(condition) has a True value.

˓→Optionally specify a reason for better reporting and run=False if you don't even

˓→want to execute the test function. If only specific exception(s) are expected, you

˓→can list them in raises, and if the test fails in other ways, it will be reported

˓→as a true failure. See http://pytest.org/latest/skipping.html

@pytest.mark.parametrize(argnames, argvalues): call a test function multiple times

˓→passing in different arguments in turn. argvalues generally needs to be a list of

˓→values if argnames specifies only one name or a list of tuples of values if

˓→argnames specifies multiple names. Example: @parametrize('arg1', [1,2]) would lead

˓→to two calls of the decorated test function, one with arg1=1 and another with

˓→arg1=2.see http://pytest.org/latest/parametrize.html for more info and examples.

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing all ˓→of the specified fixtures. see http://pytest.org/latest/fixture.html#usefixtures

@pytest.mark.tryfirst: mark a hook implementation function such that the plugin ˓→machinery will try to call it first/as early as possible.

@pytest.mark.trylast: mark a hook implementation function such that the plugin ˓→machinery will try to call it last/as late as possible.

**26.4. Working with custom markers** **205**

**pytest Documentation, Release 3.6**

For an example on how to add and work with markers from a plugin, see [Custom marker and command line option to](#page215) [control test runs](#page215).

Note: It is recommended to explicitly register markers so that:

* There is one place in your test suite defining your markers
* Asking for existing markers via pytest --markers gives good output
* Typos in function markers are treated as an error if you use the --strict option.

**Marking whole classes or modules**

You may use pytest.mark decorators with classes to apply markers to all of its test methods:

* content of test\_mark\_classlevel.py **import pytest @pytest**.mark.webtest

**class TestClass**(object): **def** test\_startup(self):

**pass**

**def** test\_startup\_and\_more(self): **pass**

This is equivalent to directly applying the decorator to the two test functions.

To remain backward-compatible with Python 2.4 you can also set a pytestmark attribute on a TestClass like this:

**import pytest**

**class TestClass**(object):

pytestmark = pytest.mark.webtest

or if you need to use multiple markers you can use a list:

**import pytest**

**class TestClass**(object):

pytestmark = [pytest.mark.webtest, pytest.mark.slowtest]

You can also set a module level marker:

**import pytest**

pytestmark = pytest.mark.webtest

in which case it will be applied to all functions and methods defined in the module.

**Marking individual tests when using parametrize**

When using parametrize, applying a mark will make it apply to each individual test. However it is also possible to apply a marker to an individual test instance:

**import pytest**

**@pytest**.mark.foo

**206** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

**@pytest**.mark.parametrize(("n","expected"), [(1, 2),

pytest.mark.bar((1, 3)),

(2, 3),

])

**def** test\_increment(n, expected):

**assert** n+1==expected

In this example the mark “foo” will apply to each of the three tests, whereas the “bar” mark is only applied to the second test. Skip and xfail marks can also be applied in this way, see [Skip/xfail with parametrize](#page78).

Note: If the data you are parametrizing happen to be single callables, you need to be careful when marking these items. pytest.mark.xfail(my\_func) won’t work because it’s also the signa-ture of a function being decorated. To resolve this ambiguity, you need to pass a reason argument: pytest.mark.xfail(func\_bar,reason="Issue#7").

**Custom marker and command line option to control test runs**

Plugins can provide custom markers and implement specific behaviour based on it. This is a self-contained example which adds a command line option and a parametrized test function marker to run tests specifies via named environ-ments:

# content of conftest.py

**import pytest**

**def** pytest\_addoption(parser):

parser.addoption("-E", action="store", metavar="NAME", help="only run tests matching the environment NAME.")

**def** pytest\_configure(config):

* register an additional marker config.addinivalue\_line("markers",

"env(name): mark test to run only on named environment")

**def** pytest\_runtest\_setup(item):

envnames = [mark.args[0] **for** mark **in** item.iter\_markers(name='env')] **if** envnames:

**if** item.config.getoption("-E") **not in** envnames:

pytest.skip("test requires env in %r" % envnames)

A test file using this local plugin:

# content of test\_someenv.py

**import pytest**

**@pytest**.mark.env("stage1")

**def** test\_basic\_db\_operation():

**pass**

and an example invocations specifying a different environment than what the test needs:

$ pytest -E stage2

=========================== test session starts ============================

platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y

**26.4. Working with custom markers** **207**

**pytest Documentation, Release 3.6**

rootdir: $REGENDOC\_TMPDIR, inifile:

collected 1 item

|  |  |
| --- | --- |
| test\_someenv.py s | [100%] |

======================== 1 skipped in 0.12 seconds =========================

and here is one that specifies exactly the environment needed:

$ pytest -E stage1

=========================== 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\_someenv.py . | [100%] |

========================= 1 passed in 0.12 seconds =========================

The --markers option always gives you a list of available markers:

$ pytest --markers

@pytest.mark.env(name): mark test to run only on named environment

@pytest.mark.skip(reason=None): skip the given test function with an optional reason.

˓→Example: skip(reason="no way of currently testing this") skips the test.

@pytest.mark.skipif(condition): skip the given test function if eval(condition)

˓→results in a True value. Evaluation happens within the module global context.

˓→Example: skipif('sys.platform == "win32"') skips the test if we are on the win32 ˓→platform. see http://pytest.org/latest/skipping.html

@pytest.mark.xfail(condition, reason=None, run=True, raises=None, strict=False): mark

˓→the test function as an expected failure if eval(condition) has a True value.

˓→Optionally specify a reason for better reporting and run=False if you don't even

˓→want to execute the test function. If only specific exception(s) are expected, you

˓→can list them in raises, and if the test fails in other ways, it will be reported

˓→as a true failure. See http://pytest.org/latest/skipping.html

@pytest.mark.parametrize(argnames, argvalues): call a test function multiple times

˓→passing in different arguments in turn. argvalues generally needs to be a list of

˓→values if argnames specifies only one name or a list of tuples of values if

˓→argnames specifies multiple names. Example: @parametrize('arg1', [1,2]) would lead

˓→to two calls of the decorated test function, one with arg1=1 and another with

˓→arg1=2.see http://pytest.org/latest/parametrize.html for more info and examples.

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing all ˓→of the specified fixtures. see http://pytest.org/latest/fixture.html#usefixtures

@pytest.mark.tryfirst: mark a hook implementation function such that the plugin ˓→machinery will try to call it first/as early as possible.

@pytest.mark.trylast: mark a hook implementation function such that the plugin ˓→machinery will try to call it last/as late as possible.

**208** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

**Passing a callable to custom markers**

Below is the config file that will be used in the next examples:

* content of conftest.py **import sys**

**def** pytest\_runtest\_setup(item):

**for** marker **in** item.iter\_markers(name='my\_marker'):

print(marker)

sys.stdout.flush()

A custom marker can have its argument set, i.e. args and kwargs properties, defined by either invoking it as a callable or using pytest.mark.MARKER\_NAME.with\_args. These two methods achieve the same effect most of the time.

However, if there is a callable as the single positional argument with no keyword arguments, using the pytest.mark.MARKER\_NAME(c) will not pass c as a positional argument but decorate c with the custom marker (see [MarkDecorator](#page51)). Fortunately, pytest.mark.MARKER\_NAME.with\_args comes to the rescue:

* content of test\_custom\_marker.py **import pytest**

**def** hello\_world(\*args,\*\*kwargs):

**return** 'Hello World'

**@pytest**.mark.my\_marker.with\_args(hello\_world)

**def** test\_with\_args():

**pass**

The output is as follows:

$ pytest -q -s

Mark(name='my\_marker', args=(<function hello\_world at 0xdeadbeef>,), kwargs={})

.

1 passed in 0.12 seconds

We can see that the custom marker has its argument set extended with the function hello\_world. This is the key difference between creating a custom marker as a callable, which invokes \_\_call\_\_ behind the scenes, and using with\_args.

**Reading markers which were set from multiple places**

If you are heavily using markers in your test suite you may encounter the case where a marker is applied several times to a test function. From plugin code you can read over all such settings. Example:

* content of test\_mark\_three\_times.py **import pytest**

pytestmark = pytest.mark.glob("module", x=1)

**@pytest**.mark.glob("class", x=2)

**class TestClass**(object):

**@pytest**.mark.glob("function", x=3)

**def** test\_something(self):

**pass**

**26.4. Working with custom markers** **209**

**pytest Documentation, Release 3.6**

Here we have the marker “glob” applied three times to the same test function. From a conftest file we can read it like this:

* content of conftest.py **import sys**

**def** pytest\_runtest\_setup(item):

**for** mark **in** item.iter\_markers(name='glob'):

print ("glob args=%s kwargs=%s" %(mark.args, mark.kwargs))

sys.stdout.flush()

Let’s run this without capturing output and see what we get:

$ pytest -q -s

glob args=('function',) kwargs={'x': 3}

glob args=('class',) kwargs={'x': 2}

glob args=('module',) kwargs={'x': 1}

.

1 passed in 0.12 seconds

**marking platform specific tests with pytest**

Consider you have a test suite which marks tests for particular platforms, namely pytest.mark.darwin, pytest.mark.win32 etc. and you also have tests that run on all platforms and have no specific marker. If you now want to have a way to only run the tests for your particular platform, you could use the following plugin:

* content of conftest.py

**import sys import pytest**

ALL = set("darwin linux win32".split())

**def** pytest\_runtest\_setup(item):

supported\_platforms = ALL.intersection(mark.name **for** mark **in** item.iter\_markers()) plat = sys.platform

**if** supported\_platforms **and** plat **not in** supported\_platforms:

pytest.skip("cannot run on platform %s" % (plat))

then tests will be skipped if they were specified for a different platform. Let’s do a little test file to show how this looks like:

# content of test\_plat.py

**import pytest**

**@pytest**.mark.darwin

**def** test\_if\_apple\_is\_evil():

**pass**

**@pytest**.mark.linux

**def** test\_if\_linux\_works():

**pass**

**@pytest**.mark.win32

**def** test\_if\_win32\_crashes():

**pass**

**210** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

**def** test\_runs\_everywhere():

**pass**

then you will see two tests skipped and two executed tests as expected:

$ pytest -rs # this option reports skip reasons

=========================== 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 4 items

|  |  |
| --- | --- |
| test\_plat.py s.s. | [100%] |

========================= short test summary info ========================== SKIP [2] $REGENDOC\_TMPDIR/conftest.py:12: cannot run on platform linux

=================== 2 passed, 2 skipped in 0.12 seconds ====================

Note that if you specify a platform via the marker-command line option like this:

$ pytest -m linux

=========================== 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 4 items / 3 deselected

|  |  |
| --- | --- |
| test\_plat.py . | [100%] |

================== 1 passed, 3 deselected in 0.12 seconds ==================

then the unmarked-tests will not be run. It is thus a way to restrict the run to the specific tests.

**Automatically adding markers based on test names**

If you a test suite where test function names indicate a certain type of test, you can implement a hook that automatically defines markers so that you can use the -m option with it. Let’s look at this test module:

# content of test\_module.py

**def** test\_interface\_simple():

**assert** 0

**def** test\_interface\_complex():

**assert** 0

**def** test\_event\_simple():

**assert** 0

**def** test\_something\_else():

**assert** 0

We want to dynamically define two markers and can do it in a conftest.py plugin:

# content of conftest.py

**import pytest**

**26.4. Working with custom markers** **211**

**pytest Documentation, Release 3.6**

**def** pytest\_collection\_modifyitems(items):

**for** item **in** items:

**if** "interface" **in** item.nodeid:

item.add\_marker(pytest.mark.interface)

**elif** "event" **in** item.nodeid:

item.add\_marker(pytest.mark.event)

We can now use the -m option to select one set:

$ pytest -m interface --tb=short

=========================== 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 4 items / 2 deselected

|  |  |
| --- | --- |
| test\_module.py FF | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_interface\_simple \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

test\_module.py:3: in test\_interface\_simple

assert 0

* assert 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_interface\_complex \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

test\_module.py:6: in test\_interface\_complex

assert 0

* assert 0

================== 2 failed, 2 deselected in 0.12 seconds ==================

or to select both “event” and “interface” tests:

$ pytest -m "interface or event" --tb=short

=========================== 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 4 items / 1 deselected

|  |  |
| --- | --- |
| test\_module.py FFF | [100%] |

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_interface\_simple \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

test\_module.py:3: in test\_interface\_simple

assert 0

* assert 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_interface\_complex \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

test\_module.py:6: in test\_interface\_complex

assert 0

* assert 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_event\_simple \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

test\_module.py:9: in test\_event\_simple

assert 0

* assert 0

================== 3 failed, 1 deselected in 0.12 seconds ==================

**212** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

**A session-fixture which can look at all collected tests**

A session-scoped fixture effectively has access to all collected test items. Here is an example of a fixture function which walks all collected tests and looks if their test class defines a callme method and calls it:

# content of conftest.py

**import pytest**

**@pytest**.fixture(scope="session", autouse=**True**)

**def** callattr\_ahead\_of\_alltests(request):

print ("callattr\_ahead\_of\_alltests called")

seen = set([**None**])

session = request.node

**for** item **in** session.items:

cls = item.getparent(pytest.Class)

**if** cls **not in** seen:

**if** hasattr(cls.obj,"callme"):

cls.obj.callme()

seen.add(cls)

test classes may now define a callme method which will be called ahead of running any tests:

# content of test\_module.py

**class TestHello**(object):

**@classmethod**

**def** callme(cls):

print ("callme called!")

**def** test\_method1(self):

print ("test\_method1 called")

**def** test\_method2(self):

print ("test\_method1 called")

**class TestOther**(object):

**@classmethod**

**def** callme(cls):

print ("callme other called")

**def** test\_other(self):

print ("test other")

* works with unittest as well ...

**import unittest**

**class SomeTest**(unittest.TestCase):

**@classmethod**

**def** callme(self):

print ("SomeTest callme called")

**def** test\_unit1(self):

print ("test\_unit1 method called")

If you run this without output capturing:

* pytest -q -s test\_module.py callattr\_ahead\_of\_alltests called

**26.5. A session-fixture which can look at all collected tests** **213**

**pytest Documentation, Release 3.6**

callme called!

callme other called

SomeTest callme called

test\_method1 called

.test\_method1 called

.test other

.test\_unit1 method called

.

4 passed in 0.12 seconds

**Changing standard (Python) test discovery**

**Ignore paths during test collection**

You can easily ignore certain test directories and modules during collection by passing the --ignore=path option on the cli. pytest allows multiple --ignore options. Example:

tests/

|-- example

* |-- test\_example\_01.py
* |-- test\_example\_02.py
* '-- test\_example\_03.py |-- foobar
* |-- test\_foobar\_01.py
* |-- test\_foobar\_02.py
* '-- test\_foobar\_03.py '-- hello

'-- world

|-- test\_world\_01.py |-- test\_world\_02.py '-- test\_world\_03.py

Now if you invoke pytest with --ignore=tests/foobar/test\_foobar\_03.py --ignore=tests/hello/, you will see that pytest only collects test-modules, which do not match the patterns specified:

========= test session starts ==========

platform darwin -- Python 2.7.10, pytest-2.8.2, py-1.4.30, pluggy-0.3.1

rootdir: $REGENDOC\_TMPDIR, inifile:

collected 5 items

tests/example/test\_example\_01.py .

tests/example/test\_example\_02.py .

tests/example/test\_example\_03.py .

tests/foobar/test\_foobar\_01.py .

tests/foobar/test\_foobar\_02.py .

======= 5 passed in 0.02 seconds =======

**Deselect tests during test collection**

Tests can individually be deselected during collection by passing the --deselect=item option. For exam-ple, say tests/foobar/test\_foobar\_01.py contains test\_a and test\_b. You can run all of the

**214** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

tests within tests/ except for tests/foobar/test\_foobar\_01.py::test\_a by invoking pytest with --deselect tests/foobar/test\_foobar\_01.py::test\_a. pytest allows multiple --deselect

options.

**Keeping duplicate paths specified from command line**

Default behavior of pytest is to ignore duplicate paths specified from the command line. Example:

pytest path\_a path\_a

...

collected 1 item

...

Just collect tests once.

To collect duplicate tests, use the --keep-duplicates option on the cli. Example:

pytest --keep-duplicates path\_a path\_a

...

collected 2 items

...

As the collector just works on directories, if you specify twice a single test file, pytest will still collect it twice, no matter if the --keep-duplicates is not specified. Example:

pytest test\_a.py test\_a.py

...

collected 2 items

...

**Changing directory recursion**

You can set the [norecursedirs](#page159) option in an ini-file, for example your pytest.ini in the project root directory:

* content of pytest.ini

[pytest]

norecursedirs = .svn \_build tmp\*

This would tell pytest to not recurse into typical subversion or sphinx-build directories or into any tmp prefixed directory.

**Changing naming conventions**

You can configure different naming conventions by setting the [python\_files](#page160), [python\_classes](#page160) and [python\_functions](#page160) configuration options. Example:

* content of pytest.ini
* can also be defined in tox.ini or setup.cfg file, although the section
* name in setup.cfg files should be "tool:pytest"

[pytest]

python\_files=check\_\*.py

**26.6. Changing standard (Python) test discovery** **215**

**pytest Documentation, Release 3.6**

python\_classes=Check

python\_functions=\*\_check

This would make pytest look for tests in files that match the check\_\* .py glob-pattern, Check prefixes in classes, and functions and methods that match \*\_check. For example, if we have:

* content of check\_myapp.py **class CheckMyApp**(object):

**def** simple\_check(self): **pass**

**def** complex\_check(self): **pass**

then the test collection looks like this:

$ 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: pytest.ini collected 2 items

<Module 'check\_myapp.py'>

<Class 'CheckMyApp'>

<Instance '()'>

<Function 'simple\_check'>

<Function 'complex\_check'>

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

Note: the python\_functions and python\_classes options has no effect for unittest.TestCase test discovery because pytest delegates detection of test case methods to unittest code.

**Interpreting cmdline arguments as Python packages**

You can use the --pyargs option to make pytest try interpreting arguments as python package names, deriving their file system path and then running the test. For example if you have unittest2 installed you can type:

pytest --pyargs unittest2.test.test\_skipping -q

which would run the respective test module. Like with other options, through an ini-file and the [addopts](#page156) option you can make this change more permanently:

* content of pytest.ini [pytest]

addopts = --pyargs

Now a simple invocation of pytest NAME will check if NAME exists as an importable package/module and other-wise treat it as a filesystem path.

**Finding out what is collected**

You can always peek at the collection tree without running tests like this:

**216** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

. $ pytest --collect-only pythoncollection.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: pytest.ini collected 3 items

<Module 'CWD/pythoncollection.py'>

<Function 'test\_function'>

<Class 'TestClass'>

<Instance '()'>

<Function 'test\_method'>

<Function 'test\_anothermethod'>

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

**Customizing test collection**

You can easily instruct pytest to discover tests from every Python file:

* content of pytest.ini

[pytest]

python\_files = \*.py

However, many projects will have a setup.py which they don’t want to be imported. Moreover, there may files only importable by a specific python version. For such cases you can dynamically define files to be ignored by listing them in a conftest.py file:

* content of conftest.py **import sys**

collect\_ignore = ["setup.py"]

**if** sys.version\_info[0]>2:

collect\_ignore.append("pkg/module\_py2.py")

and then if you have a module file like this:

* content of pkg/module\_py2.py **def** test\_only\_on\_python2():

**try**:

**assert** 0 **except** Exception, e:

**pass**

and a setup.py dummy file like this:

# content of setup.py

0/0 # will raise exception if imported

If you run with a Python 2 interpreter then you will find the one test and will leave out the setup.py file:

#$ pytest --collect-only

====== test session starts ======

platform linux2 -- Python 2.7.10, pytest-2.9.1, py-1.4.31, pluggy-0.3.1 rootdir: $REGENDOC\_TMPDIR, inifile: pytest.ini collected 1 items

<Module 'pkg/module\_py2.py'>

<Function 'test\_only\_on\_python2'>

**26.6. Changing standard (Python) test discovery** **217**

**pytest Documentation, Release 3.6**

====== no tests ran in 0.04 seconds ======

If you run with a Python 3 interpreter both the one test and the setup.py file will be left out:

$ 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: pytest.ini collected 0 items

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

**Working with non-python tests**

**A basic example for specifying tests in Yaml files**

Here is an example conftest.py (extracted from Ali Afshnars special purpose [pytest-yamlwsgi](http://bitbucket.org/aafshar/pytest-yamlwsgi/src/tip/pytest_yamlwsgi.py) plugin). This conftest.py will collect test\*.yml files and will execute the yaml-formatted content as custom tests:

# content of conftest.py

**import pytest**

**def** pytest\_collect\_file(parent, path):

**if** path.ext==".yml" **and** path.basename.startswith("test"):

**return** YamlFile(path, parent)

**class YamlFile**(pytest.File):

**def** collect(self):

**import yaml** # we need a yaml parser, e.g. PyYAML

raw = yaml.safe\_load(self.fspath.open())

**for** name, spec **in** sorted(raw.items()):

**yield** YamlItem(name,self, spec)

**class YamlItem**(pytest.Item):

**def** \_\_init\_\_(self, name, parent, spec):

super(YamlItem, self).\_\_init\_\_(name, parent)

self.spec = spec

**def** runtest(self):

**for** name, value **in** sorted(self.spec.items()):

* some custom test execution (dumb example follows) **if** name!=value:

**raise** YamlException(self, name, value)

**def** repr\_failure(self, excinfo):

""" called when self.runtest() raises an exception. """

**if** isinstance(excinfo.value, YamlException):

**return** "**\n**".join(

[

**218** **Chapter 26. Examples and customization tricks**

**pytest Documentation, Release 3.6**

"usecase execution failed",

* spec failed: %r: %r" % excinfo.value.args[1:3],
* no further details known at this point.",

]

)

**def** reportinfo(self):

**return** self.fspath,0,"usecase:%s"%self.name

**class YamlException**(Exception):

""" custom exception for error reporting. """

You can create a simple example file:

* test\_simple.yml ok:

sub1: sub1

hello:

world: world

some: other

and if you installed [PyYAML](https://pypi.org/project/PyYAML/) or a compatible YAML-parser you can now execute the test specification:

nonpython $ pytest test\_simple.yml

=========================== test session starts ============================

platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y rootdir: $REGENDOC\_TMPDIR/nonpython, inifile: collected 2 items

test\_simple.yml F. [100%]

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ usecase: hello \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

usecase execution failed

spec failed: 'some': 'other'

no further details known at this point.

==================== 1 failed, 1 passed in 0.12 seconds ====================

You get one dot for the passing sub1: sub1 check and one failure. Obviously in the above conftest.py you’ll want to implement a more interesting interpretation of the yaml-values. You can easily write your own domain specific testing language this way.

Note: repr\_failure(excinfo) is called for representing test failures. If you create custom collection nodes you can return an error representation string of your choice. It will be reported as a (red) string.

reportinfo() is used for representing the test location and is also consulted when reporting in verbose mode:

nonpython $ pytest -v

=========================== 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: .pytest\_cache

rootdir: $REGENDOC\_TMPDIR/nonpython, inifile:

collecting ... collected 2 items

**26.7. Working with non-python tests** **219**

**pytest Documentation, Release 3.6**

|  |  |  |
| --- | --- | --- |
| test\_simple.yml::hello FAILED |  | [ 50%] |
| test\_simple.yml::ok PASSED |  | [100%] |
| ================================= FAILURES | | ================================= |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | usecase: hello \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | |
| usecase execution failed |  |  |
| spec failed: 'some': 'other' |  |  |
| no further details known at | this point. |  |
| ==================== 1 failed, | 1 passed in | 0.12 seconds ==================== |
|  |  |  |

While developing your custom test collection and execution it’s also interesting to just look at the collection tree:

nonpython $ 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/nonpython, inifile: collected 2 items

<YamlFile 'test\_simple.yml'>

<YamlItem 'hello'>

<YamlItem 'ok'>

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

**220** **Chapter 26. Examples and customization tricks**

**CHAPTER 27**

**Setting up bash completion**

When using bash as your shell, pytest can use argcomplete (<https://argcomplete.readthedocs.io/>) for auto-completion. For this argcomplete needs to be installed and enabled.

Install argcomplete using:

sudo pip install 'argcomplete>=0.5.7'

For global activation of all argcomplete enabled python applications run:

sudo activate-**global**-python-argcomplete

For permanent (but not global) pytest activation, use:

register-python-argcomplete pytest >> ~/.bashrc

For one-time activation of argcomplete for pytest only, use:

eval "$(register-python-argcomplete pytest)"

**221**

**pytest Documentation, Release 3.6**

**222** **Chapter 27. Setting up bash completion**

**CHAPTER 28**

**Backwards Compatibility Policy**

Keeping backwards compatibility has a very high priority in the pytest project. Although we have deprecated func-tionality over the years, most of it is still supported. All deprecations in pytest were done because simpler or more efficient ways of accomplishing the same tasks have emerged, making the old way of doing things unnecessary.

With the pytest 3.0 release we introduced a clear communication scheme for when we will actually remove the old busted joint and politely ask you to use the new hotness instead, while giving you enough time to adjust your tests or raise concerns if there are valid reasons to keep deprecated functionality around.

To communicate changes we are already issuing deprecation warnings, but they are not displayed by default. In pytest 3.0 we changed the default setting so that pytest deprecation warnings are displayed if not explicitly silenced (with --disable-pytest-warnings).

We will only remove deprecated functionality in major releases (e.g. if we deprecate something in 3.0 we will remove it in 4.0), and keep it around for at least two minor releases (e.g. if we deprecate something in 3.9 and 4.0 is the next release, we will not remove it in 4.0 but in 5.0).

**Deprecation Roadmap**

We track deprecation and removal of features using milestones and the [deprecation](https://github.com/pytest-dev/pytest/issues?q=label%3A%22type%3A+deprecation%22) and [removal](https://github.com/pytest-dev/pytest/labels/type%3A%20removal) labels on GitHub.

Following our deprecation policy, after starting issuing deprecation warnings we keep features for at least two minor versions before considering removal.

**223**

**pytest Documentation, Release 3.6**

**224** **Chapter 28. Backwards Compatibility Policy**

**CHAPTER 29**

**Historical Notes**

This page lists features or behavior from previous versions of pytest which have changed over the years. They are kept here as a historical note so users looking at old code can find documentation related to them.

**cache plugin integrated into the core**

New in version 2.8.

The functionality of the [core cache](#page85) plugin was previously distributed as a third party plugin named pytest-cache. The core plugin is compatible regarding command line options and API usage except that you can only store/receive data between test runs that is json-serializable.

**funcargs and pytest\_funcarg\_\_**

Changed in version 2.3.

In versions prior to 2.3 there was no @pytest.fixture marker and you had to use a magic pytest\_funcarg\_\_NAME prefix for the fixture factory. This remains and will remain supported but is not anymore advertised as the primary means of declaring fixture functions.

**@pytest.yield\_fixture decorator**

Changed in version 2.10.

Prior to version 2.10, in order to use a yield statement to execute teardown code one had to mark a fixture using the yield\_fixture marker. From 2.10 onward, normal fixtures can use yield directly so the yield\_fixture decorator is no longer needed and considered deprecated.

**[pytest] header in setup.cfg**

Changed in version 3.0.

Prior to 3.0, the supported section name was [pytest]. Due to how this may collide with some distutils commands, the recommended section name for setup.cfg files is now [tool:pytest].

Note that for pytest.ini and tox.ini files the section name is [pytest].

**225**

**pytest Documentation, Release 3.6**

**Applying marks to @pytest.mark.parametrize parameters**

Changed in version 3.1.

Prior to version 3.1 the supported mechanism for marking values used the syntax:

**import pytest**

**@pytest**.mark.parametrize("test\_input,expected", [

("3+5", 8),

("2+4", 6),

pytest.mark.xfail(("6\*9", 42),),

])

**def** test\_eval(test\_input, expected):

**assert** eval(test\_input)==expected

This was an initial hack to support the feature but soon was demonstrated to be incomplete, broken for passing func-tions or applying multiple marks with the same name but different parameters.

The old syntax is planned to be removed in pytest-4.0.

**@pytest.mark.parametrize argument names as a tuple**

Changed in version 2.4.

In versions prior to 2.4 one needed to specify the argument names as a tuple. This remains valid but the simpler "name1,name2,..." comma-separated-string syntax is now advertised first because it’s easier to write and pro-duces less line noise.

**setup: is now an “autouse fixture”**

Changed in version 2.3.

During development prior to the pytest-2.3 release the name pytest.setup was used but before the release it was renamed and moved to become part of the general fixture mechanism, namely [Autouse fixtures (xUnit setup on](#page45) [steroids)](#page45)

**Conditions as strings instead of booleans**

Changed in version 2.4.

Prior to pytest-2.4 the only way to specify skipif/xfail conditions was to use strings:

**import sys**

**@pytest**.mark.skipif("sys.version\_info >= (3,3)")

**def** test\_function():

...

During test function setup the skipif condition is evaluated by calling eval('sys.version\_info >= (3,0)',namespace). The namespace contains all the module globals, and os and sys as a minimum.

Since pytest-2.4 [boolean conditions](#page73) are considered preferable because markers can then be freely imported between test modules. With strings you need to import not only the marker but all variables used by the marker, which violates encapsulation.

**226** **Chapter 29. Historical Notes**

**pytest Documentation, Release 3.6**

The reason for specifying the condition as a string was that pytest can report a summary of skip conditions based purely on the condition string. With conditions as booleans you are required to specify a reason string.

Note that string conditions will remain fully supported and you are free to use them if you have no need for cross-importing markers.

The evaluation of a condition string in pytest.mark.skipif(conditionstring) or pytest.mark.xfail(conditionstring) takes place in a namespace dictionary which is constructed as follows:

* the namespace is initialized by putting the sys and os modules and the pytest config object into it.
* updated with the module globals of the test function for which the expression is applied.

The pytest config object allows you to skip based on a test configuration value which you might have added:

**@pytest**.mark.skipif("not config.getvalue('db')")

**def** test\_function(...):

...

The equivalent with “boolean conditions” is:

**@pytest**.mark.skipif(**not** pytest.config.getvalue("db"),reason="--db was not specified")

**def** test\_function(...):

**pass**

Note: You cannot use pytest.config.getvalue() in code imported before pytest’s argument pars-ing takes place. For example, conftest.py files are imported before command line parsing and thus config.getvalue() will not execute correctly.

**pytest.set\_trace()**

Changed in version 2.4.

Previous to version 2.4 to set a break point in code one needed to use pytest.set\_trace():

**import pytest**

**def** test\_function():

...

pytest.set\_trace() # invoke PDB debugger and tracing

This is no longer needed and one can use the native import pdb;pdb.set\_trace() call directly.

For more details see [Setting breakpoints](#page17).

**29.9. pytest.set\_trace()** **227**

**pytest Documentation, Release 3.6**

**228** **Chapter 29. Historical Notes**

**CHAPTER 30**

**License**

Distributed under the terms of the [MIT](https://github.com/pytest-dev/pytest/blob/master/LICENSE) license, pytest is free and open source software.

The MIT License (MIT)

Copyright (c) 2004-2017 Holger Krekel **and** others

Permission **is** hereby granted, free of charge, to any person obtaining a copy of this software **and** associated documentation files (the "Software"), to deal **in** the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, **and**/**or** sell copies of the Software, **and** to permit persons to whom the Software **is** furnished to do so, subject to the following conditions:

The above copyright notice **and** this permission notice shall be included **in** all copies **or** substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

**229**

**pytest Documentation, Release 3.6**

**230** **Chapter 30. License**

**CHAPTER 31**

**Contribution getting started**

Contributions are highly welcomed and appreciated. Every little help counts, so do not hesitate!

Contribution links

* [Contribution getting started](#page239)

– [Feature requests and feedback](#page239)

– [Report bugs](#page239)

– [Fix bugs](#page240)

– [Implement features](#page240)

– [Write documentation](#page240)

– [Submitting Plugins to pytest-dev](#page240)

– [Preparing Pull Requests](#page241)

– [Joining the Development Team](#page243)

**Feature requests and feedback**

Do you like pytest? Share some love on Twitter or in your blog posts!

We’d also like to hear about your propositions and suggestions. Feel free to [submit them as issues](https://github.com/pytest-dev/pytest/issues) and:

* Explain in detail how they should work.
* Keep the scope as narrow as possible. This will make it easier to implement.

**Report bugs**

Report bugs for pytest in the [issue tracker](https://github.com/pytest-dev/pytest/issues).

If you are reporting a bug, please include:

* Your operating system name and version.
* Any details about your local setup that might be helpful in troubleshooting, specifically the Python interpreter version, installed libraries, and pytest version.

**231**

**pytest Documentation, Release 3.6**

• Detailed steps to reproduce the bug.

If you can write a demonstration test that currently fails but should pass (xfail), that is a very useful commit to make as well, even if you cannot fix the bug itself.

**Fix bugs**

Look through the [GitHub issues for bugs](https://github.com/pytest-dev/pytest/labels/type:%20bug).

[Talk](#page255) to developers to find out how you can fix specific bugs.

Don’t forget to check the issue trackers of your favourite plugins, too!

**Implement features**

Look through the [GitHub issues for enhancements](https://github.com/pytest-dev/pytest/labels/type:%20enhancement).

[Talk](#page255) to developers to find out how you can implement specific features.

**Write documentation**

Pytest could always use more documentation. What exactly is needed?

* More complementary documentation. Have you perhaps found something unclear?
* Documentation translations. We currently have only English.
* Docstrings. There can never be too many of them.
* Blog posts, articles and such – they’re all very appreciated.

You can also edit documentation files directly in the GitHub web interface, without using a local copy. This can be convenient for small fixes.

Note: Build the documentation locally with the following command:

$ tox -e docs

The built documentation should be available in the doc/en/\_build/.

Where ‘en’ refers to the documentation language.

**Submitting Plugins to pytest-dev**

Pytest development of the core, some plugins and support code happens in repositories living under the pytest-dev organisations:

* [pytest-dev on GitHub](https://github.com/pytest-dev)
* [pytest-dev on Bitbucket](https://bitbucket.org/pytest-dev)

**232** **Chapter 31. Contribution getting started**

**pytest Documentation, Release 3.6**

All pytest-dev Contributors team members have write access to all contained repositories. Pytest core and plugins are generally developed using [pull requests](#page241) to respective repositories.

The objectives of the pytest-dev organisation are:

* Having a central location for popular pytest plugins
* Sharing some of the maintenance responsibility (in case a maintainer no longer wishes to maintain a plugin)

You can submit your plugin by subscribing to the [pytest-dev mail list](https://mail.python.org/mailman/listinfo/pytest-dev) and writing a mail pointing to your existing pytest plugin repository which must have the following:

* PyPI presence with a setup.py that contains a license, pytest- prefixed name, version number, authors, short and long description.
* a tox.ini for running tests using [tox](https://tox.readthedocs.io).
* a README.txt describing how to use the plugin and on which platforms it runs.
* a LICENSE.txt file or equivalent containing the licensing information, with matching info in setup.py.
* an issue tracker for bug reports and enhancement requests.
* a [changelog](http://keepachangelog.com/)

If no contributor strongly objects and two agree, the repository can then be transferred to the pytest-dev organisa-tion.

Here’s a rundown of how a repository transfer usually proceeds (using a repository named joedoe/pytest-xyz as example):

* joedoe transfers repository ownership to pytest-dev administrator calvin.
* calvin creates pytest-xyz-admin and pytest-xyz-developers teams, inviting joedoe to both as maintainer.
* calvin transfers repository to pytest-dev and configures team access:

– pytest-xyz-admin admin access;

– pytest-xyz-developers write access;

The pytest-dev/Contributors team has write access to all projects, and every project administrator is in it.

We recommend that each plugin has at least three people who have the right to release to PyPI.

Repository owners can rest assured that no pytest-dev administrator will ever make releases of your repository or take ownership in any way, except in rare cases where someone becomes unresponsive after months of contact attempts. As stated, the objective is to share maintenance and avoid “plugin-abandon”.

**Preparing Pull Requests**

**Short version**

1. Fork the repository.
2. Enable and install [pre-commit](https://pre-commit.com) to ensure style-guides and code checks are followed.
3. Target master for bugfixes and doc changes.
4. Target features for new features or functionality changes.
5. Follow PEP-8 for naming and [black](https://github.com/ambv/black) for formatting.
6. Tests are run using tox:

**31.7. Preparing Pull Requests** **233**

**pytest Documentation, Release 3.6**

tox -e linting,py27,py36

The test environments above are usually enough to cover most cases locally.

1. Write a changelog entry: changelog/2574.bugfix, use issue id number and one of bugfix, removal, feature, vendor, doc or trivial for the issue type.
2. Unless your change is a trivial or a documentation fix (e.g., a typo or reword of a small section) please add yourself to the AUTHORS file, in alphabetical order.

**Long version**

What is a “pull request”? It informs the project’s core developers about the changes you want to review and merge. Pull requests are stored on [GitHub servers](https://github.com/pytest-dev/pytest/pulls). Once you send a pull request, we can discuss its potential modifications and even add more commits to it later on. There’s an excellent tutorial on how Pull Requests work in the [GitHub Help](https://help.github.com/articles/using-pull-requests/) [Center](https://help.github.com/articles/using-pull-requests/).

Here is a simple overview, with pytest-specific bits:

1. Fork the [pytest GitHub repository](https://github.com/pytest-dev/pytest). It’s fine to use pytest as your fork repository name because it will live under your user.
2. Clone your fork locally using [git](https://git-scm.com/) and create a branch:

$ git clone git@github.com:YOUR\_GITHUB\_USERNAME/pytest.git $ cd pytest

# now, to fix a bug create your own branch off "master":

* + git checkout -b your-bugfix-branch-name master
* or to instead add a feature create your own branch off "features":
  + git checkout -b your-feature-branch-name features

Given we have “major.minor.micro” version numbers, bugfixes will usually be released in micro releases whereas features will be released in minor releases and incompatible changes in major releases.

If you need some help with Git, follow this quick start guide: <https://git.wiki.kernel.org/index.php/QuickStart>

1. Install [pre-commit](https://pre-commit.com) and its hook on the pytest repo:

$ pip install --user pre-commit $ pre-commit install

Afterwards pre-commit will run whenever you commit.

<https://pre-commit.com/>is a framework for managing and maintaining multi-language pre-commit hooks to ensure code-style and code formatting is consistent.

1. Install tox

Tox is used to run all the tests and will automatically setup virtualenvs to run the tests in. (will implicitly use <http://www.virtualenv.org/en/latest/>):

$ pip install tox

1. Run all the tests

You need to have Python 2.7 and 3.6 available in your system. Now running tests is as simple as issuing this command:

**234** **Chapter 31. Contribution getting started**

**pytest Documentation, Release 3.6**

$ tox -e linting,py27,py36

This command will run tests via the “tox” tool against Python 2.7 and 3.6 and also perform “lint” coding-style checks.

1. You can now edit your local working copy and run the tests again as necessary. Please follow PEP-8 for naming.

You can pass different options to tox. For example, to run tests on Python 2.7 and pass options to pytest (e.g. enter pdb on failure) to pytest you can do:

$ tox -e py27 -- --pdb

Or to only run tests in a particular test module on Python 3.6: $ tox -e py36 -- testing/test\_config.py

When committing, pre-commit will re-format the files if necessary.

1. Commit and push once your tests pass and you are happy with your change(s):

$ git commit -a -m "<commit message>" $ git push -u

1. Create a new changelog entry in changelog. The file should be named <issueid>.<type>, where issueid is the number of the issue related to the change and type is one of bugfix, removal, feature, vendor, doc or trivial.
2. Add yourself to AUTHORS file if not there yet, in alphabetical order.
3. Finally, submit a pull request through the GitHub website using this data:

head-fork: YOUR\_GITHUB\_USERNAME/pytest

compare: your-branch-name

base-fork: pytest-dev/pytest

base: master # if it's a bugfix

base: features # if it's a feature

**Joining the Development Team**

Anyone who has successfully seen through a pull request which did not require any extra work from the development team to merge will themselves gain commit access if they so wish (if we forget to ask please send a friendly reminder). This does not mean your workflow to contribute changes, everyone goes through the same pull-request-and-review process and no-one merges their own pull requests unless already approved. It does however mean you can participate in the development process more fully since you can merge pull requests from other contributors yourself after having reviewed them.

**31.8. Joining the Development Team** **235**

**pytest Documentation, Release 3.6**

**236** **Chapter 31. Contribution getting started**

**CHAPTER 32**

**Development Guide**

Some general guidelines regarding development in pytest for maintainers and contributors. Nothing here is set in stone and can’t be changed, feel free to suggest improvements or changes in the workflow.

**Code Style**

* [PEP-8](https://www.python.org/dev/peps/pep-0008)
* [flake8](https://pypi.org/project/flake8/) for quality checks
* [invoke](http://www.pyinvoke.org/) to automate development tasks

**Branches**

We have two long term branches:

* master: contains the code for the next bugfix release.
* features: contains the code with new features for the next minor release.

The official repository usually does not contain topic branches, developers and contributors should create topic branches in their own forks.

Exceptions can be made for cases where more than one contributor is working on the same topic or where it makes sense to use some automatic capability of the main repository, such as automatic docs from readthedocs for a branch dealing with documentation refactoring.

**Issues**

Any question, feature, bug or proposal is welcome as an issue. Users are encouraged to use them whenever they need.

GitHub issues should use labels to categorize them. Labels should be created sporadically, to fill a niche; we should avoid creating labels just for the sake of creating them.

Each label should include a description in the GitHub’s interface stating its purpose.

Labels are managed using [labels](https://github.com/hackebrot/labels). All the labels in the repository are kept in .github/labels.toml, so any changes should be via PRs to that file. After a PR is accepted and merged, one of the maintainers must manually synchronize the labels file with the GitHub repository.

**237**

**pytest Documentation, Release 3.6**

**Temporary labels**

To classify issues for a special event it is encouraged to create a temporary label. This helps those involved to find the relevant issues to work on. Examples of that are sprints in Python events or global hacking events.

* temporary: EP2017 sprint: candidate issues or PRs tackled during the EuroPython 2017 Issues created at those events should have other relevant labels added as well.

Those labels should be removed after they are no longer relevant.

**Release Procedure**

Our current policy for releasing is to aim for a bugfix every few weeks and a minor release every 2-3 months. The idea is to get fixes and new features out instead of trying to cram a ton of features into a release and by consequence taking a lot of time to make a new one.

Important: pytest releases must be prepared on Linux because the docs and examples expect to be executed in that platform.

1. Create a branch release-X.Y.Z with the version for the release.
   * patch releases: from the latest master;
   * minor releases: from the latest features; then merge with the latest master; Ensure your are in a clean work tree.
2. Using tox, generate docs, changelog, announcements: $ tox -e release -- <VERSION>

This will generate a commit with all the changes ready for pushing.

1. Open a PR for this branch targeting master.
2. After all tests pass and the PR has been approved, publish to PyPI by pushing the tag: git push git**@github**.com:pytest-dev/pytest.git <VERSION>

Wait for the deploy to complete, then make sure it is [available on PyPI](https://pypi.org/project/pytest).

1. Send an email announcement with the contents from: doc/en/announce/release-<VERSION>.rst

To the following mailing lists:

* + [pytest-dev@python.org](mailto:pytest-dev@python.org) (all releases)
  + [python-announce-list@python.org](mailto:python-announce-list@python.org) (all releases)
  + [testing-in-python@lists.idyll.org](mailto:testing-in-python@lists.idyll.org) (only major/minor releases)

And announce it on [Twitter](https://twitter.com/) with the #pytest hashtag.

6. After a minor/major release, merge release-X.Y.Z into master and push (or open a PR).

**238** **Chapter 32. Development Guide**

**CHAPTER 33**

**Talks and Tutorials**

**Books**

* [Python Testing with pytest, by Brian Okken (2017)](https://pragprog.com/book/bopytest/python-testing-with-pytest).

**Talks and blog postings**

* [Pythonic testing, Igor Starikov (Russian, PyNsk, November 2016)](https://www.youtube.com/watch?v=_92nfdd5nK8).
* [pytest - Rapid Simple Testing, Florian Bruhin, Swiss Python Summit 2016](https://www.youtube.com/watch?v=rCBHkQ_LVIs).
* [Improve your testing with Pytest and Mock, Gabe Hollombe, PyCon SG 2015](https://www.youtube.com/watch?v=RcN26hznmk4).
* [Introduction to pytest, Andreas Pelme, EuroPython 2014](https://www.youtube.com/watch?v=LdVJj65ikRY).
* [Advanced Uses of py.test Fixtures, Floris Bruynooghe, EuroPython 2014](https://www.youtube.com/watch?v=IBC_dxr-4ps).
* [Why i use py.test and maybe you should too, Andy Todd, Pycon AU 2013](https://www.youtube.com/watch?v=P-AhpukDIik)
* [3-part blog series about pytest from @pydanny alias Daniel Greenfeld (January 2014)](http://pydanny.com/pytest-no-boilerplate-testing.html)
* [pytest: helps you write better Django apps, Andreas Pelme, DjangoCon Europe 2014](https://www.youtube.com/watch?v=aaArYVh6XSM).
* [pytest fixtures: explicit, modular, scalable](#page31)
* [Testing Django Applications with pytest, Andreas Pelme, EuroPython 2013](https://www.youtube.com/watch?v=aUf8Fkb7TaY).
* [Testes pythonics com py.test, Vinicius Belchior Assef Neto, Plone Conf 2013, Brazil](https://www.youtube.com/watch?v=QUKoq2K7bis).
* [Introduction to py.test fixtures, FOSDEM 2013, Floris Bruynooghe](https://www.youtube.com/watch?v=bJhRW4eZMco).
* [pytest feature and release highlights, Holger Krekel (GERMAN, October 2013)](http://pyvideo.org/video/2429/pytest-feature-and-new-release-highlights)
* [pytest introduction from Brian Okken (January 2013)](http://pythontesting.net/framework/pytest-introduction/)
* pycon australia 2012 pytest talk from Brianna Laugher ([video](http://www.youtube.com/watch?v=DTNejE9EraI), [slides](http://www.slideshare.net/pfctdayelise/funcargs-other-fun-with-pytest), [code](https://gist.github.com/3386951))
* [pycon 2012 US talk video from Holger Krekel](http://www.youtube.com/watch?v=9LVqBQcFmyw)
* [monkey patching done right](http://tetamap.wordpress.com/2009/03/03/monkeypatching-in-unit-tests-done-right/) (blog post, consult monkeypatch plugin for up-to-date API) Test parametrization:
* generating parametrized tests with fixtures.
* [test generators and cached setup](http://bruynooghe.blogspot.com/2010/06/pytest-test-generators-and-cached-setup.html)
* [parametrizing tests, generalized](http://tetamap.wordpress.com/2009/05/13/parametrizing-python-tests-generalized/) (blog post)

**239**

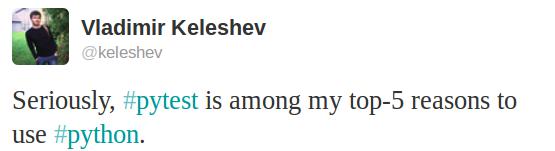
**pytest Documentation, Release 3.6**

* [putting test-hooks into local or global plugins](http://tetamap.wordpress.com/2009/05/14/putting-test-hooks-into-local-and-global-plugins/) (blog post) Assertion introspection:
* [(07/2011) Behind the scenes of pytest’s new assertion rewriting](http://pybites.blogspot.com/2011/07/behind-scenes-of-pytests-new-assertion.html) Distributed testing:
* [simultaneously test your code on all platforms](http://tetamap.wordpress.com/2009/03/23/new-simultanously-test-your-code-on-all-platforms/) (blog entry) Plugin specific examples:
* [skipping slow tests by default in pytest](http://bruynooghe.blogspot.com/2009/12/skipping-slow-test-by-default-in-pytest.html) (blog entry)
* many examples in the docs for plugins



**240** **Chapter 33. Talks and Tutorials**

**pytest Documentation, Release 3.6**



**33.2. Talks and blog postings** **241**

**pytest Documentation, Release 3.6**

**242** **Chapter 33. Talks and Tutorials**

**CHAPTER 34**

**Project examples**

Here are some examples of projects using pytest (please send notes via [Contact channels](#page255)):

* [PyPy](http://pypy.org), Python with a JIT compiler, running over [21000 tests](http://buildbot.pypy.org/summary?branch=%3Ctrunk%3E)
* the [MoinMoin](http://moinmo.in) Wiki Engine
* [sentry](https://getsentry.com/welcome/), realtime app-maintenance and exception tracking
* [Astropy](http://www.astropy.org/) and [affiliated packages](http://www.astropy.org/affiliated/index.html)
* [tox](http://testrun.org/tox), virtualenv/Hudson integration tool
* [PIDA](http://pida.co.uk) framework for integrated development
* [PyPM](http://code.activestate.com/pypm/) ActiveState’s package manager
* [Fom](http://packages.python.org/Fom/) a fluid object mapper for FluidDB
* [applib](https://github.com/ActiveState/applib) cross-platform utilities
* [six](https://pypi.org/project/six/) Python 2 and 3 compatibility utilities
* [pediapress](http://code.pediapress.com/wiki/wiki) MediaWiki articles
* [mwlib](https://pypi.org/project/mwlib/) mediawiki parser and utility library
* [The Translate Toolkit](http://translate.sourceforge.net/wiki/toolkit/index) for localization and conversion
* [execnet](http://codespeak.net/execnet) rapid multi-Python deployment
* [pylib](https://py.readthedocs.io) cross-platform path, IO, dynamic code library
* [Pacha](http://pacha.cafepais.com/) configuration management in five minutes
* [bbfreeze](https://pypi.org/project/bbfreeze/) create standalone executables from Python scripts
* [pdb++](http://bitbucket.org/antocuni/pdb) a fancier version of PDB
* [py-s3fuse](http://code.google.com/p/py-s3fuse/) Amazon S3 FUSE based filesystem
* [waskr](http://code.google.com/p/waskr/) WSGI Stats Middleware
* [guachi](http://code.google.com/p/guachi/) global persistent configs for Python modules
* [Circuits](https://pypi.org/project/circuits/) lightweight Event Driven Framework
* [pygtk-helpers](http://bitbucket.org/aafshar/pygtkhelpers-main/) easy interaction with PyGTK
* [QuantumCore](http://quantumcore.org/) statusmessage and repoze openid plugin
* [pydataportability](http://pydataportability.net/) libraries for managing the open web

**243**

**pytest Documentation, Release 3.6**

* [XIST](http://www.livinglogic.de/Python/xist/) extensible HTML/XML generator
* [tiddlyweb](https://pypi.org/project/tiddlyweb/) optionally headless, extensible RESTful datastore
* [fancycompleter](http://bitbucket.org/antocuni/fancycompleter/src) for colorful tab-completion
* [Paludis](http://paludis.exherbo.org/) tools for Gentoo Paludis package manager
* [Gerald](http://halfcooked.com/code/gerald/) schema comparison tool
* [abjad](http://code.google.com/p/abjad/) Python API for Formalized Score control
* [bu](http://packages.python.org/bu/) a microscopic build system
* [katcp](https://bitbucket.org/hodgestar/katcp) Telescope communication protocol over Twisted
* [kss plugin timer](https://pypi.org/project/kss.plugin.timer/)
* [pyudev](https://pyudev.readthedocs.io/en/latest/tests/plugins.html) a pure Python binding to the Linux library libudev
* [pytest-localserver](https://bitbucket.org/pytest-dev/pytest-localserver/) a plugin for pytest that provides an httpserver and smtpserver
* [pytest-monkeyplus](https://pypi.org/project/pytest-monkeyplus/) a plugin that extends monkeypatch

These projects help integrate pytest into other Python frameworks:

* [pytest-django](https://pypi.org/project/pytest-django/) for Django
* [zope.pytest](http://packages.python.org/zope.pytest/) for Zope and Grok
* [pytest\_gae](https://pypi.org/project/pytest_gae/0.2.1/) for Google App Engine
* There is [some work](https://github.com/Kotti/Kotti/blob/master/kotti/testing.py) underway for Kotti, a CMS built in Pyramid/Pylons

**Some organisations using pytest**

* [Square Kilometre Array, Cape Town](http://ska.ac.za/)
* [Some Mozilla QA people](http://www.theautomatedtester.co.uk/blog/2011/pytest_and_xdist_plugin.html) use pytest to distribute their Selenium tests
* [Tandberg](http://www.tandberg.com/)
* [Shootq](http://web.shootq.com/)
* [Stups department of Heinrich Heine University Duesseldorf](http://www.stups.uni-duesseldorf.de/projects.php)
* [cellzome](http://www.cellzome.com/)
* [Open End, Gothenborg](http://www.openend.se)
* [Laboratory of Bioinformatics, Warsaw](http://genesilico.pl/)
* [merlinux, Germany](http://merlinux.eu)
* [ESSS, Brazil](http://www.esss.com.br)
* many more ... (please be so kind to send a note via [Contact channels](#page255))

**244** **Chapter 34. Project examples**

**CHAPTER 35**

**Some Issues and Questions**

Note: This FAQ is here only mostly for historic reasons. Checkout [pytest Q&A at Stackoverflow](http://stackoverflow.com/search?q=pytest) for many questions and answers related to pytest and/or use [Contact channels](#page255) to get help.

**On naming, nosetests, licensing and magic**

**How does pytest relate to nose and unittest?**

pytest and [nose](https://nose.readthedocs.io/en/latest/) share basic philosophy when it comes to running and writing Python tests. In fact, you can run many tests written for nose with pytest. [nose](https://nose.readthedocs.io/en/latest/) was originally created as a clone of pytest when pytest was in the 0.8 release cycle. Note that starting with pytest-2.0 support for running unittest test suites is majorly improved.

**how does pytest relate to twisted’s trial?**

Since some time pytest has builtin support for supporting tests written using trial. It does not itself start a reactor, however, and does not handle Deferreds returned from a test in pytest style. If you are using trial’s unittest.TestCase chances are that you can just run your tests even if you return Deferreds. In addition, there also is a dedicated [pytest-twisted](https://pypi.org/project/pytest-twisted/) plugin which allows you to return deferreds from pytest-style tests, allowing the use of [pytest fixtures: explicit,](#page31) [modular, scalable](#page31) and other features.

**how does pytest work with Django?**

In 2012, some work is going into the [pytest-django plugin](https://pypi.org/project/pytest-django/). It substitutes the usage of Django’s manage.py test and allows the use of all pytest features most of which are not available from Django directly.

**What’s this “magic” with pytest? (historic notes)**

Around 2007 (version 0.8) some people thought that pytest was using too much “magic”. It had been part of the [pylib](https://py.readthedocs.io/en/latest/) which contains a lot of unrelated python library code. Around 2010 there was a major cleanup refactoring, which removed unused or deprecated code and resulted in the new pytest PyPI package which strictly contains only test-related code. This release also brought a complete pluginification such that the core is around 300 lines of code and everything else is implemented in plugins. Thus pytest today is a small, universally runnable and customizable testing framework for Python. Note, however, that pytest uses metaprogramming techniques and reading its source is thus likely not something for Python beginners.

**245**

**pytest Documentation, Release 3.6**

A second “magic” issue was the assert statement debugging feature. Nowadays, pytest explicitly rewrites assert statements in test modules in order to provide more useful [assert feedback](#page25). This completely avoids previous issues of confusing assertion-reporting. It also means, that you can use Python’s -O optimization without losing assertions in test modules.

You can also turn off all assertion interaction using the --assert=plain option.

**Why can I use both pytest and py.test commands?**

pytest used to be part of the py package, which provided several developer utilities, all starting with py.<TAB>, thus providing nice TAB-completion. If you install pip install pycmd you get these tools from a separate package. Once pytest became a separate package, the py.test name was retained due to avoid a naming conflict with another tool. This conflict was eventually resolved, and the pytest command was therefore introduced. In future versions of pytest, we may deprecate and later remove the py.test command to avoid perpetuating the confusion.

**pytest fixtures, parametrized tests**

**Is using pytest fixtures versus xUnit setup a style question?**

For simple applications and for people experienced with [nose](https://nose.readthedocs.io/en/latest/) or unittest-style test setup using xUnit style setup proba-bly feels natural. For larger test suites, parametrized testing or setup of complex test resources using fixtures may feel more natural. Moreover, fixtures are ideal for writing advanced test support code (like e.g. the monkeypatch, the tmpdir or capture fixtures) because the support code can register setup/teardown functions in a managed class/module/function scope.

**Can I yield multiple values from a fixture function?**

There are two conceptual reasons why yielding from a factory function is not possible:

* If multiple factories yielded values there would be no natural place to determine the combination policy - in real-world examples some combinations often should not run.
* Calling factories for obtaining test function arguments is part of setting up and running a test. At that point it is not possible to add new test calls to the test collection anymore.

However, with pytest-2.3 you can use the [Fixtures as Function arguments](#page31) decorator and specify params so that all tests depending on the factory-created resource will run multiple times with different parameters.

You can also use the pytest\_generate\_tests hook to implement the [parametrization scheme of your choice](http://tetamap.wordpress.com/2009/05/13/parametrizing-python-tests-generalized/).

See also [Parametrizing tests](#page200) for more examples.

**pytest interaction with other packages**

**Issues with pytest, multiprocess and setuptools?**

On Windows the multiprocess package will instantiate sub processes by pickling and thus implicitly re-import a lot of local modules. Unfortunately, setuptools-0.6.11 does not if \_\_name\_\_=='\_\_main\_\_' protect its generated command line script. This leads to infinite recursion when running a test that instantiates Processes.

As of mid-2013, there shouldn’t be a problem anymore when you use the standard setuptools (note that distribute has been merged back into setuptools which is now shipped directly with virtualenv).

**246** **Chapter 35. Some Issues and Questions**

**CHAPTER 36**

**Contact channels**

* [pytest issue tracker](https://github.com/pytest-dev/pytest/issues) to report bugs or suggest features (for version 2.0 and above).
* [pytest on stackoverflow.com](http://stackoverflow.com/search?q=pytest) to post questions with the tag pytest. New Questions will usually be seen by pytest users or developers and answered quickly.
* [Testing In Python](http://lists.idyll.org/listinfo/testing-in-python): a mailing list for Python testing tools and discussion.
* [pytest-dev at python.org (mailing list)](http://mail.python.org/mailman/listinfo/pytest-dev) pytest specific announcements and discussions.
* [pytest-commit at python.org (mailing list)](http://mail.python.org/mailman/listinfo/pytest-commit): for commits and new issues
* [contribution guide](#page239) for help on submitting pull requests to GitHub.
* #pylib on irc.freenode.net IRC channel for random questions.
* private mail to Holger.Krekel at gmail com if you want to communicate sensitive issues
* [merlinux.eu](http://merlinux.eu) offers pytest and tox-related professional teaching and consulting.

**247**

**pytest Documentation, Release 3.6**

**248** **Chapter 36. Contact channels**

Index

Symbols

\_Result (class in pluggy), [146](#page154)

A

add\_cleanup() (Config method), [136](#page144)

add\_hookcall\_monitoring() (PluginManager method), [144](#page152)

add\_hookspecs() (PluginManager method), [143](#page151)

add\_mark() (MarkInfo method), [139](#page147)

add\_marker() (Node method), [142](#page150)

add\_report\_section() (Item method), [138](#page146)

addcall() (Metafunc method), [140](#page148)

addfinalizer() (FixtureRequest method), [122](#page130)

addfinalizer() (Node method), [142](#page150)

addhooks() (PytestPluginManager method), [145](#page153)

addini() (Parser method), [143](#page151)

addinivalue\_line() (Config method), [136](#page144)

addoption() (Parser method), [142](#page150)

addopts

configuration value, [148](#page156)

applymarker() (FixtureRequest method), [122](#page130)

approx() (in module \_pytest.python\_api), [109](#page117)

args (Mark attribute), [139](#page147)

args (MarkDecorator attribute), [138](#page146)

args (MarkInfo attribute), [139](#page147)

assert\_outcomes() (RunResult method), [127](#page135)

at\_level() (LogCaptureFixture method), [124](#page132)

C

cache\_dir

configuration value, [148](#page156)

cached\_setup() (FixtureRequest method), [122](#page130)

CallInfo (class in \_pytest.runner), [135](#page143)

capfd() (in module \_pytest.capture), [120](#page128)

capfdbinary() (in module \_pytest.capture), [121](#page129)

caplog (TestReport attribute), [146](#page154) caplog() (in module \_pytest.logging), [123](#page131) capstderr (TestReport attribute), [146](#page154) capstdout (TestReport attribute), [146](#page154) capsys() (in module \_pytest.capture), [119](#page127)

capsysbinary() (in module \_pytest.capture), [120](#page128)

CaptureFixture (class in \_pytest.capture), [120](#page128)

chdir() (MonkeyPatch method), [125](#page133)

check\_pending() (PluginManager method), [144](#page152)

Class (class in \_pytest.python), [135](#page143) clear() (LogCaptureFixture method), [124](#page132) clear() (WarningsRecorder method), [128](#page136) cls (FixtureRequest attribute), [122](#page130)

cls (Metafunc attribute), [140](#page148) collect() (Collector method), [135](#page143) Collector (class in \_pytest.nodes), [135](#page143) Collector.CollectError, [135](#page143) combined\_with() (Mark method), [139](#page147) confcutdir

configuration value, [148](#page156) Config (class in \_pytest.config), [136](#page144) config (FixtureRequest attribute), [121](#page129) config (Node attribute), [141](#page149) configuration value

addopts, [148](#page156)

cache\_dir, [148](#page156)

confcutdir, [148](#page156)

console\_output\_style, [148](#page156)

doctest\_encoding, [149](#page157)

doctest\_optionflags, [149](#page157)

empty\_parameter\_set\_mark, [149](#page157)

filterwarnings, [149](#page157)

junit\_suite\_name, [149](#page157)

log\_cli\_date\_format, [149](#page157)

log\_cli\_format, [150](#page158)

log\_cli\_level, [150](#page158)

log\_date\_format, [150](#page158)

log\_file, [150](#page158)

log\_file\_date\_format, [150](#page158)

log\_file\_format, [150](#page158)

log\_file\_level, [151](#page159)

log\_format, [151](#page159)

log\_level, [151](#page159)

log\_print, [151](#page159)

markers, [151](#page159)

minversion, [151](#page159)

**249**

**pytest Documentation, Release 3.6**

norecursedirs, [151](#page159)

python\_classes, [152](#page160)

python\_files, [152](#page160)

python\_functions, [152](#page160)

testpaths, [152](#page160)

usefixtures, [153](#page161)

xfail\_strict, [153](#page161)

consider\_conftest() (PytestPluginManager method), [145](#page153) consider\_env() (PytestPluginManager method), [145](#page153) consider\_module() (PytestPluginManager method), [145](#page153) consider\_pluginarg() (PytestPluginManager method), [145](#page153) consider\_preparse() (PytestPluginManager method), [145](#page153) console\_output\_style

configuration value, [148](#page156) context() (MonkeyPatch method), [125](#page133)

D

delattr() (MonkeyPatch method), [125](#page133) delenv() (MonkeyPatch method), [125](#page133) delitem() (MonkeyPatch method), [125](#page133) deprecated\_call() (in module pytest), [114](#page122) disabled() (CaptureFixture method), [120](#page128) doctest\_encoding

configuration value, [149](#page157)

doctest\_namespace() (in module \_pytest.doctest), [121](#page129)

doctest\_optionflags

configuration value, [149](#page157) duration (TestReport attribute), [146](#page154)

E

empty\_parameter\_set\_mark

configuration value, [149](#page157)

enable\_tracing() (PluginManager method), [144](#page152) errisinstance() (ExceptionInfo method), [137](#page145) ExceptionInfo (class in \_pytest.\_code), [136](#page144) excinfo (CallInfo attribute), [135](#page143)

exconly() (ExceptionInfo method), [137](#page145) exit() (in module \_pytest.outcomes), [112](#page120) extra\_keyword\_matches (Node attribute), [141](#page149)

F

fail() (in module \_pytest.outcomes), [111](#page119) filterwarnings

configuration value, [149](#page157)

fixture() (in module pytest), [118](#page126)

FixtureDef (class in \_pytest.fixtures), [137](#page145)

fixturename (FixtureRequest attribute), [121](#page129)

fixturenames (Metafunc attribute), [140](#page148)

FixtureRequest (class in \_pytest.fixtures), [121](#page129)

fnmatch\_lines() (LineMatcher method), [127](#page135)

fnmatch\_lines\_random() (LineMatcher method), [127](#page135)

force\_result() (\_Result method), [146](#page154)

fromdictargs() (Config method), [136](#page144)

FSCollector (class in \_pytest.nodes), [137](#page145)

fspath (FixtureRequest attribute), [122](#page130)

fspath (Node attribute), [141](#page149)

Function (class in \_pytest.python), [137](#page145)

function (FixtureRequest attribute), [121](#page129)

function (Function attribute), [137](#page145)

function (Metafunc attribute), [140](#page148)

G

get() (Cache method), [119](#page127)

get\_canonical\_name() (PluginManager method), [144](#page152)

get\_closest\_marker() (Node method), [142](#page150)

get\_hookcallers() (PluginManager method), [144](#page152)

get\_lines\_after() (LineMatcher method), [127](#page135)

get\_marker() (Node method), [142](#page150)

get\_name() (PluginManager method), [144](#page152) get\_plugin() (PluginManager method), [144](#page152) get\_plugins() (PluginManager method), [143](#page151) get\_records() (LogCaptureFixture method), [123](#page131) get\_result() (\_Result method), [146](#page154) getbasetemp() (TempdirFactory method), [128](#page136) getfixturevalue() (FixtureRequest method), [122](#page130) getfuncargvalue() (FixtureRequest method), [122](#page130) getgroup() (Parser method), [142](#page150)

getini() (Config method), [136](#page144) getoption() (Config method), [136](#page144) getparent() (Node method), [142](#page150)

getplugin() (PytestPluginManager method), [145](#page153) getrepr() (ExceptionInfo method), [137](#page145) getvalue() (Config method), [136](#page144) getvalueorskip() (Config method), [136](#page144)

H

handler (LogCaptureFixture attribute), [123](#page131) has\_plugin() (PluginManager method), [144](#page152) hasplugin() (PytestPluginManager method), [145](#page153)

I

ihook (Node attribute), [141](#page149)

import\_plugin() (PytestPluginManager method), [145](#page153)

importorskip() (in module \_pytest.outcomes), [112](#page120)

instance (FixtureRequest attribute), [122](#page130)

is\_blocked() (PluginManager method), [143](#page151)

is\_registered() (PluginManager method), [143](#page151)

Item (class in \_pytest.nodes), [138](#page146)

iter\_markers() (Node method), [142](#page150)

iter\_markers\_with\_node() (Node method), [142](#page150)

J

junit\_suite\_name

configuration value, [149](#page157)

K

keywords (FixtureRequest attribute), [122](#page130)

**250** **Index**

**pytest Documentation, Release 3.6**

keywords (Node attribute), [141](#page149)

keywords (TestReport attribute), [145](#page153)

kwargs (Mark attribute), [139](#page147)

kwargs (MarkDecorator attribute), [139](#page147)

kwargs (MarkInfo attribute), [139](#page147)

L

LineMatcher (class in \_pytest.pytester), [127](#page135)

list (WarningsRecorder attribute), [128](#page136)

list\_name\_plugin() (PluginManager method), [144](#page152)

list\_plugin\_distinfo() (PluginManager method), [144](#page152)

listchain() (Node method), [141](#page149)

listextrakeywords() (Node method), [142](#page150)

load\_setuptools\_entrypoints() (PluginManager method), [144](#page152)

location (TestReport attribute), [145](#page153) log\_cli\_date\_format

configuration value, [149](#page157)

log\_cli\_format

configuration value, [150](#page158)

log\_cli\_level

configuration value, [150](#page158)

log\_date\_format

configuration value, [150](#page158)

log\_file

configuration value, [150](#page158)

log\_file\_date\_format

configuration value, [150](#page158)

log\_file\_format

configuration value, [150](#page158)

log\_file\_level

configuration value, [151](#page159)

log\_format

configuration value, [151](#page159)

log\_level

configuration value, [151](#page159)

log\_print

configuration value, [151](#page159)

LogCaptureFixture (class in \_pytest.logging), [123](#page131)

longrepr (TestReport attribute), [145](#page153)

longreprtext (TestReport attribute), [146](#page154)

M

main() (in module \_pytest.config), [112](#page120) makeconftest() (Testdir method), [126](#page134) makedir() (Cache method), [119](#page127) makepyfile() (Testdir method), [126](#page134)

Mark (class in \_pytest.mark.structures), [139](#page147) MarkDecorator (class in \_pytest.mark), [138](#page146) markers

configuration value, [151](#page159) MarkGenerator (class in \_pytest.mark), [139](#page147) MarkInfo (class in \_pytest.mark), [139](#page147) match() (ExceptionInfo method), [137](#page145)

Metafunc (class in \_pytest.python), [140](#page148) minversion

configuration value, [151](#page159)

mktemp() (TempdirFactory method), [128](#page136)

Module (class in \_pytest.python), [141](#page149)

module (FixtureRequest attribute), [122](#page130)

module (Metafunc attribute), [140](#page148)

MonkeyPatch (class in \_pytest.monkeypatch), [124](#page132)

monkeypatch() (in module \_pytest.monkeypatch), [124](#page132)

N

name (Mark attribute), [139](#page147)

name (MarkDecorator attribute), [138](#page146) name (MarkInfo attribute), [139](#page147) name (Node attribute), [141](#page149)

Node (class in \_pytest.nodes), [141](#page149) node (FixtureRequest attribute), [121](#page129) nodeid (Node attribute), [141](#page149) nodeid (TestReport attribute), [145](#page153) norecursedirs

configuration value, [151](#page159)

O

option (Config attribute), [136](#page144) originalname (Function attribute), [137](#page145) outcome (TestReport attribute), [145](#page153) own\_markers (Node attribute), [141](#page149)

P

param() (in module pytest), [112](#page120) parametrize() (Metafunc method), [116](#page124), [140](#page148) parent (Node attribute), [141](#page149)

parse\_hookimpl\_opts() (PytestPluginManager method), [145](#page153)

parse\_hookspec\_opts() (PytestPluginManager method), [145](#page153)

parse\_known\_and\_unknown\_args() (Parser method), [143](#page151) parse\_known\_args() (Parser method), [143](#page151) parseoutcomes() (RunResult method), [127](#page135)

Parser (class in \_pytest.config.argparsing), [142](#page150) PluginManager (class in pluggy), [143](#page151) pluginmanager (Config attribute), [136](#page144)

pop() (WarningsRecorder method), [128](#page136) pytest.mark.filterwarnings() (in module \_pytest.\_code),

[115](#page123)

pytest.mark.skip() (in module \_pytest.\_code), [116](#page124) pytest.mark.skipif() (in module \_pytest.\_code), [117](#page125) pytest.mark.usefixtures() (in module \_pytest.\_code), [117](#page125) pytest.mark.xfail() (in module \_pytest.\_code), [117](#page125) pytest\_addhooks() (in module \_pytest.hookspec), [130](#page138) pytest\_addoption() (in module \_pytest.hookspec), [130](#page138)

pytest\_assertrepr\_compare() (in module \_pytest.hookspec), [134](#page142)

**Index** **251**

**pytest Documentation, Release 3.6**

pytest\_cmdline\_main() (in module \_pytest.hookspec), [129](#page137)

pytest\_cmdline\_parse() (in module \_pytest.hookspec), [129](#page137)

pytest\_cmdline\_preparse() (in module \_pytest.hookspec), [129](#page137)

pytest\_collect\_directory() (in module \_pytest.hookspec), [132](#page140)

pytest\_collect\_file() (in module \_pytest.hookspec), [132](#page140) pytest\_collection() (in module \_pytest.hookspec), [132](#page140)

pytest\_collection\_modifyitems() (in module \_pytest.hookspec), [133](#page141)

pytest\_collectreport() (in module \_pytest.hookspec), [133](#page141) pytest\_collectstart() (in module \_pytest.hookspec), [133](#page141) pytest\_configure() (in module \_pytest.hookspec), [130](#page138) pytest\_configure() (PytestPluginManager method), [145](#page153) pytest\_deselected() (in module \_pytest.hookspec), [133](#page141) pytest\_enter\_pdb() (in module \_pytest.hookspec), [135](#page143)

pytest\_exception\_interact() (in module \_pytest.hookspec), [135](#page143)

pytest\_fixture\_post\_finalizer() (in module \_pytest.hookspec), [134](#page142)

pytest\_fixture\_setup() (in module \_pytest.hookspec), [134](#page142) pytest\_generate\_tests() (in module \_pytest.hookspec),

[132](#page140)

pytest\_ignore\_collect() (in module \_pytest.hookspec), [132](#page140)

pytest\_internalerror() (in module \_pytest.hookspec), [134](#page142) pytest\_itemcollected() (in module \_pytest.hookspec), [133](#page141)

pytest\_keyboard\_interrupt() (in module \_pytest.hookspec), [134](#page142)

pytest\_load\_initial\_conftests() (in module \_pytest.hookspec), [128](#page136)

pytest\_make\_parametrize\_id() (in module \_pytest.hookspec), [132](#page140)

pytest\_pycollect\_makeitem() (in module \_pytest.hookspec), [132](#page140)

pytest\_report\_collectionfinish() (in module \_pytest.hookspec), [133](#page141)

pytest\_report\_header() (in module \_pytest.hookspec), [133](#page141) pytest\_report\_teststatus() (in module \_pytest.hookspec),

[134](#page142)

pytest\_runtest\_call() (in module \_pytest.hookspec), [131](#page139) pytest\_runtest\_logfinish() (in module \_pytest.hookspec),

[131](#page139)

pytest\_runtest\_logreport() (in module \_pytest.hookspec), [134](#page142)

pytest\_runtest\_logstart() (in module \_pytest.hookspec), [131](#page139)

pytest\_runtest\_makereport() (in module \_pytest.hookspec), [132](#page140)

pytest\_runtest\_protocol() (in module \_pytest.hookspec), [131](#page139)

pytest\_runtest\_setup() (in module \_pytest.hookspec), [131](#page139)

pytest\_runtest\_teardown() (in module \_pytest.hookspec), [131](#page139)

pytest\_runtestloop() (in module \_pytest.hookspec), [131](#page139) pytest\_sessionfinish() (in module \_pytest.hookspec), [130](#page138) pytest\_sessionstart() (in module \_pytest.hookspec), [130](#page138)

pytest\_terminal\_summary() (in module \_pytest.hookspec), [134](#page142)

pytest\_unconfigure() (in module \_pytest.hookspec), [130](#page138) pytestconfig() (in module \_pytest.fixtures), [123](#page131) PytestPluginManager (class in \_pytest.config), [144](#page152) Python Enhancement Proposals

PEP 302, [95](#page103)

python\_classes

configuration value, [152](#page160)

python\_files

configuration value, [152](#page160)

python\_functions

configuration value, [152](#page160)

R

raiseerror() (FixtureRequest method), [122](#page130)

raises() (in module pytest), [112](#page120)

re\_match\_lines() (LineMatcher method), [127](#page135)

re\_match\_lines\_random() (LineMatcher method), [127](#page135)

readouterr() (CaptureFixture method), [120](#page128)

record\_property() (in module \_pytest.junitxml), [123](#page131)

record\_tuples (LogCaptureFixture attribute), [124](#page132)

records (LogCaptureFixture attribute), [124](#page132)

recwarn() (in module \_pytest.recwarn), [127](#page135)

register() (PluginManager method), [143](#page151) register() (PytestPluginManager method), [145](#page153) register\_assert\_rewrite() (in module pytest), [114](#page122) repr\_failure() (Collector method), [135](#page143) result (\_Result attribute), [146](#page154)

runpytest() (Testdir method), [126](#page134)

runpytest\_inprocess() (Testdir method), [126](#page134)

runpytest\_subprocess() (Testdir method), [126](#page134)

RunResult (class in \_pytest.pytester), [126](#page134)

runtest() (Function method), [137](#page145)

S

scope (FixtureRequest attribute), [121](#page129) sections (TestReport attribute), [146](#page154) Session (class in \_pytest.main), [145](#page153) session (FixtureRequest attribute), [122](#page130) session (Node attribute), [141](#page149) Session.Failed, [145](#page153) Session.Interrupted, [145](#page153) set() (Cache method), [119](#page127)

set\_blocked() (PluginManager method), [143](#page151) set\_level() (LogCaptureFixture method), [124](#page132) setattr() (MonkeyPatch method), [125](#page133) setenv() (MonkeyPatch method), [125](#page133) setitem() (MonkeyPatch method), [125](#page133)

**252** **Index**

**pytest Documentation, Release 3.6**

skip() (in module \_pytest.outcomes), [111](#page119)

str() (LineMatcher method), [127](#page135)

subset\_hook\_caller() (PluginManager method), [144](#page152)

syspath\_prepend() (MonkeyPatch method), [125](#page133)

T

tb (ExceptionInfo attribute), [136](#page144)

Testdir (class in \_pytest.pytester), [126](#page134)

testpaths

configuration value, [152](#page160)

TestReport (class in \_pytest.runner), [145](#page153)

text (LogCaptureFixture attribute), [123](#page131)

tmpdir() (in module \_pytest.tmpdir), [128](#page136)

traceback (ExceptionInfo attribute), [137](#page145)

type (ExceptionInfo attribute), [136](#page144)

typename (ExceptionInfo attribute), [136](#page144)

U

undo() (MonkeyPatch method), [125](#page133)

unregister() (PluginManager method), [143](#page151)

usefixtures

configuration value, [153](#page161)

user\_properties (Item attribute), [138](#page146)

user\_properties (TestReport attribute), [146](#page154)

V

value (ExceptionInfo attribute), [136](#page144)

W

warn() (Config method), [136](#page144)

warn() (Node method), [141](#page149)

WarningsRecorder (class in \_pytest.recwarn), [127](#page135)

warns() (in module pytest), [115](#page123)

when (CallInfo attribute), [135](#page143)

when (TestReport attribute), [146](#page154)

with\_args() (MarkDecorator method), [139](#page147)

X

xfail() (in module \_pytest.outcomes), [112](#page120)

xfail\_strict

configuration value, [153](#page161)

**Index** **253**