pytest-bdd

Release 8.1.0

Oleg Pidsadnyi

Jul 05, 2026

CONTENTS

1 Install pytest-bdd 3

2 Example 5

3 Scenario decorator 7

4 Step aliases 9

5 Using Asterisks in Place of Keywords 11

6 Step arguments 13

7 Override fixtures via given steps 17

8 Scenarios shortcut 19

9 Scenario outlines 21

10 Scenario Outlines with Multiple Example Tables 23

11 Handling Empty Example Cells 25

12 Rules 27

13 Datatables 29

14 Docstrings 31

15 Organizing your scenarios 33

16 Test setup 35

17 Backgrounds 37

18 Reusing fixtures 39

19 Reusing steps 41

20 Default steps 43

21 Feature file paths 45

22 Avoid retyping the feature file name 47

23 Programmatic step generation **49**

24 Hooks **53**

25 Browser testing **55**

26 Reporting **57**

27 Test code generation helpers **59**

28 Advanced code generation **61**

29 Migration of your tests from versions 5.x.x **63**
29.1 Removal of the feature examples . . . 63
29.2 Removal of the vertical examples . . . 63
29.3 Step arguments are no longer fixtures . . . 63
29.4 Variable templates in steps are only parsed for Scenario Outlines . . . 63

30 Migration of your tests from versions 4.x.x **65**
30.1 Replace usage of <parameter> inside step definitions with parsed {parameter} . . . 65
30.2 Refuse combining scenario outline and pytest parametrization . . . 66

31 Migration of your tests from versions 3.x.x **67**

32 License **69**

33 Authors **71**

34 Changelog **73**
34.1 Unreleased . . . 73
34.2 [8.1.0] - 2024-12-05 . . . 73
34.3 [8.0.0] - 2024-11-14 . . . 74
34.4 8.0.0b2 . . . 74
34.5 8.0.0b1 . . . 75
34.6 7.3.0 . . . 75
34.7 7.2.0 . . . 75
34.8 7.1.2 . . . 75
34.9 7.1.1 . . . 75
34.10 7.1 . . . 75
34.11 7.0.1 . . . 75
34.12 7.0.0 . . . 75
34.13 6.1.1 . . . 76
34.14 6.1.0 . . . 76
34.15 6.0.1 . . . 76
34.16 6.0.0 . . . 76
34.17 5.0.0 . . . 77
34.18 4.1.0 . . . 77
34.19 4.0.2 . . . 77
34.20 4.0.1 . . . 77
34.21 4.0.0 . . . 77
34.22 3.4.0 . . . 78
34.23 3.3.0 . . . 78
34.24 3.2.1 . . . 78
34.25 3.2.0 . . . 78
34.26 3.1.1 . . . 78
34.27 3.1.0 . . . 78

34.28 3.0.2 . . . 78
34.29 3.0.1 . . . 79
34.30 3.0.0 . . . 79
34.31 2.21.0 . . . 79
34.32 2.20.0 . . . 79
34.33 2.19.0 . . . 79
34.34 2.18.2 . . . 79
34.35 2.18.1 . . . 79
34.36 2.18.0 . . . 79
34.37 2.17.2 . . . 79
34.38 2.17.1 . . . 80
34.39 2.17.0 . . . 80
34.40 2.16.1 . . . 80
34.41 2.16.0 . . . 80
34.42 2.15.0 . . . 80
34.43 2.14.5 . . . 80
34.44 2.14.3 . . . 80
34.45 2.14.1 . . . 80
34.46 2.14.0 . . . 80
34.47 2.13.1 . . . 81
34.48 2.13.0 . . . 81
34.49 2.12.2 . . . 81
34.50 2.11.3 . . . 81
34.51 2.11.1 . . . 81
34.52 2.11.0 . . . 81
34.53 2.10.0 . . . 81
34.54 2.9.1 . . . 81
34.55 2.9.0 . . . 81
34.56 2.8.0 . . . 81
34.57 2.7.2 . . . 81
34.58 2.7.1 . . . 82
34.59 2.7.0 . . . 82
34.60 2.6.2 . . . 82
34.61 2.6.1 . . . 82
34.62 2.5.3 . . . 82
34.63 2.5.2 . . . 82
34.64 2.5.1 . . . 82
34.65 2.5.0 . . . 82
34.66 2.4.5 . . . 82
34.67 2.4.3 . . . 82
34.68 2.4.2 . . . 83
34.69 2.4.1 . . . 83
34.70 2.4.0 . . . 83
34.71 2.3.3 . . . 83
34.72 2.3.2 . . . 83
34.73 2.3.1 . . . 83
34.74 2.1.2 . . . 83
34.75 2.1.1 . . . 83
34.76 2.1.0 . . . 83
34.77 2.0.1 . . . 84
34.78 2.0.0 . . . 84
34.79 1.0.0 . . . 84
34.80 0.6.11 . . . 84
34.81 0.6.9 . . . 84

34.82 0.6.8 84
34.83 0.6.6 84
34.84 0.6.5 84
34.85 0.6.4 84
34.86 0.6.3 84
34.87 0.6.2 85
34.88 0.6.1 85
34.89 0.6.0 85
34.90 0.5.2 85
34.91 0.5.0 85
34.92 0.4.7 85
34.93 0.4.6 85
34.94 0.4.5 85
34.95 0.4.3 85

pytest-bdd implements a subset of the Gherkin language to enable automating project requirements testing and to facilitate behavioral driven development.

Unlike many other BDD tools, it does not require a separate runner and benefits from the power and flexibility of pytest. It enables unifying unit and functional tests, reduces the burden of continuous integration server configuration and allows the reuse of test setups.

Pytest fixtures written for unit tests can be reused for setup and actions mentioned in feature steps with dependency injection. This allows a true BDD just-enough specification of the requirements without maintaining any context object containing the side effects of Gherkin imperative declarations.

CHAPTER

ONE

INSTALL PYTEST-BDD

```
pip install pytest-bdd
```

CHAPTER

TWO

EXAMPLE

An example test for a blog hosting software could look like this. Note that pytest-splinter is used to get the browser fixture.

```
# content of publish_article.feature

Feature: Blog
    A site where you can publish your articles.

    Scenario: Publishing the article
        Given I'm an author user
        And I have an article

        When I go to the article page
        And I press the publish button

        Then I should not see the error message
        And the article should be published
```

Note that only one feature is allowed per feature file.

```
# content of test_publish_article.py

from pytest_bdd import scenario, given, when, then

@scenario('publish_article.feature', 'Publishing the article')
def test_publish():
    pass


@given("I'm an author user")
def author_user(auth, author):
    auth['user'] = author.user


@given("I have an article", target_fixture="article")
def article(author):
    return create_test_article(author=author)


@when("I go to the article page")
def go_to_article(article, browser):
```

(continues on next page)

(continued from previous page)

```
    browser.visit(urljoin(browser.url, '/manage/articles/{0}/'.format(article.id)))


@when("I press the publish button")
def publish_article(browser):
    browser.find_by_css('button[name=publish]').first.click()


@then("I should not see the error message")
def no_error_message(browser):
    with pytest.raises(ElementDoesNotExist):
        browser.find_by_css('.message.error').first


@then("the article should be published")
def article_is_published(article):
    article.refresh()  # Refresh the object in the SQLAlchemy session
    assert article.is_published
```

CHAPTER

THREE

SCENARIO DECORATOR

Functions decorated with the *scenario* decorator behave like a normal test function, and they will be executed after all scenario steps.

```
from pytest_bdd import scenario, given, when, then

@scenario('publish_article.feature', 'Publishing the article')
def test_publish(browser):
    assert article.title in browser.html
```

Note

It is however encouraged to try as much as possible to have your logic only inside the Given, When, Then steps.

CHAPTER

FOUR

STEP ALIASES

Sometimes, one has to declare the same fixtures or steps with different names for better readability. In order to use the same step function with multiple step names simply decorate it multiple times:

```
@given("I have an article")
@given("there's an article")
def article(author, target_fixture="article"):
    return create_test_article(author=author)
```

Note that the given step aliases are independent and will be executed when mentioned.

For example if you associate your resource to some owner or not. Admin user can't be an author of the article, but articles should have a default author.

```
Feature: Resource owner
    Scenario: I'm the author
        Given I'm an author
        And I have an article


    Scenario: I'm the admin
        Given I'm the admin
        And there's an article
```

CHAPTER

FIVE

USING ASTERISKS IN PLACE OF KEYWORDS

To avoid redundancy or unnecessary repetition of keywords such as “And” or “But” in Gherkin scenarios, you can use an asterisk (*) as a shorthand. The asterisk acts as a wildcard, allowing for the same functionality without repeating the keyword explicitly. It improves readability by making the steps easier to follow, especially when the specific keyword does not add value to the scenario’s clarity.

The asterisk will work the same as other step keywords - Given, When, Then - it follows.

For example:

```
Feature: Resource owner
    Scenario: I'm the author
        Given I'm an author
        * I have an article
        * I have a pen
```

```
from pytest_bdd import given

@given("I'm an author")
def _():
    pass

@given("I have an article")
def _():
    pass

@given("I have a pen")
def _():
    pass
```

In the scenario above, the asterisk (*) replaces the And or Given keywords. This allows for cleaner scenarios while still linking related steps together in the context of the scenario.

This approach is particularly useful when you have a series of steps that do not require explicitly stating whether they are part of the “Given”, “When”, or “Then” context but are part of the logical flow of the scenario.

CHAPTER

SIX

STEP ARGUMENTS

Often it's possible to reuse steps giving them a parameter(s). This allows to have single implementation and multiple use, so less code. Also opens the possibility to use same step twice in single scenario and with different arguments! And even more, there are several types of step parameter parsers at your disposal (idea taken from behave implementation):

string (the default)
: This is the default and can be considered as a *null* or *exact* parser. It parses no parameters and matches the step name by equality of strings.

parse (based on: pypi_parse)
: Provides a simple parser that replaces regular expressions for step parameters with a readable syntax like `{param:Type}`. The syntax is inspired by the Python builtin `string.format()` function. Step parameters must use the named fields syntax of pypi_parse in step definitions. The named fields are extracted, optionally type converted and then used as step function arguments. Supports type conversions by using type converters passed via *extra_types*

cfparse (extends: pypi_parse, based on: pypi_parse_type)
: Provides an extended parser with "Cardinality Field" (CF) support. Automatically creates missing type converters for related cardinality as long as a type converter for cardinality=1 is provided. Supports parse expressions like: * `{values:Type+}` (cardinality=1..N, many) * `{values:Type*}` (cardinality=0..N, many0) * `{value:Type?}` (cardinality=0..1, optional) Supports type conversions (as above).

re
: This uses full regular expressions to parse the clause text. You will need to use named groups "(?P<name>…)" to define the variables pulled from the text and passed to your `step()` function. Type conversion can only be done via *converters* step decorator argument (see example below).

The default parser is *string*, so just plain one-to-one match to the keyword definition. Parsers except *string*, as well as their optional arguments are specified like:

for *cfparse* parser

```
from pytest_bdd import parsers

@given(
    parsers.cfparse("there are {start:Number} cucumbers", extra_types={"Number": int}),
    target_fixture="cucumbers",
)
def given_cucumbers(start):
    return {"start": start, "eat": 0}
```

for *re* parser

```
from pytest_bdd import parsers
```

(continues on next page)

(continued from previous page)

```
@given(
    parsers.re(r"there are (?P<start>\d+) cucumbers"),
    converters={"start": int},
    target_fixture="cucumbers",
)
def given_cucumbers(start):
    return {"start": start, "eat": 0}
```

Example:

```
Feature: Step arguments
    Scenario: Arguments for given, when, then
        Given there are 5 cucumbers

        When I eat 3 cucumbers
        And I eat 2 cucumbers

        Then I should have 0 cucumbers
```

The code will look like:

```
from pytest_bdd import scenarios, given, when, then, parsers


scenarios("arguments.feature")


@given(parsers.parse("there are {start:d} cucumbers"), target_fixture="cucumbers")
def given_cucumbers(start):
    return {"start": start, "eat": 0}


@when(parsers.parse("I eat {eat:d} cucumbers"))
def eat_cucumbers(cucumbers, eat):
    cucumbers["eat"] += eat


@then(parsers.parse("I should have {left:d} cucumbers"))
def should_have_left_cucumbers(cucumbers, left):
    assert cucumbers["start"] - cucumbers["eat"] == left
```

Example code also shows possibility to pass argument converters which may be useful if you need to postprocess step arguments after the parser.

You can implement your own step parser. It's interface is quite simple. The code can look like:

```
import re
from pytest_bdd import given, parsers


class MyParser(parsers.StepParser):
    """Custom parser."""
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, name, **kwargs):
        """Compile regex."""
        super().__init__(name)
        self.regex = re.compile(re.sub("%(.+)%", "(?P<\1>.+)", self.name), **kwargs)

    def parse_arguments(self, name):
        """Get step arguments.

        :return: `dict` of step arguments
        """
        return self.regex.match(name).groupdict()

    def is_matching(self, name):
        """Match given name with the step name."""
        return bool(self.regex.match(name))


@given(parsers.parse("there are %start% cucumbers"), target_fixture="cucumbers")
def given_cucumbers(start):
    return {"start": start, "eat": 0}
```

CHAPTER

SEVEN

OVERRIDE FIXTURES VIA GIVEN STEPS

Dependency injection is not a panacea if you have complex structure of your test setup data. Sometimes there's a need such a given step which would imperatively change the fixture only for certain test (scenario), while for other tests it will stay untouched. To allow this, special parameter *target_fixture* exists in the *given* decorator:

```
from pytest_bdd import given

@pytest.fixture
def foo():
    return "foo"


@given("I have injecting given", target_fixture="foo")
def injecting_given():
    return "injected foo"


@then('foo should be "injected foo"')
def foo_is_foo(foo):
    assert foo == 'injected foo'
```

```
Feature: Target fixture
    Scenario: Test given fixture injection
        Given I have injecting given
        Then foo should be "injected foo"
```

In this example, the existing fixture *foo* will be overridden by given step *I have injecting given* only for the scenario it's used in.

Sometimes it is also useful to let *when* and *then* steps provide a fixture as well. A common use case is when we have to assert the outcome of an HTTP request:

```
# content of test_blog.py

from pytest_bdd import scenarios, given, when, then

from my_app.models import Article

scenarios("blog.feature")


@given("there is an article", target_fixture="article")
```

(continues on next page)

(continued from previous page)

```
def there_is_an_article():
    return Article()


@when("I request the deletion of the article", target_fixture="request_result")
def there_should_be_a_new_article(article, http_client):
    return http_client.delete(f"/articles/{article.uid}")


@then("the request should be successful")
def article_is_published(request_result):
    assert request_result.status_code == 200
```

```
# content of blog.feature

Feature: Blog
    Scenario: Deleting the article
        Given there is an article

        When I request the deletion of the article

        Then the request should be successful
```

CHAPTER

EIGHT

SCENARIOS SHORTCUT

If you have a relatively large set of feature files, it's boring to manually bind scenarios to the tests using the scenario decorator. Of course with the manual approach you get all the power to be able to additionally parametrize the test, give the test function a nice name, document it, etc, but in the majority of the cases you don't need that. Instead, you want to bind all the scenarios found in the `features` folder(s) recursively automatically, by using the `scenarios` helper.

```
from pytest_bdd import scenarios

# assume 'features' subfolder is in this file's directory
scenarios('features')
```

That's all you need to do to bind all scenarios found in the `features` folder! Note that you can pass multiple paths, and those paths can be either feature files or feature folders.

```
from pytest_bdd import scenarios

# pass multiple paths/files
scenarios('features', 'other_features/some.feature', 'some_other_features')
```

But what if you need to manually bind a certain scenario, leaving others to be automatically bound? Just write your scenario in a "normal" way, but ensure you do it **before** the call of `scenarios` helper.

```
from pytest_bdd import scenario, scenarios

@scenario('features/some.feature', 'Test something')
def test_something():
    pass

# assume 'features' subfolder is in this file's directory
scenarios('features')
```

In the example above, the `test_something` scenario binding will be kept manual, other scenarios found in the `features` folder will be bound automatically.

CHAPTER

NINE

SCENARIO OUTLINES

Scenarios can be parametrized to cover multiple cases. These are called Scenario Outlines in Gherkin, and the variable templates are written using angular brackets (e.g. <var_name>).

Example:

```
# content of scenario_outlines.feature

Feature: Scenario outlines
    Scenario Outline: Outlined given, when, then
        Given there are <start> cucumbers
        When I eat <eat> cucumbers
        Then I should have <left> cucumbers

        Examples:
        | start | eat | left |
        |  12   |  5  |  7   |
```

```
from pytest_bdd import scenarios, given, when, then, parsers


scenarios("scenario_outlines.feature")


@given(parsers.parse("there are {start:d} cucumbers"), target_fixture="cucumbers")
def given_cucumbers(start):
    return {"start": start, "eat": 0}


@when(parsers.parse("I eat {eat:d} cucumbers"))
def eat_cucumbers(cucumbers, eat):
    cucumbers["eat"] += eat


@then(parsers.parse("I should have {left:d} cucumbers"))
def should_have_left_cucumbers(cucumbers, left):
    assert cucumbers["start"] - cucumbers["eat"] == left
```

Example parameters from example tables can not only be used in steps, but also embedded directly within docstrings and datatables, allowing for dynamic substitution. This provides added flexibility for scenarios that require complex setups or validations.

Example:

```
# content of docstring_and_datatable_with_params.feature

Feature: Docstring and Datatable with example parameters
    Scenario Outline: Using parameters in docstrings and datatables
        Given the following configuration:
            """
            username: <username>
            password: <password>
            """
        When the user logs in
        Then the response should contain:
            | field     | value      |
            | username  | <username> |
            | logged_in | true       |

        Examples:
        | username  | password  |
        | user1     | pass123   |
        | user2     | 123secure |
```

```
from pytest_bdd import scenarios, given, when, then
import json

# Load scenarios from the feature file
scenarios("docstring_and_datatable_with_params.feature")


@given("the following configuration:")
def given_user_config(docstring):
    print(docstring)


@when("the user logs in")
def user_logs_in(logged_in):
    logged_in = True


@then("the response should contain:")
def response_should_contain(datatable):
    assert datatable[1][1] in ["user1", "user2"]
```

CHAPTER

TEN

SCENARIO OUTLINES WITH MULTIPLE EXAMPLE TABLES

In *pytest-bdd*, you can use multiple example tables in a scenario outline to test different sets of input data under various conditions. You can define separate *Examples* blocks, each with its own table of data, and optionally tag them to differentiate between positive, negative, or any other conditions.

Example:

```
# content of scenario_outline.feature

Feature: Scenario outlines with multiple examples tables
    Scenario Outline: Outlined with multiple example tables
        Given there are <start> cucumbers
        When I eat <eat> cucumbers
        Then I should have <left> cucumbers

        @positive
        Examples: Positive results
            | start | eat | left |
            |  12   |  5  |  7   |
            |  5    |  4  |  1   |

        @negative
        Examples: Impossible negative results
            | start | eat | left |
            |  3    |  9  |  -6  |
            |  1    |  4  |  -3  |
```

```
from pytest_bdd import scenarios, given, when, then, parsers


scenarios("scenario_outline.feature")


@given(parsers.parse("there are {start:d} cucumbers"), target_fixture="cucumbers")
def given_cucumbers(start):
    return {"start": start, "eat": 0}


@when(parsers.parse("I eat {eat:d} cucumbers"))
def eat_cucumbers(cucumbers, eat):
    cucumbers["eat"] += eat
```

(continues on next page)

(continued from previous page)

```
@then(parsers.parse("I should have {left:d} cucumbers"))
def should_have_left_cucumbers(cucumbers, left):
    assert cucumbers["start"] - cucumbers["eat"] == left
```

When you filter scenarios by a tag, only the examples associated with that tag will be executed. This allows you to run a specific subset of your test cases based on the tag. For example, in the following scenario outline, if you filter by the @positive tag, only the examples under the "Positive results" table will be executed, and the "Negative results" table will be ignored.

```
pytest -k "positive"
```

CHAPTER

ELEVEN

HANDLING EMPTY EXAMPLE CELLS

By default, empty cells in the example tables are interpreted as empty strings (“”). However, there may be cases where it is more appropriate to handle them as `None`. In such scenarios, you can use a converter with the `parsers.re` parser to define a custom behavior for empty values.

For example, the following code demonstrates how to use a custom converter to return `None` when an empty cell is encountered:

```
# content of empty_example_cells.feature

Feature: Handling empty example cells
    Scenario Outline: Using converters for empty cells
        Given I am starting lunch
        Then there are <start> cucumbers

        Examples:
        | start |
        |       |
```

```
from pytest_bdd import then, parsers


# Define a converter that returns None for empty strings
def empty_to_none(value):
    return None if value.strip() == "" else value


@given("I am starting lunch")
def _():
    pass


@then(
    parsers.re("there are (?P<start>.*?) cucumbers"),
    converters={"start": empty_to_none}
)
def _(start):
    # Example assertion to demonstrate the conversion
    assert start is None
```

Here, the *start* cell in the example table is empty. When the `parsers.re` parser is combined with the `empty_to_none` converter, the empty cell will be converted to `None` and can be handled accordingly in the step definition.

CHAPTER

TWELVE

RULES

In Gherkin, *Rules* allow you to group related scenarios or examples under a shared context. This is useful when you want to define different conditions or behaviours for multiple examples that follow a similar structure. You can use either `Scenario` or `Example` to define individual cases, as they are aliases and function identically.

Additionally, **tags** applied to a rule will be automatically applied to all the **examples or scenarios** under that rule, making it easier to organize and filter tests during execution.

Example:

```
Feature: Rules and examples

    @feature_tag
    Rule: A rule for valid cases

        @rule_tag
        Example: Valid case 1
            Given I have a valid input
            When I process the input
            Then the result should be successful

    Rule: A rule for invalid cases

        Example: Invalid case
            Given I have an invalid input
            When I process the input
            Then the result should be an error
```

CHAPTER

THIRTEEN

DATATABLES

The `datatable` argument allows you to utilise data tables defined in your Gherkin scenarios directly within your test functions. This is particularly useful for scenarios that require tabular data as input, enabling you to manage and manipulate this data conveniently.

When you use the `datatable` argument in a step definition, it will return the table as a list of lists, where each inner list represents a row from the table.

For example, the Gherkin table:

```
| name  | email            |
| John  | john@example.com |
```

Will be returned by the `datatable` argument as:

```
[
    ["name", "email"],
    ["John", "john@example.com"]
]
```

Note

When using the datatable argument, it is essential to ensure that the step to which it is applied actually has an associated data table. If the step does not have an associated data table, attempting to use the datatable argument will raise an error. Make sure that your Gherkin steps correctly reference the data table when defined.

Full example:

```
Feature: Manage user accounts

  Scenario: Creating a new user with roles and permissions
    Given the following user details:
      | name  | email             | age |
      | John  | john@example.com  | 30  |
      | Alice | alice@example.com | 25  |

    When each user is assigned the following roles:
      | Admin       | Full access to the system |
      | Contributor | Can add content           |

    And the page is saved
```

(continues on next page)

(continued from previous page)

```
    Then the user should have the following permissions:
      | permission     | allowed |
      | view dashboard | true    |
      | edit content   | true    |
      | delete content | false   |
```

```
from pytest_bdd import given, when, then


@given("the following user details:", target_fixture="users")
def _(datatable):
    users = []
    for row in datatable[1:]:
        users.append(row)

    print(users)
    return users


@when("each user is assigned the following roles:")
def _(datatable, users):
    roles = datatable
    for user in users:
        for role_row in datatable:
            assign_role(user, role_row)


@when("the page is saved")
def _():
    save_page()


@then("the user should have the following permissions:")
def _(datatable, users):
    expected_permissions = []
    for row in datatable[1:]:
        expected_permissions.append(row)

    assert users_have_correct_permissions(users, expected_permissions)
```

CHAPTER

FOURTEEN

DOCSTRINGS

The *docstring* argument allows you to access the Gherkin docstring defined in your steps as a multiline string. The content of the docstring is passed as a single string, with each line separated by \n. Leading indentation are stripped.

For example, the Gherkin docstring:

```
"""
This is a sample docstring.
It spans multiple lines.
"""
```

Will be returned as:

```
"This is a sample docstring.\nIt spans multiple lines."
```

Full example:

```
Feature: Docstring

  Scenario: Step with docstrings
    Given some steps will have docstrings

    Then a step has a docstring
    """
    This is a docstring
    on two lines
    """

    And a step provides a docstring with lower indentation
    """
This is a docstring
    """

    And this step has no docstring

    And this step has a greater indentation
    """
        This is a docstring
    """

    And this step has no docstring
```

```
from pytest_bdd import given, then


@given("some steps will have docstrings")
def _():
    pass

@then("a step has a docstring")
def _(docstring):
    assert docstring == "This is a docstring\non two lines"

@then("a step provides a docstring with lower indentation")
def _(docstring):
    assert docstring == "This is a docstring"

@then("this step has a greater indentation")
def _(docstring):
    assert docstring == "This is a docstring"

@then("this step has no docstring")
def _():
    pass
```

Note

The `docstring` argument can only be used for steps that have an associated docstring. Otherwise, an error will be thrown.

CHAPTER

FIFTEEN

ORGANIZING YOUR SCENARIOS

The more features and scenarios you have, the more important the question of their organization becomes. The things you can do (and that is also a recommended way):

- organize your feature files in the folders by semantic groups:

```
features
├──frontend
│   └──auth
│      └──login.feature
└──backend
    └──auth
       └──login.feature
```

This looks fine, but how do you run tests only for a certain feature? As pytest-bdd uses pytest, and bdd scenarios are actually normal tests. But test files are separate from the feature files, the mapping is up to developers, so the test files structure can look completely different:

```
tests
├
└──functional
   └──test_auth.py
      └ """Authentication tests."""
        from pytest_bdd import scenario

        @scenario('frontend/auth/login.feature')
        def test_logging_in_frontend():
            pass

        @scenario('backend/auth/login.feature')
        def test_logging_in_backend():
            pass
```

For picking up tests to run we can use the tests selection technique. The problem is that you have to know how your tests are organized, knowing only the feature files organization is not enough. Cucumber uses tags as a way of categorizing your features and scenarios, which pytest-bdd supports. For example, we could have:

```
@login @backend
Feature: Login

  @successful
  Scenario: Successful login
```

pytest-bdd uses pytest markers as a *storage* of the tags for the given scenario test, so we can use standard test selection:

```
pytest -m "backend and login and successful"
```

The feature and scenario markers are not different from standard pytest markers, and the `@` symbol is stripped out automatically to allow test selector expressions. If you want to have bdd-related tags to be distinguishable from the other test markers, use a prefix like `bdd`. Note that if you use pytest with the `--strict-markers` option, all Gherkin tags mentioned in the feature files should be also in the `markers` setting of the `pytest.ini` config. Also for tags please use names which are python-compatible variable names, i.e. start with a non-number, only underscores or alphanumeric characters, etc. That way you can safely use tags for tests filtering.

You can customize how tags are converted to pytest marks by implementing the `pytest_bdd_apply_tag` hook and returning `True` from it:

```
def pytest_bdd_apply_tag(tag, function):
    if tag == 'todo':
        marker = pytest.mark.skip(reason="Not implemented yet")
        marker(function)
        return True
    else:
        # Fall back to the default behavior of pytest-bdd
        return None
```

CHAPTER

SIXTEEN

TEST SETUP

Test setup is implemented within the Given section. Even though these steps are executed imperatively to apply possible side-effects, pytest-bdd is trying to benefit of the PyTest fixtures which is based on the dependency injection and makes the setup more declarative style.

```
@given("I have a beautiful article", target_fixture="article")
def article():
    return Article(is_beautiful=True)
```

The target PyTest fixture "article" gets the return value and any other step can depend on it.

```
Feature: The power of PyTest
    Scenario: Symbolic name across steps
        Given I have a beautiful article
        When I publish this article
```

The When step is referencing the `article` to publish it.

```
@when("I publish this article")
def publish_article(article):
    article.publish()
```

Many other BDD toolkits operate on a global context and put the side effects there. This makes it very difficult to implement the steps, because the dependencies appear only as the side-effects during run-time and not declared in the code. The "publish article" step has to trust that the article is already in the context, has to know the name of the attribute it is stored there, the type etc.

In pytest-bdd you just declare an argument of the step function that it depends on and the PyTest will make sure to provide it.

Still side effects can be applied in the imperative style by design of the BDD.

```
Feature: News website
    Scenario: Publishing an article
        Given I have a beautiful article
        And my article is published
```

Functional tests can reuse your fixture libraries created for the unit-tests and upgrade them by applying the side effects.

```
@pytest.fixture
def article():
    return Article(is_beautiful=True)
```

(continues on next page)

(continued from previous page)

```
@given("I have a beautiful article")
def i_have_a_beautiful_article(article):
    pass


@given("my article is published")
def published_article(article):
    article.publish()
    return article
```

This way side-effects were applied to our article and PyTest makes sure that all steps that require the "article" fixture will receive the same object. The value of the "published_article" and the "article" fixtures is the same object.

Fixtures are evaluated **only once** within the PyTest scope and their values are cached.

CHAPTER

SEVENTEEN

BACKGROUNDS

It's often the case that to cover certain feature, you'll need multiple scenarios. And it's logical that the setup for those scenarios will have some common parts (if not equal). For this, there are *backgrounds*. pytest-bdd implements Gherkin backgrounds for features.

```
Feature: Multiple site support

  Background:
    Given a global administrator named "Greg"
    And a blog named "Greg's anti-tax rants"
    And a customer named "Wilson"
    And a blog named "Expensive Therapy" owned by "Wilson"

  Scenario: Wilson posts to his own blog
    Given I am logged in as Wilson
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."

  Scenario: Greg posts to a client's blog
    Given I am logged in as Greg
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."
```

In this example, all steps from the background will be executed before all the scenario's own given steps, adding a possibility to prepare some common setup for multiple scenarios in a single feature. About best practices for Background, please read Gherkin's Tips for using Background.

Note

Only "Given" steps should be used in "Background" section. Steps "When" and "Then" are prohibited, because their purposes are related to actions and consuming outcomes; that is in conflict with the aim of "Background" - to prepare the system for tests or "put the system in a known state" as "Given" does it.

CHAPTER

EIGHTEEN

REUSING FIXTURES

Sometimes scenarios define new names for an existing fixture that can be inherited (reused). For example, if we have the pytest fixture:

```
@pytest.fixture
def article():
    """Test article."""
    return Article()
```

Then this fixture can be reused with other names using given():

```
@given('I have a beautiful article')
def i_have_an_article(article):
    """I have an article."""
```

CHAPTER

NINETEEN

REUSING STEPS

It is possible to define some common steps in the parent `conftest.py` and simply expect them in the child test file.

```
# content of common_steps.feature

Scenario: All steps are declared in the conftest
    Given I have a bar
    Then bar should have value "bar"
```

```
# content of conftest.py

from pytest_bdd import given, then


@given("I have a bar", target_fixture="bar")
def bar():
    return "bar"


@then('bar should have value "bar"')
def bar_is_bar(bar):
    assert bar == "bar"
```

```
# content of test_common.py

@scenario("common_steps.feature", "All steps are declared in the conftest")
def test_conftest():
    pass
```

There are no definitions of steps in the test file. They were collected from the parent conftest.py.

CHAPTER

TWENTY

DEFAULT STEPS

Here is the list of steps that are implemented inside pytest-bdd:

given

- trace - enters the *pdb* debugger via *pytest.set_trace()*

when

- trace - enters the *pdb* debugger via *pytest.set_trace()*

then

- trace - enters the *pdb* debugger via *pytest.set_trace()*

CHAPTER

TWENTYONE

FEATURE FILE PATHS

By default, pytest-bdd will use the current module's path as the base path for finding feature files, but this behaviour can be changed in the pytest configuration file (i.e. *pytest.ini*, *tox.ini* or *setup.cfg*) by declaring the new base path in the *bdd_features_base_dir* key. The path is interpreted as relative to the pytest root directory. You can also override the features base path on a per-scenario basis, in order to override the path for specific tests.

pytest.ini:

```
[pytest]
bdd_features_base_dir = features/
```

tests/test_publish_article.py:

```
from pytest_bdd import scenario


@scenario("foo.feature", "Foo feature in features/foo.feature")
def test_foo():
    pass


@scenario(
    "foo.feature",
    "Foo feature in tests/local-features/foo.feature",
    features_base_dir="./local-features/",
)
def test_foo_local():
    pass
```

The *features_base_dir* parameter can also be passed to the *@scenario* decorator.

CHAPTER

TWENTYTWO

AVOID RETYPING THE FEATURE FILE NAME

If you want to avoid retyping the feature file name when defining your scenarios in a test file, use `functools.partial`. This will make your life much easier when defining multiple scenarios in a test file. For example:

```
# content of test_publish_article.py

from functools import partial

import pytest_bdd


scenario = partial(pytest_bdd.scenario, "/path/to/publish_article.feature")


@scenario("Publishing the article")
def test_publish():
    pass


@scenario("Publishing the article as unprivileged user")
def test_publish_unprivileged():
    pass
```

You can learn more about functools.partial in the Python docs.

CHAPTER

TWENTYTHREE

PROGRAMMATIC STEP GENERATION

Sometimes you have step definitions that would be much easier to automate rather than writing them manually over and over again. This is common, for example, when using libraries like pytest-factoryboy that automatically creates fixtures. Writing step definitions for every model can become a tedious task.

For this reason, pytest-bdd provides a way to generate step definitions automatically.

The trick is to pass the `stacklevel` parameter to the `given`, `when`, `then`, `step` decorators. This will instruct them to inject the step fixtures in the appropriate module, rather than just injecting them in the caller frame.

Let's look at a concrete example; let's say you have a class `Wallet` that has some amount of each currency:

```
# contents of wallet.py

from dataclasses import dataclass

@dataclass
class Wallet:
    verified: bool

    amount_eur: int
    amount_usd: int
    amount_gbp: int
    amount_jpy: int
```

You can use pytest-factoryboy to automatically create model fixtures for this class:

```
# contents of wallet_factory.py

from wallet import Wallet

import factory
from pytest_factoryboy import register

class WalletFactory(factory.Factory):
    class Meta:
        model = Wallet

    verified = False
    amount_eur = 0
    amount_usd = 0
    amount_gbp = 0
    amount_jpy = 0
```

(continues on next page)

(continued from previous page)

```
register(Wallet)  # creates the "wallet" fixture
register(Wallet, "second_wallet")  # creates the "second_wallet" fixture
```

Now we can define a function `generate_wallet_steps(...)` that creates the steps for any wallet fixture (in our case, it will be `wallet` and `second_wallet`):

```
# contents of wallet_steps.py

import re
from dataclasses import fields

from pytest_bdd import given, then, parsers


def generate_wallet_steps(model_name="wallet", stacklevel=1):
    stacklevel += 1

    human_name = model_name.replace("_", " ")  # "second_wallet" -> "second wallet"

    @given(f"I have a {human_name}", target_fixture=model_name, stacklevel=stacklevel)
    def _(request):
        return request.getfixturevalue(model_name)

    # Generate steps for currency fields:
    for field in fields(Wallet):
        match = re.fullmatch(r"amount_(?P<currency>[a-z]{3})", field.name)
        if not match:
            continue
        currency = match["currency"]

        @given(
            parsers.parse(f"I have {{value:d}} {currency.upper()} in my {human_name}"),
            target_fixture=f"{model_name}__amount_{currency}",
            stacklevel=stacklevel,
        )
        def _(value: int) -> int:
            return value

        @then(
            parsers.parse(f"I should have {{value:d}} {currency.upper()} in my {human_
↪name}"),
            stacklevel=stacklevel,
        )
        def _(request, value: int, _currency=currency, _model_name=model_name) -> None:
            wallet = request.getfixturevalue(_model_name)
            assert getattr(wallet, f"amount_{_currency}") == value

# Inject the steps into the current module
generate_wallet_steps("wallet")
generate_wallet_steps("second_wallet")
```

This last file, `wallet_steps.py`, now contains all the step definitions for our “wallet” and “second_wallet” fixtures.

We can now define a scenario like this:

```
# contents of wallet.feature
Feature: A feature

    Scenario: Wallet EUR amount stays constant
        Given I have 10 EUR in my wallet
        And I have a wallet
        Then I should have 10 EUR in my wallet

    Scenario: Second wallet JPY amount stays constant
        Given I have 100 JPY in my second wallet
        And I have a second wallet
        Then I should have 100 JPY in my second wallet
```

and finally a test file that puts it all together and run the scenarios:

```
# contents of test_wallet.py

from pytest_factoryboy import scenarios

from wallet_factory import *  # import the registered fixtures "wallet" and "second_
↪wallet"
from wallet_steps import *  # import all the step definitions into this test file

scenarios("wallet.feature")
```

CHAPTER

TWENTYFOUR

HOOKS

pytest-bdd exposes several pytest hooks which might be helpful building useful reporting, visualization, etc. on top of it:

- *pytest_bdd_before_scenario(request, feature, scenario)* - Called before scenario is executed
- *pytest_bdd_after_scenario(request, feature, scenario)* - Called after scenario is executed (even if one of steps has failed)
- *pytest_bdd_before_step(request, feature, scenario, step, step_func)* - Called before step function is executed and its arguments evaluated
- *pytest_bdd_before_step_call(request, feature, scenario, step, step_func, step_func_args)* - Called before step function is executed with evaluated arguments
- *pytest_bdd_after_step(request, feature, scenario, step, step_func, step_func_args)* - Called after step function is successfully executed
- *pytest_bdd_step_error(request, feature, scenario, step, step_func, step_func_args, exception)* - Called when step function failed to execute
- *pytest_bdd_step_func_lookup_error(request, feature, scenario, step, exception)* - Called when step lookup failed

CHAPTER

TWENTYFIVE

BROWSER TESTING

Tools recommended to use for browser testing:

- pytest-splinter - pytest splinter integration for the real browser testing

CHAPTER

TWENTYSIX

REPORTING

It's important to have nice reporting out of your bdd tests. Cucumber introduced some kind of standard for json format which can be used for, for example, by this Jenkins plugin.

To have an output in json format:

```
pytest --cucumberjson=<path to json report>
```

This will output an expanded (meaning scenario outlines will be expanded to several scenarios) Cucumber format.

To enable gherkin-formatted output on terminal, use *–gherkin-terminal-reporter* in conjunction with the *-v* or *-vv* options:

```
pytest -v --gherkin-terminal-reporter
```

CHAPTER

TWENTYSEVEN

TEST CODE GENERATION HELPERS

For newcomers it's sometimes hard to write all needed test code without being frustrated. To simplify their life, a simple code generator was implemented. It allows to create fully functional (but of course empty) tests and step definitions for a given feature file. It's done as a separate console script provided by pytest-bdd package:

```
pytest-bdd generate <feature file name> .. <feature file nameN>
```

It will print the generated code to the standard output so you can easily redirect it to the file:

```
pytest-bdd generate features/some.feature > tests/functional/test_some.py
```

CHAPTER

TWENTYEIGHT

ADVANCED CODE GENERATION

For more experienced users, there's a smart code generation/suggestion feature. It will only generate the test code which is not yet there, checking existing tests and step definitions the same way it's done during the test execution. The code suggestion tool is called via passing additional pytest arguments:

```
pytest --generate-missing --feature features tests/functional
```

The output will be like:

```
============================= test session starts ==============================
platform linux2 -- Python 2.7.6 -- py-1.4.24 -- pytest-2.6.2
plugins: xdist, pep8, cov, cache, bdd, bdd, bdd
collected 2 items

Scenario is not bound to any test: "Code is generated for scenarios which are not bound␣
↪to any tests" in feature "Missing code generation" in /tmp/pytest-552/testdir/test_
↪generate_missing0/tests/generation.feature
--------------------------------------------------------------------------------

Step is not defined: "I have a custom bar" in scenario: "Code is generated for scenario␣
↪steps which are not yet defined(implemented)" in feature "Missing code generation" in /
↪tmp/pytest-552/testdir/test_generate_missing0/tests/generation.feature
--------------------------------------------------------------------------------
Please place the code above to the test file(s):

@scenario('tests/generation.feature', 'Code is generated for scenarios which are not␣
↪bound to any tests')
def test_Code_is_generated_for_scenarios_which_are_not_bound_to_any_tests():
    """Code is generated for scenarios which are not bound to any tests."""


@given("I have a custom bar")
def I_have_a_custom_bar():
    """I have a custom bar."""
```

As as side effect, the tool will validate the files for format errors, also some of the logic bugs, for example the ordering of the types of the steps.

CHAPTER

TWENTYNINE

MIGRATION OF YOUR TESTS FROM VERSIONS 5.X.X

The primary focus of the pytest-bdd is the compatibility with the latest gherkin developments e.g. multiple scenario outline example tables with tags support etc.

In order to provide the best compatibility, it is best to support the features described in the official gherkin reference. This means deprecation of some non-standard features that were implemented in pytest-bdd.

29.1 Removal of the feature examples

The example tables on the feature level are no longer supported. If you had examples on the feature level, you should copy them to each individual scenario.

29.2 Removal of the vertical examples

Vertical example tables are no longer supported since the official gherkin doesn't support them. The example tables should have horizontal orientation.

29.3 Step arguments are no longer fixtures

Step parsed arguments conflicted with the fixtures. Now they no longer define fixture. If the fixture has to be defined by the step, the target_fixture param should be used.

29.4 Variable templates in steps are only parsed for Scenario Outlines

In previous versions of pytest, steps containing `<variable>` would be parsed both by `Scenario` and `Scenario Outline`. Now they are only parsed within a `Scenario Outline`.

CHAPTER

THIRTY

MIGRATION OF YOUR TESTS FROM VERSIONS 4.X.X

30.1 Replace usage of <parameter> inside step definitions with parsed {parameter}

Templated steps (e.g. `@given("there are <start> cucumbers")`) should now the use step argument parsers in order to match the scenario outlines and get the values from the example tables. The values from the example tables are no longer passed as fixtures, although if you define your step to use a parser, the parameters will be still provided as fixtures.

```
# Old step definition:
@given("there are <start> cucumbers")
def given_cucumbers(start):
    pass


# New step definition:
@given(parsers.parse("there are {start} cucumbers"))
def given_cucumbers(start):
    pass
```

Scenario *example_converters* are removed in favor of the converters provided on the step level:

```
# Old code:
@given("there are <start> cucumbers")
def given_cucumbers(start):
    return {"start": start}

@scenario("outline.feature", "Outlined", example_converters={"start": float})
def test_outline():
    pass


# New code:
@given(parsers.parse("there are {start} cucumbers"), converters={"start": float})
def given_cucumbers(start):
    return {"start": start}

@scenario("outline.feature", "Outlined")
def test_outline():
    pass
```

30.2 Refuse combining scenario outline and pytest parametrization

The significant downside of combining scenario outline and pytest parametrization approach was an inability to see the test table from the feature file.

CHAPTER

THIRTYONE

MIGRATION OF YOUR TESTS FROM VERSIONS 3.X.X

Given steps are no longer fixtures. In case it is needed to make given step setup a fixture, the target_fixture parameter should be used.

```
@given("there's an article", target_fixture="article")
def there_is_an_article():
    return Article()
```

Given steps no longer have the *fixture* parameter. In fact the step may depend on multiple fixtures. Just normal step declaration with the dependency injection should be used.

```
@given("there's an article")
def there_is_an_article(article):
    pass
```

Strict gherkin option is removed, so the `strict_gherkin` parameter can be removed from the scenario decorators as well as `bdd_strict_gherkin` from the ini files.

Step validation handlers for the hook `pytest_bdd_step_validation_error` should be removed.

CHAPTER

THIRTYTWO

LICENSE

CHAPTER

THIRTYTHREE

AUTHORS

Oleg Pidsadnyi
: original idea, initial implementation and further improvements

Anatoly Bubenkov
: key implementation idea and realization, many new features and improvements

These people have contributed to *pytest-bdd*, in alphabetical order:

- Adam Coddington
- Albert-Jan Nijburg
- Alessio Bogon
- Andrey Makhnach
- Aron Curzon
- Dmitrijs Milajevs
- Dmitry Kolyagin
- Florian Bruhin
- Floris Bruynooghe
- Harro van der Klauw
- Hugo van Kemenade
- Kyle Adams
- Laurence Rowe
- Leonardo Santagada
- Milosz Sliwinski
- Michiel Holtkamp
- Robin Pedersen
- Sergey Kraynev

CHAPTER

THIRTYFOUR

CHANGELOG

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

34.1 Unreleased

- Relaxed *gherkin-official* dependency requirement to >=*29.0.0* to allow for newer versions of the *gherkin-official* package.
- Excluded *gherkin-official 31.0.0* and *32.0.0*, which crash with `StopIteration` when parsing empty descriptions (fixed upstream in *32.0.1*).
- Dropped support for python 3.9 (end-of-life, and its locked dev dependencies had unfixable security alerts). Supported python versions: 3.10, 3.11, 3.12, 3.13, 3.14.
- The following private attributes are not available anymore (#658): * `_pytest.reports.TestReport.scenario`; replaced by `pytest_bdd.reporting.test_report_context` WeakKeyDictionary (internal use) * `__scenario__` attribute of test functions generated by the `@scenario` (and `@scenarios`) decorator; replaced by `pytest_bdd.scenario.scenario_wrapper_template_registry` WeakKeyDictionary (internal use) * `_pytest.nodes.Item.__scenario_report__`; replaced by `pytest_bdd.reporting.scenario_reports_registry` WeakKeyDictionary (internal use) * `_pytest_bdd_step_context` attribute of internal test function markers; replaced by `pytest_bdd.steps.step_function_context_registry` WeakKeyDictionary (internal use)
- Backslashes in datatable and examples table cells are no longer over-quoted. A cell containing a single backslash now reaches the step as a single backslash, matching the Gherkin escaping rules. If you compensated by doubling backslashes in feature files, undo that. #769
- Made type annotations stronger and removed most of the `typing.Any` usages and `# type: ignore` annotations. #658
- Empty docstrings are now correctly forwarded to step functions as an empty string instead of being silently dropped, which previously caused pytest to report a missing `docstring` fixture. #809

34.2 [8.1.0] - 2024-12-05

- Step arguments `"datatable"` and `"docstring"` are now reserved, and they can't be used as step argument names. An error is raised if a step parser uses these names.
- Scenario `description` field is now set for Cucumber JSON output.
- Fixed an issue with the upcoming pytest release related to the use of `@pytest.mark.usefixtures` with an empty list.

- Render template variables in docstrings and datatable cells with example table entries, as we already do for steps definitions.

34.3 [8.0.0] - 2024-11-14

- Gherkin keyword aliases can now be used and correctly reported in json and terminal output (see Keywords for permitted list).
- Added localization support. The language of the feature file can be specified using the *# language: <language>* directive at the beginning of the file.
- Rule keyword can be used in feature files (see Rule)
- Added support for multiple example tables
- Added filtering by tags against example tables
- Since the 7.x series: * Tags can now be on multiple lines (stacked) * Continuation of steps using asterisks (`*`) instead of `And`/`But` supported. * Added `datatable` argument for steps that contain a datatable (see Data Tables). * Added `docstring` argument for steps that contain a docstring (see Doc Strings).
- Changelog format updated to follow Keep a Changelog.
- Text after the # character is no longer stripped from the Scenario and Feature name.
- Since the 7.x series:
  - Use the gherkin-official parser, replacing the custom parsing logic. This will make pytest-bdd more compatible with the Gherkin specification.
  - Multiline steps must now always use triple-quotes for the additional lines.
  - All feature files must now use the keyword `Feature:` to be considered valid.
  - Tags can no longer have spaces (e.g. `@tag one` and `@tag two` are no longer valid).
  - Text after the # character is no longer stripped from the Step name.
  - Multiline strings no longer match name based on multiple lines - only on the actual step text on the step line.
- Dropped support for python 3.8. Supported python versions: 3.9, 3.10, 3.11, 3.12, 3.13.
- Since the 7.x series:
  - Drop compatibility with pytest < 7.0.0.
- Since the 7.x series:
  - Updated documentation to clarify that `--gherkin-terminal-reporter` needs to be used with `-v` or `-vv`.

34.4 8.0.0b2

- Updated documentation to clarify that `--gherkin-terminal-reporter` needs to be used with `-v` or `-vv`.
- Drop compatibility with pytest < 7.0.0.
- Continuation of steps using asterisks instead of And/But supported.
- Added `datatable` argument for steps that contain a datatable (see Data Tables).
- Added `docstring` argument for steps that contain a docstring (see Doc Strings).
- Multiline strings no longer match name based on multiple lines - only on the actual step text on the step line.

34.5 8.0.0b1

- Use the gherkin-official parser, replacing the custom parsing logic. This will make pytest-bdd more compatible with the Gherkin specification.
- Multiline steps must now always use triple-quotes for the additional lines.
- All feature files must now use the keyword `Feature:` to be considered valid.
- Tags can no longer have spaces (e.g. `@tag one` and `@tag two` are no longer valid).
- Tags can now be on multiple lines (stacked)
- Text after the # character is no longer stripped from the Step name.

34.6 7.3.0

- Fix an issue when only the first Step would inject a fixture, while later steps would not be able to.
- Test against the latest versions of pytest (8.2, 8.3).

34.7 7.2.0

- Fix compatibility issue with Python 3.13.
- Declare compatibility with Python 3.13.

34.8 7.1.2

- Address another compatibility issue with pytest 8.1 (fixture registration). #680

34.9 7.1.1

- Address a bug introduced in pytest-bdd 7.1 caused by incorrect pytest version check.

34.10 7.1

- Address compatibility issue with pytest 8.1. #666

34.11 7.0.1

- Fix errors occurring if `pytest_unconfigure` is called before *pytest_configure*. #362 #641

34.12 7.0.0

- Backwards incompatible: - `parsers.re` now does a fullmatch instead of a partial match. This is to make it work just like the other parsers, since they don't ignore non-matching characters at the end of the string. #539
- Drop python 3.7 compatibility, as it's no longer supported. #627
- Declare official support for python 3.12 #628
- Improve parser performance by 15% #623 by @dcendents
- Add support for Scenarios and Scenario Outlines to have descriptions. #600

34.13 6.1.1

- Fix regression introduced in version 6.1.0 where the `pytest_bdd_after_scenario` hook would be called after every step instead of after the scenario. #577

34.14 6.1.0

- Fix bug where steps without parsers would take precedence over steps with parsers. #534
- Step functions can now be decorated multiple times with @given, @when, @then. Previously every decorator would override `converters` and `target_fixture` every at every application. #534 #544 #525
- Require pytest>=6.2 #534
- Using modern way to specify hook options to avoid deprecation warnings with pytest >=7.2.
- Add generic `step` decorator that will be used for all kind of steps #548
- Add `stacklevel` param to `given`, `when`, `then`, `step` decorators. This allows for programmatic step generation #548
- Hide pytest-bdd internal method in user tracebacks #557.
- Make the package PEP 561-compatible #559 #563.
- Configuration option `bdd_features_base_dir` is interpreted as relative to the pytest root directory (previously it was relative to the current working directory). #573

34.15 6.0.1

- Fix regression introduced in 6.0.0 where a step function decorated multiple using a parsers times would not be executed correctly. #530 #528

34.16 6.0.0

This release introduces breaking changes in order to be more in line with the official gherkin specification.

- Cleanup of the documentation and tests related to parametrization (elchupanebrej) #469
- Removed feature level examples for the gherkin compatibility (olegpidsadnyi) #490
- Removed vertical examples for the gherkin compatibility (olegpidsadnyi) #492
- Step arguments are no longer fixtures (olegpidsadnyi) #493
- Drop support of python 3.6, pytest 4 (elchupanebrej) #495 #504
- Step definitions can have "yield" statements again (4.0 release broke it). They will be executed as normal fixtures: code after the yield is executed during teardown of the test. (youtux) #503
- Scenario outlines unused example parameter validation is removed (olegpidsadnyi) #499
- Add type annotations (youtux) #505
- `pytest_bdd.parsers.StepParser` now is an Abstract Base Class. Subclasses must make sure to implement the abstract methods. (youtux) #505
- Angular brackets in step definitions are only parsed in "Scenario Outline" (previously they were parsed also in normal "Scenario"s) (youtux) #524.

34.17 5.0.0

This release introduces breaking changes, please refer to the *Migration of your tests from versions 4.x.x*.

- Rewrite the logic to parse Examples for Scenario Outlines. Now the substitution of the examples is done during the parsing of Gherkin feature files. You won't need to define the steps twice like `@given("there are <start> cucumbers")` and `@given(parsers.parse("there are {start} cucumbers"))`. The latter will be enough.
- Removed `example_converters` from `scenario(...)` signature. You should now use just the `converters` parameter for `given`, `when`, `then`.
- Removed `--cucumberjson-expanded` and `--cucumber-json-expanded` options. Now the JSON report is always expanded.
- Removed `--gherkin-terminal-reporter-expanded` option. Now the terminal report is always expanded.

34.18 4.1.0

- *when* and *then* steps now can provide a *target_fixture*, just like *given* does. Discussion at https://github.com/pytest-dev/pytest-bdd/issues/402.
- Drop compatibility for python 2 and officially support only python >= 3.6.
- Fix error when using *–cucumber-json-expanded* in combination with *example_converters* (marcbrossaissogeti).
- Fix *–generate-missing* not correctly recognizing steps with parsers

34.19 4.0.2

- Fix a bug that prevents using comments in the `Examples:` section. (youtux)

34.20 4.0.1

- Fixed performance regression introduced in 4.0.0 where collection time of tests would take way longer than before. (youtux)

34.21 4.0.0

This release introduces breaking changes, please refer to the *Migration of your tests from versions 3.x.x*.

- Strict Gherkin option is removed (`@scenario()` does not accept the `strict_gherkin` parameter). (olegpidsadnyi)
- `@scenario()` does not accept the undocumented parameter `caller_module` anymore. (youtux)
- Given step is no longer a fixture. The scope parameter is also removed. (olegpidsadnyi)
- Fixture parameter is removed from the given step declaration. (olegpidsadnyi)
- `pytest_bdd_step_validation_error` hook is removed. (olegpidsadnyi)
- Fix an error with pytest-pylint plugin #374. (toracle)
- Fix pytest-xdist 2.0 compatibility #369. (olegpidsadnyi)
- Fix compatibility with pytest 6 `--import-mode=importlib` option. (youtux)

34.22 3.4.0

- Parse multiline steps according to the gherkin specification #365.

34.23 3.3.0

- Drop support for pytest < 4.3.
- Fix a Python 4.0 bug.
- Fix `pytest --generate-missing` functionality being broken.
- Fix problematic missing step definition from strings containing quotes.
- Implement parsing escaped pipe characters in outline parameters (Mark90) #337.
- Disable the strict Gherkin validation in the steps generation (v-buriak) #356.

34.24 3.2.1

- Fix regression introduced in 3.2.0 where pytest-bdd would break in presence of test items that are not functions.

34.25 3.2.0

- Fix Python 3.8 support
- Remove code that rewrites code. This should help with the maintenance of this project and make debugging easier.

34.26 3.1.1

- Allow unicode string in `@given()` step names when using python2. This makes the transition of projects from python 2 to 3 easier.

34.27 3.1.0

- Drop support for pytest < 3.3.2.
- Step definitions generated by `$ pytest-bdd generate` will now raise `NotImplementedError` by default.
- `@given(...)` no longer accepts regex objects. It was deprecated long ago.
- Improve project testing by treating warnings as exceptions.
- `pytest_bdd_step_validation_error` will now always receive `step_func_args` as defined in the signature.

34.28 3.0.2

- Add compatibility with pytest 4.2 (sliwinski-milosz) #288.

34.29 3.0.1

- Minimal supported version of *pytest* is now 2.9.0 as lower versions do not support *bool* type ini options (sliwinski-milosz) #260
- Fix RemovedInPytest4Warning warnings (sliwinski-milosz) #261.

34.30 3.0.0

- Fixtures *pytestbdd_feature_base_dir* and *pytestbdd_strict_gherkin* have been removed. Check the Migration of your tests from versions 2.x.x for more information (sliwinski-milosz) #255
- Fix step definitions not being found when using parsers or converters after a change in pytest (youtux) #257

34.31 2.21.0

- Gherkin terminal reporter expanded format (pauk-slon)

34.32 2.20.0

- Added support for But steps (olegpidsadnyi)
- Fixed compatibility with pytest 3.3.2 (olegpidsadnyi)
- MInimal required version of pytest is now 2.8.1 since it doesn't support earlier versions (olegpidsadnyi)

34.33 2.19.0

- Added –cucumber-json-expanded option for explicit selection of expanded format (mjholtkamp)
- Step names are filled in when –cucumber-json-expanded is used (mjholtkamp)

34.34 2.18.2

- Fix check for out section steps definitions for no strict gherkin feature

34.35 2.18.1

- Relay fixture results to recursive call of 'get_features' (coddingtonbear)

34.36 2.18.0

- Add gherkin terminal reporter (spinus + thedrow)

34.37 2.17.2

- Fix scenario lines containing an `@` being parsed as a tag. (The-Compiler)

34.38 2.17.1

- Add support for pytest 3.0

34.39 2.17.0

- Fix FixtureDef signature for newer pytest versions (The-Compiler)
- Better error explanation for the steps defined outside of scenarios (olegpidsadnyi)
- Add a `pytest_bdd_apply_tag` hook to customize handling of tags (The-Compiler)
- Allow spaces in tag names. This can be useful when using the `pytest_bdd_apply_tag` hook with tags like `@xfail: Some reason`.

34.40 2.16.1

- Cleaned up hooks of the plugin (olegpidsadnyi)
- Fixed report serialization (olegpidsadnyi)

34.41 2.16.0

- Fixed deprecation warnings with pytest 2.8 (The-Compiler)
- Fixed deprecation warnings with Python 3.5 (The-Compiler)

34.42 2.15.0

- Add examples data in the scenario report (bubenkoff)

34.43 2.14.5

- Properly parse feature description (bubenkoff)

34.44 2.14.3

- Avoid potentially random collection order for xdist compartibility (bubenkoff)

34.45 2.14.1

- Pass additional arguments to parsers (bubenkoff)

34.46 2.14.0

- Add validation check which prevents having multiple features in a single feature file (bubenkoff)

34.47 2.13.1

- Allow mixing feature example table with scenario example table (bubenkoff, olegpidsadnyi)

34.48 2.13.0

- Feature example table (bubenkoff, sureshvv)

34.49 2.12.2

- Make it possible to relax strict Gherkin scenario validation (bubenkoff)

34.50 2.11.3

- Fix minimal *six* version (bubenkoff, dustinfarris)

34.51 2.11.1

- Mention step type on step definition not found errors and in code generation (bubenkoff, lrowe)

34.52 2.11.0

- Prefix step definition fixture names to avoid name collisions (bubenkoff, lrowe)

34.53 2.10.0

- Make feature and scenario tags to be fully compartible with pytest markers (bubenkoff, kevinastone)

34.54 2.9.1

- Fixed FeatureError string representation to correctly support python3 (bubenkoff, lrowe)

34.55 2.9.0

- Added possibility to inject fixtures from given keywords (bubenkoff)

34.56 2.8.0

- Added hook before the step is executed with evaluated parameters (olegpidsadnyi)

34.57 2.7.2

- Correct base feature path lookup for python3 (bubenkoff)

34.58 2.7.1

- Allow to pass `scope` for `given` steps (bubenkoff, sureshvv)

34.59 2.7.0

- Implemented *scenarios* shortcut to automatically bind scenarios to tests (bubenkoff)

34.60 2.6.2

- Parse comments only in the beginning of words (santagada)

34.61 2.6.1

- Correctly handle *pytest-bdd* command called without the subcommand under python3 (bubenkoff, spinus)
- Pluggable parsers for step definitions (bubenkoff, spinus)

34.62 2.5.3

- Add after scenario hook, document both before and after scenario hooks (bubenkoff)

34.63 2.5.2

- Fix code generation steps ordering (bubenkoff)

34.64 2.5.1

- Fix error report serialization (olegpidsadnyi)

34.65 2.5.0

- Fix multiline steps in the Background section (bubenkoff, arpe)
- Code cleanup (olegpidsadnyi)

34.66 2.4.5

- Fix unicode issue with scenario name (bubenkoff, aohontsev)

34.67 2.4.3

- Fix unicode regex argumented steps issue (bubenkoff, aohontsev)
- Fix steps timings in the json reporting (bubenkoff)

34.68 2.4.2

- Recursion is fixed for the –generate-missing and the –feature parameters (bubenkoff)

34.69 2.4.1

- Better reporting of a not found scenario (bubenkoff)
- Simple test code generation implemented (bubenkoff)
- Correct timing values for cucumber json reporting (bubenkoff)
- Validation/generation helpers (bubenkoff)

34.70 2.4.0

- Background support added (bubenkoff)
- Fixed double collection of the conftest files if scenario decorator is used (ropez, bubenkoff)

34.71 2.3.3

- Added timings to the cucumber json report (bubenkoff)

34.72 2.3.2

- Fixed incorrect error message using e.argname instead of step.name (hvdklauw)

34.73 2.3.1

- Implemented cucumber tags support (bubenkoff)
- Implemented cucumber json formatter (bubenkoff, albertjan)
- Added ‘trace’ keyword (bubenkoff)

34.74 2.1.2

- Latest pytest compartibility fixes (bubenkoff)

34.75 2.1.1

- Bugfixes (bubenkoff)

34.76 2.1.0

- Implemented multiline steps (bubenkoff)

34.77 2.0.1

- Allow more than one parameter per step (bubenkoff)
- Allow empty example values (bubenkoff)

34.78 2.0.0

- Pure pytest parametrization for scenario outlines (bubenkoff)
- Argumented steps now support converters (transformations) (bubenkoff)
- scenario supports only decorator form (bubenkoff)
- Code generation refactoring and cleanup (bubenkoff)

34.79 1.0.0

- Implemented scenario outlines (bubenkoff)

34.80 0.6.11

- Fixed step arguments conflict with the fixtures having the same name (olegpidsadnyi)

34.81 0.6.9

- Implemented support of Gherkin “Feature:” (olegpidsadnyi)

34.82 0.6.8

- Implemented several hooks to allow reporting/error handling (bubenkoff)

34.83 0.6.6

- Fixes to unnecessary mentioning of pytest-bdd package files in py.test log with -v (bubenkoff)

34.84 0.6.5

- Compartibility with recent pytest (bubenkoff)

34.85 0.6.4

- More unicode fixes (amakhnach)

34.86 0.6.3

- Added unicode support for feature files. Removed buggy module replacement for scenario. (amakhnach)

34.87 0.6.2

- Removed unnecessary mention of pytest-bdd package files in py.test log with -v (bubenkoff)

34.88 0.6.1

- Step arguments in whens when there are no given arguments used. (amakhnach, bubenkoff)

34.89 0.6.0

- Added step arguments support. (curzona, olegpidsadnyi, bubenkoff)
- Added checking of the step type order. (markon, olegpidsadnyi)

34.90 0.5.2

- Added extra info into output when FeatureError exception raises. (amakhnach)

34.91 0.5.0

- Added parametrization to scenarios
- Coveralls.io integration
- Test coverage improvement/fixes
- Correct wrapping of step functions to preserve function docstring

34.92 0.4.7

- Fixed Python 3.3 support

34.93 0.4.6

- Fixed a bug when py.test –fixtures showed incorrect filenames for the steps.

34.94 0.4.5

- Fixed a bug with the reuse of the fixture by given steps being evaluated multiple times.

34.95 0.4.3

- Update the license file and PYPI related documentation.