# Pytest-BDD Documentation

*Release 3.2b0*

**Oleg Pidsadnyi**

**Jun 27, 2020**

# Contents


**1 BDD library for the py.test runner** **5**
- 1.1 Install pytest-bdd . . . 5
- 1.2 Example . . . 5
- 1.3 Scenario decorator . . . 6
- 1.4 Step aliases . . . 7
- 1.5 Given step scope . . . 7
- 1.6 Step arguments . . . 8
- 1.7 Override fixtures via given steps . . . 10
- 1.8 Multiline steps . . . 11
- 1.9 Scenarios shortcut . . . 11
- 1.10 Scenario outlines . . . 12
- 1.11 Organizing your scenarios . . . 15
- 1.12 Test setup . . . 16
- 1.13 Backgrounds . . . 18
- 1.14 Reusing fixtures . . . 18
- 1.15 Reusing steps . . . 19
- 1.16 Using unicode in the feature files . . . 19
- 1.17 Default steps . . . 19
- 1.18 Feature file paths . . . 20
- 1.19 Avoid retyping the feature file name . . . 20
- 1.20 Relax strict Gherkin language validation . . . 21
- 1.21 Hooks . . . 21
- 1.22 Browser testing . . . 21
- 1.23 Reporting . . . 22
- 1.24 Test code generation helpers . . . 22
- 1.25 Advanced code generation . . . 22
- 1.26 Migration of your tests from versions 2.x.x . . . 23
- 1.27 Migration of your tests from versions 0.x.x-1.x.x . . . 23
- 1.28 License . . . 24

**2 Authors** **25**

**3 Changelog** **27**
- 3.1 Unreleased . . . 27
- 3.2 3.2b0 . . . 27
- 3.3 3.1.1 . . . 27
- 3.4 3.1.0 . . . 27

3.5 3.0.2 . . . 28
3.6 3.0.1 . . . 28
3.7 3.0.0 . . . 28
3.8 2.21.0 . . . 28
3.9 2.20.0 . . . 28
3.10 2.19.0 . . . 28
3.11 2.18.2 . . . 28
3.12 2.18.1 . . . 28
3.13 2.18.0 . . . 29
3.14 2.17.2 . . . 29
3.15 2.17.1 . . . 29
3.16 2.17.0 . . . 29
3.17 2.16.1 . . . 29
3.18 2.16.0 . . . 29
3.19 2.15.0 . . . 29
3.20 2.14.5 . . . 29
3.21 2.14.3 . . . 30
3.22 2.14.1 . . . 30
3.23 2.14.0 . . . 30
3.24 2.13.1 . . . 30
3.25 2.13.0 . . . 30
3.26 2.12.2 . . . 30
3.27 2.11.3 . . . 30
3.28 2.11.1 . . . 30
3.29 2.11.0 . . . 30
3.30 2.10.0 . . . 31
3.31 2.9.1 . . . 31
3.32 2.9.0 . . . 31
3.33 2.8.0 . . . 31
3.34 2.7.2 . . . 31
3.35 2.7.1 . . . 31
3.36 2.7.0 . . . 31
3.37 2.6.2 . . . 31
3.38 2.6.1 . . . 31
3.39 2.5.3 . . . 32
3.40 2.5.2 . . . 32
3.41 2.5.1 . . . 32
3.42 2.5.0 . . . 32
3.43 2.4.5 . . . 32
3.44 2.4.3 . . . 32
3.45 2.4.2 . . . 32
3.46 2.4.1 . . . 32
3.47 2.4.0 . . . 33
3.48 2.3.3 . . . 33
3.49 2.3.2 . . . 33
3.50 2.3.1 . . . 33
3.51 2.1.2 . . . 33
3.52 2.1.1 . . . 33
3.53 2.1.0 . . . 33
3.54 2.0.1 . . . 33
3.55 2.0.0 . . . 34
3.56 1.0.0 . . . 34
3.57 0.6.11 . . . 34
3.58 0.6.9 . . . 34

3.59 0.6.8 . . . 34
3.60 0.6.6 . . . 34
3.61 0.6.5 . . . 34
3.62 0.6.4 . . . 34
3.63 0.6.3 . . . 34
3.64 0.6.2 . . . 35
3.65 0.6.1 . . . 35
3.66 0.6.0 . . . 35
3.67 0.5.2 . . . 35
3.68 0.5.0 . . . 35
3.69 0.4.7 . . . 35
3.70 0.4.6 . . . 35
3.71 0.4.5 . . . 35
3.72 0.4.3 . . . 36

**Contents**


- *Welcome to Pytest-BDD's documentation!*
- *BDD library for the py.test runner*
  - *Install pytest-bdd*
  - *Example*
  - *Scenario decorator*
  - *Step aliases*
  - *Given step scope*
  - *Step arguments*
    - *Step arguments are fixtures as well!*
  - *Override fixtures via given steps*
  - *Multiline steps*
  - *Scenarios shortcut*
  - *Scenario outlines*
    - *Feature examples*
    - *Combine scenario outline and pytest parametrization*
  - *Organizing your scenarios*
  - *Test setup*
  - *Backgrounds*
  - *Reusing fixtures*
  - *Reusing steps*
  - *Using unicode in the feature files*
  - *Default steps*
  - *Feature file paths*
  - *Avoid retyping the feature file name*
  - *Relax strict Gherkin language validation*
  - *Hooks*
  - *Browser testing*
  - *Reporting*
  - *Test code generation helpers*
  - *Advanced code generation*
  - *Migration of your tests from versions 2.x.x*
  - *Migration of your tests from versions 0.x.x-1.x.x*
  - *License*
- *Authors*

- *Changelog*
  - *Unreleased*
  - *3.2b0*
  - *3.1.1*
  - *3.1.0*
  - *3.0.2*
  - *3.0.1*
  - *3.0.0*
  - *2.21.0*
  - *2.20.0*
  - *2.19.0*
  - *2.18.2*
  - *2.18.1*
  - *2.18.0*
  - *2.17.2*
  - *2.17.1*
  - *2.17.0*
  - *2.16.1*
  - *2.16.0*
  - *2.15.0*
  - *2.14.5*
  - *2.14.3*
  - *2.14.1*
  - *2.14.0*
  - *2.13.1*
  - *2.13.0*
  - *2.12.2*
  - *2.11.3*
  - *2.11.1*
  - *2.11.0*
  - *2.10.0*
  - *2.9.1*
  - *2.9.0*
  - *2.8.0*
  - *2.7.2*

- 2.7.1
- 2.7.0
- 2.6.2
- 2.6.1
- 2.5.3
- 2.5.2
- 2.5.1
- 2.5.0
- 2.4.5
- 2.4.3
- 2.4.2
- 2.4.1
- 2.4.0
- 2.3.3
- 2.3.2
- 2.3.1
- 2.1.2
- 2.1.1
- 2.1.0
- 2.0.1
- 2.0.0
- 1.0.0
- 0.6.11
- 0.6.9
- 0.6.8
- 0.6.6
- 0.6.5
- 0.6.4
- 0.6.3
- 0.6.2
- 0.6.1
- 0.6.0
- 0.5.2
- 0.5.0
- 0.4.7

- *0.4.6*
- *0.4.5*
- *0.4.3*

# CHAPTER 1

## BDD library for the py.test runner

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.

## 1.1 Install pytest-bdd

```
pip install pytest-bdd
```

The minimum required version of pytest is 3.3.2

## 1.2 Example

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

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: will query the database
```

Note that only one feature is allowed per feature file.

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')
def article(author):
    return create_test_article(author=author)


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

## 1.3 Scenario decorator

The scenario decorator can accept the following optional keyword arguments:

- `encoding` - decode content of feature file in specific encoding. UTF-8 is default.
- `example_converters` - mapping to pass functions to convert example values provided in feature files.

Functions decorated with the *scenario* decorator behave like a normal test function, and they will be executed after all scenario steps. You can consider it as a normal pytest test function, e.g. order fixtures there, call other functions and make assertions:

```
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
```

## 1.4 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):
    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.

```
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
```

## 1.5 Given step scope

If you need your given step to be executed less than once per scenario (for example: once for module, session), you can pass optional `scope` argument:

```
@given('there is an article', scope='session')
def article(author):
    return create_test_article(author=author)
```

```
Scenario: I'm the author
    Given I'm an author
    And there is an article


Scenario: I'm the admin
```

(continues on next page)

(continued from previous page)

```
    Given I'm the admin
    And there is an article
```

In this example, the step function for the 'there is an article' given step will be executed once, even though there are 2 scenarios using it. Note that for other step types, it makes no sense to have scope larger than 'function' (the default) as they represent an action (when step), and assertion (then step).

## 1.6 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=dict(Number=int)))
def start_cucumbers(start):
    return dict(start=start, eat=0)
```

for *re* parser

```
from pytest_bdd import parsers

@given(parsers.re(r'there are (?P<start>\d+) cucumbers'), converters=dict(start=int))
def start_cucumbers(start):
    return dict(start=start, eat=0)
```

Example:

```
Scenario: Arguments for given, when, thens
    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:

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


@scenario('arguments.feature', 'Arguments for given, when, thens')
def test_arguments():
    pass


@given(parsers.parse('there are {start:d} cucumbers'))
def start_cucumbers(start):
    return dict(start=start, eat=0)


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


@then(parsers.parse('I should have {left:d} cucumbers'))
def should_have_left_cucumbers(start_cucumbers, start, left):
    assert start_cucumbers['start'] == start
    assert start - 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 looks like:

```
import re

from pytest_bdd import given, parsers

class MyParser(parsers.StepParser):

    """Custom parser."""

    def __init__(self, name, **kwargs):
        """Compile regex."""
        super(re, self).__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()
```

(continues on next page)

(continued from previous page)

```
    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'))
def start_cucumbers(start):
    return dict(start=start, eat=0)
```

### 1.6.1 Step arguments are fixtures as well!

Step arguments are injected into pytest *request* context as normal fixtures with the names equal to the names of the arguments. This opens a number of possibilies:

- you can access step's argument as a fixture in other step function just by mentioning it as an argument (just like any othe pytest fixture)
- if the name of the step argument clashes with existing fixture, it will be overridden by step's argument value; this way you can set/override the value for some fixture deeply inside of the fixture tree in a ad-hoc way by just choosing the proper name for the step argument.

## 1.7 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'
```

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

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

## 1.8 Multiline steps

As Gherkin, pytest-bdd supports multiline steps (aka PyStrings). But in much cleaner and powerful way:

```
Scenario: Multiline step using sub indentation
    Given I have a step with:
        Some
        Extra
        Lines
    Then the text should be parsed with correct indentation
```

Step is considered as multiline one, if the **next** line(s) after it's first line, is indented relatively to the first line. The step name is then simply extended by adding further lines with newlines. In the example above, the Given step name will be:

```
'I have a step with:\nSome\nExtra\nLines'
```

You can of course register step using full name (including the newlines), but it seems more practical to use step arguments and capture lines after first line (or some subset of them) into the argument:

```
import re

from pytest_bdd import given, then, scenario


@scenario(
    'multiline.feature',
    'Multiline step using sub indentation',
)
def test_multiline():
    pass


@given(parsers.parse('I have a step with:\n{text}'))
def i_have_text(text):
    return text


@then('the text should be parsed with correct indentation')
def text_should_be_correct(i_have_text, text):
    assert i_have_text == text == 'Some\nExtra\nLines'
```

Note that *then* step definition (*text_should_be_correct*) in this example uses *text* fixture which is provided by a a *given* step (*i_have_text*) argument with the same name (*text*). This possibility is described in the *Step arguments are fixtures as well!* section.

## 1.9 Scenarios shortcut

If you have 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* scenarios found in the *feature* folder(s) recursively automatically. For this - there's a *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 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 *test_something* scenario binding will be kept manual, other scenarios found in the *features* folder will be bound automatically.

## 1.10 Scenario outlines

Scenarios can be parametrized to cover few cases. In Gherkin the variable templates are written using corner braces as <somevalue>. Gherkin scenario outlines are supported by pytest-bdd exactly as it's described in be behave docs.

Example:

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

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

pytest-bdd feature file format also supports example tables in different way:

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

    Examples: Vertical
    | start | 12 | 2 |
    | eat   | 5  | 1 |
    | left  | 7  | 1 |
```

This form allows to have tables with lots of columns keeping the maximum text width predictable without significant readability change.

The code will look like:

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


@scenario(
    'outline.feature',
    'Outlined given, when, thens',
    example_converters=dict(start=int, eat=float, left=str)
)
def test_outlined():
    pass


@given('there are <start> cucumbers')
def start_cucumbers(start):
    assert isinstance(start, int)
    return dict(start=start)


@when('I eat <eat> cucumbers')
def eat_cucumbers(start_cucumbers, eat):
    assert isinstance(eat, float)
    start_cucumbers['eat'] = eat


@then('I should have <left> cucumbers')
def should_have_left_cucumbers(start_cucumbers, start, eat, left):
    assert isinstance(left, str)
    assert start - eat == int(left)
    assert start_cucumbers['start'] == start
    assert start_cucumbers['eat'] == eat
```

Example code also shows possibility to pass example converters which may be useful if you need parameter types different than strings.

### 1.10.1 Feature examples

It's possible to declare example table once for the whole feature, and it will be shared among all the scenarios of that feature:

```
Feature: Outline

    Examples:
    | start | eat | left |
    |  12   |  5  |  7   |
    |  5    |  4  |  1   |

    Scenario Outline: Eat cucumbers
        Given there are <start> cucumbers
        When I eat <eat> cucumbers
        Then I should have <left> cucumbers

    Scenario Outline: Eat apples
```

(continues on next page)

(continued from previous page)

```
        Given there are <start> apples
        When I eat <eat> apples
        Then I should have <left> apples
```

For some more complex case, you might want to parametrize on both levels: feature and scenario. This is allowed as long as parameter names do not clash:

```
Feature: Outline

    Examples:
    | start | eat | left |
    |  12   |  5  |  7   |
    |  5    |  4  |  1   |

    Scenario Outline: Eat fruits
        Given there are <start> <fruits>
        When I eat <eat> <fruits>
        Then I should have <left> <fruits>

        Examples:
        | fruits  |
        | oranges |
        | apples  |

    Scenario Outline: Eat vegetables
        Given there are <start> <vegetables>
        When I eat <eat> <vegetables>
        Then I should have <left> <vegetables>

        Examples:
        | vegetables |
        | carrots    |
        | tomatoes   |
```

### 1.10.2 Combine scenario outline and pytest parametrization

It's also possible to parametrize the scenario on the python side. The reason for this is that it is sometimes not needed to mention example table for every scenario.

The code will look like:

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


# Here we use pytest to parametrize the test with the parameters table
@pytest.mark.parametrize(
    ['start', 'eat', 'left'],
    [(12, 5, 7)])
@scenario(
    'parametrized.feature',
    'Parametrized given, when, thens',
)
# Note that we should take the same arguments in the test function that we use
# for the test parametrization either directly or indirectly (fixtures depend on␣
↪them).
```

(continues on next page)

(continued from previous page)

```
def test_parametrized(start, eat, left):
    """We don't need to do anything here, everything will be managed by the scenario␣
↪decorator."""


@given('there are <start> cucumbers')
def start_cucumbers(start):
    return dict(start=start)


@when('I eat <eat> cucumbers')
def eat_cucumbers(start_cucumbers, start, eat):
    start_cucumbers['eat'] = eat


@then('I should have <left> cucumbers')
def should_have_left_cucumbers(start_cucumbers, start, eat, left):
    assert start - eat == left
    assert start_cucumbers['start'] == start
    assert start_cucumbers['eat'] == eat
```

With a parametrized.feature file:

```
Feature: parametrized

Scenario: Parametrized given, when, thens
    Given there are <start> cucumbers
    When I eat <eat> cucumbers
    Then I should have <left> cucumbers
```

The significant downside of this approach is inability to see the test table from the feature file.

## 1.11 Organizing your scenarios

The more features and scenarios you have, the more important becomes the question about their organization. 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 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 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 tags introduce standard 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:

```
py.test -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 prefix like *bdd*. Note that if you use pytest *–strict* option, all bdd 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-compartible variable names, eg starts with a non-number, underscore alphanumberic, etc. That way you can safely use tags for tests filtering.

You can customize how hooks 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 pytest-bdd's default behavior
        return None
```

## 1.12 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')
def article():
    return Article(is_beautiful=True)
```

This also declares a PyTest fixture "article" and any other step can depend on it.

```
Given I have a beautiful article
When I publish this article
```

When step is referring the article to publish it.

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

Many other BDD toolkits operate 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 in the 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.

```
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.

```
given('I have a beautiful article', fixture='article')

@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. In case of Given steps and the step arguments mentioning the same given step makes no sense. It won't be executed second time.

```
Given I have a beautiful article
And some other thing
And I have a beautiful article  # Won't be executed, exception is raised
```

pytest-bdd will raise an exception even in the case of the steps that use regular expression patterns to get arguments.

```
Given I have 1 cucumbers
And I have 2 cucumbers  # Exception is raised
```

Will raise an exception if the step is using the regular expression pattern.

```
@given(re.compile('I have (?P<n>\d+) cucumbers'))
def cucumbers(n):
    return create_cucumbers(n)
```

## 1.13 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 possibility to prepare some common setup for multiple scenarios in a single feature. About background best practices, please read here.

---

**Note:** There is only step "Given" should be used in "Background" section, steps "When" and "Then" are prohibited, because their purpose are related to actions and consuming outcomes, that is conflict with "Background" aim - prepare system for tests or "put the system in a known state" as "Given" does it. The statement above is applied for strict Gherkin mode, which is enabled by default.

---

## 1.14 Reusing fixtures

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

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

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

```
given('I have beautiful article', fixture='article')
```

This will be equivalent to:

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

## 1.15 Reusing steps

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

common_steps.feature:

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

conftest.py:

```
from pytest_bdd import given, then


@given('I have a bar')
def bar():
    return 'bar'


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

test_common.py:

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

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

## 1.16 Using unicode in the feature files

As mentioned above, by default, utf-8 encoding is used for parsing feature files. For steps definition, you should use unicode strings, which is the default in python 3. If you are on python 2, make sure you use unicode strings by prefixing them with the *u* sign.

```
@given(parsers.re(u"      '{0}'".format(u'(?P<content>.+)')))
def there_is_a_string_with_content(content, string):
    """Create string with unicode content."""
    string['content'] = content
```

## 1.17 Default steps

Here is the list of steps that are implemented inside of the 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()*

## 1.18 Feature file paths

By default, pytest-bdd will use current module's path as 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 working directory when starting pytest. You can also override 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.

## 1.19 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:

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
```

(continues on next page)

(continued from previous page)

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

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

## 1.20 Relax strict Gherkin language validation

If your scenarios are not written in *proper* Gherkin language, e.g. they are more like textual scripts, then you might find it hard to use *pytest-bdd* as by default it validates the order of step types (given-when-then). To relax that validation, just pass `strict_gherkin=False` to the `scenario` and `scenarios` decorators:

test_publish_article.py:

```
from pytest_bdd import scenario

@scenario('publish_article.feature', 'Publishing the article in a weird way', strict_
↪gherkin=False)
def test_publish():
    pass
```

## 1.21 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 it's 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_validation_error(request, feature, scenario, step, step_func, step_func_args, exception) - Called when step failed to validate
- pytest_bdd_step_func_lookup_error(request, feature, scenario, step, exception) - Called when step lookup failed

## 1.22 Browser testing

Tools recommended to use for browser testing:

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

## 1.23 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 this jenkins plugin

To have an output in json format:

```
py.test --cucumberjson=<path to json report>
```

This will output an expanded (meaning scenario outlines will be expanded to several scenarios) cucumber format. To also fill in parameters in the step name, you have to explicitly tell pytest-bdd to use the expanded format:

```
py.test --cucumberjson=<path to json report> --cucumberjson-expanded
```

To enable gherkin-formatted output on terminal, use

```
py.test --gherkin-terminal-reporter
```

Terminal reporter supports expanded format as well

```
py.test --gherkin-terminal-reporter-expanded
```

## 1.24 Test code generation helpers

For newcomers it's sometimes hard to write all needed test code without being frustrated. To simplify their life, simple code generator was implemented. It allows to create fully functional but of course empty tests and step definitions for given a 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
```

## 1.25 Advanced code generation

For more experienced users, there's 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:

```
py.test --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
```

(continues on next page)

(continued from previous page)

```
--------------------------------------------------------------------------------

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.

## 1.26 Migration of your tests from versions 2.x.x

In version 3.0.0, the fixtures `pytestbdd_feature_base_dir` and `pytestbdd_strict_gherkin` have been removed.

If you used `pytestbdd_feature_base_dir` fixture to override the path discovery, you can instead configure it in `pytest.ini`:

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

For more details, check the *Feature file paths* section.

If you used `pytestbdd_strict_gherkin` fixture to relax the parser, you can instead specify `strict_gherkin=False` in the declaration of your scenarios, or change it globally in the pytest configuration file:

```
[pytest]
bdd_strict_gherkin = false
```

For more details, check the *Relax strict Gherkin language validation* section.

## 1.27 Migration of your tests from versions 0.x.x-1.x.x

In version 2.0.0, the backwards-incompartible change was introduced: scenario function can now only be used as a decorator. Reasons for that:

- test code readability is much higher using normal python function syntax;
- pytest-bdd internals are much cleaner and shorter when using single approach instead of supporting two;

- after moving to parsing-on-import-time approach for feature files, it's not possible to detect whether it's a decorator more or not, so to support it along with functional approach there needed to be special parameter for that, which is also a backwards-incompartible change.

To help users migrate to newer version, there's migration subcommand of the *pytest-bdd* console script:

```
# run migration script
pytest-bdd migrate <your test folder>
```

Under the hood the script does the replacement from this:

```
test_function = scenario('publish_article.feature', 'Publishing the article')
```

to this:

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

## 1.28 License

This software is licensed under the MIT license.

CHAPTER 2

# 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
- Laurence Rowe
- Leonardo Santagada
- Milosz Sliwinski
- Michiel Holtkamp
- Robin Pedersen
- Sergey Kraynev

CHAPTER 3

# Changelog

## 3.1 Unreleased

## 3.2 3.2b0

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

## 3.3 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.

## 3.4 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.

## 3.5 3.0.2

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

## 3.6 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.

## 3.7 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

## 3.8 2.21.0

- Gherkin terminal reporter expanded format (pauk-slon)

## 3.9 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)

## 3.10 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)

## 3.11 2.18.2

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

## 3.12 2.18.1

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

## 3.13 2.18.0

- Add gherkin terminal reporter (spinus + thedrow)

## 3.14 2.17.2

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

## 3.15 2.17.1

- Add support for pytest 3.0

## 3.16 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`.

## 3.17 2.16.1

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

## 3.18 2.16.0

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

## 3.19 2.15.0

- Add examples data in the scenario report (bubenkoff)

## 3.20 2.14.5

- Properly parse feature description (bubenkoff)

## 3.21 2.14.3

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

## 3.22 2.14.1

- Pass additional arguments to parsers (bubenkoff)

## 3.23 2.14.0

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

## 3.24 2.13.1

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

## 3.25 2.13.0

- Feature example table (bubenkoff, sureshvv)

## 3.26 2.12.2

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

## 3.27 2.11.3

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

## 3.28 2.11.1

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

## 3.29 2.11.0

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

## 3.30 2.10.0

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

## 3.31 2.9.1

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

## 3.32 2.9.0

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

## 3.33 2.8.0

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

## 3.34 2.7.2

- Correct base feature path lookup for python3 (bubenkoff)

## 3.35 2.7.1

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

## 3.36 2.7.0

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

## 3.37 2.6.2

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

## 3.38 2.6.1

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

## 3.39 2.5.3

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

## 3.40 2.5.2

- Fix code generation steps ordering (bubenkoff)

## 3.41 2.5.1

- Fix error report serialization (olegpidsadnyi)

## 3.42 2.5.0

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

## 3.43 2.4.5

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

## 3.44 2.4.3

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

## 3.45 2.4.2

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

## 3.46 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)

## 3.47 2.4.0

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

## 3.48 2.3.3

- Added timings to the cucumber json report (bubenkoff)

## 3.49 2.3.2

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

## 3.50 2.3.1

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

## 3.51 2.1.2

- Latest pytest compartibility fixes (bubenkoff)

## 3.52 2.1.1

- Bugfixes (bubenkoff)

## 3.53 2.1.0

- Implemented multiline steps (bubenkoff)

## 3.54 2.0.1

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

## 3.55 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)

## 3.56 1.0.0

- Implemented scenario outlines (bubenkoff)

## 3.57 0.6.11

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

## 3.58 0.6.9

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

## 3.59 0.6.8

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

## 3.60 0.6.6

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

## 3.61 0.6.5

- Compartibility with recent pytest (bubenkoff)

## 3.62 0.6.4

- More unicode fixes (amakhnach)

## 3.63 0.6.3

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

## 3.64 0.6.2

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

## 3.65 0.6.1

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

## 3.66 0.6.0

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

## 3.67 0.5.2

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

## 3.68 0.5.0

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

## 3.69 0.4.7

- Fixed Python 3.3 support

## 3.70 0.4.6

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

## 3.71 0.4.5

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

## 3.72 0.4.3

- Update the license file and PYPI related documentation.