behave Documentation

Release 1.4.0.dev0

Jens Engel, Benno Rice and Richard Jones

2026-06-12

CONTENTS

behave is behaviour-driven development, Python style.

Behavior-driven development (or BDD) is an agile software development technique that encourages collaboration between developers, QA and non-technical or business participants in a software project. We have a page further describing this *philosophy*.

behave uses tests written in a natural language style, backed up by Python code.

Once you've *installed behave*, we recommend reading the

- *tutorial* first and then
- *feature test setup*,
- *behave API* and
- *related software* (things that you can combine with behave)
- finally: *how to use and configure* the behave tool.

There is also a *comparison* with the other tools available.

CHAPTER

ONE

CONTENTS

1.1 Installation

1.1.1 Using pip (or . . .)

Category
Stable version

Precondition
pip (or setuptools) is installed

Execute the following command to install behave with pip:

```
pip install behave
```

To update an already installed behave version, use:

```
pip install -U behave
```

Hint

See also pip related information for installing Python packages.

1.1.2 Using a Source Distribution

After unpacking the behave source distribution, enter the newly created directory "behave-<version>" and run:

```
pip install .
```

1.1.3 Using the GitHub Repository

Category
Bleeding edge

Precondition
pip is installed

Run the following command to install the newest version from the GitHub repository:

```
pip install git+https://github.com/behave/behave
```

To install a tagged version from the GitHub repository, use:

```
pip install git+https://github.com/behave/behave@<TAG>
```

where <TAG> is the placeholder for an existing tag.

When installing extras, use `<TAG>#egg=behave[...]`, e.g.:

```
pip install git+https://github.com/behave/behave@v1.3.1#egg=behave[develop]
```

1.1.4 Optional Dependencies

If needed, additional dependencies ("extras") can be installed using `pip install` with one of the following installation targets.

Installation Target	Description
`behave[docs]`	Include packages needed for building Behave's documentation.
`behave[develop]`	Optional packages helpful for local development.
`behave[formatters]`	Install formatters from behave-contrib to extend the list of *formatters* provided by default.

1.1.5 Specify Dependency to "behave"

Use the following recipe in the `"pyproject.toml"` config-file if:

- your project depends on behave and
- you use a `version` from the git-repository (or a `git branch`)

EXAMPLE:

```
# -- FILE: my-project/pyproject.toml
# SCHEMA: Use "behave" from git-repository (instead of: https://pypi.org/ )
#   "behave @ git+https://github.com/behave/behave.git@<TAG>"
#   "behave @ git+https://github.com/behave/behave.git@<BRANCH>"
#   "behave[VARIANT] @ git+https://github.com/behave/behave.git@<TAG>" # with␣
↪VARIANT=develop, docs, ...
# SEE: https://peps.python.org/pep-0508/

[project]
name = "my-project"
dependencies = [
    "behave @ git+https://github.com/behave/behave.git@v1.3.1",
    # OR: "behave[develop] @ git+https://github.com/behave/behave.git@main",
]
```

1.2 Tutorial

First, *install behave*.

Now make a directory called "features". In that directory create a file called "tutorial.feature" containing:

```
Feature: showing off behave

  Scenario: run a simple test
     Given we have behave installed
      When we implement a test
      Then behave will test it for us!
```

Make a new directory called "features/steps". In that directory create a file called "tutorial.py" containing:

```
from behave import *

@given('we have behave installed')
def step_impl(context):
```

(continues on next page)

(continued from previous page)

```
    pass

@when('we implement a test')
def step_impl(context):
    assert True is not False

@then('behave will test it for us!')
def step_impl(context):
    assert context.failed is False
```

Run behave:

```
% behave
Feature: showing off behave # features/tutorial.feature:1

  Scenario: run a simple test        # features/tutorial.feature:3
    Given we have behave installed   # features/steps/tutorial.py:3
    When we implement a test         # features/steps/tutorial.py:7
    Then behave will test it for us! # features/steps/tutorial.py:11

1 feature passed, 0 failed, 0 skipped
1 scenario passed, 0 failed, 0 skipped
3 steps passed, 0 failed, 0 skipped, 0 undefined
```

Now, continue reading to learn how to make the most of *behave*.

1.2.1 Features

behave operates on directories containing:

1. *feature files* written by your Business Analyst / Sponsor / whoever with your behaviour scenarios in it, and
2. a "steps" directory with *Python step implementations* for the scenarios.

You may optionally include some *environmental controls* (code to run before and after steps, scenarios, features or the whole shooting match).

The minimum requirement for a features directory is:

```
features/
features/everything.feature
features/steps/
features/steps/steps.py
```

A more complex directory might look like:

```
features/
features/signup.feature
features/login.feature
features/account_details.feature
features/environment.py
features/steps/
features/steps/website.py
features/steps/utils.py
```

If you're having trouble setting things up and want to see what *behave* is doing in attempting to find your features use the "-v" (verbose) command-line switch.

1.2.2 Feature Files

A feature file has a *natural language format* describing a feature or part of a feature with representative examples of expected outcomes. They're plain-text (encoded in UTF-8) and look something like:

```
Feature: Fight or flight
  In order to increase the ninja survival rate,
  As a ninja commander
  I want my ninjas to decide whether to take on an
  opponent based on their skill levels

  Scenario: Weaker opponent
    Given the ninja has a third level black-belt
     When attacked by a samurai
     Then the ninja should engage the opponent

  Scenario: Stronger opponent
    Given the ninja has a third level black-belt
     When attacked by Chuck Norris
     Then the ninja should run for his life
```

The "Given", "When" and "Then" parts of this prose form the actual steps that will be taken by *behave* in testing your system. These map to *Python step implementations*. As a general guide:

Given we *put the system in a known state* before the user (or external system) starts interacting with the system (in the When steps). Avoid talking about user interaction in givens.

When we *take key actions* the user (or external system) performs. This is the interaction with your system which should (or perhaps should not) cause some state to change.

Then we *observe outcomes*.

You may also include "And" or "But" as a step - these are renamed by *behave* to take the name of their preceding step, so:

```
Scenario: Stronger opponent
  Given the ninja has a third level black-belt
   When attacked by Chuck Norris
   Then the ninja should run for his life
    And fall off a cliff
```

In this case *behave* will look for a step definition for `"Then fall off a cliff"`.

Scenario Outlines

Sometimes a scenario should be run with a number of variables giving a set of known states, actions to take and expected outcomes, all using the same basic actions. You may use a Scenario Outline to achieve this:

```
Scenario Outline: Blenders
   Given I put <thing> in a blender,
    When I switch the blender on
    Then it should transform into <other thing>

 Examples: Amphibians
   | thing         | other thing |
   | Red Tree Frog | mush        |

 Examples: Consumer Electronics
   | thing         | other thing |
   | iPhone        | toxic waste |
   | Galaxy Nexus  | toxic waste |
```

behave will run the scenario once for each (non-heading) line appearing in the example data tables.

Step Data

Sometimes it's useful to associate a table of data with your step.

Any text block following a step wrapped in """ lines will be associated with the step. For example:

```
Scenario: some scenario
  Given a sample text loaded into the frobulator
     """
     Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
     eiusmod tempor incididunt ut labore et dolore magna aliqua.
     """
 When we activate the frobulator
 Then we will find it similar to English
```

The text is available to the Python step code as the ".text" attribute in the *Context* variable passed into each step function.

You may also associate a table of data with a step by simply entering it, indented, following the step. This can be useful for loading specific required data into a model.

```
Scenario: some scenario
  Given a set of specific users
     | name       | department  |
     | Barry      | Beer Cans   |
     | Pudey      | Silly Walks |
     | Two-Lumps  | Silly Walks |

 When we count the number of people in each department
 Then we will find two people in "Silly Walks"
  But we will find one person in "Beer Cans"
```

The table is available to the Python step code as the ".table" attribute in the *Context* variable passed into each step function. The table for the example above could be accessed like so:

```
@given('a set of specific users')
def step_impl(context):
    for row in context.table:
        model.add_user(name=row['name'], department=row['department'])
```

There's a variety of ways to access the table data - see the *Table* API documentation for the full details.

1.2.3 Python Step Implementations

Steps used in the scenarios are implemented in Python files in the "steps" directory. You can call these whatever you like as long as they use the python `*.py` file extension. You don't need to tell *behave* which ones to use - it'll use all of them.

The full detail of the Python side of *behave* is in the *API documentation*.

Steps are identified using decorators which match the predicate from the feature file: **given**, **when**, **then** and **step** (variants with Title case are also available if that's your preference.) The decorator accepts a string containing the rest of the phrase used in the scenario step it belongs to.

Given a Scenario:

```
Scenario: Search for an account
   When I search for a valid account
   Then I will see the account details
```

Step code implementing the two steps here might look like (using selenium webdriver and some other helpers):

```
@when('I search for a valid account')
def step_impl(context):
    context.browser.get('http://localhost:8000/index')
    form = get_element(context.browser, tag='form')
    get_element(form, name="msisdn").send_keys('61415551234')
    form.submit()

@then('I will see the account details')
def step_impl(context):
    elements = find_elements(context.browser, id='no-account')
    eq_(elements, [], 'account not found')
    h = get_element(context.browser, id='account-head')
    ok_(h.text.startswith("Account 61415551234"),
        'Heading %r has wrong text' % h.text)
```

The `step` decorator matches the step to *any* step type, "given", "when" or "then". The "and" and "but" step types are renamed internally to take the preceding step's keyword (so an "and" following a "given" will become a "given" internally and use a **given** decorated step).

> **Note**
>
> Step function names do not need to have a unique symbol name, because the text matching selects the step function from the step registry before it is called as anonymous function. Hence, when *behave* prints out the missing step implementations in a test run, it uses "step_impl" for all functions by default.

If you find you'd like your step implementation to invoke another step you may do so with the *`Context`* method *`execute_steps()`*.

This function allows you to, for example:

```
@when('I do the same thing as before')
def step_impl(context):
    context.execute_steps('''
        when I press the big red button
         and I duck
    ''')
```

This will cause the "when I do the same thing as before" step to execute the other two steps as though they had also appeared in the scenario file.

Step Parameters

Steps sometimes include very common phrases with only one variation (one word is different or some words are different). For example:

```
# -- FILE: features/example_step_parameters.feature
Scenario: look up a book
  When I search for a valid book
  Then the result page will include "success"

Scenario: look up an invalid book
  When I search for a invalid book
  Then the result page will include "failure"
```

You can define one Python step-definition that handles both cases by using *step parameters* . In this case, the *Then* step verifies the `context.response` parameter that was stored in the `context` by the *When* step:

```
# -- FILE: features/steps/example_steps_with_step_parameters.py
# HINT: Step-matcher "parse" is the DEFAULT step-matcher class.
from behave import then

@then('the result page will include "{text}"')
def step_impl(context, text):
    if text not in context.response:
        fail('%r not in %r' % (text, context.response))
```

There are several step-matcher classes available in **behave** that can be used for *step parameters*. You can select another step-matcher class by using the *behave.use_step_matcher()* function:

```
# -- FILE: features/steps/example_use_step_matcher_in_steps.py
# HINTS:
#   * "parse" in the DEFAULT step-matcher
#   * Use "use_step_matcher(...)" in "features/environment.py" file
#     to define your own own default step-matcher.
from behave import given, when, use_step_matcher

use_step_matcher("cfparse")

@given('some event named "{event_name}" happens')
def step_given_some_event_named_happens(context, event_name):
    pass    # ... DETAILS LEFT OUT HERE.

use_step_matcher("re")

@when('a person named "(?P<name>...)" enters the room')
def step_when_person_enters_room(context, name):
    pass    # ... DETAILS LEFT OUT HERE.
```

Step-matchers

There are several step-matcher classes available in **behave** that can be used for parsing *step parameters*:

- **parse** (default step-matcher class, based on: parse):
- **cfparse** (extends: parse, requires: parse_type):
- **re** (step-matcher class is based on regular expressions):

Step-matcher: parse

This step-matcher class provides a parser based on: parse module.

It 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 parse in step definitions. The named fields are extracted, optionally type converted and then used as step function arguments.

FEATURES:

- Supports named step parameters (and unnamed step parameters)
- Supports **type conversions** by using type converters (see *register_type()*).

Step-matcher: cfparse

This step-matcher class extends the `parse` step-matcher and provides an extended parser with "Cardinality Field" (CF) support.

It automatically creates missing type converters for other cardinalities as long as a type converter for cardinality=1 is provided.

It supports parse expressions like:

- `{values:Type+}` (cardinality=1..N, many)
- `{values:Type*}` (cardinality=0..N, many0)
- `{value:Type?}` (cardinality=0..1, optional).

FEATURES:

- Supports named step parameters (and unnamed step parameters)
- Supports **type conversions** by using type converters (see *register_type()*).

Step-matcher: re

This step-matcher provides step-matcher class is based on regular expressions. It 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.

> **Hint**
>
> Type conversion is **not supported**.
>
> A step function writer may implement type conversion inside the step function (implementation).

To specify which parser to use, call the *use_step_matcher()* function with the name of the step-matcher class to use.

You can change the step-matcher class at any time to suit your needs. The following step-definitions use the current step-matcher class.

FEATURES:

- Supports named step parameters (and unnamed step parameters)
- Supports no type conversions

VARIANTS:

- `"re0"`: Provides a regex matcher that is compatible with `cucumber` (regex based step-matcher).

Context

You'll have noticed the "context" variable that's passed around. It's a clever place where you and *behave* can store information to share around. It runs at three levels, automatically managed by *behave*.

When *behave* launches into a new feature or scenario it adds a new layer to the context, allowing the new activity level to add new values, or overwrite ones previously defined, for the duration of that activity. These can be thought of as scopes.

You can define values in your *environmental controls* file which may be set at the feature level and then overridden for some scenarios. Changes made at the scenario level won't permanently affect the value set at the feature level.

You may also use it to share values between steps. For example, in some steps you define you might have:

```
@when('I request a new widget for an account via SOAP')
def step_impl(context):
    client = Client("http://127.0.0.1:8000/soap/")
    context.response = client.Allocate(customer_first='Firstname',
        customer_last='Lastname', colour='red')

@then('I should receive an OK SOAP response')
def step_impl(context):
    eq_(context.response['ok'], 1)
```

There's also some values added to the context by *behave* itself:

table
: This holds any table data associated with a step.

text
: This holds any multi-line text associated with a step.

failed
: This is set at the root of the context when any step fails. It is sometimes useful to use this combined with the `--stop` command-line option to prevent some mis-behaving resource from being cleaned up in an `after_feature()` or similar (for example, a web browser being driven by Selenium.)

The *context* variable in all cases is an instance of *behave.runner.Context*.

1.2.4 Environmental Controls

The environment.py module may define code to run before and after certain events during your testing:

before_step(context, step), after_step(context, step)
: These run before and after every step.

before_scenario(context, scenario), after_scenario(context, scenario)
: These run before and after each scenario is run.

before_feature(context, feature), after_feature(context, feature)
: These run before and after each feature file is exercised.

before_tag(context, tag), after_tag(context, tag)
: These run before and after a section tagged with the given name. They are invoked for each tag encountered in the order they're found in the feature file. See *controlling things with tags*.

before_all(context), after_all(context)
: These run before and after the whole shooting match.

The feature, scenario and step objects represent the information parsed from the feature file. They have a number of attributes:

keyword
: "Feature", "Scenario", "Given", etc.

name
: The name of the step (the text after the keyword.)

tags
: A list of the tags attached to the section or step. See *controlling things with tags*.

filename and line
: The file name (or "<string>") and line number of the statement.

A common use-case for environmental controls might be to set up a web server and browser to run all your tests in. For example:

```
# -- FILE: features/environment.py
from behave import fixture, use_fixture
from behave4my_project.fixtures import wsgi_server
from selenium import webdriver

@fixture
def selenium_browser_chrome(context):
    # -- HINT: @behave.fixture is similar to @contextlib.contextmanager
    context.browser = webdriver.Chrome()
    yield context.browser
    # -- CLEANUP-FIXTURE PART:
    context.browser.quit()

def before_all(context):
    use_fixture(wsgi_server, context, port=8000)
    use_fixture(selenium_browser_chrome, context)
    # -- HINT: CLEANUP-FIXTURE is performed after after_all() hook is called.

def before_feature(context, feature):
    model.init(environment='test')
```

```
# -- FILE: behave4my_project/fixtures.py
# ALTERNATIVE: Place fixture in "features/environment.py" (but reuse is harder)
from behave import fixture
import threading
from wsgiref import simple_server
from my_application import model
from my_application import web_app

@fixture
def wsgi_server(context, port=8000):
    context.server = simple_server.WSGIServer(('', port))
    context.server.set_app(web_app.main(environment='test'))
    context.thread = threading.Thread(target=context.server.serve_forever)
    context.thread.start()
    yield context.server
    # -- CLEANUP-FIXTURE PART:
    context.server.shutdown()
    context.thread.join()
```

Of course, if you wish, you could have a new browser for each feature, or to retain the database state between features or even initialise the database for each scenario.

1.2.5 Controlling Things With Tags

You may also "tag" parts of your feature file. At the simplest level this allows *behave* to selectively check parts of your feature set.

Given a feature file with:

```
Feature: Fight or flight
  In order to increase the ninja survival rate,
  As a ninja commander
  I want my ninjas to decide whether to take on an
  opponent based on their skill levels

  @slow
```

(continues on next page)

(continued from previous page)

```
  Scenario: Weaker opponent
    Given the ninja has a third level black-belt
    When attacked by a samurai
    Then the ninja should engage the opponent

  Scenario: Stronger opponent
    Given the ninja has a third level black-belt
    When attacked by Chuck Norris
    Then the ninja should run for his life
```

then running `behave --tags=slow` will run just the scenarios tagged `@slow`. If you wish to check everything *except* the slow ones then you may run `behave --tags="not @slow"`.

Another common use-case is to tag a scenario you're working on with `@wip` and then `behave --tags=wip` to just test that one case.

Tag selection on the command-line may be combined:

- **`--tags="@wip or @slow"`**
  This will select all the cases tagged *either* "wip" or "slow".
- **`--tags="@wip and @slow"`**
  This will select all the cases tagged *both* "wip" and "slow".

If a feature or scenario is tagged and then skipped because of a command-line control then the *before_* and *after_* environment functions will not be called for that feature or scenario. Note that *behave* has additional support specifically for testing *works in progress*.

The tags attached to a feature and scenario are available in the environment functions via the "feature" or "scenario" object passed to them. On those objects there is an attribute called "tags" which is a list of the tag names attached, in the order they're found in the features file.

There are also *environmental controls* specific to tags, so in the above example *behave* will attempt to invoke an `environment.py` function `before_tag` and `after_tag` before and after the Scenario tagged `@slow`, passing in the name "slow". If multiple tags are present then the functions will be called multiple times with each tag in the order they're defined in the feature file.

Re-visiting the example from above; if only some of the features required a browser and web server then you could tag them `@fixture.browser`:

```
# -- FILE: features/environment.py
# HINT: Reusing some code parts from above.
...

def before_feature(context, feature):
    model.init(environment='test')
    if "fixture.browser" in feature.tags:
        use_fixture(wsgi_server, context)
        use_fixture(selenium_browser_chrome, context)
```

1.2.6 Works In Progress

behave supports the concept of a highly-unstable "work in progress" scenario that you're actively developing. This scenario may produce strange logging, or odd output to stdout or just plain interact in unexpected ways with *behave*'s scenario runner.

To make testing such scenarios simpler we've implemented a "-w" command-line flag. This flag:

1. turns off stdout capture
2. turns off logging capture; you will still need to configure your own logging handlers - we recommend a `before_all()` with:

```
if not context.config.log_capture:
    logging.basicConfig(level=logging.DEBUG)
```

3. turns off pretty output - no ANSI escape sequences to confuse your scenario's output
4. only runs scenarios tagged with "@wip"
5. stops at the first error

1.2.7 Fixtures

Fixtures simplify the setup/cleanup tasks that are often needed during test execution.

```
# -- FILE: behave4my_project/fixtures.py  (or in: features/environment.py)
from behave import fixture
from somewhere.browser.firefox import FirefoxBrowser

# -- FIXTURE: Use generator-function
@fixture
def browser_firefox(context, timeout=30, **kwargs):
    # -- SETUP-FIXTURE PART:
    context.browser = FirefoxBrowser(timeout, **kwargs)
    yield context.browser
    # -- CLEANUP-FIXTURE PART:
    context.browser.shutdown()
```

See *Fixtures* for more information.

1.2.8 Debug-on-Error (in Case of Step Failures)

A "debug on error/failure" functionality can easily be provided, by using the `after_step()` hook. The debugger is started when a step fails.

It is in general a good idea to enable this functionality only when needed (in interactive mode). The functionality is enabled (in this example) by using the user-specific configuration data. A user can:

- provide a userdata define on command-line
- store a value in the "behave.userdata" section of behave's configuration file

```
# -- FILE: features/environment.py
# USE: behave -D BEHAVE_DEBUG_ON_ERROR         (to enable  debug-on-error)
# USE: behave -D BEHAVE_DEBUG_ON_ERROR=yes     (to enable  debug-on-error)
# USE: behave -D BEHAVE_DEBUG_ON_ERROR=no      (to disable debug-on-error)

BEHAVE_DEBUG_ON_ERROR = False

def setup_debug_on_error(userdata):
    global BEHAVE_DEBUG_ON_ERROR
    BEHAVE_DEBUG_ON_ERROR = userdata.getbool("BEHAVE_DEBUG_ON_ERROR")

def before_all(context):
    setup_debug_on_error(context.config.userdata)

def after_step(context, step):
    if BEHAVE_DEBUG_ON_ERROR and step.status == "failed":
        # -- ENTER DEBUGGER: Zoom in on failure location.
        # NOTE: Use IPython debugger, same for pdb (basic python debugger).
        import ipdb
        ipdb.post_mortem(step.exc_traceback)
```

1.3 Behavior Driven Development

Behavior-driven development (or BDD) is an agile software development technique that encourages collaboration between developers, QA and non-technical or business participants in a software project. It was originally named in 2003 by Dan North as a response to test-driven development (TDD), including acceptance test or customer test driven development practices as found in extreme programming. It has evolved over the last few years.

On the "Agile specifications, BDD and Testing eXchange" in November 2009 in London, Dan North gave the following definition of BDD:

> BDD is a second-generation, outside–in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.

BDD focuses on obtaining a clear understanding of desired software behavior through discussion with stakeholders. It extends TDD by writing test cases in a natural language that non-programmers can read. Behavior-driven developers use their native language in combination with the ubiquitous language of domain-driven design to describe the purpose and benefit of their code. This allows the developers to focus on why the code should be created, rather than the technical details, and minimizes translation between the technical language in which the code is written and the domain language spoken by the business, users, stakeholders, project management, etc.

1.3.1 BDD practices

The practices of BDD include:

- Establishing the goals of different stakeholders required for a vision to be implemented
- Drawing out features which will achieve those goals using feature injection
- Involving stakeholders in the implementation process through outside–in software development
- Using examples to describe the behavior of the application, or of units of code
- Automating those examples to provide quick feedback and regression testing
- Using 'should' when describing the behavior of software to help clarify responsibility and allow the software's functionality to be questioned
- Using 'ensure' when describing responsibilities of software to differentiate outcomes in the scope of the code in question from side-effects of other elements of code.
- Using mocks to stand-in for collaborating modules of code which have not yet been written

1.3.2 Outside–in

BDD is driven by business value; that is, the benefit to the business which accrues once the application is in production. The only way in which this benefit can be realized is through the user interface(s) to the application, usually (but not always) a GUI.

In the same way, each piece of code, starting with the UI, can be considered a stakeholder of the other modules of code which it uses. Each element of code provides some aspect of behavior which, in collaboration with the other elements, provides the application behavior.

The first piece of production code that BDD developers implement is the UI. Developers can then benefit from quick feedback as to whether the UI looks and behaves appropriately. Through code, and using principles of good design and refactoring, developers discover collaborators of the UI, and of every unit of code thereafter. This helps them adhere to the principle of YAGNI, since each piece of production code is required either by the business, or by another piece of code already written.

1.3.3 The Gherkin language

The requirements of a retail application might be, "Refunded or exchanged items should be returned to stock." In BDD, a developer or QA engineer might clarify the requirements by breaking this down into specific examples. The language of the examples below is called Gherkin and is used by *behave* as well as many other tools.

```
Scenario: Refunded items should be returned to stock
  Given a customer previously bought a black sweater from me
    and I currently have three black sweaters left in stock.
   When he returns the sweater for a refund
   then I should have four black sweaters in stock.,

Scenario: Replaced items should be returned to stock
  Given that a customer buys a blue garment
    and I have two blue garments in stock
    and three black garments in stock.
   When he returns the garment for a replacement in black,
   then I should have three blue garments in stock
    and two black garments in stock.
```

Each scenario is an example, designed to illustrate a specific aspect of behavior of the application.

When discussing the scenarios, participants question whether the outcomes described always result from those events occurring in the given context. This can help to uncover further scenarios which clarify the requirements. For instance, a domain expert noticing that refunded items are not always returned to stock might reword the requirements as "Refunded or replaced items should be returned to stock, unless faulty.".

This in turn helps participants to pin down the scope of requirements, which leads to better estimates of how long those requirements will take to implement.

The words "Given", "When" and "Then" are often used to help drive out the scenarios, but are not mandatory.

These scenarios can also be automated, if an appropriate tool exists to allow automation at the UI level. If no such tool exists then it may be possible to automate at the next level in, i.e.: if an MVC design pattern has been used, the level of the Controller.

1.3.4 Programmer-domain examples and behavior

The same principles of examples, using contexts, events and outcomes are used to drive development at the level of abstraction of the programmer, as opposed to the business level. For instance, the following examples describe different aspects of the list's behavior:

```
Scenario: New lists are empty
  Given a new list
   then the list should be empty.

Scenario: Lists with things in them are not empty.
  Given a new list
   when we add an object
   then the list should not be empty.
```

Both these examples are required to describe the boolean nature of a list in Python and to derive the benefit of the nature. These examples are usually automated using TDD frameworks. In BDD these examples are often encapsulated in a single method, with the name of the method being a complete description of the behavior. Both examples are required for the code to be valuable, and encapsulating them in this way makes it easy to question, remove or change the behavior.

For instance as unit tests, the above examples might become:

```
class TestList:
   def test_empty_list_is_false(self):
       list = []
       assertEqual(bool(list), False)

   def test_populated_list_is_true(self):
       list = []
```

(continues on next page)

(continued from previous page)

```
        list.append('item')
        assertEqual(bool(list), True)
```

Sometimes the difference between the context, events and outcomes is made more explicit. For instance:

```
class TestWindow:
    def test_window_close(self):
        # Given
        window = gui.Window("My Window")
        frame = gui.Frame(window)

        # When
        window.close()

        # Then
        assert_(not frame.isVisible())
```

However the example is phrased, the effect describes the behavior of the code in question. For instance, from the examples above one can derive:

- lists should know when they are empty
- window.close() should cause contents to stop being visible

The description is intended to be useful if the test fails, and to provide documentation of the code's behavior. Once the examples have been written they are then run and the code implemented to make them work in the same way as TDD. The examples then become part of the suite of regression tests.

1.3.5 Using mocks

BDD proponents claim that the use of "should" and "ensureThat" in BDD examples encourages developers to question whether the responsibilities they're assigning to their classes are appropriate, or whether they can be delegated or moved to another class entirely. Practitioners use an object which is simpler than the collaborating code, and provides the same interface but more predictable behavior. This is injected into the code which needs it, and examples of that code's behavior are written using this object instead of the production version.

These objects can either be created by hand, or created using a mocking framework such as mock.

Questioning responsibilities in this way, and using mocks to fulfill the required roles of collaborating classes, encourages the use of Role-based Interfaces. It also helps to keep the classes small and loosely coupled.

1.3.6 Acknowledgement

This text is partially taken from the wikipedia text on Behavior Driven Development with modifications where appropriate to be more specific to *behave* and Python.

1.4 Feature Testing Setup

1.4.1 Feature Testing Layout

behave works with three types of files:

1. *Feature files* written by your Business Analyst / Sponsor / whoever with your behaviour scenarios in it, and
2. a "steps" directory with *Python step implementations* for the scenarios.
3. optionally some *environmental controls* (code to run before and after steps, scenarios, features or the whole shooting match).

These files are typically stored in a directory called "features". The minimum requirement for a features directory is:

```
+--features/
|   +--steps/       # -- Steps directory
|   |    +-- *.py   # -- Step implementation or use step-library python files.
|   +-- *.feature   # -- Feature files.
```

A more complex directory might look like:

```
+-- features/
|     +-- steps/
|     |    +-- website_steps.py
|     |    +-- utils.py
|     |
|     +-- environment.py      # -- Environment file with behave hooks, etc.
|     +-- signup.feature
|     +-- login.feature
|     +-- account_details.feature
```

Layout Variations

behave has some flexibility built in. It will actually try quite hard to find feature specifications. When launched you may pass on the command line:

nothing
: In the absence of any information *behave* will attempt to load your features from a subdirectory called "features" in the directory you launched *behave*.

a features directory path
: This is the path to a features directory laid out as described above. It may be called anything but *must* contain at least one "*name*.feature" file and a directory called "steps". The "environment.py" file, if present, must be in the same directory that contains the "steps" directory (not *in* the "steps" directory).

the path to a "*name*.feature" file
: This tells *behave* where to find the feature file. To find the steps directory *behave* will look in the directory containing the feature file. If it is not present, *behave* will look in the parent directory, and then its parent, and so on until it hits the root of the filesystem. The "environment.py" file, if present, must be in the same directory that contains the "steps" directory (not *in* the "steps" directory).

a directory containing your feature files
: Similar to the approach above, you're identifying the directory where your "*name*.feature" files are, and if the "steps" directory is not in the same place then *behave* will search for it just like above. This allows you to have a layout like:

```
+--tests/
|    +-- steps/
|    |    +-- use_steplib_xyz.py
|    |    +-- website_steps.py
|    |    +-- utils.py
|    +-- environment.py
|    +-- signup.feature
|    +-- login.feature
|    +-- account_details.feature
```

  Note that with this approach, if you want to execute *behave* without having to explicitly specify the directory (first option) you can set the `paths` setting in your *configuration file* (e.g. `paths=tests`).

If you're having trouble setting things up and want to see what *behave* is doing in attempting to find your features use the "-v" (verbose) command-line switch.

1.4.2 Gherkin: Feature Testing Language

behave *features* are written using a language called Gherkin (with *some modifications*) and are named "*name*.feature".

These files should be written using natural language - ideally by the non-technical business participants in the software project. Feature files serve two purposes – documentation and automated tests.

It is very flexible but has a few simple rules that writers need to adhere to.

Line endings terminate statements (eg, steps). Either spaces or tabs may be used for indentation (but spaces are more portable). Indentation is almost always ignored - it's a tool for the feature writer to express some structure in the text. Most lines start with a keyword ("Feature", "Scenario", "Given", ...)

Comment lines are allowed anywhere in the file. They begin with zero or more spaces, followed by a sharp sign (#) and some amount of text.

Features

Features are composed of scenarios. They may optionally have a description, a background and a set of tags. In its simplest form a feature looks like:

```
Feature: feature name

  Scenario: some scenario
    Given some condition
     Then some result is expected.
```

In all its glory it could look like:

```
@tags @tag
Feature: feature name
  description
  further description

  Background: some requirement of this test
    Given some setup condition
      And some other setup action

  Scenario: some scenario
      Given some condition
       When some action is taken
       Then some result is expected.

  Scenario: some other scenario
      Given some other condition
       When some action is taken
       Then some other result is expected.

  Scenario: ...
```

The feature name should just be some reasonably descriptive title for the feature being tested, like "the message posting interface". The following description is optional and serves to clarify any potential confusion or scope issue in the feature name. The description is for the benefit of humans reading the feature text.

The Background part and the Scenarios will be discussed in the following sections.

Background

A background consists of a series of steps similar to *scenarios*. It allows you to add some context to the scenarios of a feature. A background is executed before each scenario of this feature but after any of the before hooks. It is useful for performing setup operations like:

- logging into a web browser or
- setting up a database with test data used by the scenarios.

The background description is for the benefit of humans reading the feature text. Again the background name should just be a reasonably descriptive title for the background operation being performed or requirement being met.

A background section may exist only once within a feature file. In addition, a background must be defined before any scenario or scenario outline.

It contains *steps* as described below.

Good practices for using Background

Don't use "Background" to set up complicated state unless that state is actually something the client needs to know.
: For example, if the user and site names don't matter to the client, you should use a high-level step such as "Given that I am logged in as a site owner".

Keep your "Background" section short.
: You're expecting the user to actually remember this stuff when reading your scenarios. If the background is more than 4 lines long, can you move some of the irrelevant details into high-level steps? See *calling steps from other steps*.

Make your "Background" section vivid.
: You should use colorful names and try to tell a story, because the human brain can keep track of stories much better than it can keep track of names like "User A", "User B", "Site 1", and so on.

Keep your scenarios short, and don't have too many.
: If the background section has scrolled off the screen, you should think about using higher-level steps, or splitting the features file in two.

Scenarios

Scenarios describe the discrete behaviours being tested. They are given a title which should be a reasonably descriptive title for the scenario being tested. The scenario description is for the benefit of humans reading the feature text.

Scenarios are composed of a series of *steps* as described below. The steps typically take the form of "given some condition" "then we expect some test will pass." In this simplest form, a scenario might be:

```
Scenario: we have some stock when we open the store
  Given that the store has just opened
   then we should have items for sale.
```

There may be additional conditions imposed on the scenario, and these would take the form of "when" steps following the initial "given" condition. If necessary, additional "and" or "but" steps may also follow the "given", "when" and "then" steps if more conditions need to be tested. A more complex example of a scenario might be:

```
Scenario: Replaced items should be returned to stock
  Given that a customer buys a blue garment
    and I have two blue garments in stock
    but I have no red garments in stock
    and three black garments in stock.
   When he returns the garment for a replacement in black,
   then I should have three blue garments in stock
    and no red garments in stock,
    and two black garments in stock.
```

It is good practise to have a scenario test only one behaviour or desired outcome.

Scenarios contain *steps* as described below.

Scenario Outlines

These may be used when you have a set of expected conditions and outcomes to go along with your scenario *steps*.

An outline includes keywords in the step definitions which are filled in using values from example tables. You may have a number of example tables in each scenario outline.

```
Scenario Outline: Blenders
   Given I put <thing> in a blender,
    when I switch the blender on
    then it should transform into <other thing>

 Examples: Amphibians
   | thing         | other thing |
   | Red Tree Frog | mush        |

 Examples: Consumer Electronics
   | thing         | other thing |
   | iPhone        | toxic waste |
   | Galaxy Nexus  | toxic waste |
```

behave will run the scenario once for each (non-heading) line appearing in the example data tables.

The values to replace are determined using the name appearing in the angle brackets "<*name*>" which must match a headings of the example tables. The name may include almost any character, though not the close angle bracket ">".

Substitution may also occur in *step data* if the "<*name*>" texts appear within the step data text or table cells.

Steps

Steps take a line each and begin with a *keyword* - one of "given", "when", "then", "and" or "but".

In a formal sense the keywords are all Title Case, though some languages allow all-lowercase keywords where that makes sense.

Steps should not need to contain significant degree of detail about the mechanics of testing; that is, instead of:

```
Given a browser client is used to load the URL "https://website.example/website/home.
↪html"
```

the step could instead simply say:

```
Given we are looking at the home page
```

Steps are implemented using Python code which is implemented in the "steps" directory in Python modules (files with Python code which are named "*name*.py".) The naming of the Python modules does not matter. *All* modules in the "steps" directory will be imported by *behave* at startup to discover the step implementations.

Given, When, Then (And, But)

behave doesn't technically distinguish between the various kinds of steps. However, we strongly recommend that you do! These words have been carefully selected for their purpose, and you should know what the purpose is to get into the BDD mindset.

Given

The purpose of givens is to **put the system in a known state** before the user (or external system) starts interacting with the system (in the When steps). Avoid talking about user interaction in givens. If you had worked with usecases, you would call this preconditions.

Examples:

- Create records (model instances) / set up the database state.
- It's ok to call directly into your application model here.
- Log in a user (An exception to the no-interaction recommendation. Things that "happened earlier" are ok).

You might also use Given with a multiline table argument to set up database records instead of fixtures hard-coded in steps. This way you can read the scenario and make sense out of it without having to look elsewhere (at the fixtures).

When

Each of these steps should **describe the key action** the user (or external system) performs. This is the interaction with your system which should (or perhaps should not) cause some state to change.

Examples:

- Interact with a web page (Requests/Twill/Selenium *interaction* etc should mostly go into When steps).
- Interact with some other user interface element.
- Developing a library? Kicking off some kind of action that has an observable effect somewhere else.

Then

Here we **observe outcomes**. The observations should be related to the business value/benefit in your feature description. The observations should also be on some kind of *output* - that is something that comes *out* of the system (report, user interface, message) and not something that is deeply buried inside it (that has no business value).

Examples:

- Verify that something related to the Given+When is (or is not) in the output
- Check that some external system has received the expected message (was an email with specific content sent?)

While it might be tempting to implement Then steps to just look in the database - resist the temptation. You should only verify outcome that is observable for the user (or external system) and databases usually are not.

And, But

If you have several givens, whens or thens you could write:

```
Scenario: Multiple Givens
  Given one thing
  Given another thing
  Given yet another thing
   When I open my eyes
   Then I see something
   Then I don't see something else
```

Or you can make it read more fluently by writing:

```
Scenario: Multiple Givens
  Given one thing
```

(continues on next page)

(continued from previous page)

```
    And another thing
    And yet another thing
   When I open my eyes
   Then I see something
    But I don't see something else
```

The two scenarios are identical to *behave* - steps beginning with "and" or "but" are exactly the same kind of steps as all the others. They simply mimic the step that precedes them.

Step Data

Steps may have some text or a table of data attached to them.

Substitution of scenario outline values will be done in step data text or table data if the "<*name*>" texts appear within the step data text or table cells.

Text

Any text block following a step wrapped in """ lines will be associated with the step. This is the one case where indentation is actually parsed: the leading whitespace is stripped from the text, and successive lines of the text should have at least the same amount of whitespace as the first line.

So for this rather contrived example:

```
Scenario: some scenario
  Given a sample text loaded into the frobulator
     """
     Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
     eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
     enim ad minim veniam, quis nostrud exercitation ullamco laboris
     nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in
     reprehenderit in voluptate velit esse cillum dolore eu fugiat
     nulla pariatur. Excepteur sint occaecat cupidatat non proident,
     sunt in culpa qui officia deserunt mollit anim id est laborum.
     """
 When we activate the frobulator
 Then we will find it similar to English
```

The text is available to the Python step code as the ".text" attribute in the *Context* variable passed into each step function. The text supplied on the first step in a scenario will be available on the context variable for the duration of that scenario. Any further text present on a subsequent step will overwrite previously-set text.

Table

You may associate a table of data with a step by simply entering it, indented, following the step. This can be useful for loading specific required data into a model.

The table formatting doesn't have to be strictly lined up but it does need to have the same number of columns on each line. A column is anything appearing between two vertical bars "|". Any whitespace between the column content and the vertical bar is removed.

```
Scenario: some scenario
  Given a set of specific users
     | name      | department  |
     | Barry     | Beer Cans   |
     | Pudey     | Silly Walks |
     | Two-Lumps | Silly Walks |
```

(continues on next page)

(continued from previous page)

```
 When we count the number of people in each department
 Then we will find two people in "Silly Walks"
  But we will find one person in "Beer Cans"
```

The table is available to the Python step code as the ".table" attribute in the *Context* variable passed into each step function. The table is an instance of *Table* and for the example above could be accessed like so:

```
@given('a set of specific users')
def step_impl(context):
    for row in context.table:
        model.add_user(name=row['name'], department=row['department'])
```

There's a variety of ways to access the table data - see the *Table* API documentation for the full details.

Tags

You may also "tag" parts of your feature file. At the simplest level this allows *behave* to selectively check parts of your feature set.

You may tag features, scenarios or scenario outlines but nothing else. Any tag that exists in a feature will be inherited by its scenarios and scenario outlines.

Tags appear on the line preceding the feature or scenario you wish to tag. You may have many space-separated tags on a single line.

A tag takes the form of the at symbol "@" followed by a word (which may include underscores "_"). Valid tag lines include:

```
@slow
@wip
@needs_database
```

For example:

```
@wip @slow
Feature: annual reporting
  Some description of a slow reporting system.
```

or:

```
@wip
@slow
Feature: annual reporting
  Some description of a slow reporting system.
```

Tags may be used to *control your test run* by only including certain features or scenarios based on tag selection. The tag information may also be accessed from the *Python code backing up the tests*.

Controlling Your Test Run With Tags

Given a feature file with:

```
Feature: Fight or flight
  In order to increase the ninja survival rate,
  As a ninja commander
  I want my ninjas to decide whether to take on an
  opponent based on their skill levels

@slow
```

(continues on next page)

(continued from previous page)

```
Scenario: Weaker opponent
  Given the ninja has a third level black-belt
  When attacked by a samurai
  Then the ninja should engage the opponent

Scenario: Stronger opponent
  Given the ninja has a third level black-belt
  When attacked by Chuck Norris
  Then the ninja should run for his life
```

Running `behave --tags=slow` will run just the scenarios tagged `@slow`. If you wish to check everything *except* the slow ones then you may run `behave --tags="not slow"`.

Another common use-case is to tag a scenario you're working on with `@wip` and then `behave --tags=wip` to just test that one case.

Tag selection on the command-line may be combined:

–tags="wip or slow"
: This will select all the cases tagged *either* "wip" or "slow".

–tags="wip and slow"
: This will select all the cases tagged *both* "wip" and "slow".

If a feature or scenario is tagged and then skipped because of a command-line control then the *before_* and *after_* environment functions will not be called for that feature or scenario.

Accessing Tag Information In Python

The tags attached to a feature and scenario are available in the environment functions via the "feature" or "scenario" object passed to them. On those objects there is an attribute called "tags" which is a list of the tag names attached, in the order they're found in the features file.

There are also *environmental controls* specific to tags, so in the above example *behave* will attempt to invoke an `environment.py` function `before_tag` and `after_tag` before and after the Scenario tagged `@slow`, passing in the name "slow". If multiple tags are present then the functions will be called multiple times with each tag in the order they're defined in the feature file.

Re-visiting the example from above; if only some of the features required a browser and web server then you could tag them `@fixture.browser` and `@fixture.webserver`:

```
# -- FILE: features/environment.py
from behave.fixture import fixture, use_fixture_by_tag

@fixture
def webserver(context):
    # -- STEP: Setup browser fixture
    context.server = simple_server.WSGIServer(("", 8000))
    context.server.set_app(web_app.main(environment='test'))
    context.thread = threading.Thread(target=context.server.serve_forever)
    context.thread.start()
    yield context.server
    # -- STEP: Teardown/cleanup fixture
    context.server.shutdown()
    context.thread.join()

@fixture
def browser_chrome(context):
    # -- STEP: Setup browser fixture
    context.browser = webdriver.Chrome()
```

(continues on next page)

(continued from previous page)

```
    yield context.browser
    # -- STEP: Teardown/cleanup fixture
    context.browser.quit()


fixture_registry = {
    "fixture.browser":   browser_chrome,
    "fixture.webserver": webserver,
}

# -- BEHAVE HOOKS:
def before_feature(context, feature):
    model.init(environment='test')

def before_tag(context, tag):
    if tag.startswith("fixture."):
        # USE-FIXTURE FOR TAGS: @fixture.browser, @fixture.webserver
        return use_fixture_by_tag(tag, context, fixture_registry):
```

```
# -- FILE: features/use_browser.feature
# HINT: Fixture are automatically setutp and teardown via fixture-tags.
@fixture.webserver
@fixture.browser
Feature: Use Webserver and browser
    ...
```

Languages Other Than English

English is the default language used in parsing feature files. If you wish to use a different language you should check to see whether it is available:

```
behave --lang-list
```

This command lists all the supported languages. If yours is present then you have two options:

1. add a line to the top of the feature files like (for French):

   # – FILE: features/some.feature # language: fr

2. use the command-line switch `--lang`:

   ```
   behave --lang=fr
   ```

The feature file keywords will now use the French translations. To see what the language equivalents recognised by *behave* are, use:

```
behave --lang-help fr
```

Modifications to the Gherkin Standard

behave can parse standard Gherkin files and extends Gherkin to allow lowercase step keywords because these can sometimes allow more readable feature specifications.

1.5 Tag Expressions

1.5.1 Tag-Expressions v2

Tag-Expressions v2 are based on cucumber-tag-expressions with some extensions:

- Tag-Expressions v2 provide *boolean logic expression* (with `and`, `or` and `not` operators and parenthesis for grouping expressions)
- Tag-Expressions v2 are far more readable and composable than Tag-Expressions v1
- Some boolean-logic-expressions where not possible with Tag-Expressions v1
- Therefore, Tag-Expressions v2 supersedes the old-style tag-expressions.

Listing 1: TAG-EXPRESSION EXAMPLES

```
# -- EXAMPLE 1: Select features/scenarios that have the tags: @a and @b
@a and @b

# -- EXAMPLE 2: Select features/scenarios that have the tag: @a or @b
@a or @b

# -- EXAMPLE 3: Select features/scenarios that do not have the tag: @a
not @a

# -- EXAMPLE 4: Select features/scenarios that have the tags: @a but not @b
@a and not @b

# -- EXAMPLE 5: Select features/scenarios that have the tags: (@a or @b) but not @c
# HINT: Boolean expressions can be grouped with parenthesis.
(@a or @b) and not @c
```

COMMAND-LINE EXAMPLE:

Listing 2: USING: Tag-Expressions v2 with behave

```
# -- SELECT-BY-TAG-EXPRESSION (with tag-expressions v2):
# Select all features / scenarios with both "@foo" and "@bar" tags.
$ behave --tags="@foo and @bar" features/

# -- EXAMPLE: Use default_tags from config-file "behave.ini".
# Use placeholder "{config.tags}" to refer to this tag-expression.
# HERE: config.tags = "not (@xfail or @not_implemented)"
$ behave --tags="(@foo or @bar) and {config.tags}" --tags-help
...
CURRENT TAG_EXPRESSION: ((foo or bar) and not (xfail or not_implemented))

# -- EXAMPLE: Uses Tag-Expression diagnostics with --tags-help option
$ behave --tags="(@foo and @bar) or @baz" --tags-help
$ behave --tags="(@foo and @bar) or @baz" --tags-help --verbose
```

See also

- https://docs.cucumber.io/cucumber/api/#tag-expressions
- cucumber-tag-expressions (Python package)

1.5.2 Tag Matching with Tag-Expressions

Tag-Expressions v2 support **partial string/tag matching** with wildcards. This supports tag-expressions:

Tag Matching Idiom	Example 1	Example 2	Description
`tag.starts_with`	`@foo.*`	`foo.*`	Search for tags that start with a `prefix`.
`tag.ends_with`	`@*.one`	`*.one`	Search for tags that end with a `suffix`.
`tag.contains`	`@*foo*`	`*foo*`	Search for tags that contain a `part`.

Listing 3: FILE: features/one.feature

```
Feature: Alice

  @foo.one
  Scenario: Alice.1
    ...

  @foo.two
  Scenario: Alice.2
    ...

  @bar
  Scenario: Alice.3
    ...
```

The following command-line will select all features / scenarios with tags that start with "@foo.":

Listing 4: USAGE EXAMPLE: Run behave with tag-matching expressions

```
$ behave -f plain --tags="@foo.*" features/one.feature
Feature: Alice

  Scenario: Alice.1
    ...

  Scenario: Alice.2
    ...

# -- HINT: Only Alice.1 and Alice.2 are matched (not: Alice.3).
```

> **Note**
>
> - Filename matching wildcards are supported. See `fnmatch` (Unix style filename matching).
> - The tag matching functionality is an extension to cucumber-tag-expressions.

1.5.3 Select the Tag-Expression Version to Use

The tag-expression version, that should be used by behave, can be specified in the behave config-file.

This allows a user to select:

- Tag-Expressions v1 (if needed)
- Tag-Expressions v2 when it is feasible

EXAMPLE:

Listing 5: FILE: behave.ini

```
# SPECIFY WHICH TAG-EXPRESSION-PROTOCOL SHOULD BE USED:
#   SUPPORTED VALUES: v1, v2, auto_detect
#   CURRENT DEFAULT:  auto_detect
[behave]
tag_expression_protocol = v1    # -- Use Tag-Expressions v1.
```

1.5.4 Tag-Expressions v1

Tag-Expressions v1 are becoming deprecated (but are currently still supported). Use **Tag-Expressions v2** instead.

> **Note**
>
> Tag-Expressions v1 support will be dropped in `behave v1.4.0`.

1.6 Using *behave*

The command-line tool *behave* has a bunch of *command-line arguments* and is also configurable using *configuration files*.

Values defined in the configuration files are used as defaults which the command-line arguments may override.

1.6.1 Command-Line Arguments

You may see the same information presented below at any time using `behave -h`.

`-C, --no-color`
: Disable colored mode.

`--color` `COLORED`
: Use colored mode or not (default: auto).

`-d, --dry-run`
: Invokes formatters without executing the steps.

`-D` `NAME=VALUE, --define` `NAME=VALUE`
: Define user-specific data for the config.userdata dictionary. Example: -D foo=bar to store it in config.userdata[“foo”].

-e PATTERN, **--exclude** PATTERN

Don't run feature files matching regular expression PATTERN.

-i PATTERN, **--include** PATTERN

Only run feature files matching regular expression PATTERN.

--no-junit

Don't output JUnit-compatible reports.

--junit

Output JUnit-compatible reports. When junit is enabled, all stdout and stderr will be redirected and dumped to the junit report, regardless of the "–capture" and "–no-capture" options.

--junit-directory PATH

Directory in which to store JUnit reports.

-j NUMBER, **--jobs** NUMBER, **--parallel** NUMBER

Number of concurrent jobs to use (default: 1). Only supported by test runners that support parallel execution.

-f FORMATTER, **--format** FORMATTER

Specify a formatter. If none is specified the default formatter is used. Pass "–format help" to get a list of available formatters.

--steps-catalog

Show a catalog of all available step definitions. SAME AS: "– format=steps.catalog –dry-run –no-summary -q".

--no-skipped

Don't print skipped steps (due to tags).

--show-skipped

Print skipped steps. This is the default behaviour. This switch is used to override a configuration file setting.

--no-snippets

Don't print snippets for unimplemented steps.

--snippets

Print snippets for unimplemented steps. This is the default behaviour. This switch is used to override a configuration file setting.

--no-multiline

Don't print multiline strings and tables under steps.

--multiline

Print multiline strings and tables under steps. This is the default behaviour. This switch is used to override a configuration file setting.

-n NAME_PATTERN, **--name** NAME_PATTERN

Select feature elements (scenarios, ...) to run which match part of the given name (regex pattern). If this option is given more than once, it will match against all the given names.

--capture

Enable capture mode (stdout/stderr/log-output). Any capture output will be printed on a failure/error.

--no-capture

Disable capture mode (stdout/stderr/log-output).

--capture-stdout

Enable capture of stdout.

--no-capture-stdout

Disable capture of stdout.

--capture-stderr
Enable capture of stderr.

--no-capture-stderr
Disable capture of stderr.

--capture-log, --logcapture
Enable capture of logging output.

--no-capture-log, --no-logcapture
Disable capture of logging output.

--capture-hooks
Enable capture of hooks (except: before_all).

--no-capture-hooks
Disable capture of hooks.

--logging-level LOG_LEVEL
Specify a level to capture logging at. The default is INFO - capturing everything.

--logging-format LOG_FORMAT
Specify custom format to print statements. Uses the same format as used by standard logging handlers. The default is "%(levelname)s:%(name)s:%(message)s".

--logging-datefmt LOG_DATE_FORMAT
Specify custom date/time format to print statements. Uses the same format as used by standard logging handlers.

--logging-filter LOG_FILTER
Specify which statements to filter in/out. By default, everything is captured. If the output is too verbose, use this option to filter out needless output. Example: –logging-filter=foo will capture statements issued ONLY to foo or foo.what.ever.sub but not foobar or other logger. Specify multiple loggers with comma: filter=foo,bar,baz. If any logger name is prefixed with a minus, eg filter=-foo, it will be excluded rather than included.

--logging-clear-handlers
Clear existing logging handlers (during capture-log).

--no-logging-clear-handlers
Keep existing logging handlers (during capture-log).

--no-summary
Don't display the summary at the end of the run.

--summary
Display the summary at the end of the run.

-o FILENAME, **--outfile** FILENAME
Write formatter output to output-file (default: stdout).

-q, --quiet
Alias for –no-snippets –no-source.

-r RUNNER_CLASS, **--runner** RUNNER_CLASS
Use own runner class, like: "behave.runner:Runner"

--no-source
Don't print the file and line of the step definition with the steps.

--show-source
Print the file and line of the step definition with the steps. This is the default behaviour. This switch is used to override a configuration file setting.

--stage TEXT

Defines the current test stage. The test stage name is used as name prefix for the environment file and the steps directory (instead of default path names).

--stop

Stop running tests at the first failure.

-t TAG_EXPRESSION, **--tags** TAG_EXPRESSION

Only execute features or scenarios with tags matching TAG_EXPRESSION. Use *--tags-help* option for more information.

-T, --no-timings

Don't print the time taken for each step.

--show-timings

Print the time taken, in seconds, of each step after the step has completed. This is the default behaviour. This switch is used to override a configuration file setting.

-v, --verbose

Show the files and features loaded.

-w, --wip

Only run scenarios tagged with "wip". Additionally: use the "plain" formatter, do not capture stdout or logging output and stop at the first failure.

--lang LANG

Use keywords for a language other than English.

--lang-list

List the languages available for –lang.

--lang-help LANG

List the translations accepted for one language.

--tags-help

Show help for tag expressions.

--version

Show version.

Tag Expression

TAG-EXPRESSIONS selects Features/Rules/Scenarios by using their tags. A TAG-EXPRESSION is a boolean expression that references some tags.

EXAMPLES:

–tags=@smoke –tags="not @xfail" –tags="@smoke or @wip" –tags="@smoke and @wip" –tags="(@slow and not @fixme) or @smoke" –tags="not (@fixme or @xfail)" –tags="@smoke and {config.tags}"

NOTES:

- The tag-prefix "@" is optional.
- An empty tag-expression is "true" (select-anything).
- Use "{config.tags}" placeholder on command-line to use tag-expressions from the config-file (from: "tags" or "default_tags").

TAG-INHERITANCE:

- A Rule inherits the tags of its Feature
- A Scenario inherits the tags of its Feature or Rule.

- A Scenario of a ScenarioOutline/ScenarioTemplate inherit tags from this ScenarioOutline/ScenarioTemplate and its Example table.

1.6.2 Configuration Files

Configuration files for *behave* are called either ".behaverc", "behave.ini", "setup.cfg", "tox.ini", or "pyproject.toml" (your preference) and are located in one of three places:

1. the current working directory (good for per-project settings),
2. your home directory ($HOME), or
3. on Windows, in the %APPDATA% directory.

If you are wondering where *behave* is getting its configuration defaults from you can use the "-v" command-line argument and it'll tell you.

Configuration files **must** start with the label "[behave]" and are formatted in the Windows INI style, for example:

```
[behave]
default_format = plain
default_tags = not (@xfail or @not_implemented)
junit = true
junit_directory = build/behave.reports
logging_level = WARNING
```

Alternatively, if using "pyproject.toml" instead (note the "tool." prefix):

```
[tool.behave]
default_format = "plain"
default_tags = "not (@xfail or @not_implemented)"
junit = true
junit_directory = "build/behave.reports"
logging_level = "WARNING"
```

NOTE: toml does not support *'%'* interpolations.

Configuration File Parameter Types

The following types are supported (and used):

text
: This just assigns whatever text you supply to the configuration setting.

bool
: This assigns a boolean value to the configuration setting. The text describes the functionality when the value is true. True values are "1", "yes", "true", and "on". False values are "0", "no", "false", and "off". TOML: toml only accepts its native *true*

sequence<text>
: These fields accept one or more values on new lines, for example a tag expression might look like:

```
default_tags= (@foo or not @bar) and @zap
```

which is the equivalent of the command-line usage:

```
--tags="(@foo or not @bar) and @zap"
```

TOML: toml can use arrays natively.

Configuration File Parameters

color : Colored (Enum)

Use colored mode or not (default: auto).

dry_run : bool

Invokes formatters without executing the steps.

exclude_re : text

Don't run feature files matching regular expression PATTERN.

include_re : text

Only run feature files matching regular expression PATTERN.

junit : bool

Output JUnit-compatible reports. When junit is enabled, all stdout and stderr will be redirected and dumped to the junit report, regardless of the "–capture" and "–no-capture" options.

junit_directory : text

Directory in which to store JUnit reports.

jobs : positive_number

Number of concurrent jobs to use (default: 1). Only supported by test runners that support parallel execution.

default_format : text

Specify default formatter (default: pretty).

format : sequence<text>

Specify a formatter. If none is specified the default formatter is used. Pass "–format help" to get a list of available formatters.

steps_catalog : bool

Show a catalog of all available step definitions. SAME AS: "– format=steps.catalog –dry-run –no-summary -q".

scenario_outline_annotation_schema : text

Specify name annotation schema for scenario outline (default="{name} – @{row.id} {examples.name}").

use_nested_step_modules : bool

Use subdirectories of steps directory to import steps (default: false).

show_skipped : bool

Print skipped steps. This is the default behaviour. This switch is used to override a configuration file setting.

show_snippets : bool

Print snippets for unimplemented steps. This is the default behaviour. This switch is used to override a configuration file setting.

show_multiline : bool

Print multiline strings and tables under steps. This is the default behaviour. This switch is used to override a configuration file setting.

name : sequence<text>

Select feature elements (scenarios, ...) to run which match part of the given name (regex pattern). If this option is given more than once, it will match against all the given names.

capture : bool

Enable capture mode (stdout/stderr/log-output). Any capture output will be printed on a failure/error.

capture_stdout : bool

Enable capture of stdout.

capture_stderr : bool

Enable capture of stderr.

capture_log : bool

Enable capture of logging output.

capture_hooks : bool

Enable capture of hooks (except: before_all).

logging_level : text

Specify a level to capture logging at. The default is INFO - capturing everything.

logging_format : text

Specify custom format to print statements. Uses the same format as used by standard logging handlers. The default is "%(levelname)s:%(name)s:%(message)s".

logging_datefmt : text

Specify custom date/time format to print statements. Uses the same format as used by standard logging handlers.

logging_filter : text

Specify which statements to filter in/out. By default, everything is captured. If the output is too verbose, use this option to filter out needless output. Example: `logging_filter = foo` will capture statements issued ONLY to "foo" or "foo.what.ever.sub" but not "foobar" or other logger. Specify multiple loggers with comma: `logging_filter = foo,bar,baz`. If any logger name is prefixed with a minus, eg `logging_filter = -foo`, it will be excluded rather than included.

logging_clear_handlers : bool

Clear existing logging handlers (during capture-log).

summary : bool

Display the summary at the end of the run.

outfiles : sequence<text>

Write formatter output to output-file (default: stdout).

paths : sequence<text>

Specify default feature paths, used when none are provided.

tag_expression_protocol : TagExpressionProtocol (Enum)

Specify the tag-expression protocol to use (default: v2). Only tag- expressions v2 are supported (since: behave v1.4.0).

quiet : bool

Alias for –no-snippets –no-source.

runner : text

Use own runner class, like: "behave.runner:Runner"

show_source : bool

Print the file and line of the step definition with the steps. This is the default behaviour. This switch is used to override a configuration file setting.

stage : text

Defines the current test stage. The test stage name is used as name prefix for the environment file and the steps directory (instead of default path names).

stop : bool

Stop running tests at the first failure.

default_tags : text

Use default tags when non are provided. Alternative to *tags : text* (if missing).

tags : text

Select a subset of features/rules/scenarios to execute based on the tag expression. See below for how to code tag expressions in configuration files.

show_timings : bool

Print the time taken, in seconds, of each step after the step has completed. This is the default behaviour. This switch is used to override a configuration file setting.

verbose : bool

Show the files and features loaded.

wip : bool

Only run scenarios tagged with "wip". Additionally: use the "plain" formatter, do not capture stdout or logging output and stop at the first failure.

lang : text

Use keywords for a language other than English.

Additional Configuration File Sections

Section: behave.userdata

This section is used to define user-specific paramters (aka: userdata) for the config.userdata dictionary.

Listing 6: FILE: behave.ini

```
[behave.userdata]
foo = Alice
bar = Bon
```

Alternatively, if using "pyproject.toml":

Listing 7: FILE: pyproject.toml

```
[tool.behave.userdata]
foo = "Alice"
bar = "Bob"
```

which is the equivalent of the command-line usage:

Listing 8: SHELL

```
behave -D foo=Alice -D bar=Bob ...
```

See *Userdata* for usage examples, type conversion and advanced use cases.

Section: behave.formatters

This configuration file section is used to:

- Define aliases for own formatters
- Override the mapping of builtin formatters

Listing 9: FILE: behave.ini

```
[behave.formatters]
allure = allure_behave.formatter:AllureFormatter
html   = behave_html_formatter:HTMLFormatter
html-pretty = behave_html_pretty_formatter:PrettyHTMLFormatter
```

Listing 10: FILE: pyproject.toml

```
[tool.behave.formatters]
allure = "allure_behave.formatter:AllureFormatter"
html   = "behave_html_formatter:HTMLFormatter"
html-pretty = "behave_html_pretty_formatter:PrettyHTMLFormatter"
```

You can then use this formatter alias on the command-line (or in the config-file):

Listing 11: SHELL

```
behave -f html --output=report.html ...
```

See *Formatters and Reporters* for more information.

Section: behave.runners

This configuration file section is used to:

- Define aliases for own test runners
- Override the mapping of builtin test runners

Listing 12: FILE: behave.ini

```
[behave.runners]
mine = behave4me.runner:SuperDuperRunner
```

Listing 13: FILE: pyproject.toml

```
[behave.runners]
mine = "behave4me.runner:SuperDuperRunner"
```

You can then use this runner alias on the command-line:

Listing 14: SHELL

```
behave --runner=mine ...
```

See *Runners* for more information.

1.7 Behave API Reference

This reference is meant for people actually writing step implementations for feature tests. It contains way more information than a typical step implementation will need: most implementations will only need to look at the basic implementation of *step functions* and *maybe environment file functions*.

The model stuff is for people getting really *serious* about their step implementations.

> **Note**
>
> Anywhere this document says "string" it means "unicode string" in Python 2.x
>
> *behave* works exclusively with unicode strings internally.

1.7.1 Step Functions

Step functions are implemented in the Python modules present in your "steps" directory. All Python files (files ending in ".py") in that directory will be imported to find step implementations. They are all loaded before *behave* starts executing your feature tests.

Step functions are identified using step decorators. All step implementations **should normally** start with the import line:

```
from behave import *
```

This line imports several decorators defined by *behave* to allow you to identify your step functions. These are available in both PEP-8 (all lowercase) and traditional (title case) versions: "given", "when", "then" and the generic "step". See the *full list of variables imported* in the above statement.

The decorators all take a single string argument: the string to match against the feature file step text (and they must match *exactly*). So the following step implementation code:

```
@given('some known state')
def step_impl(context):
    setup_something(some, state)
```

will match the "Given" step from the following feature:

```
Scenario: test something
  Given some known state
  Then  some observed outcome.
```

You don't need to import the decorators: they're automatically available to your step implementation modules as *global variables*.

The keywords "Given", "When" and "Then" are chosen for their BDD connotations. Other than matching the different names of the decorators, however, their implementations are identical.

Steps beginning with "and" or "but" in the feature file are renamed to take the name of their preceding keyword, so given the following feature file:

```
Given some known state
  And some other known state
 When some action is taken
 Then some outcome is observed
  But some other outcome is not observed.
```

the first "and" step will be renamed internally to "given" and *behave* will look for a step implementation decorated with either "given" or "step":

```
@given('some other known state')
def step_impl(context):
    setup_something(some, other, state)
```

and similarly the "but" would be renamed internally to "then". Multiple "and" or "but" steps in a row would inherit the non-"and" or "but" keyword.

The function decorated by the step decorator will be passed at least one argument. The first argument is always the `Context` variable. Additional arguments come from *step parameters*, if any.

Step Parameters

You may additionally use *parameters* in your step names. These will be handled by either the default simple parser (parse), its extension "cfparse" or by regular expressions if you invoke *use_step_matcher()*.

behave.**use_step_matcher**(*name*)

Changes the step-matcher class to use while parsing step definitions. This allows to use multiple step-matcher classes:

- in the same steps module
- in different step modules

There are several step-matcher classes available in **behave**:

- **parse** (the default, based on: parse):
- **cfparse** (extends: parse, requires: parse_type)
- **re** (using regular expressions)

Parameters
name – Name of the step-matcher class.

Returns
Current step-matcher class that is now in use.

You may add new types to the default parser by invoking *register_type()*.

behave.**register_type**(***kwargs*)

Registers one (or more) custom type that will be available by some matcher classes, like the `ParseMatcher` and its derived classes, for type conversion during step matching.

Converters should be supplied as `name=callable` arguments (or as dict). A type converter should follow the rules of its `Matcher` class.

You may define a new parameter matcher by subclassing *behave.matchers.Matcher* and registering it with `behave.matchers.matcher_mapping` which is a dictionary of "matcher name" to *Matcher* class.

class behave.matchers.**Matcher**(*func*, *pattern*, *step_type=None*)

Provides an abstract base class for step-matcher classes.

Matches steps from `*.feature` files (Gherkin files) and extracts step-parameters for these steps.

RESPONSIBILITIES:

- Matches steps from `*.feature` files (or not)
- Returns *Match* objects if this step-matcher matches that is used to run the step-definition function w/ its parameters.
- Compile parse-expression/regular-expression to detect BAD STEP-DEFINITION(s) early.

pattern

The match pattern attached to the step function.

func

The associated step-definition function to use for this pattern.

location

File location of the step-definition function.

check_match(*step_text*)

Match me against the supplied "step_text".

Return None, if I don't match otherwise return a list of matches as `Argument` instances.

The return value from this function will be converted into a *Match* instance by *behave*.

Parameters
step_text – Step text that should be matched (as string).

Returns
A list of matched-arguments (on match). None, on mismatch.

Raises
ValueError, re.error, ...

compile()

Compiles the regular-expression pattern (if necessary).

NOTES: - This allows to detect some errors with BAD regular expressions early. - Must be implemneted by derived classes.

Returns
Self (to support daisy-chaining)

describe(*schema=None*)

Provide a textual description of the step function/matcher object.

Parameters
schema – Text schema to use.

Returns
Textual description of this step definition (matcher).

matches(*step_text*)

Checks if `step_text` parameter matches this step-definition (step-matcher).

Parameters
step_text – Step text to check.

Returns
True, if step is matched. False, otherwise.

property regex_pattern

Return the used textual regex pattern.

classmethod register_type(***kwargs*)

Register one (or more) user-defined types used for matching types in step patterns of this matcher.

class behave.model_type.**Argument**(*start*, *end*, *original*, *value*, *name=None*)

An argument found in a *feature file* step name.

The attributes are:

original

The actual text matched in the step name.

value

The potentially type-converted value of the argument.

name

The name of the argument. This will be None if the parameter is anonymous.

start

The start index in the step name of the argument. Used for display.

end

The end index in the step name of the argument. Used for display.

class behave.matchers.**Match**(*func*, *arguments=None*)

An parameter-matched step name extracted from a *feature file*.

func
: The step function that this match will be applied to.

arguments
: A list of `Argument` instances containing the matched parameters from the step name.

Step Macro: Calling Steps From Other Steps

If you find you'd like your step implementation to invoke another step you may do so with the *`Context`* method *`execute_steps()`*.

This function allows you to, for example:

```
@when('I do the same thing as before with the {color:w} button')
def step_impl(context, color):
    context.execute_steps('''
        When I press the big {color} button
         And I duck
    '''.format(color=color))
```

This will cause the "when I do the same thing as before with the red button" step to execute the other two steps as though they had also appeared in the scenario file.

from behave import *

The import statement:

```
from behave import *
```

is written to introduce a restricted set of variables into your code:

Name	Kind	Description
given, when, then, step	Decorator	Decorators for step implementations.
use_step_matcher(name)	Function	Selects current step matcher (parser).
register_type(Type=func)	Function	Registers a type converter.

See also the description in *step parameters*.

1.7.2 Environment File Functions

The environment.py module may define code to run before and after certain events during your testing:

before_all(context), after_all(context)
: These run before and after the whole shooting match.

before_feature(context, feature), after_feature(context, feature)
: These run before and after each feature is executed. The feature object, that is passed in, is an instance of *`Feature`*.

before_rule(context, rule), after_rule(context, rule)
: These run before and after each rule is executed. The rule object, that is passed in, is an instance of *`Rule`*.

before_scenario(context, scenario), after_scenario(context, scenario)
: These run before and after each scenario is run. The scenario object, that is passed in, is an instance of *`Scenario`*.

before_step(context, step), after_step(context, step)
: These run before and after every step. The step object, that is passed in, is an instance of *`Step`*.

before_tag(context, tag), after_tag(context, tag)
: These run before and after a section tagged with the given name. They are invoked for each tag encountered in the order they're found in the feature file. See *Controlling Things With Tags*.

Taggable statements are: Feature, Rule, Scenario, ScenarioOutline, Examples.

The tag, that is passed in, is an instance of *Tag* and because it's a subclass of string you can do simple tests like:

```
# -- ASSUMING: tags @browser.chrome or @browser.any are used.
# BETTER: Use Fixture for this example.
def before_tag(context, tag):
    if tag.startswith("browser."):
        browser_type = tag.replace("browser.", "", 1)
        if browser_type == "chrome":
            context.browser = webdriver.Chrome()
        else:
            context.browser = webdriver.PlainVanilla()
```

Some Useful Environment Ideas

Here's some ideas for things you could use the environment for.

Logging Setup

The following recipe works in all cases (log-capture on or off). If you want to use/configure logging, you should use the following snippet:

```
# -- FILE:features/environment.py
def before_all(context):
    # -- SET LOG LEVEL: behave --logging-level=ERROR ...
    # on behave command-line or in "behave.ini".
    context.config.setup_logging()

    # -- ALTERNATIVE: Setup logging with a configuration file.
    # context.config.setup_logging(configfile="behave_logging.ini")
```

Capture Logging in Hooks

If you wish to capture any logging generated during an environment hook function's invocation, you may use the *capture()* decorator, like:

```
# -- FILE:features/environment.py
from behave.log_capture import capture

@capture
def after_scenario(context):
    ...
```

This will capture any logging done during the call to *after_scenario* and print it out.

Detecting that user code overwrites behave Context attributes

The *context* variable in all cases is an instance of *behave.runner.Context*.

class behave.runner.**Context**(*runner*)

Hold contextual information during the running of tests.

This object is a place to store information related to the tests you're running. You may add arbitrary attributes to it of whatever value you need.

During the running of your tests the object will have additional layers of namespace added and removed automatically. There is a "root" namespace and additional namespaces for features and scenarios.

Certain names are used by *behave*; be wary of using them yourself as *behave* may overwrite the value you set. These names are:

feature

This is set when we start testing a new feature and holds a *Feature*. It will not be present outside of a feature (i.e. within the scope of the environment before_all and after_all).

scenario

This is set when we start testing a new scenario (including the individual scenarios of a scenario outline) and holds a *Scenario*. It will not be present outside of the scope of a scenario.

tags

The current set of active tags (as a Python set containing instances of *Tag* which are basically just glorified strings) combined from the feature and scenario. This attribute will not be present outside of a feature scope.

aborted

This is set to true in the root namespace when the user aborts a test run (KeyboardInterrupt exception). Initially: False.

failed

This is set to true in the root namespace as soon as a step fails. Initially: False.

table

This is set at the step level and holds any *Table* associated with the step.

text

This is set at the step level and holds any multiline text associated with the step.

config

The configuration of *behave* as determined by configuration files and command-line options. The attributes of this object are the same as the configuration file section names.

active_outline

This is set for each scenario in a scenario outline and references the *Row* that is active for the current scenario. It is present mostly for debugging, but may be useful otherwise.

captured

If any output capture is enabled, provides access to a Captured object that contains a snapshot of all captured data (stdout/stderr/log).

Added in version 1.3.0.

A *behave.runner.ContextMaskWarning* warning will be raised if user code attempts to overwrite one of these variables, or if *behave* itself tries to overwrite a user-set variable.

You may use the "in" operator to test whether a certain value has been set on the context, for example:

```
"feature" in context
```

checks whether there is a "feature" value in the context.

Values may be deleted from the context using "del" but only at the level they are set. You can't delete a value set by a feature at a scenario level but you can delete a value set for a scenario in that scenario.

abort(*reason=None*)

Abort the test run.

This sets the *aborted* attribute to true. Any test runner evaluates this attribute to abort a test run.

Added in version 1.2.7.

add_cleanup(*cleanup_func*, **args*, ***kwargs*)

Adds a cleanup function that is called when `Context._pop()` is called. This is intended for user-cleanups.

Parameters

- **cleanup_func** – Callable function
- **args** – Args for cleanup_func() call (optional).
- **kwargs** – Kwargs for cleanup_func() call (optional).

Note

RESERVED `layer` : optional-string

The keyword argument `layer="LAYER_NAME"` can to be used to assign the `cleanup_func` to specific a layer on the context stack (instead of the current layer).

Known layer names are: "testrun", "feature", "rule", "scenario"

See also

`Context.LAYER_NAMES`

attach(*mime_type*, *data*)

Embeds data (e.g. a screenshot) in reports for all formatters that support it, such as the JSON formatter.

Parameters

- **mime_type** – MIME type of the binary data.
- **data** – Bytes-like object to embed.

execute_steps(*steps_text*)

The steps identified in the "steps" text string will be parsed and executed in turn just as though they were defined in a feature file.

If the execute_steps call fails (either through error or failure assertion) then the step invoking it will need to catch the resulting exceptions.

Parameters
steps_text – Text with the Gherkin steps to execute (as string).

Returns
True, if the steps executed successfully.

Raises
AssertionError, if a step failure occurs.

Raises
ValueError, if invoked without a feature context.

use_or_assign_param(*name*, *value*)

Use an existing context parameter (aka: attribute) or assign a value to new context parameter (if it does not exist yet).

Parameters

- **name** – Context parameter name (as string)
- **value** – Parameter value for new parameter.

Returns
Existing or newly created parameter.

Added in version 1.2.7.

use_or_create_param(*name*, *factory_func*, **args*, ***kwargs*)

Use an existing context parameter (aka: attribute) or create a new parameter if it does not exist yet.

Parameters

- **name** – Context parameter name (as string)
- **factory_func** – Factory function, used if parameter is created.
- **args** – Positional args for `factory_func()` on create.
- **kwargs** – Named args for `factory_func()` on create.

Returns

Existing or newly created parameter.

Added in version 1.2.7.

use_with_user_mode()

Provides a context manager for using the context in USER mode.

class behave.runner.**ContextMaskWarning**

Raised if a context variable is being overwritten in some situations.

If the variable was originally set by user code then this will be raised if *behave* overwrites the value.

If the variable was originally set by *behave* then this will be raised if user code overwrites the value.

1.7.3 Fixtures

Provide a Fixture

behave.fixture.**fixture**(*func=None*, *name=None*, *pattern=None*)

Fixture decorator (currently mostly syntactic sugar).

```
# -- FILE: features/environment.py
# CASE FIXTURE-GENERATOR-FUNCTION (like @contextlib.contextmanager):
@fixture
def foo(context, *args, **kwargs):
    the_fixture = setup_fixture_foo(*args, **kwargs)
    context.foo = the_fixture
    yield the_fixture
    cleanup_fixture_foo(the_fixture)

# CASE FIXTURE-FUNCTION: No cleanup or cleanup via context-cleanup.
@fixture(name="fixture.bar")
def bar(context, *args, **kwargs):
    the_fixture = setup_fixture_bar(*args, **kwargs)
    context.bar = the_fixture
    context.add_cleanup(cleanup_fixture_bar, the_fixture.cleanup)
    return the_fixture
```

Parameters

name – Specifies the fixture tag name (as string).

See also

- `contextlib.contextmanager()` decorator
- @pytest.fixture

Use Fixtures

behave.fixture.**use_fixture**(*fixture_func*, *context*, **fixture_args*, ***fixture_kwargs*)

Use fixture (function) and call it to perform its setup-part.

The fixture-function is similar to a `contextlib.contextmanager()` (and contains a yield-statement to separate setup and cleanup part). If it contains a yield-statement, it registers a context-cleanup function to the context object to perform the fixture-cleanup at the end of the current scoped when the context layer is removed (and all context-cleanup functions are called).

Therefore, fixture-cleanup is performed after scenario, feature or test-run (depending when its fixture-setup is performed).

```
# -- FILE: behave4my_project/fixtures.py (or: features/environment.py)
from behave import fixture
from somewhere.browser import FirefoxBrowser

@fixture(name="fixture.browser.firefox")
def browser_firefox(context, *args, **kwargs):
    # -- SETUP-FIXTURE PART:
    context.browser = FirefoxBrowser(*args, **kwargs)
    yield context.browser
    # -- CLEANUP-FIXTURE PART:
    context.browser.shutdown()
```

```
# -- FILE: features/environment.py
from behave import use_fixture
from behave4my_project.fixtures import browser_firefox

def before_tag(context, tag):
    if tag == "fixture.browser.firefox":
        use_fixture(browser_firefox, context, timeout=10)
```

Parameters

- **fixture_func** – Fixture function to use.
- **context** – Context object to use
- **fixture_kwargs** – Positional args, passed to the fixture function.
- **fixture_kwargs** – Additional kwargs, passed to the fixture function.

Returns

Setup result object (may be None).

behave.fixture.**use_fixture_by_tag**(*tag*, *context*, *fixture_registry*)

Process any fixture-tag to perform *use_fixture()* for its fixture. If the fixture-tag is known, the fixture data is retrieved from the fixture registry.

```
# -- FILE: features/environment.py
from behave.fixture import use_fixture_by_tag
from behave4my_project.fixtures import browser_firefox, browser_chrome

# -- SCHEMA 1: fixture_func
fixture_registry1 = {
    "fixture.browser.firefox": browser_firefox,
    "fixture.browser.chrome":  browser_chrome,
}
# -- SCHEMA 2: fixture_func, fixture_args, fixture_kwargs
```

(continues on next page)

(continued from previous page)

```
fixture_registry2 = {
    "fixture.browser.firefox": (browser_firefox, (), dict(timeout=10)),
    "fixture.browser.chrome":  (browser_chrome,  (), dict(timeout=12)),
}

def before_tag(context, tag):
    if tag.startswith("fixture."):
        return use_fixture_by_tag(tag, context, fixture_registry1):
    # -- MORE: Tag processing steps ...
```

Parameters

- **tag** – Fixture tag to process.
- **context** – Runtime context object, used for *use_fixture()*.
- **fixture_registry** – Registry maps fixture-tag to fixture data.

Returns

Fixture-setup result (same as: use_fixture())

Raises

- **LookupError** – If fixture-tag/fixture is unknown.
- **ValueError** – If fixture data type is not supported.

behave.fixture.**use_composite_fixture_with**(*context*, *fixture_funcs_with_params*)

Helper function when complex fixtures should be created and safe-cleanup is needed even if an setup-fixture-error occurs.

This function ensures that fixture-cleanup is performed for every fixture that was setup before the setup-error occurred.

```
# -- BAD-EXAMPLE: Simplistic composite-fixture
# NOTE: Created fixtures (fixture1) are not cleaned up.
@fixture
def foo_and_bad0(context, *args, **kwargs):
    the_fixture1 = setup_fixture_foo(*args, **kwargs)
    the_fixture2 = setup_fixture_bar_with_error("OOPS-HERE")
    yield (the_fixture1, the_fixture2)  # NOT_REACHED.
    # -- NOT_REACHED: Due to fixture2-setup-error.
    the_fixture1.cleanup()  # NOT-CALLED (SAD).
    the_fixture2.cleanup()  # OOPS, the_fixture2 is None (if called).
```

```
# -- GOOD-EXAMPLE: Sane composite-fixture
# NOTE: Fixture foo.cleanup() occurs even after fixture2-setup-error.
@fixture
def foo(context, *args, **kwargs):
    the_fixture = setup_fixture_foo(*args, **kwargs)
    yield the_fixture
    cleanup_fixture_foo(the_fixture)

@fixture
def bad_with_setup_error(context, *args, **kwargs):
    raise RuntimeError("BAD-FIXTURE-SETUP")

# -- SOLUTION 1: With use_fixture()
@fixture
```

(continues on next page)

(continued from previous page)

```
def foo_and_bad1(context, *args, **kwargs):
    the_fixture1 = use_fixture(foo, context, *args, **kwargs)
    the_fixture2 = use_fixture(bad_with_setup_error, context, "OOPS")
    return (the_fixture1, the_fixture2) # NOT_REACHED

# -- SOLUTION 2: With use_composite_fixture_with()
@fixture
def foo_and_bad2(context, *args, **kwargs):
    the_fixture = use_composite_fixture_with(context, [
        fixture_call_params(foo, *args, **kwargs),
        fixture_call_params(bad_with_setup_error, "OOPS")
     ])
    return the_fixture
```

Parameters

- **context** – Runtime context object, used for all fixtures.
- **fixture_funcs_with_params** – List of fixture functions with params.

Returns

List of created fixture objects.

1.7.4 Runner Operation

The execution of code is based on the Gherkin description in *.*feature* files. The following section provides a short overview of the hierarchical containment that is possible in the Gherkin grammar:

```
# -- SIMPLIFIED GHERKIN GRAMMAR (for Gherkin v6):
# CARDINALITY DECORATOR: '*' means 0..N (many), '?' means 0..1 (optional)
# EXAMPLE: Feature
#   A Feature can have many Tags (as TaggableStatement: zero or more tags before its␣
↪keyword).
#   A Feature can have an optional Background.
#   A Feature can have many Scenario(s), meaning zero or more Scenarios.
#   A Feature can have many ScenarioOutline(s).
#   A Feature can have many Rule(s).
Feature(TaggableStatement):
    Background?
    Scenario*
    ScenarioOutline*
    Rule*

Background:
    Step*           # Background steps are injected into any Scenario of its scope.

Scenario(TaggableStatement):
    Step*

ScenarioOutline(ScenarioTemplateWithPlaceholders):
    Scenario*       # Rendered Template by using ScenarioOutline.Examples.rows␣
↪placeholder values.

Rule(TaggableStatement):
    Background?     # Behave-specific extension (after removal from final Gherkin v6).
    Scenario*
    ScenarioOutline*
```

Given all the code that could be run by *behave*, this is the order in which that code is invoked (if they exist.)

```
# -- PSEUDO-CODE:
# HOOK: before_tag(), after_tag() is called for Feature, Rule, Scenario
ctx = createContext()
call-optional-hook before_all(ctx)
for feature in all_features:
    for tag in feature.tags: call-optional-hook before_tag(ctx, tag)
    call-optional-hook before_feature(ctx, feature)
    for run_item in feature.run_items:  # CAN BE: Rule, Scenario, ScenarioOutline
        execute_run_item(run_item, ctx)
    call-optional-hook after_feature(ctx, feature)
    for tag in feature.tags: call-optional-hook after_tag(ctx, tag)
call-optional-hook after_all(ctx)

function execute_run_item(run_item, ctx):
    if run_item isa Rule:
        # -- CASE: Rule
        rule = run_item
        for tag in rule.tags: call-optional-hook before_tag(ctx, tag)
        call-optional-hook before_rule(ctx, rule)
        for run_item in rule.run_items:     # CAN BE: Scenario, ScenarioOutline
            execute_run_item(run_item, ctx)
        call-optional-hook after_rule(ctx, rule)
        for tag in rule.tags: call-optional-hook after_tag(ctx, tag)
    else if run_item isa ScenarioOutline:
        # -- CASE: ScenarioOutline
        # HINT: All Scenarios are already created from Example(s) rows.
        scenario_outline = run_item
        for scenario in scenario_outline.scenarios:
            execute_run_item(scenario, ctx)
    else if run_item isa Scenario:
        # -- CASE: Scenario
        # HINT: Background steps are injected before scenario steps.
        scenario = run_item
        for tag in scenario.tags: call-optional-hook before_tag(ctx, tag)
        call-optional-hook before_scenario(ctx, scenario)
        for step in scenario.steps:
            call-optional-hook before_step(ctx, step)
            step.run(ctx)
            call-optional-hook after_step(ctx, step)
        call-optional-hook after_scenario(ctx, scenario)
        for tag in scenario.tags: call-optional-hook after_tag(ctx, tag)
```

1.7.5 Model Objects

The feature, scenario and step objects represent the information parsed from the feature file. They have a number of common attributes:

keyword
: "Feature", "Scenario", "Given", etc.

name
: The name of the step (the text after the keyword.)

filename and line
: The file name (or "<string>") and line number of the statement.

The structure of model objects parsed from a *feature file* will typically be:

Tag (as Feature.tags)

```
Feature : TaggableModelElement
    Description (as Feature.description)

    Background
        Step
            Table (as Step.table)
            MultiLineText (as Step.text)

    Tag (as Scenario.tags)
    Scenario : TaggableModelElement
        Description (as Scenario.description)
        Step
            Table (as Step.table)
            MultiLineText (as Step.text)

    Tag (as ScenarioOutline.tags)
    ScenarioOutline : TaggableModelElement
        Description (as ScenarioOutline.description)
        Step
            Table (as Step.table)
            MultiLineText (as Step.text)
        Examples
            Table
```

class behave.model.**Feature**(*filename*, *line*, *keyword*, *name*, *tags=None*, *description=None*, *scenarios=None*, *background=None*, *language=None*)

A feature parsed from a *feature file*.

The attributes are:

keyword

This is the keyword as seen in the *feature file*. In English this will be "Feature".

name

The name of the feature (the text after "Feature".)

description

The description of the feature as seen in the *feature file*. This is stored as a list of text lines.

background

The *Background* for this feature, if any.

scenarios

A list of *Scenario* making up this feature.

tags

A list of @tags (as *Tag* which are basically glorified strings) attached to the feature. See *Controlling Things With Tags*.

status

Read-Only. A summary status of the feature's run. If read before the feature is fully tested it will return "untested" otherwise it will return one of:

Status.untested
The feature was has not been completely tested yet.

Status.skipped
One or more steps of this feature was passed over during testing.

Status.passed
The feature was tested successfully.

Status.failed
One or more steps of this feature failed.

Changed in version 1.2.6: Use Status enum class (was: string).

hook_failed

Indicates if a hook failure occurred while running this feature.

Added in version 1.2.6.

duration

The time, in seconds, that it took to test this feature. If read before the feature is tested it will return 0.0.

filename

The file name (or "<string>") of the *feature file* where the feature was found.

line

The line number of the *feature file* where the feature was found.

language

Indicates which spoken language (English, French, German, ..) was used for parsing the feature file and its keywords. The I18N language code indicates which language is used. This corresponds to the language tag at the beginning of the feature file.

Added in version 1.2.6.

class behave.model.**Rule**(*filename*, *line*, *keyword*, *name*, *tags=None*, *description=None*, *scenarios=None*, *background=None*, *parent=None*)

A rule parsed from a *feature file*.

```
# -- FILE: *.feature
Feature: ...
    Description....

    Background?
    Scenario*
    ScenarioOutline*

    @tag1 @tag2
    Rule: Some Rule Title
      Description?      # CARDINALITY: 0..1 (optional).
      Background?       # CARDINALITY: 0..1 (optional).
      Scenario*         # CARDINALITY: 0..N (many)
      ScenarioOutline*  # CARDINALITY: 0..N (many)
```

The attributes are:

keyword

This is the keyword as seen in the *feature file*. In English this will be "Feature".

name

The name of the feature (the text after "Feature".)

description

The description of the feature as seen in the *feature file*. This is stored as a list of text lines.

background

The *Background* for this feature, if any.

scenarios

A list of *Scenario* making up this feature.

`tags`

A list of @tags (as `Tag` which are basically glorified strings) attached to the feature. See *Controlling Things With Tags*.

`status`

Read-Only. A summary status of the feature's run. If read before the feature is fully tested it will return "untested" otherwise it will return one of:

Status.untested
The feature was has not been completely tested yet.

Status.skipped
One or more steps of this feature was passed over during testing.

Status.passed
The feature was tested successfully.

Status.failed
One or more steps of this feature failed.

Changed in version 1.2.6: Use Status enum class (was: string).

`hook_failed`

Indicates if a hook failure occurred while running this feature.

Added in version 1.2.6.

`duration`

The time, in seconds, that it took to test this feature. If read before the feature is tested it will return 0.0.

`filename`

The file name (or "<string>") of the *feature file* where the feature was found.

`line`

The line number of the *feature file* where the feature was found.

`language`

Indicates which spoken language (English, French, German, ..) was used for parsing the feature file and its keywords. The I18N language code indicates which language is used. This corresponds to the language tag at the beginning of the feature file.

Added in version 1.2.7.

`class behave.model.Background(`*filename*, *line*, *keyword='Background'*, *name=''*, *steps=None*, *description=None*`)`

A background parsed from a *feature file*.

Behaviour:

- Each scenario of a scenario container (Feature, Rule) inherits the Background of its scenario container
- Background steps in a scenario are executed before scenario steps
- Rule Background inherits the Feature Background (outer background) if any
- Inherited Background steps are used/executed first
- Optionally, background inheritance can be disabled (normally: by using a fixture/fixture-tag)

The attributes are:

`keyword`

This is the keyword as seen in the *feature file*. In English this will typically be "Background".

`name`

The name of the background (the text after "Background:".)

steps

A list of *Step* making up this background.

duration

The time, in seconds, that it took to run this background. If read before the background is run it will return 0.0.

filename

The file name (or "<string>") of the *feature file* where the background was found.

line

The line number of the *feature file* where the background was found.

description

Optional description (text, as list of lines).

Added in version 1.2.7: (supported since Gherkin v6 or earlier)

class behave.model.**Scenario**(*filename*, *line*, *keyword*, *name*, *tags=None*, *steps=None*, *description=None*, *parent=None*, *background=None*, *background_steps=None*)

A scenario parsed from a *feature file*.

The attributes are:

keyword

This is the keyword as seen in the *feature file*. In English this will typically be "Scenario".

name

The name of the scenario (the text after "Scenario:".)

description

The description of the scenario as seen in the *feature file*. This is stored as a list of text lines.

feature

The *Feature* this scenario belongs to.

steps

A list of *Step* making up this scenario.

tags

A list of @tags (as *Tag* which are basically glorified strings) attached to the scenario. See *Controlling Things With Tags*.

effective_tags

A set of tags that apply to this scenario, including those inherited from parent entities (Feature, Rule, etc.).

This is different from `tags`, which only lists the tags explicitly declared on the scenario itself.

Added in version 1.2.7.

status

Read-Only. A summary status of the scenario's run. If read before the scenario is fully tested it will return "untested" otherwise it will return one of:

Status.untested
The scenario was has not been completely tested yet.

Status.skipped
One or more steps of this scenario was passed over during testing.

Status.passed
The scenario was tested successfully.

Status.failed
One or more steps of this scenario failed.

Changed in version 1.2.6: Use Status enum class (was: string)

`hook_failed`

Indicates if a hook failure occurred while running this scenario.

Added in version 1.2.6.

`duration`

The time, in seconds, that it took to test this scenario. If read before the scenario is tested it will return 0.0.

`filename`

The file name (or "<string>") of the *feature file* where the scenario was found.

`line`

The line number of the *feature file* where the scenario was found.

`parent`

Points to parent entity that contains this scenario (Feature, Rule, ...).

Added in version 1.2.7.

`class behave.model.ScenarioOutline(`*filename*, *line*, *keyword*, *name*, *tags=None*, *steps=None*, *examples=None*, *description=None*`)`

A scenario outline parsed from a *feature file*.

A scenario outline extends the existing `Scenario` class with the addition of the `Examples` tables of data from the *feature file*.

The attributes are:

`keyword`

This is the keyword as seen in the *feature file*. In English this will typically be "Scenario Outline".

`name`

The name of the scenario (the text after "Scenario Outline:".)

`description`

The description of the scenario outline as seen in the *feature file*. This is stored as a list of text lines.

`feature`

The `Feature` this scenario outline belongs to.

`steps`

A list of `Step` making up this scenario outline.

`examples`

A list of `Examples` used by this scenario outline.

`tags`

A list of @tags (as `Tag` which are basically glorified strings) attached to the scenario. See *Controlling Things With Tags*.

`status`

Read-Only. A summary status of the scenario outlines's run. If read before the scenario is fully tested it will return "untested" otherwise it will return one of:

Status.untested
The scenario was has not been completely tested yet.

Status.skipped
One or more scenarios of this outline was passed over during testing.

Status.passed
The scenario was tested successfully.

Status.failed

One or more scenarios of this outline failed.

Changed in version 1.2.6: Use Status enum class (was: string)

duration

The time, in seconds, that it took to test the scenarios of this outline. If read before the scenarios are tested it will return 0.0.

filename

The file name (or "<string>") of the *feature file* where the scenario was found.

line

The line number of the *feature file* where the scenario was found.

class behave.model.**Examples**(*filename*, *line*, *keyword*, *name*, *tags=None*, *table=None*)

A table parsed from a scenario outline in a *feature file*.

The attributes are:

keyword

This is the keyword as seen in the *feature file*. In English this will typically be "Example".

name

The name of the example (the text after "Example:".)

table

An instance of `Table` that came with the example in the *feature file*.

filename

The file name (or "<string>") of the *feature file* where the example was found.

line

The line number of the *feature file* where the example was found.

modified

Marker to indicate if this Examples table data was modified (by a user).

Added in version 1.2.7.

class behave.model.**Tag**(*name*, *line*)

Tags appear may be associated with Features or Scenarios.

They're a subclass of regular strings (unicode pre-Python 3) with an additional `line` number attribute (where the tag was seen in the source feature file.

See *Controlling Things With Tags*.

class behave.model.**Step**(*filename*, *line*, *keyword*, *step_type*, *name*, *text=None*, *table=None*)

A single step parsed from a *feature file*.

The attributes are:

keyword

This is the keyword as seen in the *feature file*. In English this will typically be "Given", "When", "Then" or a number of other words.

name

The name of the step (the text after "Given" etc.)

`step_type`

The type of step as determined by the keyword. If the keyword is "and" then the previous keyword in the *feature file* will determine this step's step_type.

`text`

An instance of *Text* that came with the step in the *feature file*.

`table`

An instance of *Table* that came with the step in the *feature file*.

`status`

Read-Only. A summary status of the step's run. If read before the step is tested it will return "untested" otherwise it will return one of:

Status.untested
This step was not run (yet).

Status.skipped
This step was skipped during testing.

Status.passed
The step was tested successfully.

Status.failed
The step failed.

Status.undefined
The step has no matching step implementation.

Changed in version Use: Status enum class (was: string).

`hook_failed`

Indicates if a hook failure occurred while running this step.

Added in version 1.2.6.

`duration`

The time, in seconds, that it took to test this step. If read before the step is tested it will return 0.0.

`error_message`

If the step failed then this will hold any error information, as a single string. It will otherwise be None.

Changed in version 1.2.6: (moved to base class)

`filename`

The file name (or "<string>") of the *feature file* where the step was found.

`line`

The line number of the *feature file* where the step was found.

Tables may be associated with either Examples or Steps:

`class behave.model.Table(`*headings*, *rows=None*, *line=None*`)`

A table extracted from a *feature file*.

Table instance data is accessible using a number of methods:

iteration
Iterating over the Table will yield the *Row* instances from the .rows attribute.

indexed access
Individual rows may be accessed directly by index on the Table instance; table[0] gives the first non-heading row and table[-1] gives the last row.

The attributes are:

headings

The headings of the table as a list of strings.

rows

An list of instances of *Row* that make up the body of the table in the *feature file*.

modified

Indicates if this table was modified (or not rendered yet).

Added in version 1.2.7.

Tables are also comparable, for what that's worth. Headings and row data are compared.

class behave.model.**Row**(*headings*, *cells*, *line=None*, *comments=None*)

One row of a table parsed from a *feature file*.

Row data is accessible using a number of methods:

iteration
Iterating over the Row will yield the individual cells as strings.

named access
Individual cells may be accessed by heading name; row["name"] would give the cell value for the column with heading "name".

indexed access
Individual cells may be accessed directly by index on the Row instance; row[0] gives the first cell and row[-1] gives the last cell.

The attributes are:

cells

The list of strings that form the cells of this row.

headings

The headings of the table as a list of strings.

Rows are also comparable, for what that's worth. Only the cells are compared.

And Text may be associated with Steps:

class behave.model.**Text**(*value*, *content_type='text/plain'*, *line=0*)

Store multiline text from a Step definition.

The attributes are:

value

The actual text parsed from the *feature file*.

content_type

Currently only "text/plain".

1.7.6 Logging Capture

The logging capture *behave* uses by default is implemented by the class *LoggingCapture*. It has methods

class behave.log_capture.**LoggingCapture**(*config*, *level=None*)

Capture logging events in a memory buffer for later display or query.

Captured logging events are stored on the attribute *buffer*:

buffer

This is a list of captured logging events as logging.LogRecords.

By default the format of the messages will be:

```
'%(levelname)s:%(name)s:%(message)s'
```

This may be overridden using standard logging formatter names in the configuration variable `logging_format`.

The level of logging captured is set to `logging.NOTSET` by default. You may override this using the configuration setting `logging_level` (which is set to a level name.)

Finally there may be filtering of logging events specified by the configuration variable `logging_filter`.

abandon()

Turn off logging capture.

If other handlers were removed by *inveigle()* then they are reinstated.

any_errors()

Search through the buffer for any ERROR or CRITICAL events.

Returns boolean indicating whether a match was found.

find_event(*pattern*)

Search through the buffer for a message that matches the given regular expression.

Returns boolean indicating whether a match was found.

flush()

Override to implement custom flushing behaviour.

This version just zaps the buffer to empty.

inveigle()

Turn on logging capture by replacing all existing handlers configured in the logging module.

If the config var logging_clear_handlers is set then we also remove all existing handlers.

We also set the level of the root logger.

The opposite of this is *abandon()*.

The *log_capture* module also defines a handy logging capture decorator that's intended to be used on your *environment file functions*.

behave.log_capture.**capture**(**args*, ***kw*)

Decorator to wrap an *environment file function* in log file capture.

It configures the logging capture using the *behave* context, the first argument to the function being decorated (so don't use this to decorate something that doesn't have *context* as the first argument).

The basic usage is:

Tip

Use `behave.capture.capture_output()` decorator instead.

Supports capturing of any-output.

It is mostly useful for debugging in situations where you are seeing a message like:

```
No handlers could be found for logger "name"
```

The decorator takes an optional "level" keyword argument which limits the level of logging captured, overriding the level in the run's configuration:

This would limit the logging captured to just ERROR and above, and thus only display logged events if they are interesting.

Parameters

- **level** – Logging level threshold to use.
- **show_on_success** – Use true, to always show captured log records.

Changed in version 1.2.7: Log records are now only shown if an error occurs or `show_on_success` is true.

- Use `capture.show_on_success = True` to enable the old behavior.
- Parameter `show_on_success : bool = CAPTURE_SHOW_ON_SUCCESS` was added.`

1.7.7 Configuration

The configuration object, that is using *Configuration* class, contains the data from the command-line options and the configuration file(s).

class behave.configuration.**Configuration**(*command_args=None*, *load_config=True*, *verbose=None*, ***kwargs*)

Configuration object for behave and behave runners.

tag_expression: behave.tag_expression.v2.TagExpression

Provides the Tag-Expression object based on the `--tags` option(s) on command-line and *:conf-val:`default_tags* parameter in the config-file.

static build_name_re(*names*)

Build regular expression for scenario selection by name by using a list of name parts or name regular expressions.

Parameters
names – List of name parts or regular expressions (as text).

Returns
Compiled regular expression to use.

init(*verbose=None*, ***kwargs*)

(Re-)Init this configuration object.

setup_formats()

Register more, user-defined formatters by name.

setup_logging(*level=None*, *filename=None*, *configfile=None*, ***kwargs*)

Support simple setup of logging subsystem. Ensures that the logging level is set.

> **Note**
> Logging setup should occur only once.

SETUP MODES:

- `logging.config.fileConfig()`, if `configfile` is provided.
- `logging.basicConfig()`, otherwise.

Parameters

- **level** – Logging level of root logger. If None, use `logging_level` value.
- **filename** – Log to file with this filename (optional).
- **configfile** – Configuration filename for fileConfig() setup.
- **kwargs** – Passed to `logging.basicConfig()` or `logging.config.fileConfig()`

Changed in version 1.2.7:

- Parameter "filename" was added to simplify logging to a file.

setup_stage(*stage=None*)

Set up the test stage that selects a different set of steps and environment implementations.

Parameters
stage – Name of current test stage (as string or None).

EXAMPLE:

```
# -- SETUP DEFAULT TEST STAGE (unnamed):
config = Configuration()
config.setup_stage()
assert config.steps_dir == "steps"
assert config.environment_file == "environment.py"

# -- SETUP PRODUCT TEST STAGE:
config.setup_stage("product")
assert config.steps_dir == "product_steps"
assert config.environment_file == "product_environment.py"
```

setup_tag_expression(*tags=None*)

Build the tag_expression object from:

- command-line tags (as tag-expression text)
- config-file tags (as tag-expression text)

show_bad_formats_and_fail(*parser*)

Show any BAD-FORMATTER(s) and fail with `ParseError` if any exists.

update_userdata(*data*)

Update userdata with data and reapply userdata defines (cmdline). :param data: Provides (partial) userdata (as dict)

See also

See chapter *Using behave* for command-line options and configuration file parameter.

1.8 Fixtures

A common task during test execution is to:

- setup a functionality when a test-scope is entered
- cleanup (or teardown) the functionality at the end of the test-scope

Fixtures are provided as concept to simplify this setup/cleanup task in behave.

1.8.1 Providing a Fixture

```
# -- FILE: behave4my_project/fixtures.py  (or in: features/environment.py)
from behave import fixture
from somewhere.browser.firefox import FirefoxBrowser

# -- FIXTURE-VARIANT 1: Use generator-function
@fixture
def browser_firefox(context, timeout=30, **kwargs):
```

(continues on next page)

(continued from previous page)

```
    # -- SETUP-FIXTURE PART:
    context.browser = FirefoxBrowser(timeout, **kwargs)
    yield context.browser
    # -- CLEANUP-FIXTURE PART:
    context.browser.shutdown()
```

```
# -- FIXTURE-VARIANT 2: Use normal function
from somewhere.browser.chrome import ChromeBrowser

@fixture
def browser_chrome(context, timeout=30, **kwargs):
    # -- SETUP-FIXTURE PART: And register as context-cleanup task.
    browser = ChromeBrowser(timeout, **kwargs)
    context.browser = browser
    context.add_cleanup(browser.shutdown)
    return browser
    # -- CLEANUP-FIXTURE PART: browser.shutdown()
    # Fixture-cleanup is called when current context-layer is removed.
```

See also

A *fixture* is similar to:

- a `contextlib.contextmanager()`
- a pytest.fixture
- the scope guard idiom

1.8.2 Using a Fixture

In many cases, the usage of a fixture is triggered by the `fixture-tag` in a feature file. The `fixture-tag` marks that a fixture should be used in this scenario/feature (as test-scope).

```
# -- FILE: features/use_fixture1.feature
Feature: Use Fixture on Scenario Level

    @fixture.browser.firefox
    Scenario: Use Web Browser Firefox
        Given I load web page "https://somewhere.web"
        ...
    # -- AFTER-SCENARIO: Cleanup fixture.browser.firefox
```

```
# -- FILE: features/use_fixture2.feature
@fixture.browser.firefox
Feature: Use Fixture on Feature Level

    Scenario: Use Web Browser Firefox
        Given I load web page "https://somewhere.web"
        ...

    Scenario: Another Browser Test
        ...

# -- AFTER-FEATURE: Cleanup fixture.browser.firefox
```

A **fixture** can be used by calling the `use_fixture()` function. The `use_fixture()` call performs the

SETUP-FIXTURE part and returns the setup result. In addition, it ensures that CLEANUP-FIXTURE part is called later-on when the current context-layer is removed. Therefore, any manual cleanup handling in the `after_tag()` hook is not necessary.

```
# -- FILE: features/environment.py
from behave import use_fixture
from behave4my_project.fixtures import browser_firefox

def before_tag(context, tag):
    if tag == "fixture.browser.firefox":
        use_fixture(browser_firefox, context, timeout=10)
```

Realistic Example

A more realistic example by using a fixture registry is shown below:

```
# -- FILE: features/environment.py
from behave.fixture import use_fixture_by_tag, fixture_call_params
from behave4my_project.fixtures import browser_firefox, browser_chrome

# -- REGISTRY DATA SCHEMA 1: fixture_func
fixture_registry1 = {
    "fixture.browser.firefox": browser_firefox,
    "fixture.browser.chrome":  browser_chrome,
}
# -- REGISTRY DATA SCHEMA 2: (fixture_func, fixture_args, fixture_kwargs)
fixture_registry2 = {
    "fixture.browser.firefox": fixture_call_params(browser_firefox),
    "fixture.browser.chrome":  fixture_call_params(browser_chrome, timeout=12),
}

def before_tag(context, tag):
    if tag.startswith("fixture."):
        return use_fixture_by_tag(tag, context, fixture_registry1):
    # -- MORE: Tag processing steps ...
```

```
# -- FILE: behave/fixture.py
# ...
def use_fixture_by_tag(tag, context, fixture_registry):
    fixture_data = fixture_registry.get(tag, None)
    if fixture_data is None:
        raise LookupError("Unknown fixture-tag: %s" % tag)

    # -- FOR DATA SCHEMA 1:
    fixture_func = fixture_data
    return use_fixture(fixture_func, context)

    # -- FOR DATA SCHEMA 2:
    fixture_func, fixture_args, fixture_kwargs = fixture_data
    return use_fixture(fixture_func, context, *fixture_args, **fixture_kwargs)
```

Hint

Naming Convention for Fixture Tags

Fixture tags should start with `"@fixture.*"` prefix to improve readability and understandibilty in feature files (Gherkin).

Tags are used for different purposes. Therefore, it should be clear when a `fixture-tag` is used.

1.8.3 Fixture Cleanup Points

The point when a fixture-cleanup is performed depends on the scope where `use_fixture()` is called (and the fixture-setup is performed).

Context Layer	Fixture-Setup Point	Fixture-Cleanup Point
test run	In `before_all()` hook	After `after_all()` at end of test-run.
feature	In `before_feature()`	After `after_feature()`, at end of feature.
feature	In `before_tag()`	After `after_feature()` for feature tag.
scenario	In `before_scenario()`	After `after_scenario()`, at end of scenario.
scenario	In `before_tag()`	After `after_scenario()` for scenario tag.
scenario	In a step	After `after_scenario()`. Fixture is usable until end of scenario.

1.8.4 Fixture Setup/Cleanup Semantics

If an error occurs during fixture-setup (meaning an exception is raised):

- Feature/scenario execution is aborted
- Any remaining fixture-setups are skipped
- After feature/scenario hooks are processed
- All fixture-cleanups and context cleanups are performed
- The feature/scenario is marked as failed

If an error occurs during fixture-cleanup (meaning an exception is raised):

- All remaining fixture-cleanups and context cleanups are performed
- First cleanup-error is reraised to pass failure to user (test runner)
- The feature/scenario is marked as failed

1.8.5 Ensure Fixture Cleanups with Fixture Setup Errors

Fixture-setup errors are special because a cleanup of a fixture is in many cases not necessary (or rather difficult because the fixture object is only partly created, etc.). Therefore, if an error occurs during fixture-setup (meaning: an exception is raised), the fixture-cleanup part is normally not called.

If you need to ensure that the fixture-cleanup is performed, you need to provide a slightly different fixture implementation:

```
# -- FILE: behave4my_project/fixtures.py  (or: features/environment.py)
from behave import fixture
from somewhere.browser.firefox import FirefoxBrowser

def setup_fixture_part2_with_error(arg):
    raise RuntimeError("OOPS-FIXTURE-SETUP-ERROR-HERE)

# -- FIXTURE-VARIANT 1: Use generator-function with try/finally.
@fixture
def browser_firefox(context, timeout=30, **kwargs):
    try:
```

(continues on next page)

(continued from previous page)

```
        browser = FirefoxBrowser(timeout, **kwargs)
        browser.part2 = setup_fixture_part2_with_error("OOPS")
        context.browser = browser   # NOT_REACHED
        yield browser
        # -- NORMAL FIXTURE-CLEANUP PART: NOT_REACHED due to setup-error.
    finally:
        browser.shutdown()  # -- CLEANUP: When generator-function is left.
```

```
# -- FIXTURE-VARIANT 2: Use normal function and register cleanup-task early.
from somewhere.browser.chrome import ChromeBrowser

@fixture
def browser_chrome(context, timeout=30, **kwargs):
    browser = ChromeBrowser(timeout, **kwargs)
    context.browser = browser
    context.add_cleanup(browser.shutdown)   # -- ENSURE-CLEANUP EARLY
    browser.part2 = setup_fixture_part2_with_error("OOPS")
    return browser  # NOT_REACHED
    # -- CLEANUP: browser.shutdown() when context-layer is removed.
```

Note

An fixture-setup-error that occurs when the browser object is created, is not covered by these solutions and not so easy to solve.

1.8.6 Composite Fixtures

The last section already describes some problems when you use complex or *composite fixtures*. It must be ensured that cleanup of already created fixture parts is performed even when errors occur late in the creation of a *composite fixture*. This is basically a scope guard problem.

Solution 1:

```
# -- FILE: behave4my_project/fixtures.py
# SOLUTION 1: Use "use_fixture()" to ensure cleanup even in case of errors.
from behave import fixture, use_fixture

@fixture
def foo(context, *args, **kwargs):
    pass    # -- FIXTURE IMPLEMENTATION: Not of interest here.

@fixture
def bar(context, *args, **kwargs):
    pass    # -- FIXTURE IMPLEMENTATION: Not of interest here.

# -- SOLUTION: With use_fixture()
# ENSURES: foo-fixture is cleaned up even when setup-error occurs later.
@fixture
def composite1(context, *args, **kwargs):
    the_fixture1 = use_fixture(foo, context)
    the_fixture2 = use_fixture(bar, context)
    return [the_fixture1, the_fixture2]
```

Solution 2:

```
# -- ALTERNATIVE SOLUTION: With use_composite_fixture_with()
from behave import fixture
from behave.fixture import use_composite_fixture_with, fixture_call_params

@fixture
def composite2(context, *args, **kwargs):
    the_composite = use_composite_fixture_with(context, [
        fixture_call_params(foo, name="foo"),
        fixture_call_params(bar, name="bar"),
    ])
    return the_composite
```

1.9 Userdata

The userdata functionality allows a user to specify their own parameters and values (or more complex configuration data) to the test run.

1.9.1 Overview

Userdata can be provided in three ways:

- As command-line options: `-D name=value` or `--define name=value`
- In the behave configuration file under section `[behave.userdata]`
- Loaded programmatically in the `before_all()` hook

> **Note**
>
> Command-line definitions take precedence over configuration file settings. Therefore, command-line defintions override userdata parameters in the config-file.

1.9.2 Basic Usage

Configuration File

Define userdata in your configuration file:

Listing 15: FILE: behave.ini

```
[behave.userdata]
browser = firefox
server  = asterix
```

Alternatively, if using `pyproject.toml`:

Listing 16: FILE: pyproject.toml

```
[tool.behave.userdata]
browser = "firefox"
server = "asterix"
```

Command Line

Override or provide userdata via command-line:

```
# Provide specific values
behave -D server=obelix features/
behave --define browser=chrome features/

# Boolean flags (value becomes "true")
behave -D debug_mode features/
```

> **Note**
>
> If the command-line contains no value part, like `-D NEEDS_CLEANUP`, its value becomes `"true"` (as boolean value).

Accessing Userdata

Access userdata in your hooks and steps using the `context.config.userdata` dictionary:

Listing 17: FILE: features/environment.py

```
def before_all(context):
    browser = context.config.userdata.get("browser", "chrome")
    setup_browser(browser)
```

Listing 18: FILE: features/steps/userdata_example_steps.py

```
@given('I setup the system with the user-specified server"')
def step_setup_system_with_userdata_server(context):
    server_host = context.config.userdata.get("server", "beatrix")
    context.xxx_client = xxx_protocol.connect(server_host)
```

Other examples for user-specific data are:

- Passing a URL to an external resource that should be used in the tests
- Turning off cleanup mechanisms implemented in environment hooks, for debugging purposes.

1.9.3 Type Converters

The userdata object provides basic type conversion methods, similar to the configparser module:

- `Userdata.getint(name, default=0)`
- `Userdata.getfloat(name, default=0.0)`
- `Userdata.getbool(name, default=False)`
- `Userdata.getas(convert_func, name, default=None, ...)`

> **Note**
>
> Type conversion may raise a `ValueError` exception if the conversion fails.

Example

Listing 19: FILE: features/environment.py

```
def before_all(context):
    userdata = context.config.userdata
    server_name  = userdata.get("server", "beatrix")
    int_number   = userdata.getint("port", 80)
    bool_answer  = userdata.getbool("are_you_sure", True)
    float_number = userdata.getfloat("temperature_threshold", 50.0)
    ...
```

1.9.4 Advanced Use Cases

Loading JSON Configuration

For complex configuration needs, you can load additional data from JSON files:

Listing 20: FILE: features/environment.py

```
import json
import os.path

def before_all(context):
    """Load and update userdata from JSON configuration file."""
    userdata = context.config.userdata
    configfile = userdata.get("configfile", "userconfig.json")

    if os.path.exists(configfile):
        assert configfile.endswith(".json")
        with open(configfile) as f:
            more_userdata = json.load(f)
        context.config.update_userdata(more_userdata)
        # NOTE: Reapplies userdata_defines from command-line too
```

Create a JSON configuration file:

Listing 21: FILE: userconfig.json

```
{
    "browser": "firefox",
    "server":  "asterix",
    "count":   42,
    "cleanup": true
}
```

Then use it:

Listing 22: SHELL

```
behave -D configfile=userconfig.json features/
```

Configuration Profiles

Implement configuration profiles for different environments:

Listing 23: FILE: features/environment.py

```
import json
import os.path

def before_all(context):
    profile = context.config.userdata.get("profile", "default")
    config_file = f"config/{profile}.json"

    if os.path.exists(config_file):
        with open(config_file) as f:
            profile_config = json.load(f)
        context.config.update_userdata(profile_config)
```

Usage:

Listing 24: SHELL

```
# -- EXAMPLE: Use different configuration profiles
behave -D profile=staging features/
behave -D profile=production features/
behave -D profile=local features/
```

1.10 Django Test Integration

There are now at least 2 projects that integrate Django and behave. Both use a LiveServerTestCase to spin up a runserver for the tests automatically, and shut it down when done with the test run. The approach used for integrating Django, though, varies slightly.

behave-django
: Provides a dedicated management command. Easy, automatic integration (thanks to monkey patching). Behave tests are run with `python manage.py behave`. Allows running tests against an existing database as a special feature. See setup behave-django and usage instructions.

django-behave
: Provides a Django-specific TestRunner for Behave, which is set with the TEST_RUNNER property in your settings. Behave tests are run with the usual `python manage.py test <app_name>` by default. See setup django-behave instructions.

1.10.1 Manual Integration

Alternatively, you can integrate Django using the following boilerplate code in your `environment.py` file:

```
# -- FILE: my_django/behave_fixtures.py
from behave import fixture
import django
from django.test.runner import DiscoverRunner
from django.test.testcases import LiveServerTestCase
```

(continues on next page)

(continued from previous page)

```
@fixture
def django_test_runner(context):
    django.setup()
    context.test_runner = DiscoverRunner()
    context.test_runner.setup_test_environment()
    context.old_db_config = context.test_runner.setup_databases()
    yield
    context.test_runner.teardown_databases(context.old_db_config)
    context.test_runner.teardown_test_environment()

@fixture
def django_test_case(context):
    context.test_case = LiveServerTestCase
    context.test_case.setUpClass()
    yield
    context.test_case.tearDownClass()
    del context.test_case
```

```
# -- FILE: features/environment.py
from behave import use_fixture
from my_django.behave_fixtures import django_test_runner, django_test_case
import os

os.environ["DJANGO_SETTINGS_MODULE"] = "test_project.settings"

def before_all(context):
    use_fixture(django_test_runner, context)

def before_scenario(context, scenario):
    use_fixture(django_test_case, context)
```

Taken and adapted from Andrey Zarubin's blog post "BDD. PyCharm + Python & Django".

1.10.2 Strategies and Tooling

See *Practical Tips on Testing* for automation libraries and implementation tips on your BDD tests.

1.11 Flask Test Integration

Integrating your Flask application with behave is done via boilerplate code in your `environment.py` file.

The Flask documentation on testing explains how to use the Werkzeug test client for running tests in general.

1.11.1 Integration Example

The example below is an integration boilerplate derived from the official Flask documentation, featuring the Flaskr sample application from the Flask tutorial.

```
# -- FILE: features/environment.py
import os
import tempfile
from behave import fixture, use_fixture
# flaskr is the sample application we want to test
from flaskr import app, init_db

@fixture
```

(continues on next page)

(continued from previous page)

```
def flaskr_client(context, *args, **kwargs):
    context.db, app.config['DATABASE'] = tempfile.mkstemp()
    app.testing = True
    context.client = app.test_client()
    with app.app_context():
        init_db()
    yield context.client
    # -- CLEANUP:
    os.close(context.db)
    os.unlink(app.config['DATABASE'])

def before_feature(context, feature):
    # -- HINT: Recreate a new flaskr client before each feature is executed.
    use_fixture(flaskr_client, context)
```

Taken and adapted from Ismail Dhorat's BDD testing example on Flaskr.

1.11.2 Strategies and Tooling

See *Practical Tips on Testing* for automation libraries and implementation tips on your BDD tests.

1.12 Practical Tips on Testing

This chapter contains a collection of tips on test strategies and tools, such as test automation libraries, that help you make BDD a successful experience.

1.12.1 Seriously, Don't Test the User Interface

> **Warning**
>
> While you can use behave to drive the "user interface" (UI) or front-end, interacting with the model layer or the business logic, e.g. by using a REST API, is often the better choice.
>
> And keep in mind, BDD advises your to test **WHAT** your application should do and not **HOW** it is done.

If you want to test/exercise also the "user interface", it may be a good idea to reuse the feature files, that test the model layer, by just replacing the test automation layer (meaning mostly the step implementations). This approach ensures that your feature files are technology-agnostic, meaning they are independent how you interact with "system under test" (SUT) or "application under test" (AUT).

For example, if you want to use the feature files in the same directory for testing the model layer and the UI layer, this can be done by using the `--stage` option, like with:

```
$ behave --stage=model features/
$ behave --stage=ui    features/  # NOTE: Normally used on a subset of features.
```

See the *More Information about Behave* chapter for additional hints.

1.12.2 Automation Libraries

With *behave* you can test anything on your application stack: front-end behavior, RESTful APIs, you can even drive your unit tests using Gherkin language. Any library that helps you with that you usually integrate by adding start-up code in `before_all()` and tear-down code in `after_all()`.

The following examples show you how to interact with your Python application by using the web interface (see *Seriously, Don't Test the User Interface* above to learn about entry points for test automation that may be better suited for your use case).

Selenium (Example)

To start a web browser for interaction with the front-end using selenium your `environment.py` may look like this:

```
# -- FILE: features/environment.py
# CONTAINS: Browser fixture setup and teardown
from behave import fixture, use_fixture
from selenium.webdriver import Firefox

@fixture
def browser_firefox(context):
    # -- BEHAVE-FIXTURE: Similar to @contextlib.contextmanager
    context.browser = Firefox()
    yield context.browser
    # -- CLEANUP-FIXTURE PART:
    context.browser.quit()

def before_all(context):
    use_fixture(browser_firefox, context)
    # -- NOTE: CLEANUP-FIXTURE is called after after_all() hook.
```

In your step implementations you can use the `context.browser` object to access Selenium features. See the Selenium docs (`remote.webdriver`) for details. Example using behave-django:

```
# -- FILE: features/steps/browser_steps.py
from behave import given, when, then

@when('I visit "{url}"')
def step_impl(context, url):
    context.browser.get(context.get_url(url))
```

Splinter (Example)

To start a web browser for interaction with the front-end using splinter your `environment.py` may look like this:

```
# -- FILE: features/environment.py
# CONTAINS: Browser fixture setup and teardown
from behave import fixture, use_fixture
from splinter.browser import Browser

@fixture
def splinter_browser(context):
    context.browser = Browser()
    yield context.browser
    context.browser.quit()

def before_all(context):
    use_fixture(splinter_browser, context)
```

In your step implementations you can use the `context.browser` object to access Splinter features. See the Splinter docs for details. Example using *behave-django*:

```
# -- FILE: features/steps/browser_steps.py
from behave import given, when, then

@when('I visit "{url}"')
def step_impl(context, url):
    context.browser.visit(context.get_url(url))
```

Visual Testing

Visually checking your front-end on regression is integrated into *behave* in a straight-forward manner, too. Basically, what you do is drive your application using the front-end automation library of your choice (such as Selenium, Splinter, etc.) to the test location, take a screenshot and compare it with an earlier, approved screenshot (your "baseline").

A list of visual testing tools and services is available from Dave Haeffner's How to Do Visual Testing blog post.

1.13 Comparison With Other Tools

There are other options for doing Gherkin-based BDD in Python. We've listed the main ones below and why we feel you're better off using behave. Obviously this comes from our point of view and you may disagree. That's cool. We're not worried whichever way you go.

This page may be out of date as the projects mentioned will almost certainly change over time. If anything on this page is out of date, please contact us.

1.13.1 Cucumber

You can actually use Cucumber to run test code written in Python. It uses "rubypython" (dead) to fire up a Python interpreter inside the Ruby process though and this can be somewhat brittle. Obviously we prefer to use something written in Python but if you've got an existing workflow based around Cucumber and you have code in multiple languages, Cucumber may be the one for you.

1.13.2 Lettuce

lettuce is similar to behave in that it's a fairly straight port of the basic functionality of Cucumber. The main differences with behave are:

- Single decorator for step definitions, `@step`.
- The context variable, `world`, is simply a shared holder of attributes. It never gets cleaned up during the run.
- Hooks are declared using decorators rather than as simple functions.
- No support for tags.
- Step definition code files can be anywhere in the feature directory hierarchy.

The issues we had with Lettuce that stopped us using it were:

- Lack of tags (which are supported by now, at least since v0.2.20).
- The hooks functionality was patchy. For instance it was very hard to clean up the `world` variable between scenario outlines. Behave clears the scenario-level context between outlines automatically.
- Lettuce's handling of stdout would occasionally cause it to crash mid-run in such a way that cleanup hooks were never run.
- Lettuce uses import hackery so .pyc files are left around and the module namespace is polluted.

1.13.3 *Freshen*

freshen is a plugin for nose that implements a Gherkin-style language with Python step definitions. The main differences with behave are:

- Operates as a plugin for nose, and is thus tied to the nose runner and its output model.
- Has some additions to its Gherkin syntax allowing it to specify specific step definition modules for each feature.
- Has separate context objects for various levels: `glc`, `ftc` and `scc`. These relate to global, feature and scenario levels respectively.

The issues we had with Freshen that stopped us using it were:

- The integration with the nose runner made it quite hard to properly debug how and why tests were failing. Quite often you'd get a rather cryptic message with the actual exception having been swallowed.
- The feature-specific step includes could lead to specific sets of step definitions for each feature despite them warning against doing that.
- The output being handled by nose meant that you couldn't do cucumber-style output without the addition of more plugins.
- The context variable names are cryptic and moving context data from one level to another takes a certain amount of work finding and renaming. The behave *context* variable is much more flexible.
- Step functions must have unique names, even though they're decorated to match different strings.
- As with Lettuce, Freshen uses import hackery so .pyc files are left around and the module namespace is polluted.
- Only Before and no contextual before/after control, thus requiring use of atexit for teardown operations and no fine-grained control.

1.14 New and Noteworthy

In the good tradition of the Eclipse IDE, a number of news, changes and improvements are described here to provide better background information about what has changed and how to make use of it.

This page orders the information by newest version first.

1.14.1 Noteworthy in Version 1.4.0

Remove Support for Tag-Expressions v1

- Tag-Expressions v1 are supersed by *Tag-Expressions v2*.
- Support for Tag-Expressions v1 was removed.
- Use *Tag-Expressions v2* instead.

AFFECTED: By this change

- The `--tags` command-line option and config-file parameter `tags : text` and `default_tags : text` has now the type `text` (was: `sequence<text>`).
- The `--tags` command-line option can now be used only once.

See also

SEE: *Tag Expressions* for details.

1.14.2 Noteworthy in Version 1.3.2

Support for Nested Step Modules

Support for nested step modules inside of the `steps` directory was added in *behave v1.3.0*.

This functionality has caused some problems, if Python package(s) are placed in the `steps` directory that uses relative-import statements. Therefore, the loading of nested step module is disabled by default since this version.

An experienced user can enable this feature by providing:

Listing 25: FILE: behave.ini

```
[behave]
use_nested_step_modules = true
```

See also

- features/runner.use_nested_step_modules.feature
- features/runner.use_substep_dirs.feature (RELATED: older solution)

Hint

BEST PRACTICE:

- Use a step-library instead.
- DO NOT put Python packages in the `steps` directory.

Python packages belong on the Python search path:

- OPTION 0: Install the Python packages in a virtual environment (if ypu use one).
- OPTION 1: Setup the Python search path by using the `PYTHONPATH` environment variable.
- OPTION 2: Setup the Python search path in the `features/environment.py` file.

SEE ALSO:

- features/step.use_step_library.feature

1.14.3 Noteworthy in Version 1.3.0

This version was released as `behave v1.3.0` but is based on the changes from `behave v1.2.7` (which was not released except for pre-releases).

See also

SEE: *Noteworthy in Version 1.2.7* for details.

1.14.4 Noteworthy in Version 1.2.7

Summary:

- Support Gherkin v6 grammar (to simplify usage of Example Mapping)
- Use/Support cucumber-tag-expressions (supersedes: old-style tag-expressions)
- cucumber-tag-expressions are extended by "tag-matching" to match partial tag names, like: `@foo.*`
- *Select-by-location for Scenario Containers* (Feature, Rule, ScenarioOutline)
- *Support for emojis in feature files and steps*
- *Extension Point: Runner*
- *Improve Active-Tags Logic*
- *Active-Tags: Use ValueObject for better Comparisons*
- *Detect bad step definitions*
- *Distinguish between Failures and Errors*
- *Support for Pending Steps*
- *Step definitions with Cucumber-Expressions*
- *Improved Logging Support*
- *Improved Capture Support*

- *Step Decorators: Support for Async-Steps*

BREAKING CHANGES:

- *Gherkin Parser strips no longer trailing colon from step*
- Capture related command line options changed (some in incompatible ways).

Support Gherkin v6 Grammar

Grammar changes:

- `Rule` concept added to better correspond to Example Mapping concepts
- Add aliases for `Scenario` and `Scenario Outline` (for similar reasons)

Listing 26: FEATURE GRAMMAR (PSEUDO-CODE)

```
@tag1 @tag2
Feature: Optional Feature Title...

    Description?        #< CARDINALITY: 0..1 (optional)
    Background?         #< CARDINALITY: 0..1 (optional)
    Scenario*           #< CARDINALITY: 0..N (many)
    ScenarioOutline*    #< CARDINALITY: 0..N (many)
    Rule*               #< CARDINALITY: 0..N (many)
```

A Rule (or: business rule) allows to group multiple Scenario(s)/Example(s):

Listing 27: RULE GRAMMAR (PSEUDO-CODE)

```
@tag1 @tag2
Rule: Optional Rule Title...

    Description?        #< CARDINALITY: 0..1 (optional)
    Background?         #< CARDINALITY: 0..1 (optional)
    Scenario*           #< CARDINALITY: 0..N (many)
    ScenarioOutline*    #< CARDINALITY: 0..N (many)
```

Gherkin v6 keyword aliases:

Concept	Preferred Keyword	Alias(es)
Scenario	Example	Scenario
Scenario Outline	Scenario Outline	Scenario Template
Examples	Examples	Scenarios

EXAMPLE:

Listing 28: FILE: features/example_with_rules.feature

```
# -- USING: Gherkin v6
Feature: With Rules

  Background: Feature.Background
    Given feature background step_1

  Rule: Rule_1
    Background: Rule_1.Background
      Given rule_1 background step_1
```

(continues on next page)

(continued from previous page)

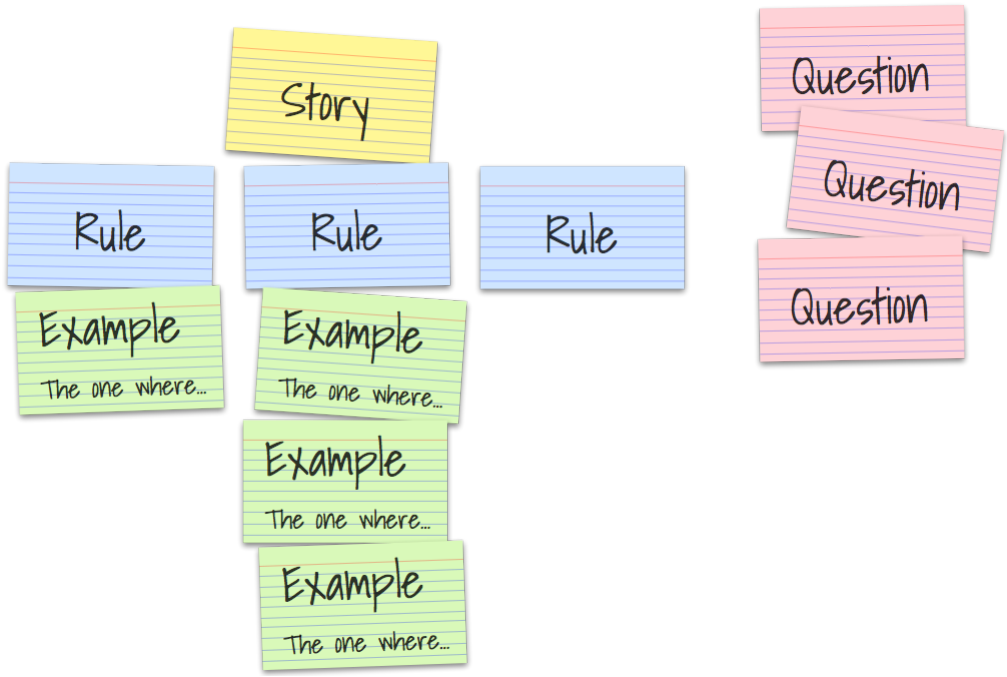
```
  Example: Rule_1.Example_1
    Given rule_1 scenario_1 step_1

Rule: Rule_2

  Example: Rule_2.Example_1
    Given rule_2 scenario_1 step_1

Rule: Rule_3
  Background: Rule_3.EmptyBackground
  Example: Rule_3.Example_1
    Given rule_3 scenario_1 step_1
```

Overview of the Example Mapping concepts:



See also

Gherkin v6:

- https://docs.cucumber.io/gherkin/reference/
- Gherkin v6 grammar

Example Mapping:

- Cucumber: Introduction to Example Mapping (by: Matt Wynne)
- Cucumber: Example Mapping Webinar
- https://docs.cucumber.io/bdd/example-mapping/

More on Example Mapping:

- https://speakerdeck.com/mattwynne/rules-vs-examples-bddx-london-2014
- https://lisacrispin.com/2016/06/02/experiment-example-mapping/
- https://tobythetesterblog.wordpress.com/2016/05/25/how-to-do-example-mapping/

Hint

Gherkin v6 Grammar Issues

- cucumber issue #632: Rule tags are currently only supported in *behave*. The Cucumber Gherkin v6 grammar currently lacks this functionality.
- cucumber issue #590: Rule Background: A proposal is pending to remove Rule Backgrounds again

Tag-Expressions v2

Tag-Expressions v2 are based on cucumber-tag-expressions with some extensions:

- Tag-Expressions v2 provide *boolean logic expression* (with `and`, `or` and `not` operators and parenthesis for grouping expressions)
- Tag-Expressions v2 are far more readable and composable than Tag-Expressions v1
- Some boolean-logic-expressions where not possible with Tag-Expressions v1
- Therefore, Tag-Expressions v2 supersedes the old-style tag-expressions.

Listing 29: TAG-EXPRESSION EXAMPLES

```
# -- EXAMPLE 1: Select features/scenarios that have the tags: @a and @b
@a and @b

# -- EXAMPLE 2: Select features/scenarios that have the tag: @a or @b
@a or @b

# -- EXAMPLE 3: Select features/scenarios that do not have the tag: @a
not @a

# -- EXAMPLE 4: Select features/scenarios that have the tags: @a but not @b
@a and not @b

# -- EXAMPLE 5: Select features/scenarios that have the tags: (@a or @b) but not @c
# HINT: Boolean expressions can be grouped with parenthesis.
(@a or @b) and not @c
```

COMMAND-LINE EXAMPLE:

Listing 30: USING: Tag-Expressions v2 with `behave`

```
# -- SELECT-BY-TAG-EXPRESSION (with tag-expressions v2):
# Select all features / scenarios with both "@foo" and "@bar" tags.
$ behave --tags="@foo and @bar" features/

# -- EXAMPLE: Use default_tags from config-file "behave.ini".
# Use placeholder "{config.tags}" to refer to this tag-expression.
# HERE: config.tags = "not (@xfail or @not_implemented)"
$ behave --tags="(@foo or @bar) and {config.tags}" --tags-help
...
CURRENT TAG_EXPRESSION: ((foo or bar) and not (xfail or not_implemented))

# -- EXAMPLE: Uses Tag-Expression diagnostics with --tags-help option
$ behave --tags="(@foo and @bar) or @baz" --tags-help
$ behave --tags="(@foo and @bar) or @baz" --tags-help --verbose
```

See also

- https://docs.cucumber.io/cucumber/api/#tag-expressions
- cucumber-tag-expressions (Python package)

Tag Matching with Tag-Expressions

Tag-Expressions v2 support **partial string/tag matching** with wildcards. This supports tag-expressions:

Tag Matching Idiom	Example 1	Example 2	Description
`tag.starts_with`	`@foo.*`	`foo.*`	Search for tags that start with a `prefix`.
`tag.ends_with`	`@*.one`	`*.one`	Search for tags that end with a `suffix`.
`tag.contains`	`@*foo*`	`*foo*`	Search for tags that contain a `part`.

Listing 31: FILE: features/one.feature

```
Feature: Alice

  @foo.one
  Scenario: Alice.1
    ...

  @foo.two
  Scenario: Alice.2
    ...

  @bar
  Scenario: Alice.3
    ...
```

The following command-line will select all features / scenarios with tags that start with "@foo.":

Listing 32: USAGE EXAMPLE: Run behave with tag-matching expressions

```
$ behave -f plain --tags="@foo.*" features/one.feature
Feature: Alice

  Scenario: Alice.1
    ...

  Scenario: Alice.2
    ...

# -- HINT: Only Alice.1 and Alice.2 are matched (not: Alice.3).
```

> **Note**
>
> - Filename matching wildcards are supported. See `fnmatch` (Unix style filename matching).
> - The tag matching functionality is an extension to cucumber-tag-expressions.

Select the Tag-Expression Version to Use

The tag-expression version, that should be used by behave, can be specified in the behave config-file.

This allows a user to select:

- Tag-Expressions v1 (if needed)
- Tag-Expressions v2 when it is feasible

EXAMPLE:

Listing 33: FILE: behave.ini

```
# SPECIFY WHICH TAG-EXPRESSION-PROTOCOL SHOULD BE USED:
#   SUPPORTED VALUES: v1, v2, auto_detect
#   CURRENT DEFAULT:  auto_detect
[behave]
tag_expression_protocol = v1    # -- Use Tag-Expressions v1.
```

Tag-Expressions v1

Tag-Expressions v1 are becoming deprecated (but are currently still supported). Use **Tag-Expressions v2** instead.

> **Note**
>
> Tag-Expressions v1 support will be dropped in `behave v1.4.0`.

Select-by-location for Scenario Containers

In the past, it was already possible to scenario(s) by using its **file-location**.

A **file-location** has the schema: `<FILENAME>:<LINE_NUMBER>`. Example: `features/alice.feature:12` (refers to `line 12` in `features/alice.feature` file).

Rules to select **Scenarios** by using the file-location:

- **Scenario:** Use a file-location that points to the keyword/title or its steps (until next Scenario/Entity starts).
- **Scenario of a ScenarioOutline:** Use the file-location of its Examples row.

Now you can select all entities of a **Scenario Container** (Feature, Rule, ScenarioOutline):

- **Feature:** Use file-location before first contained entity/Scenario starts.
- **Rule:** Use file-location from keyword/title line to line before its first Scenario/Background.
- **ScenarioOutline:** Use file-location from keyword/title line to line before its Examples rows.

A file-location into a **Scenario Container** selects all its entities (Scenarios, ...).

Support for Emojis in Feature Files and Steps

- Emojis can now be used in `*.feature` files.
- Emojis can now be used in step definitions.
- You can now use `language=emoji (em)` in `*.feature` files ;-)

Listing 34: FILE: features/i18n_emoji.feature

```
# language: em
# SOURCE: https://github.com/cucumber/cucumber/blob/master/gherkin/testdata/good/i18n_
↪emoji.feature
# HINT:
#   Temporarily disabled on os=win32 (Windows) until unicode encoding issues are␣
↪fixed.
#   Try with environment variable: PYTHONUTF8=1

@not.with_os=win32
:

  :
```

Listing 35: FILE: features/steps/i18n_emoji_steps.py

```
# -- NEEDED-BY: features/i18n_emoji.feature
from behave import given


@given('')
def step_impl(context):
    """Step implementation example with emoji(s)."""
    pass
```

Extension Point: Runner

behave has now an extension point to supply an own test runner. See *Runners* for more details.

Improve Active-Tags Logic

The active-tag computation logic was slightly changed (and fixed):

- if multiple active-tags with same category are used
- combination of positive active-tags (`use.with_{category}={value}`) and negative active-tags (`not.with_{category}={value}`) with same category are now supported

All active-tags with same category are combined into one category tag-group. The following logical expression is used for active-tags with the same category:

Listing 36: ALGORITHM: Active-Tag Expressions

```
category_tag_group.enabled := positive-tag-group-expression and not negative-tag-
↪group-expression
positive-tag-group-expression := enabled(tag1) or enabled(tag2) or ...
negative-tag-group-expression := enabled(tag3) or enabled(tag4) or ...

enabled(tag) := TRUE, if positive/negative active-tag condition is TRUE.
POSITIVE TAGS: tag1, tag2  --  @use.with_{category}={value}
NEGATIVE TAGS: tag3, tag4  --  @not.with_{category}={value}
```

EXAMPLE:

Listing 37: FILE: features/active_tag.example.feature

```
Feature: Active-Tag Example

  @use.with_browser=Safari
  @use.with_browser=Chrome
  @not.with_browser=Firefox
  Scenario: Use one active-tag group/category

    HINT: Only executed with web browser Safari and Chrome, Firefox is explicitly␣
↪excluded.
    ...

  @use.with_browser=Firefox
  @use.with_os=linux
  @use.with_os=darwin
  Scenario: Use two active-tag groups/categories

    HINT 1: Only executed with browser: Firefox
    HINT 2: Only executed on OS: Linux and Darwin (macOS)
    ...
```

Active-Tags: Use ValueObject for better Comparisons

The current mechanism of active-tags only supports the `equals / equal-to` comparison mechanism to determine if the `tag.value` matches the `current.value`, like:

Listing 38: NAME SCHEMA FOR: Active-tags

```
# -- SCHEMA: "@use.with_{category}={value}" or "@not.with_{category}={value}"
@use.with_browser=Safari     # HINT: tag.value = "Safari"
@not.with_browser=Safari     # HINT: tag.value = "Safari"

ACTIVE TAG MATCHES, if:
    current.value == tag.value  (using "@use..." for string values)
    current.value != tag.value  (using "@not..." for string values)
```

The `equals-to` comparison method is sufficient for many situations. But in some situations, you want to use other comparison methods. The `behave.tag_matcher.ValueObject` class was added to allow the user to provide an own comparison method (and type conversion support).

EXAMPLE 1:

Listing 39: FILE: features/active_tags.example1.feature

```
Feature: Active-Tag Example 1 with ValueObject

  @use.with_temperature.min_value=15
  Scenario: Only run if temperature >= 15 degrees Celsius
    ...
```

Listing 40: FILE: features/environment.py

```
import operator
from behave.tag_matcher import ActiveTagMatcher, ValueObject
from my_system.sensors import Sensors

# -- SIMPLIFIED: Better use behave.tag_matcher.NumberValueObject
```

(continues on next page)

(continued from previous page)

```
# CTOR: ValueObject(value, compare=operator.eq)
# HINT: Parameter "value" can be a getter-function (w/o args).
class NumberValueObject(ValueObject):
    def matches(self, tag_value):
        tag_number = int(tag_value)
        return self.compare(self.value, tag_number)

current_temperature = Sensors().get_temperature()
active_tag_value_provider = {
    # -- COMPARISON:
    # temperature.value:     current.value == tag.value  -- DEFAULT: equals  (eq)
    # temperature.min_value: current.value >= tag.value  -- greater_or_equal (ge)
    "temperature.value":     NumberValueObject(current_temperature),
    "temperature.min_value": NumberValueObject(current_temperature, operator.ge),
}
active_tag_matcher = ActiveTagMatcher(active_tag_value_provider)

# -- HOOKS SETUP FOR ACTIVE-TAGS: ... (omitted here)
```

EXAMPLE 2:

A slightly more complex situation arises, if you need to constrain the execution of an scenario to a temperature range, like:

Listing 41: FILE: features/active_tags.example2.feature

```
Feature: Active-Tag Example 2 with Min/Max Value Range

  @use.with_temperature.min_value=10
  @use.with_temperature.max_value=70
  Scenario: Only run if temperature is between 10 and 70 degrees Celsius
    ...
```

Listing 42: FILE: features/environment.py

```
...
current_temperature = Sensors().get_temperature()
active_tag_value_provider = {
    # -- COMPARISON:
    # temperature.min_value:  current.value >= tag.value
    # temperature.max_value:  current.value <= tag.value
    "temperature.min_value": NumberValueObject(current_temperature, operator.ge),
    "temperature.max_value": NumberValueObject(current_temperature, operator.le),
}
...
```

EXAMPLE 3:

Listing 43: FILE: features/active_tags.example3.feature

```
Feature: Active-Tag Example 3 with Contains/Contained-in Comparison

  @use.with_supported_payment_method=VISA
  Scenario: Only run if VISA is one of the supported payment methods
    ...

  # OR: @use.with_supported_payment_methods.contains_value=VISA
```

Listing 44: FILE: features/environment.py

```
# -- NORMALLY:
#  from my_system.payment import get_supported_payment_methods
#  payment_methods = get_supported_payment_methods()
...
payment_methods = ["VISA", "MasterCard", "paycheck"]
active_tag_value_provider = {
    # -- COMPARISON:
    # supported_payment_method: current.value contains tag.value
    "supported_payment_method": ValueObject(payment_methods, operator.contains),
}
...
```

Detect Bad Step Definitions

The **regular expression** (`re`) module in Python has increased the checks when bad regular expression patterns are used. Since *Python >= 3.11*, an `re.error` exception may be raised on some regular expressions. The exception is raised when the bad regular expression is compiled (on `re.compile()`).

`behave` has added the following support:

- Detects a bad step-definition when they are added to the step-registry.
- Reports a bad step-definition and their exception during this step.
- bad step-definitions are not registered in the step-registry.
- A bad step-definition is like an UNDEFINED step-definition.
- A `BadStepsFormatter` formatter was added that shows any BAD STEP DEFINITIONS

Note

More Information on BAD STEP-DEFINITIONS:

- features/formatter.steps_bad.feature
- features/runner.bad_steps.feature

Gherkin Parser strips no longer trailing colon from step

In the past, the Gherkin parser removed a trailing colon (:) on steps that had a text or table section.

EXAMPLE:

Listing 45: FILE: features/parser_example.feature

```
Feature:
  Scenario:
    Given a file named "some_file.txt" with:
      """
      Lorem ipsum, ipsum lorem, ...
      """
```

Listing 46: FILE: features/steps/filesystem_steps.py

```
# -- OLD IMPLEMENTATION:
from behave import given, when, then
from pathlib import Path
```

(continues on next page)

(continued from previous page)

```
@given('a file named "{filename}" with')  #< HINT: Ends without colon
def step_write_file_with_contents(ctx, filename):
    Path(filename).write_text(ctx.text, encoding="UTF-8")
```

The behaviour of the Gherkin parser was changed:

- Trailing colon character is no longer removed on steps with text/table section

REASONS:

- The new behaviour is more natural and much simpler.
- The step writer can define whatever is needed.
- Fixes a problem in PyCharm IDE where the lookup of the step-definition in such a case is not working.

EXAMPLE 2:

Listing 47: FILE: features/steps/filesystem_steps.py

```
# -- NEW IMPLEMENTATION:
from behave import given, when, then
from pathlib import Path

@given('a file named "{filename}" with:')  #< HINT: Ends with colon
def step_write_file_with_contents(ctx, filename):
    Path(filename).write_text(ctx.text, encoding="UTF-8")
```

Hint

The old behaviour of the Gherkin parser can be (re-)enabled by setting the following environment variable before using behave:

Listing 48: On UNIX: Using bash shell

```
export BEHAVE_STRIP_STEPS_WITH_TRAILING_COLON="yes"
```

Listing 49: On WINDOWS: Using cmd shell

```
set BEHAVE_STRIP_STEPS_WITH_TRAILING_COLON="yes"
```

Distinguish between Failures and Errors

behave distinguishes now between failures and errors:

- a **failure** is caused by an assert-mismatch (or: `AssertionError` is raised)
- an **error** is caused normally when an "unexpected" exception is caught.

In addition, an **error** occurs if:

- a hook error occurs
- a cleanup error occurs (and is not ignored)
- a pending step is detected (`behave.api.pending_step.StepNotImplementedError`)
- a undefined step is detected

Support for Pending Steps

behave provides now better support for **pending steps**.

- A pending step has a binding between the step-pattern and its step-function.
- Therefore, a pending step registers itself in the step registry
- But a pending step is not yet implemented (marked-by: `behave.api.pending_step.StepNotImplementedError`)

A **pending step** looks like:

Listing 50: FILE: features/steps/pending_step_example.py

```
from behave import given, when, then
from behave.api.pending_step import StepNotImplementedError

@given('a pending step')
def step_given_a_pending_step(ctx):
    raise StepNotImplementedError("Given a pending step")
```

A pending step causes an error during the test run. But you can mark a scenario temporarily with the *@wip* tag to let any of its pending steps pass:

Listing 51: FILE: features/pending_step.feature

```
Feature: Example

  @wip
  Scenario: With @wip tag and pending step
    Given a step passes
    When a pending step is used
    Then another step passes
```

Listing 52: shell: Run behave tests

```
$ behave -f plain features/pending_step.feature
Feature: Example

  Scenario: With @wip tag and pending step
    Given a step passes ... passed
    When a pending step is used ... pending_warn
    When another step passes ... passed

...
1 scenario passed, 0 failed, 0 skipped
2 steps passed, 0 failed, 0 skipped, 1 pending_warn
```

Without the *@wip* marker, a scenario with pending steps causes an error:

Listing 53: shell: Run behave tests

```
$ behave -f plain features/other_pending_step.feature
Feature: Example 2

  Scenario: Without @wip tag but with pending step
    Given a step passes ... passed
    When a pending step is used ... pending

...
```

(continues on next page)

(continued from previous page)

```
0 scenarios passed, 0 failed, 1 error, 0 skipped
1 step passed, 0 failed, 1 skipped, 1 pending
```

> **Note**
>
> More Information on **pending steps** and **undefined steps**:
>
> - features/step.pending_steps.feature
> - features/step.undefined_steps.feature

Step Definitions with Cucumber-Expressions

Support for a step definitions with cucumber-expressions was added to behave by providing the `behave.cucumber_expression` module.

Cucumber-expressions:

- Provide a simple syntax for step-parameters (aka: placeholders) compared to regular-expressions
- Provide a simple syntax for optional or alternative (unmatched) text parts.
- Provide support for parameter types and type conversions
- Provide a number of predefined parameter types, like: `{int}`, `{word}`, `{string}`, ...
- Similar to `parse-expressions` that are normally used in behave (hint: `parse-expressions` was one of the descendants that lead to the development of cucumber-expressions)

EXAMPLE 1:

Use the `use_step_matcher_for_cucumber_expressions()` function to enable this step-matcher before any step definitions with cucumber-expressions are used. It is possible to do this:

- in the `features/environment.py` file (as default step-matcher)
- in each `features/steps/*.py` steps file

Listing 54: FILE: features/environment.py

```
from behave.cucumber_expression import use_step_matcher_for_cucumber_expressions

# -- HINT: Use StepMatcher4CucumberExpressions as default step-matcher.
use_step_matcher_for_cucumber_expressions()
```

In this example, we want to use the `Color` enum as `parameter_type` (placeholder) in the steps definitions:

Listing 55: FILE: example4me/color.py

```
from enum import Enum

class Color(Enum):
    red = 1
    green = 2
    blue = 3

    @classmethod
    def from_name(cls, text: str):
        text = text.lower()
        for enum_item in iter(cls):
            if enum_item.name == text:
                return enum_item
```

(continues on next page)

(continued from previous page)

```
        # -- OOPS:
        raise ValueError("UNEXPECTED: {}".format(text))
```

We provide the necessary steps with the additional `parameter_type=color` by using the `Color.from_name()` function as type converter/transformer.

Listing 56: FILE: features/steps/color_steps.py

```
# -- REQUIRES: Python3
from behave import when, then
from behave.cucumber_expression import (
    ParameterType,
    define_parameter_type,
    # -- SIMILAR-TO: define_parameter_type_with
)
from example4me.color import Color

# -- REGISTER PARAMETER TYPES:
# OR: Use define_parameter_type_with(name="color", ...)
define_parameter_type(ParameterType(
    name="color",
    regexp="red|green|blue",
    type=Color,
    transformer=Color.from_name
))

...
```

After the `parameter_type=color` is defined, we can use it as `{color}` placeholder in the step definitions:

Listing 57: FILE: features/steps/color_steps.py (continued)

```
# -- STEP DEFINITIONS:
@when('I select the "{color}" theme colo(u)r')
def step_when_select_color_theme(ctx, color: Color):
    assert isinstance(color, Color)
    ctx.selected_color = color

@then('the profile colo(u)r should be "{color}"')
def step_then_profile_color_should_be(ctx, the_color: Color):
    assert isinstance(the_color, Color)
    assert ctx.selected_color == the_color
```

Listing 58: FILE: features/cucumber_expression.feature

```
Feature: Use CucumberExpressions in Step Definitions
    Scenario: User selects a color twice
      Given I am on the profile settings page
      When I select the "red" theme colour
      But  I select the "blue" theme color
      Then the profile color should be "blue"
```

EXAMPLE 2: Use TypeBuilder.make_enum()

The solution in "EXAMPLE 1" can be simplified by using the `TypeBuilder` class. It provides a `TypeBuilder.make_enum()` that generates a `parse-function` for an `Enum` class or a dict-mapping. This `parse-function` provides a type converter/transformer and its regular-expression pattern (as attribute), like:

Listing 59: FILE: features/steps/color_steps.py (ALTERNATIVE)

```
# -- USING: TypeBuilder.make_enum()
from behave import given, when, then
from behave.cucumber_expression import (
    TypeBuilder,
    define_parameter_type_with
)
from example4me.color import Color

parse_color = TypeBuilder.make_enum(Color)

# -- REGISTER PARAMETER TYPES:
define_parameter_type_with(
    name="color",
    regexp=parse_color.pattern,
    type=Color,
    transformer=parse_color
)

...
```

MORE:

In addition, the `TypeBuilder` class provides support to compose `parse-functions` (aka: type converters) and regular-expression patterns from other `parse-functions` or data, like:

- `TypeBuilder.make_enum()`: Builds a `parse-function` and regex-pattern for an `Enum` class or a key/value mapping (aka: `dict`).
- `TypeBuilder.make_choice()`: Builds a `parse-function` and regex-pattern for a list of string values.
- `TypeBuilder.make_variant()`: Builds a `parse-function` and regex-pattern from a list of `parse-functions` (and their patterns) as alternative types.
- `TypeBuilder.with_many()`: Builds a `parse-function` and regex-pattern for **many items** based on the `parse-function` of one item

See also

cucumber-expressions

- Repository: https://github.com/cucumber/cucumber-expressions
- features/step_matcher.cucumber_expressions.feature

See also

parse-expressions

- parse
- parse-type (and `TypeBuilder` core functionality)

Note

A `parameter_type` can only be defined once (maybe: Use the `environment-file` or ...).

Improved Logging Support

It is now simpler to set up the logging to a file in *behave*:

Listing 60: FILE: features/environment.py

```
def before_all(ctx):
    log_format = "LOGFILE.{levelname} -- {name}: {message}"
    ctx.config.setup_logging(filename="behave.log", format=log_format)

# -- NOTE: Setup with logging configuration file was needed before.
```

Tip

behave supports now the newer, additional format styles for log record formats:

- f-string format style, like: `{message}`
- shell placeholder format style, like: `${message}`

Only the percent-string placeholder style was supported before (like: `%(message)s`).

See also

- features/logging.to_file.feature
- features/logging.setup_with_configfile.feature
- python docs: howto/logging-cookbook.html#use-of-alternative-formatting-styles

Improved Capture Support

The capture of hooks is now supported (special case: `before_all()` hook). To better support this, the formatter(s) are now called before the `before_feature/before_scenario/before_tag` hooks are called. This ensures that the Feature/Scenario name is shown (as context) before the any captured output of `before_feature/before_scenario/before_tag` hooks is printed.

AFFECTED FORMATTERS:

- pretty
- plain

See also

- features/capture.on_hooks.feature
- features/runner.hook_errors.feature
- features/runner.context_cleanup.feature

CHANGES (partly incompatible):

The name of capture related command line options have been changed slightly:

Option Name	Old Option Name	Description
`--capture`	—	NEW: Enable/disable capture mode for stdout/stderr/log.
`--capture-hooks`	—	NEW: Enable/disable capture of hooks.
`--capture-stdout`	`--capture`	Enable/disable capture of stdout.
`--capture-stderr`	`--capture-stderr`	Enable/disable capture of stderr.
`--capture-log`	`--logcapture`	Enable/disable capture of log-output.

The *`Configuration`* class attribute names were adapted accordingly to better correspond to the command line options:

Attribute Name	Old Attribute Name	Description
`capture`	—	NEW: Enable/disable capture mode for stdout/stderr/log.
`capture_hooks`	—	NEW: Enable/disable capture of hooks.
`capture_stdout`	`stdout_capture`	Enable/disable capture mode for stdout.
`capture_stderr`	`stderr_capture`	Enable/disable capture mode for stderr.
`capture_log`	`log_capture`	Enable/disable capture mode for log output.

The `CaptureController` class attribute names were renamed accordingly to better correspond to the naming scheme:

Attribute Name	Old Attribute Name	Description
`capture_stdout`	`stdout_capture`	Used to capture stdout output.
`capture_stderr`	`stderr_capture`	Used to capture stderr output.
`capture_log`	`log_capture`	Used to capture log output.

Note

A deprecating warning will be emitted if you use the old names.

Step Decorators: Support for Async-Steps

To simplify usage, the normal step decorators directly support async-steps now, like:

Listing 61: FILE: features/steps/async_steps.py

```
# -- NOW:
import asyncio
from behave import given, when, then, step

@step('a coroutine step waits "{duration:f}" seconds')
async def step_coroutine_waits_seconds(ctx, duration: float):
```

(continues on next page)

(continued from previous page)

```
    await asyncio.sleep(duration)

@when('I execute the long running command', timeout=20.0)
async def step_execute_long_running_command(ctx):
    # -- HINT: Step fails if step duration exceeds 20 seconds.
    pass  # ...
```

Listing 62: FILE: features/steps/async_steps_before.py

```
# -- BEFORE:
import asyncio
from behave import step
from behave.api.async_step import async_run_until_complete

@step('a coroutine step waits "{duration:f}" seconds')
@async_run_until_complete
async def step_async_step_waits_seconds(ctx, duration: float):
    await asyncio.sleep(duration)

@when('I execute the long running command')
@async_run_until_complete(timeout=20.0)
async def step_execute_long_running_command(ctx):
    # -- HINT: Fails if step duration exceeds 20 seconds.
    pass  # ...
```

DEPRECATING `@async_run_until_complete` decorator

- BETTER: Use nornmal step decorators instead.
- The support for `@async_run_until_complete` decorator will be removed in behave v1.4.0.

See also

- features/step.async_steps.feature

Changes

`ModelRunner`:

- Simplify signature on method `run_hook(context, *args)` to `run_hook(*args)`

NEW `behave.model_type` module:

- Moved generic model classes here from `behave.model_core` module, like: `behave.model_core.Status`, `behave.model_core.FileLocation`.

1.14.5 Noteworthy in Version 1.2.6

Summary:

- Tagged Examples: Examples in a ScenarioOutline can now have tags.
- Feature model elements have now language attribute based on language tag in feature file (or the default language tag that was used by the parser).
- Gherkin parser: Supports escaped-pipe in Gherkin table cell value
- Configuration: Supports now to define default tags in configfile

- Configuration: language data is now used as default-language that should be used by the Gherkin parser. Language tags in the Feature file override this setting.
- Runner: Can continue after a failed step in a scenario
- Runner: Hooks processing handles now exceptions. Hook errors (exception in hook processing) lead now to scenario failures (even if no step fails).
- Testing support for asynchronuous frameworks or protocols (`asyncio` based)
- Context-cleanups: Register cleanup functions that are executed at the end of the test-scope (scenario, feature or test-run) via *`add_cleanup()`*.
- *Fixtures*: Simplify setup/cleanup in scenario, feature or test-run

Scenario Outline Improvements

Tagged Examples

Since
behave 1.2.6.dev0

The Gherkin parser (and the model) supports now to use tags with the `Examples` section in a `Scenario Outline`. This functionality can be used to provide multiple `Examples` sections, for example one section per testing stage (development, integration testing, system testing, ...) or one section per test team.

The following feature file provides a simple example of this functionality:

Listing 63: FILE: features/tagged_examples.feature

```
Feature:
  Scenario Outline: Wow
    Given an employee "<name>"

    @develop
    Examples: Araxas
      | name  | birthyear |
      | Alice |  1985     |
      | Bob   |  1975     |

    @integration
    Examples:
      | name   | birthyear |
      | Charly |  1995     |
```

Note

The generated scenarios from a ScenarioOutline inherit the tags from the ScenarioOutline and its Examples section:

```
# -- FOR scenario in scenario_outline.scenarios:
scenario.tags = scenario_outline.tags + examples.tags
```

To run only the first `Examples` section, you use:

Listing 64: shell

```
behave --tags=@develop features/tagged_examples.feature
```

Listing 65: Command output

```
Scenario Outline: Wow -- @1.1 Araxas  # features/tagged_examples.feature:7
  Given an employee "Alice"

Scenario Outline: Wow -- @1.2 Araxas  # features/tagged_examples.feature:8
  Given an employee "Bob"
```

Tagged Examples with Active Tags and Userdata

An even more natural fit is to use `tagged examples` together with `active tags` and `userdata`:

Listing 66: FILE: features/tagged_examples2.feature

```
# -- VARIANT 2: With active tags and userdata.
Feature:
  Scenario Outline: Wow
    Given an employee "<name>"

    @use.with_stage=develop
    Examples: Araxas
      | name  | birthyear |
      | Alice |  1985     |
      | Bob   |  1975     |

    @use.with_stage=integration
    Examples:
      | name   | birthyear |
      | Charly |  1995     |
```

Select the `Examples` section now by using:

Listing 67: shell

```
# -- VARIANT 1: Use userdata
behave -D stage=integration features/tagged_examples2.feature

# -- VARIANT 2: Use stage mechanism
behave --stage=integration features/tagged_examples2.feature
```

Listing 68: FILE: features/environment.py

```
from behave.tag_matcher import ActiveTagMatcher, setup_active_tag_values
import sys

# -- ACTIVE TAG SUPPORT: @use.with_{category}={value}, ...
active_tag_value_provider = {
    "stage":   "develop",
}
active_tag_matcher = ActiveTagMatcher(active_tag_value_provider)

# -- BEHAVE HOOKS:
def before_all(context):
    userdata = context.config.userdata
    stage = context.config.stage or userdata.get("stage", "develop")
    userdata["stage"] = stage
    setup_active_tag_values(active_tag_value_provider, userdata)
```

(continues on next page)

(continued from previous page)

```
def before_scenario(context, scenario):
    if active_tag_matcher.should_exclude_with(scenario.tags):
        sys.stdout.write("ACTIVE-TAG DISABLED: Scenario %s\n" % scenario.name)
        scenario.skip(active_tag_matcher.exclude_reason)
```

Tip

Since behave v1.2.7, better use:

```
def before_scenario(context, scenario):
    # -- ALTERNATIVE: if active_tag_matcher.should_skip(scenario):
    if active_tag_matcher.should_skip_with_tags(scenario.tags):
        scenario.skip(active_tag_matcher.skip_reason)
```

Gherkin Parser Improvements

Escaped-Pipe Support in Tables

It is now possible to use the "|" (pipe) symbol in Gherkin tables by escaping it. The pipe symbol is normally used as column separator in tables.

EXAMPLE:

Listing 69: FILE: features/escaped_pipe.feature

```
Scenario: Use escaped-pipe symbol
  Given I use table data with:
    | name  | value |
    | alice | one\|two\|three\|four  |
  Then table data for "alice" is "one|two|three|four"
```

See also

- issue.features/issue0302.feature for details

Configuration Improvements

Language Option

The interpretation of the `language-tag` comment in feature files (Gherkin) and the configuration `lang` option on command-line and in the configuration file changed slightly.

If a `language-tag` is used in a feature file, it is now preferred over the command-line/configuration file settings. This is especially useful when your feature files use multiple spoken languages (in different files).

EXAMPLE:

Listing 70: FILE: features/french_1.feature

```
# language: fr
Fonctionnalité: Alice
    ...
```

Listing 71: FILE: behave.ini

```
[behave]
lang = de       # Default (spoken) language to use: German
...
```

Note

The feature object contains now a `language` attribute that contains the information which language was used during Gherkin parsing.

Default Tags

It is now possible to define `default tags` in the configuration file. `Default tags` are used when you do not specify tags on the command-line.

EXAMPLE:

Listing 72: FILE: behave.ini

```
# -- Exclude/skip any feature/scenario with @xfail or @not_implemented tags
[behave]
default_tags = not (@xfail or @not_implemented)
...
```

Runner Improvements

Hook Errors cause Failures

The behaviour of hook errors, meaning uncaught exceptions while processing hooks, is changed in this release. The new behaviour causes the entity (test-run, feature, scenario), for which the hook is executed, to fail. In addition, a hook error in a `before_all()`, `before_feature()`, `before_scenario()`, and `before_tag()` hook causes its corresponding entity to be skipped.

See also

- features/runner.hook_errors.feature for the detailed specification

Option: Continue after Failed Step in a Scenario

This behaviour is sometimes desired, when you want to see what happens in the remaining steps of a scenario.

EXAMPLE:

Listing 73: FILE: features/environment.py

```
from behave.model import Scenario

def before_all(context):
    userdata = context.config.userdata
    continue_after_failed = userdata.getbool("runner.continue_after_failed_step",␣
↪False)
    Scenario.continue_after_failed_step = continue_after_failed
```

```
# -- ENABLE OPTION: Use userdata on command-line
behave -D runner.continue_after_failed_step=true features/
```

Note

A failing step normally causes correlated failures in most of the following steps. Therefore, this behaviour is normally not desired.

See also

- features/runner.continue_after_failed_step.feature for the detailed specification

Testing asyncio Frameworks

Since
behave 1.2.6.dev0

The following support was added to simplify testing asynchronuous framework and protocols that are based on `asyncio` module (since Python 3.4).

There are basically two use cases:

- async-steps (with event_loop.run_until_complete() semantics)
- async-dispatch step(s) with async-collect step(s) later on

Async-steps

It is now possible to use `async-steps` in `behave`. An async-step is basically a coroutine as step-implementation for behave. The async-step is wrapped into an `event_loop.run_until_complete()` call by using the `@async_run_until_complete` step-decorator.

This avoids another layer of indirection that would otherwise be necessary, to use the coroutine.

A simple example for the implementation of the async-steps is shown for:

- Python 3.5 with new `async/await` keywords
- Python 3.4 with `@asyncio.coroutine` decorator and `yield from` keyword

Listing 74: FILE: features/steps/async_steps35.py

```
# -- REQUIRES: Python >= 3.5
import asyncio
from behave import step
from behave.api.async_step import async_run_until_complete


@step('an async-step waits {duration:f} seconds')
@async_run_until_complete
async def step_async_step_waits_seconds_py35(context, duration):
    """Simple example of a coroutine as async-step (in Python 3.5 or newer)"""
    await asyncio.sleep(duration)
```

When you use the async-step from above in a feature file and run it with behave:

Listing 75: shell

```
# -- TEST-RUN OUTPUT:
$ behave -f plain features/async_run.feature
Feature:

  Scenario:
```

(continues on next page)

(continued from previous page)

```
    Given an async-step waits 0.3 seconds ... passed in 0.307s

1 feature passed, 0 failed, 0 skipped
1 scenario passed, 0 failed, 0 skipped
1 step passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.307s
```

Note

The async-step is wrapped with an `event_loop.run_until_complete()` call. As the timings show, it actually needs approximately 0.3 seconds to run.

Async-dispatch and async-collect

The other use case with testing async frameworks is that

- you dispatch one or more async-calls
- you collect (and verify) the results of the async-calls later-on

A simple example of this approach is shown in the following feature file:

Listing 76: FILE: features/async_dispatch.feature

```
@use.with_python.min_version=3.5
Feature:
  Scenario:
    Given I dispatch an async-call with param "Alice"
    And   I dispatch an async-call with param "Bob"
    Then the collected result of the async-calls is "ALICE, BOB"
```

When you run this feature file:

Listing 77: shell

```
# -- TEST-RUN OUTPUT:
$ behave -f plain features/async_dispatch.feature
Feature:

  Scenario:
    Given I dispatch an async-call with param "Alice" ... passed in 0.001s
    And I dispatch an async-call with param "Bob" ... passed in 0.000s
    Then the collected result of the async-calls is "ALICE, BOB" ... passed in 0.206s

1 feature passed, 0 failed, 0 skipped
1 scenario passed, 0 failed, 0 skipped
3 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.208s
```

Note

The final async-collect step needs approx. 0.2 seconds until the two dispatched async-tasks have finished. In contrast, the async-dispatch steps basically need no time at all.

An `AsyncContext` object is used on the context, to hold the event loop information and the async-tasks that are of interest.

The implementation of the steps from above:

Listing 78: FILE: features/steps/async_dispatch_steps.py

```
# -- REQUIRES: Python >= 3.5
import sys
from behave import given, then
from behave.api.async_step import use_or_create_async_context
from behave.python_feature import PythonFeature
from hamcrest import assert_that, equal_to, empty
import asyncio


PYTHON_VERSION = sys.version_info[:2]


# -----------------------------------------------------------------------------
# ASYNC EXAMPLE FUNCTION:
# -----------------------------------------------------------------------------
if PythonFeature.has_async_function():
    # -- USE: async-function as coroutine-function
    # SINCE: Python 3.5 (preferred)
    async def async_func(param):
        await asyncio.sleep(0.2)
        return str(param).upper()


# -----------------------------------------------------------------------------
# STEPS:
# -----------------------------------------------------------------------------
@given('I dispatch an async-call with param "{param}"')
def step_dispatch_async_call(context, param):
    async_context = use_or_create_async_context(context, "async_context1")
    task = async_context.loop.create_task(async_func(param))
    async_context.tasks.append(task)


@then('the collected result of the async-calls is "{expected}"')
def step_collected_async_call_result_is(context, expected):
    async_context = context.async_context1
    wait_kwargs = {}
    if PYTHON_VERSION < (3, 10):
        # -- HINT on asyncio.wait: loop parameter was removed in Python 3.10
        wait_kwargs = dict(loop=async_context.loop)
    done, pending = async_context.loop.run_until_complete(
        asyncio.wait(async_context.tasks, **wait_kwargs))

    parts = [task.result() for task in done]
    joined_result = ", ".join(sorted(parts))
    assert_that(joined_result, equal_to(expected))
    assert_that(pending, empty())
```

Context-based Cleanups

It is now possible to register cleanup functions with the context object. This functionality is normally used in:

- hooks (`before_all()`, `before_feature()`, `before_scenario()`, ...)
- step implementations

- ...

Listing 79: SIGNATURE: `Context.add_cleanup(cleanup_func, *args, **kwargs)`

```
# -- CLEANUP CALL EXAMPLES:
context.add_cleanup(cleanup0)                       # CALLS LATER: cleanup0()
context.add_cleanup(cleanup1, 1, 2)                 # CALLS LATER: cleanup1(1, 2)
context.add_cleanup(cleanup2, name="Alice")         # CALLS LATER: cleanup2(name=
↪"Alice")
context.add_cleanup(cleanup3, 1, 2, name="Bob")     # CALLS LATER: cleanup3(1, 2,␣
↪name="Bob")
```

The registered cleanup will be performed when the context layer is removed. This depends on the the context layer when the cleanup function was registered (test-run, feature, scenario).

Example:

Listing 80: FILE: features/environment.py

```
def before_all(context):
    context.add_cleanup(cleanup_me)
    # -- ON CLEANUP: Calls cleanup_me()
    # Called after test-run.

def before_tag(context, tag):
    if tag == "foo":
        context.foo = setup_foo()
        context.add_cleanup(cleanup_foo, context.foo)
        # -- ON CLEANUP: Calls cleanup_foo(context.foo)
        # CASE scenario tag: cleanup_foo() will be called after this scenario.
        # CASE feature  tag: cleanup_foo() will be called after this feature.
```

See also

For more details, see features/runner.context_cleanup.feature .

Fixtures

Fixtures simplify setup/cleanup tasks that are often needed for testing.

Providing a Fixture

Listing 81: FILE: behave4my_project/fixtures.py

```
# -- ALTERNATIVE-FILE-OPTION: features/environment.py
from behave import fixture
from somewhere.browser.firefox import FirefoxBrowser

# -- FIXTURE-VARIANT 1: Use generator-function
@fixture
def browser_firefox(context, timeout=30, **kwargs):
    # -- SETUP-FIXTURE PART:
    context.browser = FirefoxBrowser(timeout, **kwargs)
    yield context.browser
    # -- CLEANUP-FIXTURE PART:
    context.browser.shutdown()
```

Using a Fixture

Listing 82: FILE: features/use_fixture1.feature

```
Feature: Use Fixture on Scenario Level

    @fixture.browser.firefox
    Scenario: Use Web Browser Firefox
        Given I load web page "https://somewhere.web"
        ...
    # -- AFTER-SCENARIO: Cleanup fixture.browser.firefox
```

Listing 83: FILE: features/environment.py

```
from behave import use_fixture
from behave4my_project.fixtures import browser_firefox

def before_tag(context, tag):
    if tag == "fixture.browser.firefox":
        use_fixture(browser_firefox, context, timeout=10)
```

See also

- *Fixtures* description for details
- features/fixture.feature

1.14.6 Noteworthy in Version 1.2.5

Scenario Outline Improvements

Better represent Example/Row

Since
behave 1.2.5a1

Covers
Name annotation, file location

A scenario outline basically a parametrized scenario template. It represents a macro/script that is executed for a data-driven set of examples (parametrized data). Therefore, a scenario outline generates several scenarios, each representing one example/row combination.

```
# -- file:features/xxx.feature
Feature:
  Scenario Outline: Wow            # line 2
    Given an employee "<name>"

    Examples: Araxas
      | name  | birthyear |
      | Alice |  1985     |         # line 7
      | Bob   |  1975     |         # line 8

    Examples:
      | name    | birthyear |
      | Charly |  1995     |        # line 12
```

Up to now, the following scenarios were generated from the scenario outline:

```
Scenario Outline: Wow          # features/xxx.feature:2
  Given an employee "Alice"

Scenario Outline: Wow          # features/xxx.feature:2
  Given an employee "Bob"

Scenario Outline: Wow          # features/xxx.feature:2
  Given an employee "Charly"
```

Note that all generated scenarios had the:

- same name (scenario_outline.name)
- same file location (scenario_outline.file_location)

From now on, the generated scenarios better represent the example/row combination within a scenario outline:

```
Scenario Outline: Wow -- @1.1 Araxas  # features/xxx.feature:7
  Given an employee "Alice"

Scenario Outline: Wow -- @1.2 Araxas  # features/xxx.feature:8
  Given an employee "Bob"

Scenario Outline: Wow -- @2.1         # features/xxx.feature:12
  Given an employee "Charly"
```

Note that:

- scenario name is now unique for any examples/row combination
- scenario name optionally contains the examples (group) name (if one exists)
- each scenario has a unique file location, based on the row's file location

Therefore, each generated scenario from a scenario outline can be selected via its file location (and run on its own). In addition, if one fails, it is now possible to rerun only the failing example/row combination(s).

The name annotations schema for the generated scenarios from above provides the new default name annotation schema. It can be adapted/overwritten in "behave.ini":

Listing 84: FILE: behave.ini

```
[behave]
scenario_outline_annotation_schema = {name} -- @{row.id} {examples.name}

# -- REVERT TO: Old naming schema:
# scenario_outline_annotation_schema = {name}
```

The following additional placeholders are provided within a scenario outline to support this functionality. They can be used anywhere within a scenario outline.

Placeholder	Description
examples.name	Refers name of the example group, may be an empty string.
examples.index	Index of the example group (range=1..N).
row.index	Index of the current row within an example group (range=1..R).
row.id	Shortcut for schema: "<examples.index>.<row.index>"

Name may contain Placeholders

Since
behave 1.2.5a1

A scenario outline can now use placeholders from example/rows in its name or its examples name. When the scenarios a generated, these placeholders will be replaced with the values of the example/row.

Up to now this behavior did only apply to steps of a scenario outline.

EXAMPLE:

Listing 85: FILE: features/some.feature

```
Feature:
  Scenario Outline: Wow <name>-<birthyear>  # line 2
    Given an employee "<name>"

    Examples:
      | name  | birthyear |
      | Alice |  1985     |         # line 7
      | Bob   |  1975     |         # line 8

    Examples: Benares-<ID>
      | name   | birthyear | ID |
      | Charly |  1995     | 42 |   # line 12
```

This leads to the following generated scenarios, one for each examples/row combination:

Listing 86: shell: behave -f pretty features/some.feature

```
Scenario Outline: Wow Alice-1985 -- @1.1          # features/some.feature:7
  Given an employee "Alice"

Scenario Outline: Wow Bob-1975 -- @1.2            # features/some.feature:8
  Given an employee "Bob"

Scenario Outline: Wow Charly-1885 -- @2.1 Benares-42 # features/some.feature:12
  Given an employee "Charly"
```

Tags may contain Placeholders

Since
behave 1.2.5a1

Tags from a Scenario Outline are also part of the parametrized template. Therefore, you may also use placeholders in the tags of a Scenario Outline.

Note

- Placeholder names, that are used in tags, should not contain whitespace.
- Placeholder values, that are used in tags, are transformed to contain no whitespace characters.

EXAMPLE:

Listing 87: FILE: features/some.feature

```
Feature:
```

(continues on next page)

(continued from previous page)

```
  @foo.group<examples.index>
  @foo.row<row.id>
  @foo.name.<name>
  Scenario Outline: Wow            # line 6
    Given an employee "<name>"

    Examples: Araxas
      | name  | birthyear |
      | Alice |  1985     |         # line 11
      | Bob   |  1975     |         # line 12

    Examples: Benares
      | name    | birthyear | ID |
      | Charly |  1995     | 42 |   # line 16
```

This leads to the following generated scenarios, one for each examples/row combination:

Listing 88: shell: behave -f pretty features/some.feature

```
@foo.group1 @foo.row1.1 @foo.name.Alice
Scenario Outline: Wow -- @1.1 Araxas   # features/some.feature:11
  Given an employee "Alice"

@foo.group1 @foo.row1.2 @foo.name.Bob
Scenario Outline: Wow -- @1.2 Araxas   # features/some.feature:12
  Given an employee "Bob"

@foo.group2 @foo.row2.1 @foo.name.Charly
Scenario Outline: Wow -- @2.1 Benares  # features/some.feature:16
  Given an employee "Charly"
```

It is now possible to run only the examples group “Araxas” (examples group 1) by using the select-by-tag mechanism:

Listing 89: shell

```
$ behave --tags=@foo.group1 -f progress3 features/some.feature
...  # features/some.feature
  Wow -- @1.1 Araxas  .
  Wow -- @1.2 Araxas  .
```

Run examples group via select-by-name

Since
behave 1.2.5a1

The improvements on unique generated scenario names for a scenario outline (with name annotation) can now be used to run all rows of one examples group.

EXAMPLE:

Listing 90: FILE: features/some.feature

```
Feature:
  Scenario Outline: Wow            # line 2
    Given an employee "<name>"

    Examples: Araxas
```

(continues on next page)

(continued from previous page)

```
      | name  | birthyear |
      | Alice |  1985     |         # line 7
      | Bob   |  1975     |         # line 8

    Examples: Benares
      | name   | birthyear |
      | Charly |  1995     |         # line 12
```

This leads to the following generated scenarios (when the feature is executed):

Listing 91: shell: behave -f pretty features/some.feature

```
Scenario Outline: Wow -- @1.1 Araxas  # features/some.feature:7
  Given an employee "Alice"

Scenario Outline: Wow -- @1.2 Araxas   # features/some.feature:8
  Given an employee "Bob"

Scenario Outline: Wow -- @2.1 Benares  # features/some.feature:12
  Given an employee "Charly"
```

You can now run all rows of the "Araxas" examples (group) by selecting it by name (name part or regular expression):

Listing 92: shell

```
$ behave --name=Araxas -f progress3 features/some.feature
...  # features/some.feature
  Wow -- @1.1 Araxas  .
  Wow -- @1.2 Araxas  .

$ behave --name='-- @.* Araxas' -f progress3 features/some.feature
...  # features/some.feature
  Wow -- @1.1 Araxas  .
  Wow -- @1.2 Araxas  .
```

Exclude Feature/Scenario at Runtime

Since
behave 1.2.5a1

A test writer can now provide a runtime decision logic to exclude a feature, scenario or scenario outline from a test run within the following hooks:

- `before_feature()` for a feature
- `before_scenario()` for a scenario
- step implementation (normally only: given step)

by using the `skip()` method before a feature or scenario is run.

Listing 93: FILE: features/environment.py

```
# -- EXAMPLE 1: Exclude scenario from run-set at runtime.
import sys

def should_exclude_scenario(scenario):
    # -- RUNTIME DECISION LOGIC: Will exclude
    #  * Scenario: Alice
```

(continues on next page)

(continued from previous page)

```
    #  * Scenario: Alice in Wonderland
    #  * Scenario: Bob and Alice2
    return "Alice" in scenario.name

def before_scenario(context, scenario):
    if should_exclude_scenario(scenario):
        scenario.skip()  #< EXCLUDE FROM RUN-SET.
        # -- OR WITH REASON:
        # reason = "RUNTIME-EXCLUDED"
        # scenario.skip(reason)
```

Listing 94: FILE: features/steps/my_steps.py

```
# -- EXAMPLE 2: Skip remaining steps in step implementation.
from behave import given

@given('the assumption "{assumption}" is met')
def step_check_assumption(context, assumption):
    if not is_assumption_valid(assumption):
        # -- SKIP: Remaining steps in current scenario.
        context.scenario.skip("OOPS: Assumption not met")
        return

    # -- NORMAL CASE:
    ...
```

Test Stages

Since
behave 1.2.5a1

Intention
Use different Step Implementations for Each Stage

A test stage allows the user to provide different step and environment implementation for each stage. Examples for test stages are:

- develop (example: development environment with simple database)
- product (example: use the real product and its database)
- sysint (system integration)
- ...

Each test stage may have a different test environment and needs to fulfill different testing constraints.

EXAMPLE DIRECTORY LAYOUT (with `stage=testlab` and default stage):

```
features/
  +-- steps/                # -- Step implementations for default stage.
  |   +-- foo_steps.py
  +-- testlab_steps/        # -- Step implementations for stage=testlab.
  |   +-- foo_steps.py
  +-- environment.py          # -- Environment for default stage.
  +-- testlab_environment.py  # -- Environment for stage=testlab.
  +-- *.feature
```

To use the `stage=testlab`, you run behave with:

or define the environment variable `BEHAVE_STAGE=testlab`.

Userdata

Since
behave 1.2.5a1

Intention
User-specific Configuration Data

The userdata functionality allows a user to provide its own configuration data:

- as command-line option `-D name=value` or `--define name=value`
- with the behave configuration file in section `behave.userdata`
- load more configuration data in `before_all()` hook

Listing 95: FILE: behave.ini

```
[behave.userdata]
browser = firefox
server  = asterix
```

Note

Command-line definitions override userdata definitions in the configuration file.

If the command-line contains no value part, like in `-D NEEDS_CLEANUP`, its value is `"true"`.

The userdata settings can be accessed as dictionary in hooks and steps by using the `context.config.userdata` dictionary.

Listing 96: FILE: features/environment.py

```
def before_all(context):
    browser = context.config.userdata.get("browser", "chrome")
    setup_browser(browser)
```

Listing 97: FILE: features/steps/userdata_example_steps.py

```
@given('I setup the system with the user-specified server"')
def step_setup_system_with_userdata_server(context):
    server_host = context.config.userdata.get("server", "beatrix")
    context.xxx_client = xxx_protocol.connect(server_host)
```

Listing 98: shell

```
# -- ADAPT TEST-RUN: With user-specific data settings.
behave -D server=obelix features/
behave --define server=obelix features/
```

Other examples for user-specific data are:

- Passing a URL to an external resource that should be used in the tests
- Turning off cleanup mechanisms implemented in environment hooks, for debugging purposes.

Type Converters

The userdata object provides basic support for "type conversion on demand", similar to the `configparser` module. The following type conversion methods are provided:

- `Userdata.getint(name, default=0)`

- `Userdata.getfloat(name, default=0.0)`
- `Userdata.getbool(name, default=False)`
- `Userdata.getas(convert_func, name, default=None, ...)`

Type conversion may raise a `ValueError` exception if the conversion fails.

The following example shows how the type converter functions for integers are used:

Listing 99: FILE: features/environment.py

```
def before_all(context):
    userdata = context.config.userdata
    server_name  = userdata.get("server", "beatrix")
    int_number   = userdata.getint("port", 80)
    bool_answer  = userdata.getbool("are_you_sure", True)
    float_number = userdata.getfloat("temperature_threshold", 50.0)
    ...
```

Advanced Cases

The last section described the basic use cases of userdata. For more complicated cases, it is better to provide your own configuration setup in the `before_all()` hook.

This section describes how to load a JSON configuration file and store its data in the `userdata` dictionary.

Listing 100: FILE: features/environment.py

```
import json
import os.path

def before_all(context):
    """Load and update userdata from JSON configuration file."""
    userdata = context.config.userdata
    configfile = userdata.get("configfile", "userconfig.json")
    if os.path.exists(configfile):
        assert configfile.endswith(".json")
        more_userdata = json.load(open(configfile))
        context.config.update_userdata(more_userdata)
        # -- NOTE: Reapplies userdata_defines from command-line, too.
```

Provide the file "userconfig.json" with:

Listing 101: FILE: userconfig.json

```
{
    "browser": "firefox",
    "server":  "asterix",
    "count":   42,
    "cleanup": true
}
```

Other advanced use cases:

- support configuration profiles via cmdline "... -D PROFILE=xxx ..." (uses profile-specific configuration file or profile-specific config section)
- provide test stage specific configuration data

Active Tags

Since
behave 1.2.5a1

Active tags are used when it is necessary to decide at runtime which features or scenarios should run (and which should be skipped). The runtime decision is based on which:

- platform the tests run (like: Windows, Linux, MACOSX, ...)
- runtime environment resources are available (by querying the "testbed")
- runtime environment resources should be used (via *userdata* or ...)

Therefore, for *active tags* it is decided at runtime if a tag is enabled or disabled. The runtime decision logic excludes features/scenarios with disabled active tags before they are run.

Note

The active tag mechanism is applied after the normal tag filtering that is configured on the command-line.

The active tag mechanism uses the `ActiveTagMatcher` for its core functionality.

Active Tag Logic

- A (positive) active tag is enabled, if its value matches the current value of its category.
- A negated active tag (starting with "not") is enabled, if its value does not match the current value of its category.
- A sequence of active tags is enabled, if all its active tags are enabled (logical-and operation).
- Active tags are evaluated isolated on each feature/rule/scenario.
- An active tag can be overridden and made more specific, if this active tag is enabled on an outer model element (like on a feature/rule for a scenario).

Active Tag Schema

The following two tag schemas are supported for active tags (by default).

Dialect 1 (preferred):

```
@use.with_{category}={value}
@not.with_{category}={value}
@only.with_{category}={value}   # -- HINT: Avoid to use.
```

Dialect 2:

```
@active.with_{category}={value}
@not_active.with_{category}={value}
```

Example 1

Assuming you have the feature file where:

- scenario `Alice` should only run if `Chrome` browser is used
- scenario `Bob` should only run if `Safari` browser is used

Listing 102: FILE: features/alice.feature

```
Feature:

    @use.with_browser=chrome
    Scenario: Alice (Run only with Browser Chrome)
        Given I do something
        ...

    @use.with_browser=safari
    Scenario: Bob (Run only with Browser Safari)
        Given I do something else
        ...
```

Listing 103: FILE: features/environment.py

```
# -- EXAMPLE: ACTIVE TAGS, exclude scenario from run-set at runtime.
# NOTE: ActiveTagMatcher implements the runtime decision logic.
from behave.tag_matcher import ActiveTagMatcher
import os
import sys

active_tag_value_provider = {
    "browser": "chrome"
}
active_tag_matcher = ActiveTagMatcher(active_tag_value_provider)

def before_all(context):
    # -- SETUP ACTIVE-TAG MATCHER VALUE(s):
    active_tag_value_provider["browser"] = os.environ.get("BROWSER", "chrome")

def before_scenario(context, scenario):
    if active_tag_matcher.should_exclude_with(scenario.tags):
        # -- NOTE: Exclude any with @use.with_browser=<other_browser>
        scenario.skip(reason="DISABLED ACTIVE-TAG")
```

Note

By using this mechanism, the `@use.with_browser=*` tags become **active tags**. The runtime decision logic decides when these tags are enabled or disabled (and uses them to exclude their scenario/feature).

Tip

Since behave `v1.2.7`, better use:

```
def before_scenario(context, scenario):
    # -- ALTERNATIVE: if active_tag_matcher.should_skip(scenario):
    if active_tag_matcher.should_skip_with_tags(scenario.tags):
        scenario.skip(active_tag_matcher.skip_reason)
```

Example 2

Assuming you have scenarios with the following runtime conditions:

- Run scenario Alice only on Windows OS

- Run scenario Bob only with browser Chrome

Listing 104: FILE: features/alice.feature

```
# -- TAG SCHEMA: @use.with_{category}={value}, ...
Feature:

  @use.with_os=win32
  Scenario: Alice (Run only on Windows)
    Given I do something
    ...

  @use.with_browser=chrome
  Scenario: Bob (Run only with Web-Browser Chrome)
    Given I do something else
    ...
```

Listing 105: FILE: features/environment.py

```
from behave.tag_matcher import ActiveTagMatcher
import sys

# -- MATCHES ANY TAGS: @use.with_{category}={value}
# NOTE: active_tag_value_provider provides category values for active tags.
active_tag_value_provider = {
    "browser": os.environ.get("BEHAVE_BROWSER", "chrome"),
    "os":      sys.platform,
}
active_tag_matcher = ActiveTagMatcher(active_tag_value_provider)

# -- BETTER USE: from behave.tag_matcher import setup_active_tag_values
def setup_active_tag_values(active_tag_values, data):
    for category in active_tag_values.keys():
        if category in data:
            active_tag_values[category] = data[category]

def before_all(context):
    # -- SETUP ACTIVE-TAG MATCHER (with userdata):
    # USE: behave -D browser=safari ...
    setup_active_tag_values(active_tag_value_provider, context.config.userdata)

def before_feature(context, feature):
    if active_tag_matcher.should_exclude_with(feature.tags):
        feature.skip(reason="DISABLED ACTIVE-TAG")

def before_scenario(context, scenario):
    if active_tag_matcher.should_exclude_with(scenario.tags):
        scenario.skip("DISABLED ACTIVE-TAG")
```

By using the *userdata* mechanism, you can now define on command-line which browser should be used when you run behave.

Listing 106: shell

```
# -- SHELL: Run behave with browser=safari, ... by using userdata.
# TEST VARIANT 1: Run tests with browser=safari
behave -D browser=safari features/
```

(continues on next page)

(continued from previous page)

```
# TEST VARIANT 2: Run tests with browser=chrome
behave -D browser=chrome features/
```

Note

Unknown categories, missing in the `active_tag_value_provider` are ignored.

User-defined Formatters

Since
behave 1.2.5a1

Behave formatters are a typical candidate for an extension point. You often need another formatter that provides the desired output format for a test-run.

Therefore, behave supports now formatters as extension point (or plugin). It is now possible to use own, user-defined formatters in two ways:

- Use formatter class (as "scoped class name") as `--format` option value
- Register own formatters by name in behave's configuration file

Note

Scoped class name (schema):

- `my.module:MyClass` (preferred)
- `my.module::MyClass` (alternative; with double colon as separator)

User-defined Formatter on Command-line

Just use the formatter class (as "scoped class name") on the command-line as value for the `-format` option (short option: `-f`):

```
behave -f my.own_module:SimpleFormatter ...
behave -f behave.formatter.plain:PlainFormatter ...
```

Listing 107: FILE: my/own_module.py

```
# -- NOTE: or installed as Python module: my.own_module
from behave.formatter.base import Formatter

class SimpleFormatter(Formatter):
    description = "A very simple NULL formatter"
```

Register User-defined Formatter by Name

It is also possible to extend behave's built-in formatters by registering one or more user-defined formatters by name in the configuration file:

Listing 108: FILE: behave.ini

```
[behave.formatters]
foo = behave_contrib.formatter.foo:FooFormatter
bar = behave_contrib.formatter.bar:BarFormatter
```

Listing 109: FILE: behave_contrib/formatter/foo.py

```
from behave.formatter.base import Formatter

class FooFormatter(Formatter):
    description = "A FOO formatter"
    ...
```

Now you can use the name for any registered, user-defined formatter:

Listing 110: shell

```
# -- NOTE: Use FooFormatter that was registered by name "foo".
behave -f foo ...
```

1.14.7 Noteworthy in Version 1.2.4

Diagnostics: Start Debugger on Error

Since
behave 1.2.4a1

See also *Debug-on-Error (in Case of Step Failures)* .

1.15 More Information about Behave

1.15.1 Tutorials

For new users, that want to read, understand and explore the concepts in Gherkin and behave (after reading the behave documentation):

- "Behave by Example" (on github)

The following small tutorials provide an introduction how you use behave in a specific testing domain:

- Phillip Johnson, Getting Started with Behavior Testing in Python with Behave, 2015-10-15.
- Nicole Harris, Beginning BDD with Django (part 1 and 2), 2015-03-16.
- TestingBot, Bdd with Python, Behave and WebDriver
- Wayne Witzel III, Using Behave with Pyramid, 2014-01-10.

Warning

A word of caution if you are new to **"behaviour-driven development" (BDD)**. In general, you want to avoid "user interface" (UI) details in your scenarios, because they describe **how something is implemented** (in this case the UI itself), like:

- `press this button`
- then `enter this text into the text field`
- ...

In **BDD** (or testing in general), you should describe **what should be done** (meaning the intention). This will make your scenarios much more robust and stable because you can change the underlying implementation of:

- the "system under test" (SUT) or
- the test automation layer, that interacts with the SUT.

without changing the scenarios.

1.15.2 Books

Behave is covered in the following books:

> Harry Percival, Test-Driven Development with Python, 2nd Edition, O'Reilly, August 2017 (focuses on Web development with Django, covers Behave in Appendix E: BDD)

1.15.3 Presentation Videos

- Benno Rice: Making Your Application Behave (30min), 2012-08-12, PyCon Australia.
- Selenium: First behave python tutorial with selenium (8min), 2015-01-28,
- Jessica Ingrasselino: Automation with Python and Behave (67min), 2015-12-16
- Selenium Python Webdriver Tutorial - Behave (BDD) (14min), 2016-01-21
- Front-end integration testing with splinter (30min), 2017-08-05

1.15.4 Tool-oriented Tutorials

JetBrains PyCharm:

- Blog: In-Depth Screencast on Testing (2016-04-11; video offset=2:10min)
- Docs: BDD Testing Framework Support in PyCharm 2024.2

1.15.5 Find more Information

See also

- python-behave examples (Google)
- python-behave tutorials (Google)
- python-behave videos (Google)

1.15.6 Get Involved

- GitHub discussions (user questions and discussions)
- GitHub issues (discuss bugs and features)
- GitHub pull requests (propose changes, fixes, features)
- StackOverflow (tag: python-behave)

1.16 Contributing

If you find a bug you can fix or want to contribute an enhancement you're welcome to open an issue on GitHub or create a pull request directly.

1.16.1 Using `invoke` for Development

For most development tasks we have invoke commands.

Install all requirements for running tasks using Pip, e.g.

```
python3 -m pip install -r tasks/py.requirements.txt
```

Display all available `invoke` commands like this:

```
invoke -l
```

If you're curious, all `invoke` tasks are located in the `tasks/` folder.

1.16.2 Update Gherkin Language Specification

An `invoke` command will download the latest Gherkin language specification and update the behave/i18n.py module:

```
invoke develop.update-gherkin
```

If there were changes this command will have updated two files:

1. `etc/gherkin/gherkin-languages.json` (original Cucumber JSON spec)
2. behave/i18n.py (Python module generated from the JSON spec)

Put both under version control and open a PR to merge them.

1.16.3 Update Documentation

Our documentation is written in reStructuredText, and built and hosted on ReadTheDocs. Make your changes to the files in the `docs/` folder and build the documentation with:

```
invoke docs
```

or, alternatively, using Tox:

```
tox -e docs
```

Hint

Building the docs requires Sphinx and DocUtils. If your build fails because those are missing, run:

```
python3 -m pip install -r py.requirements/docs.txt
```

Once the docs are built successfully, `sphinx` will tell you where it generated the HTML output (typically `build/docs/html`), which you can then inspect locally.

1.17 Appendix

Contents:

1.17.1 Formatters and Reporters

behave provides 2 different concepts for reporting results of a test run:

- formatters
- reporters

A slightly different interface is provided for each "formatter" concept. The `Formatter` is informed about each step that is taken. The `Reporter` has a more coarse-grained API.

Reporters

The following reporters are currently supported:

Name	Description
junit	Provides JUnit XML-like output.
summary	Provides a summary of the test run.

Formatters

The following formatters are currently supported:

Name	Mode	Description
help	normal	Shows all registered formatters.
captured	normal	Inspect captured output.
json	normal	JSON dump of test run
json.pretty	normal	JSON dump of test run (human readable)
plain	normal	Very basic formatter with maximum compatibility
pretty	normal	Standard colourised pretty formatter
progress	normal	Shows dotted progress for each executed scenario.
progress2	normal	Shows dotted progress for each executed step.
progress3	normal	Shows detailed progress for each step of a scenario.
rerun	normal	Emits scenario file locations of failing scenarios
sphinx.steps	dry-run	Generate sphinx-based documentation for step definitions.
steps	dry-run	Shows step definitions (step implementations).
steps.bad	dry-run	Shows BAD STEP-DEFINITION(s) (if any exist).
steps.catalog	dry-run	Shows non-technical documentation for step definitions.
steps.code	dry-run	Shows executed steps combined with their code.
steps.doc	dry-run	Shows documentation for step definitions.
steps.usage	dry-run	Shows how step definitions are used by steps (in feature files).
tags	dry-run	Shows tags (and how often they are used).
tags.location	dry-run	Shows tags and the location where they are used.

Note

You can use more than one formatter during a test run. But in general you have only one formatter that writes to `stdout`.

The "Mode" column indicates if a formatter is intended to be used in dry-run (`--dry-run` command-line option) or normal mode.

User-Defined Formatters

Behave allows you to provide your own formatter (class):

Listing 111: SHELL

```
# -- USE: Formatter class "Json2Formatter" in python module "foo.bar"
# NOTE: Formatter must be importable from python search path.
behave -f foo.bar:Json2Formatter ...
```

The usage of a user-defined formatter can be simplified by providing an alias name for it in the configuration file:

Listing 112: FILE: behave.ini

```
# ALIAS SUPPORTS: behave -f json2 ...
# NOTE: Formatter aliases may override builtin formatters.
[behave.formatters]
json2 = foo.bar:Json2Formatter
```

If your formatter can be configured, you should use the userdata concept to provide them. The formatter should use the attribute schema:

Listing 113: FILE: behave.ini

```
# SCHEMA: behave.formatter.<FORMATTER_NAME>.<ATTRIBUTE_NAME>
[behave.userdata]
behave.formatter.json2.use_pretty = true

# -- SUPPORTS ALSO:
#    behave -f json2 -D behave.formatter.json2.use_pretty ...
```

Use `behave -f help` to:

- Inspect which formatters are currently defined/supported in your workspace
- Check if a formatter definition has a problem (and which), like: `ModuleNotFoundError`

Listing 114: SHELL

```
$ behave -f help
AVAILABLE FORMATTERS:
  captured       Inspect captured output.
  html           Very basic HTML formatter
  json           JSON dump of test run
  json.pretty    JSON dump of test run (human readable)
  null           Provides formatter that does not output anything.
  plain          Very basic formatter with maximum compatibility
  pretty         Standard colourised pretty formatter
  progress       Shows dotted progress for each executed scenario.
  progress2      Shows dotted progress for each executed step.
  progress3      Shows detailed progress for each step of a scenario.
  rerun          Emits scenario file locations of failing scenarios
  sphinx.steps   Generate sphinx-based documentation for step definitions.
  steps          Shows step definitions (step implementations).
  steps.bad      Shows BAD STEP-DEFINITION(s) (if any exist).
  steps.catalog  Shows non-technical documentation for step definitions.
  steps.code     Shows executed steps combined with their code.
  steps.doc      Shows documentation for step definitions.
  steps.missing  Shows undefined/missing steps definitions, implements them.
  steps.usage    Shows how step definitions are used by steps.
  tags           Shows tags (and how often they are used).
  tags.location  Shows tags and the location where they are used.

UNAVAILABLE FORMATTERS:
  allure         ModuleNotFoundError: No module named 'allure_behave'
```

DESIGN CONSTRAINTS:

A formatter class must implement the following interface:

- `behave.formatter.api:IFormatter`
- `behave.formatter.api:IFormatter2` (alternative)

A formatter class should be derived from the following class:

- `behave.formatter.api:Formatter` (aka: *behave.formatter.base:Formatter*) for `api:IFormatter`
- `behave.formatter.api:BaseFormatter2` (aka: *behave.formatter.base2:BaseFormatter2*) for `api:IFormatter2`

More Formatters

The following contributed formatters are currently known:

Name	Description
allure	allure-behave, an Allure formatter for behave.
html	behave-html-formatter, a simple HTML formatter for behave.
html-pretty	behave-html-pretty-formatter, a pretty HTML formatter for behave.
teamcity	behave-teamcity, a formatter for JetBrains TeamCity CI testruns with behave.

The usage of a custom formatter can be simplified if a formatter alias is defined for.

EXAMPLE:

Listing 115: FILE: behave.ini

```
# FORMATTER ALIASES: "behave -f allure" and others...
[behave.formatters]
allure = allure_behave.formatter:AllureFormatter
html = behave_html_formatter:HTMLFormatter
html-pretty = behave_html_pretty_formatter:PrettyHTMLFormatter
teamcity = behave_teamcity:TeamcityFormatter
```

Embedding Screenshots / Data in Reports

Hint 1
Only supported by JSON formatter

Hint 2
Binary attachments may require base64 encoding.

You can embed data in reports with the *Context* method *attach()*, if you have configured a formatter that supports it. Currently only the JSON formatter supports embedding data.

For example:

Listing 116: FILE: features/steps/screenshot_example_steps.py

```
from behave import given, when
from behave4example.web_browser.util import take_screenshot_and_attach_to_scenario

@given('I open the Google webpage')
@when('I open the Google webpage')
def step_open_google_webpage(ctx):
    ctx.browser.get("https://www.google.com")
    take_screenshot_and_attach_to_scenario(ctx)
```

Listing 117: FILE: behave4example/web_browser/util.py

```
# HINTS:
#   * EXAMPLE CODE ONLY
#   * BROWSER-SPECIFIC: Implementation may depend on browser driver.
def take_screenshot_and_attach_to_scenario(ctx):
    # -- HINT: SELENIUM WITH CHROME: ctx.browser.get_screenshot_as_base64()
    screenshot_image = ctx.browser.get_full_page_screenshot_as_png() # OUTDATED
    ctx.attach("image/png", screenshot_image)
```

Listing 118: FILE: features/environment.py

```
# EXAMPLE REQUIRES: This browser driver setup code (or something similar).
from selenium import webdriver

def before_all(ctx):
    ctx.browser = webdriver.Firefox()
```

See also

- Selenium Python SDK: https://www.selenium.dev/selenium/docs/api/py/
- Playwright Python SDK: https://playwright.dev/python/docs/intro

RELATED: Selenium webdriver details:

- Selenium: Take a screenshot
- Selenium webdriver (for Firefox)
- Selenium webdriver (for Chrome)

RELATED: Playwright details:

- https://playwright.dev/python/docs/api/class-locator#locator-screenshot
- https://playwright.dev/python/docs/api/class-page#page-screenshot

1.17.2 Runners

behave provides an extension point for builtin and user-defined test runners.

The following runners are currently supported:

Name	Runner Class	Description
default	`behave.runner:Runner`	The default test runner provided by behave.
help	—	Shows which runners are currently available in your context.

You specify a runner by using the `-r <RUNNER>` or `--runner=<RUNNER>` command-line option. A `<RUNNER>` option value can be:

- a runner alias, defined in `[behave.runners]` section in the `behave.ini` config-file
- a scoped class name, like: `<SCOPED_MODULE_NAME>:<RUNNER_CLASS_NAME>`

Attention

If you provide an own test runner, you are in the inner parts of behave:

- You should know what you are doing.
- Inner parts of behave may change without notice.
- While it is not intended to break your test runner implementation, changes in the inner parts of behave may occur, that will break parts of your implementation.

User-Defined Runners

behave allows you to provide your own test runner (class):

Listing 119: SHELL

```
# USING COMMAND-LINE OPTION: -r/--runner=<RUNNER>
$ behave --runner=behave.runner:Runner ...
```

The usage of a user-defined runner can be simplified by providing an alias name for it in the configuration file:

Listing 120: FILE: behave.ini

```
# ALIAS SUPPORTS: behave -r default ...
[behave.runners]
default = behave.runner:Runner
```

Use `behave --runner=help` to:

- Inspect which runners are currently defined/supported in your workspace
- Check if the runner definitions have a problem, like: *ModuleNotFoundError*

Listing 121: SHELL

```
$ behave --runner=help
AVAILABLE RUNNERS:
  default  = behave.runner:Runner
```

DESIGN CONSTRAINTS:

- A runner class must implement the `behave.api.runner.ITestRunner` interface

Tip

See also features/runner.use_runner_class.feature for more information.

Failure Syndromes with User-Defined Runners

Exception	Failure Kinde	Description
`ModuleNotFoundErro:`	User Error	Python package with runner is probably not installed yet.
`ClassNotFoundError`	User or Devel Error	Python package is installed but class is not found (maybe: misspelled).
`InvalidClassError`	Developer Error	Runner class is not valid (for one of several reaons).

There are a number of reasons why the `InvalidClassError` exception occurs, like:

- The `ITestRunner` interface is not implemented.

- The `ITestRunner` interface contract is broken.
- The `ITestRunner` interface is only partially.

Tip

See also features/runner.use_runner_class.feature for more information and the different failure syndromes that may occur.

1.17.3 Context Attributes

A context object (*Context*) is handed to

- step definitions (step implementations)
- behave hooks (`before_all()`, `before_feature()`, ..., `after_all()`)

Behave Attributes

The behave runner assigns a number of attributes to the context object during a test run.

Attribute Name	Layer	Type	Description
config	test run	*Configuratio*	Configuration that is used.
aborted	test run	bool	Set to true if test run is aborted by the user.
failed	test run	bool	Set to true if a step fails.
feature	feature	*Feature*	Current feature.
rule	rule	*Feature*	Current rule.
tags	feature, rule, scenario	list<*Tag*>	Effective tags of current feature, rule, scenario, scenario outline.
active_outline	scenario outline	*Row*	Current row in a scenario outline (in examples table).
scenario	scenario	*Scenario*	Current scenario.
table	step	*Table*	Contains step's table, otherwise None.
text	step	String	Contains step's multi-line text (unicode), otherwise None.

Note

Behave attributes in the context object should not be modified by a user. See *Context* class description for more details.

Deprecated since version v1.2.7:

Attribute Name	Layer	Type	Description
log_capture	scenario	*LoggingCapture*	If logging capture is enabled.
stdout_capture	scenario	`StringIO`	If stdout capture is enabled.
stderr_capture	scenario	`StringIO`	If stderr capture is enabled.

User Attributes

A user can assign (or modify) own attributes to the context object. But these attributes will be removed again from the context object depending where these attributes are defined.

Kind	Assign Location	Lifecycle Layer (Scope)
Hook	`before_all()`	test run
Hook	`after_all()`	test run
Hook	`before_tags()`	feature, rule or scenario
Hook	`after_tags()`	feature, rule or scenario
Hook	`before_feature()`	feature
Hook	`after_feature()`	feature
Hook	`before_rule()`	rule
Hook	`after_rule()`	rule
Hook	`before_scenario()`	scenario
Hook	`after_scenario()`	scenario
Hook	`before_step()`	scenario
Hook	`after_step()`	scenario
Step	Step definition	scenario

1.17.4 Environment Variables

behave uses the following environment variables:

Environment Variable	Description
BE- HAVE_STRIP_STEPS_WITH_TI	Gherkin parser strips trailing colon from steps in Gherkin file (if enabled). This is only the case, if a step has a text/table section.

Environment Variable	Type	Values	Default Value
BEHAVE_STRIP_STEPS_WITH_TRAILING_COLON	bool	`yes`, `no`	`no`

1.17.5 Status Values

The `behave.model_core.Status` enum-class provides

Status	Description
untested	INITIAL VALUE (before test if executed).
untested_pending	RESERVED: Pending steps can not be detected in dry-run mode.
untested_undefined	Used for undefined steps, detected in dry-run mode.
skipped	Used if a model element is skipped (not part of the run set).
passed	Used if a model element has passed successfully.
failed	Used if a failure occurred: assert-mismatch.
error	Used if an error occurs, normally a unexpected exception is raised.
hook_error	Used if a hook fails (with exception or assert-mismatch).
pending	Used for pending steps (as error).
pending_warn	Used for pending steps (as passed step with *@wip* tag).
undefined	Used for undefined steps (as error).

Common Status Values

The following status values are used for:

- Features
- Rules
- Scenarios
- Steps

Status	Error?	Failed?
untested	no	
skipped	no	
passed	no	
failed	no	yes
error	yes	
hook_error	yes	

Specific Status Values for Steps

The following status values are only used for steps.

Status	Error?	Untested?	Pending?	Undefined?
untested_pending	no	yes	yes	no
untested_undefined	no	yes	no	yes
pending	yes	no	yes	no
pending_warn	no	no	yes	no
undefined	yes	no	no	yes

From Inner Status to Outer Status

The following table provides an overview how the outer status is derived from the status of contained model elements.

EXAMPLE:

- *hook_error* on a step leads to *error* in scenario.

Inner Status	Outer Status	Description
untested	*untested*	If no *passed* occurs.
untested_pending	*untested*	Like *untested*.
untested_undefined	*untested*	Like *untested*.
skipped	*skipped*	Same.
passed	*passed*	Same.
failed	*failed*	Same.
error	*error*	Same.
hook_error	*error*	
pending	*error*	
pending_warn	*passed*	
undefined	*error*	

1.17.6 Parse Expressions

Parse expressions are a simplified form of regular expressions. The actual regular expression is hidden behind the **type** name / hint.

Parse expressions are used in step definitions as a simplified alternative to regular expressions. They are used for parameters and type conversions (which are not supported for regular expression patterns).

```
# -- FILE: features/steps/example_steps.py
from behave import when

@when('we implement {number:d} tests')
def step_impl(context, number):  # -- NOTE: number is converted into integer
```

(continues on next page)

(continued from previous page)

```
    assert number > 1 or number == 0
    context.tests_count = number
```

The following tables provide a overview of the *parse expressions* syntax. See also Python regular expressions description in the Python re module.

Type	Characters Matched	Output
l	Letters (ASCII)	str
w	Letters, numbers and underscore	str
W	Not letters, numbers and underscore	str
s	Whitespace	str
S	Non-whitespace	str
d	Digits (effectively integer numbers)	int
D	Non-digit	str
n	Numbers with thousands separators (, or .)	int
%	Percentage (converted to value/100.0)	float
f	Fixed-point numbers	float
F	Decimal numbers	Decimal
e	Floating-point numbers with exponent e.g. 1.1e-10, NAN (all case insensitive)	float
g	General number format (either d, f or e)	float
b	Binary numbers	int
o	Octal numbers	int
x	Hexadecimal numbers (lower and upper case)	int
ti	ISO 8601 format date/time e.g. 1972-01-20T10:21:36Z (“T” and “Z” optional)	datetime
te	RFC2822 e-mail format date/time e.g. Mon, 20 Jan 1972 10:21:36 +1000	datetime
tg	Global (day/month) format date/time e.g. 20/1/1972 10:21:36 AM +1:00	datetime
ta	US (month/day) format date/time e.g. 1/20/1972 10:21:36 PM +10:30	datetime
tc	ctime() format date/time e.g. Sun Sep 16 01:03:52 1973	datetime
th	HTTP log format date/time e.g. 21/Nov/2011:00:07:11 +0000	datetime
ts	Linux system log format date/time e.g. Nov 9 03:37:44	datetime
tt	Time e.g. 10:21:36 PM -5:30	time

If parse_type module is used, the cardinality of a type can be specified, too (by using the CardinalityField support):

Cardinality	Description
?	Pattern with cardinality 0..1: optional part (question mark).
*	Pattern with cardinality zero or more, 0.. (asterisk).
+	Pattern with cardinality one or more, 1.. (plus sign).

1.17.7 Cucumber-Expressions

Cucumber-expressions:

- Similar idea like *parse expressions*
- Provide a compact, readable placeholder syntax for step-parameters in step definitions
- Support pre-defined parameter types with type conversion
- Support to define own parameter types
- Much easier to use compared to *regular expressions*

Listing 122: EXAMPLES: Step definitions with cucumber-expression

```
I have {int} cucumbers in my belly
I have {float} cucumbers in my belly
I have a {color} ball
```

Example

Listing 123: FILE: features/environment.py

```
from behave.cucumber_expression import use_step_matcher_for_cucumber_expressions

# -- DEFINE: cucumber-expressions as default step-matcher.
use_step_matcher_for_cucumber_expressions()
```

Listing 124: FILE: features/steps/example_steps_with_cucumber_expression.py

```
from behave import when
from behave.cucumber_expression import use_step_matcher_for_cucumber_expressions

# -- REQUIRED: If multiple step-matcher variants are used.
use_step_matcher_for_cucumber_expressions()

# -- MATCHES STEPS:
#   When I eat 1 apple
#   When I eat 10 apples
@when('I eat {int} apple(s)')
def step_when_eat_apples(ctx, number: int):
    assert isinstance(number, int)
    assert number >= 0
    ctx.apples_count = number
```

Predefined Parameter Types

Parameter-Type	Type	Description
{int}	int	Matches an 32-bit integer number (int) sand converts to it, like: 42
{float}	float	Matches float (as 32-bit float), like: 3.6, .8, -9.2
{word}	string	Matches one word without whitespace, like: banana (not: banana split).
{string}	string	Matches double-/single-quoted strings, for example "banana split" (not: banana split).
{}	string	Matches anything, like re_pattern = ".*"
{bigdecimal}	Decimal	Matches float, but converts to BigDecimal if platform supports it.
{double}	float	Matches float, but converts to 64-bit float number if platform supports it.
{biginteger}	int	Matches int, but converts to "BigInteger" if platform supports it.
{byte}	int	Matches int, but converts to 8-bit signed integer if platform supports it.
{short}	int	Matches int, but converts to 16-bit signed integer if platform supports it.
{long}	int	Matches int, but converts to 64-bit signed integer if platform supports it.

Use Optional Text

Sometimes, it is useful to match optional text (example: singular/plural).

Listing 125: EXAMPLE: Use optional word part

```
I have {int} cucumber(s) in my belly

# -- MATCHES:
I have 1 cucumber in my belly
I have 42 cucumbers in my belly
```

Use Alternative Words

Sometimes, it is useful to match two word alternatives.

Listing 126: EXAMPLE: Use word alternatives

```
I have {int} cucumber(s) in my belly/stomach

# -- MATCHES:
I have 1 cucumber in my belly
I have 42 cucumbers in my stomach
```

Escaping for Parenthesis/Braces

Listing 127: EXAMPLE: Use ESCAPING

```
I have {int} \{what} cucumber(s) in my belly \(amazing!)

# -- MATCHES:
I have 1 {what} cucumber in my belly (amazing!)
I have 42 {what} cucumbers in my belly (amazing!)
```

User-Defined Types

EXAMPLE 1: Number

Listing 128: FILE: features/steps/example1_number_steps.py

```
from behave import when
from behave.cucumber_expression import (
    define_parameter_type_with,
    use_step_matcher_for_cucumber_expressions
)
import parse

# -- TYPE-CONVERTER (aka: parse-function)
# HINT: Proof-of-concept (BETTER USE: {int})
@parse.with_pattern(r"\d+")
def parse_number(text):
    return int(text)

# -- DEFINE PARAMETER-TYPE:
# HINT: Each parameter type can be defined only once.
define_parameter_type_with(
    name="number",
    regexp=parse_number.pattern,
    type=int,
    transformer=parse_number
)
```

(continues on next page)

(continued from previous page)

```
use_step_matcher_for_cucumber_expressions()

# -- MATCHES STEPS:
#   When I eat 1 apple
#   When I eat 10 apples
@when('I eat {number} apple(s)')
def step_when_eat_apples(ctx, the_number: int):
    assert isinstance(the_number, int)
    assert the_number >= 0
    ctx.apples_count = the_number
```

EXAMPLE 2: Parameter Type for `Color` enum

Listing 129: FILE: example4me/color.py

```
from enum import Enum

class Color(Enum):
    red = 1
    green = 2
    blue = 3
```

Listing 130: FILE: features/steps/example2_color_steps.py

```
from behave import when
from behave.cucumber_expression import (
    TypeBuilder,    # -- BASED ON: parse_type.TypeBuilder
    define_parameter_type_with,
    use_step_matcher_for_cucumber_expressions
)
from example4me.color import Color

# -- TYPE-CONVERTER(s):
parse_color = TypeBuilder.make_enum(Color)

# -- DEFINE PARAMETER-TYPE:
define_parameter_type_with(
    name="color",
    regexp=parse_color.pattern,
    type=Color,
    transformer=parse_color
)

use_step_matcher_for_cucumber_expressions()

# -- MATCHES STEPS:
# When I use the red color
# When I use the blue color
@when('I use the {color} color')
def step_when_use_color(ctx, the_color: Color):
    assert isinstance(the_color, Color)
    ctx.selected_color = the_color
```

EXAMPLE 3: Using more than one Parameter Type in a Step Definition

- Ordering of parameters in step-function must match the ordering in the step definition text

Listing 131: FILE: features/steps/example3_colored_fruit_steps.py

```
from behave import when
from behave.cucumber_expression import (
    TypeBuilder,    # -- BASED ON: parse_type.TypeBuilder
    define_parameter_type_with,
    use_step_matcher_for_cucumber_expressions
)
from example4me.color import Color

# -- TYPE-CONVERTER(s):
SUPPORTED_FRUITS = ["apple", "banana", "pear"]
parse_color = TypeBuilder.make_enum(Color)
parse_fruit = TypeBuilder.make_choice(SUPPORTED_FRUITS)

# -- DEFINE PARAMETER-TYPE:
# HINT: Each parameter type can be defined only once.
define_parameter_type_with(
    name="color",
    regexp=parse_color.pattern,
    type=Color,
    transformer=parse_color
)
define_parameter_type_with(
    name="fruit",
    regexp=parse_fruit.pattern,
    type=str,
    transformer=parse_fruit
)

use_step_matcher_for_cucumber_expressions()

# -- MATCHES STEPS:
# When I eat the red apple
# When I eat the green banana
# When I eat the blue pear
@when('I eat the {color} {fruit}')
def step_when_eat_fruit(ctx, the_color: Color, the_fruit: str):
    assert isinstance(the_color, Color)
    assert isinstance(the_fruit, str)
    ctx.selected_fruit_combination = (the_color, the_fruit)
```

See also

Cucumber-Expressions:

- Repository: github:/cucumber/cucumber-expressions
- features/step_matcher.cucumber_expressions.feature
- parse_type: `TypeBuilder` functionality and more.

1.17.8 Regular Expressions

The following tables provide a overview of the regular expressions syntax. See also Python regular expressions description in the Python re module.

Special Characters	Description
`.`	Matches any character (dot).
`^`	"^...", matches start-of-string (caret).
`$`	"...$", matches end-of-string (dollar sign).
`|`	"A|B", matches "A" or "B".
`\`	Escape character.
`\.`	EXAMPLE: Matches character '.' (dot).
`\\`	EXAMPLE: Matches character '\' (backslash).

To select or match characters from a special set of characters, a character set must be defined.

Character sets	Description
`[...]`	Define a character set, like `[A-Za-z]`.
`\d`	Matches digit character: [0-9]
`\D`	Matches non-digit character.
`\s`	Matches whitespace character: `[ \t\n\r\f\v]`
`\S`	Matches non-whitespace character
`\w`	Matches alphanumeric character: `[a-zA-Z0-9_]`
`\W`	Matches non-alphanumeric character.

A text part must be group to extract it as part (parameter).

Grouping	Description
`(...)`	Group a regular expression pattern (anonymous group).
`\number`	Matches text of earlier group by index, like: "\1".
`(?P<name>...)`	Matches pattern and stores it in parameter "name".
`(?P=name)`	Match whatever text was matched by earlier group "name".
`(?:...)`	Matches pattern, but does non capture any text.
`(?#...)`	Comment (is ignored), describes pattern details.

If a *group*, *character* or *character set* should be repeated several times, it is necessary to specify the cardinality of the regular expression pattern.

Cardinality	Description
?	Pattern with cardinality 0..1: optional part (question mark).
*	Pattern with cardinality zero or more, 0.. (asterisk).
+	Pattern with cardinality one or more, 1.. (plus sign).
{m}	Matches m repetitions of a pattern.
{m,n}	Matches from m to n repetitions of a pattern.
[A-Za-z]+	EXAMPLE: Matches one or more alphabetical characters.

1.17.9 Testing Domains

Behave and other BDD frameworks allow you to provide **step libraries** to reuse step definitions in similar projects that address the same problem domain.

Step Libraries

Support of the following testing domains is currently known:

Testing Domain	Name	Description
Command-line	behave4cmd	Test command-line tools, like behave, etc. (coming soon).
Web Apps	behave-django	Test Django Web apps with behave (solution 1).
Web Apps	django-behave	Test Django Web apps with behave (solution 2).
Web, SMS, ...	behaving	Test Web Apps, Email, SMS, Personas (step library).

Step Usage Examples

This examples show how you can use behave for testing a specific problem domain. This examples are normally not a full-blown step library (that can be reused), but give you an example (or prototype), how the problem can be solved.

Testing Domain	Name	Description
GUI	Squish test	Use Squish and Behave for GUI testing (cross-platform).
Robot Control	behave4poppy	Use behave to control a robot via pypot.

See also

- google-search: behave python example

1.17.10 Behave Ecosystem

The following tools and extensions try to simplify the work with behave.

See also

- Are there any non-developer tools for writing Gherkin files ? (`*.feature` files)

Behave related Projects to Github

Use the following URL to find behave related projects on Github:

- https://github.com/topics/behave?l=python

Behave related Projects to pypi.org

Use the following URL to find behave related projects on the new pypi repository (supersedes: `https://pypi.python.org`):

- https://pypi.org/search/?q=behave

IDE Plugins

IDE	Plugin	Description
PyCharm	PyCharm BDD	PyCharm 4 (Professional edition) has **built-in support** for behave.
PyCharm	Gherkin	PyCharm/IDEA editor support for Gherkin.
Eclipse	Cucumber-Eclipse	Plugin contains editor support for Gherkin.
VisualStudio	cuke4vs	VisualStudio plugin with editor support for Gherkin.
VSCode	cucumber-official	Official VSCode extension for Cucumber, with Gherkin and behave support.
VSCode	gherkin-behave	Full behave support including **stages**, **step libraries**, **step refactoring**, **run/debug**, etc.

Editors and Editor Plugins

Editor	Plugin	Description
Gherkin editor	—	An editor for writing `*.feature` files.
Notepad++	NP++ gherkin	Notepad++ editor syntax highlighting for Gherkin.
Sublime Text	Cucumber (ST Bundle)	Gherkin editor support, table formatting.
Sublime Text	Behave Step Finder	Helps to navigate to steps in behave.
vim	vim-behave	vim plugin: Port of vim-cucumber to Python behave.
Neovim	behave-lsp.nvim	Neovim editor support, helps navigate between feature files and steps.

1.17.11 Software that Enhances *behave*

- Mock
- nose.tools and nose.twistedtools
- mechanize for pretending to be a browser
- selenium webdriver for actually driving a browser
- wsgi_intercept for providing more easily testable WSGI servers
- BeautifulSoup, lxml and html5lib for parsing HTML
- …

See also

- behave.example: Behave Examples and Tutorials (HTML)
- Peter Parente: BDD and Behave (tutorial)

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

B

C

D

E

F

S

T

U

V

W