behave Documentation

Release 1.2.6

Benno Rice, Richard Jones and Jens Engel

2020-12-15
## Contents

1. **Contents**
   - 1.1 Installation .................................................. 3
   - 1.2 Tutorial ...................................................... 4
   - 1.3 Behavior Driven Development ............................... 12
   - 1.4 Feature Testing Setup ..................................... 15
   - 1.5 Using *behave* .............................................. 24
   - 1.6 Behave API Reference ..................................... 30
   - 1.7 Fixtures ....................................................... 46
   - 1.8 Django Test Integration ................................... 50
   - 1.9 Flask Test Integration ..................................... 51
   - 1.10 Practical Tips on Testing ................................. 52
   - 1.11 Comparison With Other Tools .............................. 54
   - 1.12 New and Noteworthy ....................................... 55
   - 1.13 More Information about Behave ........................... 73
   - 1.14 Appendix ..................................................... 75

2. **Indices and tables**

Index ................................................................. 85
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.
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
```

As an alternative, you can also use `easy_install` to install `behave`:

```
easy_install behave  # CASE: New installation.
easy_install -U behave  # CASE: Upgrade existing installation.
```

**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:

```
python setup.py 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:
To install a tagged version from the Github repository, use:

```
ip install git+https://github.com/behave/behave@<tag>
```

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

## 1.2 Tutorial

First, **install behave**.

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

```plaintext
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:

```python
from behave import *

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

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

def 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 environment controls (code to run before and after steps, scenarios, features or the whole shooting match).

The minimum requirement for a features directory is:

<table>
<thead>
<tr>
<th>features/</th>
</tr>
</thead>
<tbody>
<tr>
<td>features/everything.feature</td>
</tr>
<tr>
<td>features/steps/</td>
</tr>
<tr>
<td>features/steps/steps.py</td>
</tr>
</tbody>
</table>

A more complex directory might look like:

<table>
<thead>
<tr>
<th>features/</th>
</tr>
</thead>
<tbody>
<tr>
<td>features/signup.feature</td>
</tr>
<tr>
<td>features/login.feature</td>
</tr>
<tr>
<td>features/account_details.feature</td>
</tr>
<tr>
<td>features/environment.py</td>
</tr>
<tr>
<td>features/steps/</td>
</tr>
<tr>
<td>features/steps/website.py</td>
</tr>
<tr>
<td>features/steps/utils.py</td>
</tr>
</tbody>
</table>

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

(continues on next page)
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
<table>
<thead>
<tr>
<th>thing</th>
<th>other thing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red Tree Frog</td>
<td>mush</td>
</tr>
</tbody>
</table>

Examples: Consumer Electronics
<table>
<thead>
<tr>
<th>thing</th>
<th>other thing</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPhone</td>
<td>toxic waste</td>
</tr>
<tr>
<td>Galaxy Nexus</td>
<td>toxic waste</td>
</tr>
</tbody>
</table>

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
<table>
<thead>
<tr>
<th>name</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barry</td>
<td>Beer Cans</td>
</tr>
<tr>
<td>Pudey</td>
<td>Silly Walks</td>
</tr>
<tr>
<td>Two-Lumps</td>
<td>Silly Walks</td>
</tr>
</tbody>
</table>

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
    Given 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):

@given('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**

You may find that your feature steps sometimes include very common phrases with only some variation. For example:

```plaintext
Scenario: look up a book
  Given I search for a valid book
  Then the result page will include "success"

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

You may define a single Python step that handles both of those Then clauses (with a Given step that puts some text into `context.response`):

```python
@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 parsers available in `behave` (by default):

- **parse** *(the default, based on: parse)* Provides a simple parser that replaces regular expressions for step parameters with a readable syntax like `{param:Type}`. The syntax is inspired by the Python built-in `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.
  
  Supports type conversions by using type converters (see `register_type()`).

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

  Supports type conversions (as above).

- **re** This uses full regular expressions to parse the clause text. You will need to use named groups “(?P<name>...)” to define the variables pulled from the text and passed to your `step()` function.

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

To specify which parser to use invoke `use_step_matcher()` with the name of the matcher to use. You may change matcher to suit specific step functions - the last call to `use_step_matcher` before a step function declaration will be the one it uses.

**Note:** The function `step_matcher()` is becoming deprecated. Use `use_step_matcher()` instead.
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:

```python
@given('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:
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

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=-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,slow  This will select all the cases tagged either “wip” or “slow”.
• --tags=wip --tags=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 @browser:

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

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

```python
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.

```python
# -- 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

```python
# -- 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 exemplar, 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 mandated.

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 an aspect of behavior of a list:

# 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:

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

    def test_populated_list_is_true(self):
        list = []
        list.append('item')
        assertEqual(bool(list), True)
```
Sometimes the difference between the context, events and outcomes is made more explicit. For instance:

```python
class TestWindow(object):
    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/
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

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

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 (e.g., 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 needs 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 practice 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:**

<table>
<thead>
<tr>
<th>phenomenon</th>
<th>thing</th>
<th>other thing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amphibians</td>
<td>Red Tree Frog</td>
<td>mush</td>
</tr>
</tbody>
</table>

**Examples:** Consumer Electronics

<table>
<thead>
<tr>
<th>phenomenon</th>
<th>thing</th>
<th>other thing</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPhone</td>
<td>toxic waste</td>
<td></td>
</tr>
<tr>
<td>Galaxy Nexus</td>
<td>toxic waste</td>
<td></td>
</tr>
</tbody>
</table>

*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 "http://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:

```plaintext
Scenario: Multiple Givens
  Given one thing
  Given an other thing
  Given yet an other 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:

```plaintext
Scenario: Multiple Givens
  Given one thing
  And an other thing
  And yet an other 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 |

  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:

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

For example:

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

or:

```plaintext
@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:

```plaintext
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
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=-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,slow` This will select all the cases tagged *either* “wip” or “slow”.
- `--tags=wip --tags=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 `@browser`:

```python
def before_feature(context, feature):
    model.init(environment='test')
    if 'browser' in feature.tags:
        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()
        context.browser = webdriver.Chrome()

def after_feature(context, feature):
    if 'browser' in feature.tags:
        context.server.shutdown()
        context.thread.join()
        context.browser.quit()
```

### 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):
   ```
   # 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 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.

1.5.1 Command-Line Arguments

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

- `--no-color`
  Disable the use of ANSI color escapes.

- `--color`
  Use ANSI color escapes. This is the default behaviour. This switch is used to override a configuration file setting.

- `--dry-run`
  Invokes formatters without executing the steps.

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

- `--exclude`
  Don’t run feature files matching regular expression PATTERN.

- `--include`
  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`
  Directory in which to store JUnit reports.

- `--format`
  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.

-m, --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
Only execute the feature elements which match part of the given name. If this option is given more than once, it will match against all the given names.

--no-capture
Don’t capture stdout (any stdout output will be printed immediately.)

--capture
Capture stdout (any stdout output will be printed if there is a failure.) This is the default behaviour. This switch is used to override a configuration file setting.

--no-capture-stderr
Don’t capture stderr (any stderr output will be printed immediately.)

--capture-stderr
Capture stderr (any stderr output will be printed if there is a failure.) This is the default behaviour. This switch is used to override a configuration file setting.

--no-logcapture
Don’t capture logging. Logging configuration will be left intact.

--logcapture
Capture logging. All logging during a step will be captured and displayed in the event of a failure. This is the default behaviour. This switch is used to override a configuration file setting.

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

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

--logging-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 all other logging handlers.

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

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

-o, --outfile
Write to specified file instead of stdout.
-q, --quiet
Alias for --no-snippets --no-source.

-s, --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
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, --tags
Only execute features or scenarios with tags matching TAG_EXPRESSION. Pass “--tags-help” 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.

-x, --expand
Expand scenario outline tables in output.

--lang
Use keywords for a language other than English.

--lang-list
List the languages available for --lang.

--lang-help
List the translations accepted for one language.

--tags-help
Show help for tag expressions.

--version
Show version.

Tag Expression
Scenarios inherit tags declared on the Feature level. The simplest TAG_EXPRESSION is simply a tag:

```
--tags @dev
```

You may even leave off the “@” - behave doesn’t mind.

When a tag in a tag expression starts with a ~, this represents boolean NOT:

```
--tags ~@dev
```

A tag expression can have several tags separated by a comma, which represents logical OR:
The `--tags` option can be specified several times, and this represents logical AND, for instance this represents the boolean expression "(@foo or not @bar) and @zap":

```
--tags @foo,~@bar --tags @zap.
```

Beware that if you want to use several negative tags to exclude several tags you have to use logical AND:

```
--tags ~@fixme --tags ~@buggy.
```

### 1.5.2 Configuration Files

Configuration files for `behave` are called either “.behaverc”, “behave.ini”, “setup.cfg” or “tox.ini” (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]
format=plain
logging_clear_handlers=yes
logging_filter=-suds
```

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

  ```
  tags=@foo,~@bar
  @zap
  ```

  which is the equivalent of the command-line usage:

  ```
  --tags @foo,~@bar --tags @zap
  ```

### Configuration Parameters

- **color**  :  **bool**
  Use ANSI color escapes. This is the default behaviour. This switch is used to override a configuration file setting.
- **dry_run**  :  **bool**
  Invokes formatters without executing the steps.
userdata_defines : sequence<text>
  Define user-specific data for the config.userdata dictionary. Example: -D foo=bar to store it in con-
  fig.userdata["foo"].

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.

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}’).

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>
  Only execute the feature elements which match part of the given name. If this option is given more than
  once, it will match against all the given names.

stdout_capture : bool
  Capture stdout (any stdout output will be printed if there is a failure.) This is the default behaviour. This
  switch is used to override a configuration file setting.

stderr_capture : bool
  Capture stderr (any stderr output will be printed if there is a failure.) This is the default behaviour. This
  switch is used to override a configuration file setting.

log_capture : bool
  Capture logging. All logging during a step will be captured and displayed in the event of a failure. This is
  the default behaviour. This switch is used to override a configuration file setting.

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 all other logging handlers.

summary : bool
   Display the summary at the end of the run.

outfiles : sequence<text>
   Write to specified file instead of stdout.

paths : sequence<text>
   Specify default feature paths, used when none are provided.

quiet : bool

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
   Define default tags when non are provided. See –tags for more information.

tags : sequence<text>
   Only execute certain features or scenarios based on the tag expression given. 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.

expand : bool
   Expand scenario outline tables in output.

lang : text
   Use keywords for a language other than English.
1.6 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.

1.6.1 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:

```python
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 exactly. So the following step implementation code:

```python
@given('some known state')
def step_impl(context):
    set_up(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.

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”:

```python
@given('some other known state')
def step_impl(context):
    set_up(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)
Change the parameter matcher used in parsing step text.

The change is immediate and may be performed between step definitions in your step implementation
modules - allowing adjacent steps to use different matchers if necessary.

There are several parsers available in behave (by default):

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

Supports type conversions by using type converters (see register_type()).

cfparse (extends: parse, requires: parse_type) Provides an extended parser with “Cardinality Field”
(CF) support. Automatically creates missing type converters for related cardinality as long as a type
converter for cardinality=1 is provided. Supports parse expressions like:

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

Supports type conversions (as above).

re This uses full regular expressions to parse the clause text. You will need to use named groups
“(?:<name>..)” to define the variables pulled from the text and passed to your step() function.

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

You may define your own matcher.

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

behave.register_type(**kw)
Registers a custom type that will be available to “parse” for type conversion during step matching.

Converters should be supplied as name=callable arguments (or as dict).

A type converter should follow parse module rules. In general, a type converter is a function that converts
text (as string) into a value-type (type converted value).

EXAMPLE:

from behave import register_type, given
import parse

# -- TYPE CONVERTER: For a simple, positive integer number.
@parse.with_pattern(r"\d+")
def parse_number(text):
    return int(text)

(continues on next page)
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)

Pull parameters out of step names.

pattern
The match pattern attached to the step function.

func
The step function the pattern is being attached to.

check_match (step)
Match me against the "step" name supplied.
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`.

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).

regex_pattern
Return the used textual regex pattern.

class behave.model_core.Argument (start, end, original, value, name=None)
An argument found in a feature file step name and extracted using step decorator parameters.
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 feature file step name extracted using step decorator parameters.

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.
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:

```python
@when('I do the same thing as before')
def step_impl(context):
    context.execute_steps(u'''
      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.

```python
from behave import *
```

The import statement:

```python
from behave import *
```

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>given, when, then, step</td>
<td>Decorator</td>
<td>Decorators for step implementations.</td>
</tr>
<tr>
<td>use_step_matcher(name)</td>
<td>Function</td>
<td>Selects current step matcher (parser).</td>
</tr>
<tr>
<td>register_type(Type=func)</td>
<td>Function</td>
<td>Registers a type converter.</td>
</tr>
</tbody>
</table>

See also the description in `step parameters`.

1.6.2 Environment File Functions

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. The step passed in is an instance of `Step`.

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

`before_feature(context, feature)`, `after_feature(context, feature)` These run before and after each feature file is exercised. The feature passed in is an instance of `Feature`.

`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`. The tag passed in is an instance of `Tag` and because it’s a subclass of string you can do simple tests like:

```python
# -- ASSUMING: tags @browser.chrome or @browser.any are used.
if tag.startswith("browser."):  
    browser_type = tag.replace("browser.", ",", 1)  
    if browser_type == "chrome":  
        context.browser = webdriver.Chrome()  
    else:  
        context.browser = webdriver.PlainVanilla()
```

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

```python
# -- 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:

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

```python
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.

**log_capture**
If logging capture is enabled then this attribute contains the captured logging as an instance of `LoggingCapture`. It is not present if logging is not being captured.

**stdout_capture**
If stdout capture is enabled then this attribute contains the captured output as a `StringIO` instance. It is not present if stdout is not being captured.

**stderr_capture**
If stderr capture is enabled then this attribute contains the captured output as a `StringIO` instance. It is not present if stderr is not being captured.

If an attempt made by user code to overwrite one of these variables, or indeed by `behave` to overwrite a user-set variable, then a `behave.runner.ContextMaskWarning` warning will be raised.

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.

**add_cleanup** (`cleanup_func`, *args, **kwargs)
Adds a cleanup function that is called when `Context._pop()` is called. This is intended for user-cleanup.

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

**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.


behave Documentation, Release 1.2.6

**Raises** ValueError, if invoked without a feature context.

```python
use_with_user_mode()
```

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

```python
class behave.runner.ContextMaskWarning
```

Raisd 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.

### 1.6.3 Fixtures

**Provide a Fixture**

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

Fixture decorator (currently mostly syntactic sugar).

```python
# -- 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-cleanups.
@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 `@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).

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

(continues on next page)
@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
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.

```python
# -- 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
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.

### 1.6.4 Runner Operation

Given all the code that could be run by `behave`, this is the order in which that code is invoked (if they exist.)
before_all
for feature in all_features:
    before_feature
    for scenario in feature.scenarios:
        before_scenario
        for step in scenario.steps:
            before_step
            step.run()
            after_step
            after_scenario
        after_scenario
    after_feature
after_all

If the feature contains scenario outlines then there is an additional loop over all the scenarios in the outline making the running look like this:

before_all
for feature in all_features:
    before_feature
    for outline in feature.scenarios:
        for scenario in outline.scenarios:
            before_scenario
            for step in scenario.steps:
                before_step
                step.run()
                after_step
                after_scenario
            after_scenario
        after_scenario
    after_feature
after_all

1.6.5 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:

<table>
<thead>
<tr>
<th>Tag (as Feature.tags)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature : TaggableModelElement</td>
</tr>
<tr>
<td>Description (as Feature.description)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Background</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step</td>
</tr>
<tr>
<td>Table (as Step.table)</td>
</tr>
<tr>
<td>MultiLineText (as Step.text)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag (as Scenario.tags)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario : TaggableModelElement</td>
</tr>
<tr>
<td>Description (as Scenario.description)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Tag (as ScenarioOutline.tags)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScenarioOutline : TaggableModelElement</td>
</tr>
</tbody>
</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 occured while running this feature.

New 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.

New in version 1.2.6.
class behave.model.Background(filename, line, keyword, name, steps=None)
A background 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 “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.

class behave.model.Scenario(filename, line, keyword, name, tags=None, steps=None, description=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.

**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 occured while running this scenario.

New 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.

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.

```python
class behave.model.Tag
```
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](#).*

```python
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 occured while running this step.

*New 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:

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

Tables are also comparable, for what that’s worth. Heads and row data are compared.

```python
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.

**indexed access**  Individual cells may be accessed by heading name; row[“name”] would give the cell value for the column with heading “name”.

**named 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:

```python
class behave.model.Text
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”.

---

Chapter 1. Contents
## 1.6.6 Logging Capture

The logging capture `behave` uses by default is implemented by the class `LoggingCapture`. It has methods defined by `behave.log_capture.LoggingCapture(config, level=None)`.

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

`LoggingCapture(config, level=None)`

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:

  The function prints any captured logging (at the level determined by the `log_level` configuration setting) directly to stdout, regardless of error conditions.

  It is mostly useful for debugging in situations where you are seeing a message like:
The decorator takes an optional "level" keyword argument which limits the level of logging captured, over-
riding 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.

1.7 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.

1.7.1 Providing a Fixture

```python
# -- 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):
    # -- 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

1.7.2 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).
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.

Realistic Example

A more realistic example by using a fixture registry is shown below:
# -- 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 understandability in feature files (Gherkin).

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

---

### 1.7.3 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).

<table>
<thead>
<tr>
<th>Context Layer</th>
<th>Fixture-Setup Point</th>
<th>Fixture-Cleanup Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>test run</td>
<td>In before_all()</td>
<td>After after_all() at end of test-run.</td>
</tr>
<tr>
<td>feature</td>
<td>In before_feature()</td>
<td>After after_feature() at end of feature.</td>
</tr>
<tr>
<td>feature</td>
<td>In before_tag()</td>
<td>After after_feature() for feature tag.</td>
</tr>
<tr>
<td>scenario</td>
<td>In before_scenario()</td>
<td>After after_scenario() at end of scenario.</td>
</tr>
<tr>
<td>scenario</td>
<td>In a step</td>
<td>After after_scenario() for scenario tag.</td>
</tr>
</tbody>
</table>
| scenario      |                     | After after_scenario() for scenario tag. | Fixure is usable until end of sce-

---

### 1.7.4 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)
1.7.5 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:

```python
# -- 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:
        browser = FirefoxBrowser(timeout, **kwargs)
        browser.part2 = setup_fixture_part2_with_error("OOPS")
        context.browser = browser  # NOT_REACHED
        yield browser
    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  # NOT_REACHED
    context.add_cleanup(browser.shutdown)  # -- ENSURE-CLEANUP EARLY
    browser.part2 = setup_fixture_part2_with_error("OOPS")
    return browser  # NOT_REACHED
```

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.

1.7.6 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:

```python
# -- 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

1.8 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.

1.8.1 Manual Integration

Alternatively, you can integrate Django using the following boilerplate code in your environment.py file:
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 from Andrey Zarubin’s blog post “BDD. PyCharm + Python & Django”.

1.8.2 Strategies and Tooling

See Practical Tips on Testing for automation libraries and implementation tips on your BDD tests.

1.9 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.

1.9.1 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
def flaskr_client(context, *args, **kwargs):

(continues on next page)
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 Flask.

1.9.2 Strategies and Tooling

See Practical Tips on Testing for automation libraries and implementation tips on your BDD tests.

1.10 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.

1.10.1 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.

1.10.2 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:

```python
# -- 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:

```python
# -- FILE: features/steps/browser_steps.py
from behave import given, when, then
@when(u'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:

```python
# -- 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:

```python
# -- FILE: features/steps/browser_steps.py
from behave import given, when, then
@when(u'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](https://example.com).

1.11 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.

1.11.1 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.

1.11.2 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.

1.11.3 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.
behave Documentation, Release 1.2.6

• 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.

1.12 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.

1.12.1 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 asynchronous frameworks or protocols (asyncio based)
• Context-cleanup: 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:

```plaintext
# -- 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:

```plaintext
# -- FOR scenario in scenario_outline.scenarios:
scenario.tags = scenario_outline.tags + examples.tags
```

To run only the first Examples section, you use:

```
behave --tags=@develop features/tagged_examples.feature
```

Tagged Examples with Active Tags and Userdata

An even more natural fit is to use tagged examples together with active tags and userdata:

```plaintext
# -- 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      |
```

(continues on next page)
Select the Examples section now by using:

```bash
# -- VARIANT 1: Use userdata
behave --D stage=integration features/tagged_examples2.feature
# -- VARIANT 2: Use stage mechanism
behave --stage=integration features/tagged_examples2.feature
```

```python
# -- 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)

def before_scenario(context, scenario):
    if active_tag_matcher.should_exclude_with(scenario.effective_tags):
        sys.stdout.write("ACTIVE-TAG DISABLED: Scenario %s\n" % scenario.name)
        scenario.skip(active_tag_matcher.exclude_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:**

```
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"
```
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:**

```plaintext
# -- FILE: features/french_1.feature
# language: fr
Fonctionnalité: Alice
...
```

```plaintext
# -- 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:**

```plaintext
# -- FILE: behave.ini
# Exclude/skip any feature/scenario with @xfail or @not_implemented tags
[behave]
default_tags = -@xfail -@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:**
# -- 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 asynchronous 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

# -- FILE: features/steps/async_steps35.py
# -- REQUIRES: Python >= 3.5
from behave import step
from behave.api.async_step import async_run_until_complete
import asyncio

@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)""
    await asyncio.sleep(duration)
When you use the async-step from above in a feature file and run it with behave:

```
$ behave -f plain features/async_run.feature
Feature:
    Scenario:
        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:

```
# -- FILE: features/async_dispatch.feature
@use.with_python.version=3.4
@use.with_python.version=3.5
@use.with_python.version=3.6
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:

```
# -- 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.
```

(continues on next page)
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:

```python
# -- FILE: features/steps/async_dispatch_steps.py
# REQUIRES: Python 3.4 or newer
# -*- coding: UTF-8 -*-
# REQUIRES: Python >= 3.5
from behave import given, then, step
from behave.api.async_step import use_or_create_async_context, AsyncContext
from hamcrest import assert_that, equal_to, empty
import asyncio

@asyncio.coroutine
def async_func(param):
yield from asyncio.sleep(0.2)
return str(param).upper()

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
    done, pending = async_context.loop.run_until_complete(
        asyncio.wait(async_context.tasks, loop=async_context.loop))

    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
- ...

```python
# -- 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 context layer when the cleanup function was registered (test-run, feature, scenario).

Example:

```python
# -- 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

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

Using a Fixture

```python
# -- 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
```
See also:

- *Fixtures* description for details
- *features.fixture.feature*

## 1.12.2 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: Charly
  | 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:
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”:

```ini
# -- 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.

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>examples.name</td>
<td>Refers name of the example group, may be an empty string.</td>
</tr>
<tr>
<td>examples.index</td>
<td>Index of the example group (range=1..N).</td>
</tr>
<tr>
<td>row.index</td>
<td>Index of the current row within an example group (range=1..R).</td>
</tr>
<tr>
<td>row.id</td>
<td>Shortcut for schema: “&lt;examples.index&gt;.&lt;row.index&gt;”</td>
</tr>
</tbody>
</table>

**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:**

```
# -- file:features/xxx.feature
Feature:
  Scenario Outline: Wow <name>-% <birthyear>  # line 2
  Given an employee "<name>"

Examples:
<table>
<thead>
<tr>
<th>name</th>
<th>birthyear</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1985</td>
</tr>
<tr>
<td>Bob</td>
<td>1975</td>
</tr>
</tbody>
</table>

Examples: Benares-%<ID>
```
This leads to the following generated scenarios, one for each examples/row combination:

<table>
<thead>
<tr>
<th>name</th>
<th>birthyear</th>
<th>ID</th>
<th># line 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charly</td>
<td>1995</td>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

Scenario Outline: Wow Alice-1985 -- @1.1 # features/xxx.feature:7
Given an employee "Alice"

Scenario Outline: Wow Bob-1975 -- @1.2 # features/xxx.feature:8
Given an employee "Bob"

Scenario Outline: Wow Charly-1885 -- @2.1 Benares-42 # features/xxx.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:

```feature
# -- file:features/xxx.feature
Feature:
@foo.group<examples.index>
@foo.row<row.id>
@foo.name.<name>
Scenario Outline: Wow # line 6
Given an employee "<name>"

Examples:
<table>
<thead>
<tr>
<th>name</th>
<th>birthyear</th>
<th>ID</th>
<th># line 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>1985</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Examples: Benares
<table>
<thead>
<tr>
<th>name</th>
<th>birthyear</th>
<th>ID</th>
<th># line 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>1975</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Charly</td>
<td>1995</td>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>
```

This leads to the following generated scenarios, one for each examples/row combination:

@foo.group1 @foo.row1.1 @foo.name.Alice
Scenario Outline: Wow -- @1.1 Araxas # features/xxx.feature:11
Given an employee "Araxas"

@foo.group1 @foo.row1.2 @foo.name.Bob
Scenario Outline: Wow -- @1.2 Araxas # features/xxx.feature:12
Given an employee "Bob"

@foo.group2 @foo.row2.1 @foo.name.Charly

(continues on next page)
Scenario Outline: Wow -- @2.1 Benares # features/xxx.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:

```
$ behave --tags=@foo.group1 -f progress3 features/xxx.feature
... # features/xxx.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:

```
# -- 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: Benares
    | name | birthyear |
    | Charly | 1995 | # line 12
```

This leads to the following generated scenarios (when the feature is executed):

```
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 Benares # features/xxx.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):

```
$ behave --name=Araxas -f progress3 features/xxx.feature
... # features/xxx.feature
Wow -- @1.1 Araxas .
Wow -- @1.2 Araxas .
```

```
$ behave --name='-- @.* Araxas' -f progress3 features/xxx.feature
... # features/xxx.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.

```python
# -- 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
    # * 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)
```

```python
# -- 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)
- systemint (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.
```

To use the stage=testlab, you run behave with:

```
behave --stage=testlab ...
```

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

**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.

```
# -- FILE: behave.ini
[behave.userdata]
browser = firefox
server = asterix
```

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:

```python
# -- 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.

```python
# -- 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:

```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 Tag Schema

The following two tag schemas are supported for active tags (by default).

Dialect 1:
• @active.with_{category}={value}
• @not_active.with_{category}={value}

Dialect 2:
• @use.with_{category}={value}
• @not.with_{category}={value}
• @only.with_{category}={value}

Example 1

Assuming you have the feature file where:
• scenario “Alice” should only run when browser “Chrome” is used
• scenario “Bob” should only run when browser “Safari” is used
# -- 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
  ...

# -- 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):
  # -- NOTE: scenario.effective_tags := scenario.tags + feature.tags
  if active_tag_matcher.should_exclude_with(scenario.effective_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).

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

# -- 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
  ...
By using the *userdata* mechanism, you can now define on command-line which browser should be used when you run behave.

```
# -- SHELL: Run behave with browser=safari, ... by using userdata.
# TEST VARIANT 1: Run tests with browser=safari
behave -D browser=safari features/
# 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)
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 ...
```

```python
# -- FILE: my/own_module.py
# (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:

```ini
# -- FILE: behave.ini
[behave.formatters]
foo = behave_contrib.formatter.foo:FooFormatter
bar = behave_contrib.formatter.bar:BarFormatter
```

```python
# -- 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:

```
# -- NOTE: Use FooFormatter that was registered by name "foo".
behave -f foo ...
```

1.12.3 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).

1.13 More Information about Behave

1.13.1 Tutorials

For new users, that want to read, understand and explore the concepts in Gherkin and behave (after reading the behave documentation):
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*
- Bdd with Python, Behave and WebDriver

**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.

### 1.13.2 Books

Behave is covered in the following books:


### 1.13.3 Presentation Videos

- Benno Rice: *Making Your Application Behave* (30min), 2012-08-12, PyCon Australia.
- Jessica Ingrasselino: *Automation with Python and Behave* (67min), 2015-12-16
- Selenium Python Webdriver Tutorial - Behave (BDD) (14min), 2016-01-21

### 1.13.4 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 2016.1

### 1.13.5 Find more Information

See also:

- google:python-behave examples
- google:python-behave tutorials
1.14 Appendix

Contents:

1.14.1 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>junit</td>
<td>Provides JUnit XML-like output.</td>
</tr>
<tr>
<td>summary</td>
<td>Provides a summary of the test run.</td>
</tr>
</tbody>
</table>

Formatters

The following formatters are currently supported:

<table>
<thead>
<tr>
<th>Name</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td>normal</td>
<td>Shows all registered formatters.</td>
</tr>
<tr>
<td>json</td>
<td>normal</td>
<td>JSON dump of test run</td>
</tr>
<tr>
<td>json.pretty</td>
<td>normal</td>
<td>JSON dump of test run (human readable)</td>
</tr>
<tr>
<td>plain</td>
<td>normal</td>
<td>Very basic formatter with maximum compatibility</td>
</tr>
<tr>
<td>pretty</td>
<td>normal</td>
<td>Standard colourised pretty formatter</td>
</tr>
<tr>
<td>progress</td>
<td>normal</td>
<td>Shows dotted progress for each executed scenario.</td>
</tr>
<tr>
<td>progress2</td>
<td>normal</td>
<td>Shows dotted progress for each executed step.</td>
</tr>
<tr>
<td>progress3</td>
<td>normal</td>
<td>Shows detailed progress for each step of a scenario.</td>
</tr>
<tr>
<td>rerun</td>
<td>normal</td>
<td>Emits scenario file locations of failing scenarios</td>
</tr>
<tr>
<td>sphinx.steps</td>
<td>dry-run</td>
<td>Generate sphinx-based documentation for step definitions.</td>
</tr>
<tr>
<td>steps</td>
<td>dry-run</td>
<td>Shows step definitions (step implementations).</td>
</tr>
<tr>
<td>steps.doc</td>
<td>dry-run</td>
<td>Shows documentation for step definitions.</td>
</tr>
<tr>
<td>steps.usage</td>
<td>dry-run</td>
<td>Shows how step definitions are used by steps (in feature files).</td>
</tr>
<tr>
<td>tags</td>
<td>dry-run</td>
<td>Shows tags (and how often they are used).</td>
</tr>
<tr>
<td>tags.location</td>
<td>dry-run</td>
<td>Shows tags and the location where they are used.</td>
</tr>
</tbody>
</table>

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 (\(--\text{dry-run} \) command-line option) or normal mode.
User-Defined Formatters

Behave allows you to provide your own formatter (class):

```python
# -- 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:

```ini
# -- 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:

```ini
# -- FILE: behave.ini
# SCHEMA: behave.formatter.<FORMATTER_NAME>.<ATTRIBUTE_NAME>
[behave.userdata]
behave.formatter.json2.use_pretty = true
```

More Formatters

The following formatters are currently known:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allure</td>
<td>allure-behave, an Allure formatter for behave: allure_behave.formatter:AllureFormatter</td>
</tr>
<tr>
<td>teamcity</td>
<td>behave-teamcity, a formatter for JetBrains TeamCity CI testruns with behave.</td>
</tr>
</tbody>
</table>

```ini
# -- FILE: behave.ini
# FORMATTER ALIASES: behave -f allure ...
[behave.formatters]
allure = allure_behave.formatter:AllureFormatter
teamcity = behave_teamcity:TeamcityFormatter
```

1.14.2 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

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Layer</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>config</td>
<td>test run</td>
<td>Configuration</td>
<td>Configuration that is used.</td>
</tr>
<tr>
<td>aborted</td>
<td>test run</td>
<td>bool</td>
<td>Set to true if test run is aborted by the user.</td>
</tr>
<tr>
<td>failed</td>
<td>test run</td>
<td>bool</td>
<td>Set to true if a step fails.</td>
</tr>
<tr>
<td>feature</td>
<td>feature</td>
<td>Feature</td>
<td>Current feature.</td>
</tr>
<tr>
<td>tags</td>
<td>feature, scenario</td>
<td>list&lt;Tag&gt;</td>
<td>Effective tags of current feature, scenario, scenario outline.</td>
</tr>
<tr>
<td>active_outline</td>
<td>scenario outline</td>
<td>Row</td>
<td>Current row in a scenario outline (in examples table).</td>
</tr>
<tr>
<td>scenario</td>
<td>scenario</td>
<td>Scenario</td>
<td>Current scenario.</td>
</tr>
<tr>
<td>log_capture</td>
<td>scenario</td>
<td>LoggingCapture</td>
<td>If logging capture is enabled.</td>
</tr>
<tr>
<td>stdout_capture</td>
<td>scenario</td>
<td>StringIO</td>
<td>If stdout capture is enabled.</td>
</tr>
<tr>
<td>stderr_capture</td>
<td>scenario</td>
<td>StringIO</td>
<td>If stderr capture is enabled.</td>
</tr>
<tr>
<td>table</td>
<td>step</td>
<td>Table</td>
<td>Contains step’s table, otherwise None.</td>
</tr>
<tr>
<td>text</td>
<td>step</td>
<td>String</td>
<td>Contains step’s multi-line text (unicode), otherwise None.</td>
</tr>
</tbody>
</table>

**Note:** *Behave attributes* in the context object should not be modified by a user. See *Context* class description for more details.

### 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.

<table>
<thead>
<tr>
<th>Kind</th>
<th>Assign Location</th>
<th>Lifecycle Layer (Scope)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hook</td>
<td>before_all()</td>
<td>test run</td>
</tr>
<tr>
<td>Hook</td>
<td>after_all()</td>
<td>test run</td>
</tr>
<tr>
<td>Hook</td>
<td>before_tags()</td>
<td>feature or scenario</td>
</tr>
<tr>
<td>Hook</td>
<td>after_tags()</td>
<td>feature or scenario</td>
</tr>
<tr>
<td>Hook</td>
<td>before_feature()</td>
<td>feature</td>
</tr>
<tr>
<td>Hook</td>
<td>after_feature()</td>
<td>feature</td>
</tr>
<tr>
<td>Hook</td>
<td>before_scenario()</td>
<td>scenario</td>
</tr>
<tr>
<td>Hook</td>
<td>after_scenario()</td>
<td>scenario</td>
</tr>
<tr>
<td>Hook</td>
<td>before_step()</td>
<td>scenario</td>
</tr>
<tr>
<td>Hook</td>
<td>after_step()</td>
<td>scenario</td>
</tr>
<tr>
<td>Step</td>
<td>Step definition</td>
<td>scenario</td>
</tr>
</tbody>
</table>

### 1.14.3 Predefined Data Types in parse

*behave* uses the *parse* module (inverse of Python *string.format*) under the hoods to parse parameters in step definitions. This leads to rather simple and readable parse expressions for step parameters.

```python
# -- FILE: features/steps/type_transform_example_steps.py
from behave import given
@given('I have {number:d} friends')  # < Convert 'number' into int type.
def step_given_i_have_number_friends(context, number):
    assert number > 0
... ```
Therefore, the following parse types are already supported in step definitions without registration of any user-defined type:

<table>
<thead>
<tr>
<th>Type</th>
<th>Characters Matched</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>Letters and underscore</td>
<td>str</td>
</tr>
<tr>
<td>W</td>
<td>Non-letter and underscore</td>
<td>str</td>
</tr>
<tr>
<td>s</td>
<td>Whitespace</td>
<td>str</td>
</tr>
<tr>
<td>S</td>
<td>Non-whitespace</td>
<td>str</td>
</tr>
<tr>
<td>d</td>
<td>Digits (effectively integer numbers)</td>
<td>int</td>
</tr>
<tr>
<td>D</td>
<td>Non-digit</td>
<td>str</td>
</tr>
<tr>
<td>n</td>
<td>Numbers with thousands separators (, or .)</td>
<td>int</td>
</tr>
<tr>
<td>%</td>
<td>Percentage (converted to value/100.0)</td>
<td>float</td>
</tr>
<tr>
<td>f</td>
<td>Fixed-point numbers</td>
<td>float</td>
</tr>
<tr>
<td>e</td>
<td>Floating-point numbers with exponent e.g. 1.1e-10, NAN (all case insensitive)</td>
<td>float</td>
</tr>
<tr>
<td>g</td>
<td>General number format (either d, f or e)</td>
<td>float</td>
</tr>
<tr>
<td>b</td>
<td>Binary numbers</td>
<td>int</td>
</tr>
<tr>
<td>o</td>
<td>Octal numbers</td>
<td>int</td>
</tr>
<tr>
<td>x</td>
<td>Hexadecimal numbers (lower and upper case)</td>
<td>int</td>
</tr>
<tr>
<td>ti</td>
<td>ISO 8601 format date/time e.g. 1972-01-20T10:21:36Z</td>
<td>datetime</td>
</tr>
<tr>
<td>te</td>
<td>RFC2822 e-mail format date/time e.g. Mon, 20 Jan 1972 10:21:36 +1000</td>
<td>datetime</td>
</tr>
<tr>
<td>tg</td>
<td>Global (day/month) format date/time e.g. 20/1/1972 10:21:36 AM +1:00</td>
<td>datetime</td>
</tr>
<tr>
<td>ta</td>
<td>US (month/day) format date/time e.g. 1/20/1972 10:21:36 PM +10:30</td>
<td>datetime</td>
</tr>
<tr>
<td>tc</td>
<td>ctime() format date/time e.g. Sun Sep 16 01:03:52 1973</td>
<td>datetime</td>
</tr>
<tr>
<td>th</td>
<td>HTTP log format date/time e.g. 21/Nov/2011:00:07:11 +0000</td>
<td>datetime</td>
</tr>
<tr>
<td>tt</td>
<td>Time e.g. 10:21:36 PM -5:30</td>
<td>time</td>
</tr>
</tbody>
</table>

### 1.14.4 Regular Expressions

The following tables provide an overview of the regular expressions syntax. See also Python regular expressions description in the Python re module.

<table>
<thead>
<tr>
<th>Special Characters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Matches any character (dot).</td>
</tr>
<tr>
<td>^</td>
<td><strong>“^”, matches start-of-string (caret).</strong></td>
</tr>
<tr>
<td>$</td>
<td><strong>“$”, matches end-of-string (dollar sign).</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>\</td>
<td><strong>Escape character.</strong></td>
</tr>
<tr>
<td>.\</td>
<td><strong>EXAMPLE: Matches character ‘.’ (dot).</strong></td>
</tr>
<tr>
<td>\</td>
<td><strong>EXAMPLE: Matches character ‘\’ (backslash).</strong></td>
</tr>
</tbody>
</table>

To select or match characters from a special set of characters, a character set must be defined.

<table>
<thead>
<tr>
<th>Character sets</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[…]</td>
<td>Define a character set, like [A-Za-z].</td>
</tr>
<tr>
<td>\d</td>
<td>Matches digit character: [0-9]</td>
</tr>
<tr>
<td>\D</td>
<td>Matches non-digit character.</td>
</tr>
<tr>
<td>\s</td>
<td>Matches whitespace character: [ \t\n\r\f\v]</td>
</tr>
<tr>
<td>\S</td>
<td>Matches non-whitespace character</td>
</tr>
<tr>
<td>\w</td>
<td>Matches alphanumeric character: [a-zA-Z0-9_.]</td>
</tr>
<tr>
<td>\W</td>
<td>Matches non-alphanumeric character.</td>
</tr>
</tbody>
</table>

A text part must be group to extract it as part (parameter).
Grouping

<table>
<thead>
<tr>
<th>Description</th>
<th>Grouping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group a regular expression pattern (anonymous group).</td>
<td>...</td>
</tr>
<tr>
<td>Matches text of earlier group by index, like: “\1”.</td>
<td>\number</td>
</tr>
<tr>
<td>Matches pattern and stores it in parameter “name”.</td>
<td>(?P&lt;name&gt;...)</td>
</tr>
<tr>
<td>Match whatever text was matched by earlier group “name”.</td>
<td>(?P=name)</td>
</tr>
<tr>
<td>Matches pattern, but does non capture any text.</td>
<td>(?)....</td>
</tr>
<tr>
<td>Comment (is ignored), describes pattern details.</td>
<td>(#....)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern with cardinality 0..1: optional part (question mark).</td>
<td>?</td>
</tr>
<tr>
<td>Pattern with cardinality zero or more, 0.. (asterisk).</td>
<td>*</td>
</tr>
<tr>
<td>Pattern with cardinality one or more, 1.. (plus sign).</td>
<td>+</td>
</tr>
<tr>
<td>Matches m repetitions of a pattern.</td>
<td>{m}</td>
</tr>
<tr>
<td>Matches from m to n repetitions of a pattern.</td>
<td>{m,n}</td>
</tr>
<tr>
<td>EXAMPLE: Matches one or more alphabetical characters.</td>
<td>[A-Za-z]+</td>
</tr>
</tbody>
</table>

1.14.5 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:

<table>
<thead>
<tr>
<th>Testing Domain</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-line</td>
<td>behave4cmd</td>
<td>Test command-line tools, like behave, etc. (coming soon).</td>
</tr>
<tr>
<td>Web Apps</td>
<td>behave-django</td>
<td>Test Django Web apps with behave (solution 1).</td>
</tr>
<tr>
<td>Web Apps</td>
<td>django-behave</td>
<td>Test Django Web apps with behave (solution 2).</td>
</tr>
<tr>
<td>Web, SMS, …</td>
<td>behaving</td>
<td>Test Web Apps, Email, SMS, Personas (step library).</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Testing Domain</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI</td>
<td>Squish test</td>
<td>Use Squish and Behave for GUI testing (cross-platform).</td>
</tr>
<tr>
<td>Robot Control</td>
<td>behave4poppy</td>
<td>Use behave to control a robot via pypot.</td>
</tr>
<tr>
<td>Web</td>
<td>pyramid_behave</td>
<td>Use behave to test pyramid.</td>
</tr>
<tr>
<td>Web</td>
<td>pycabara-tutorial</td>
<td>Use pycabara (with behave and Selenium).</td>
</tr>
</tbody>
</table>

See also:

- google-search: behave python example
1.14.6 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

IDE Plugins

<table>
<thead>
<tr>
<th>IDE</th>
<th>Plugin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCharm</td>
<td>PyCharm BDD</td>
<td>PyCharm 4 (Professional edition) has <strong>built-in support</strong> for behave.</td>
</tr>
<tr>
<td>PyCharm</td>
<td>Gherkin</td>
<td>PyCharm/IDEA editor support for Gherkin.</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Cucumber-Eclipse</td>
<td>Plugin contains editor support for Gherkin.</td>
</tr>
<tr>
<td>VisualStudio</td>
<td>cuke4vs</td>
<td>VisualStudio plugin with editor support for Gherkin.</td>
</tr>
</tbody>
</table>

Editors and Editor Plugins

<table>
<thead>
<tr>
<th>Editor</th>
<th>Plugin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gedit</td>
<td>gedit_behave</td>
<td>gedit plugin for jumping between feature and step files.</td>
</tr>
<tr>
<td>Gherkin editor</td>
<td>—</td>
<td>An editor for writing *.feature files.</td>
</tr>
<tr>
<td>Notepad++</td>
<td>NP++ gherkin</td>
<td>Notepad++ editor syntax highlighting for Gherkin.</td>
</tr>
<tr>
<td>Sublime Text</td>
<td>Cucumber (ST Bundle)</td>
<td>Gherkin editor support, table formatting.</td>
</tr>
<tr>
<td>Sublime Text</td>
<td>Behave Step Finder</td>
<td>Helps to navigate to steps in behave.</td>
</tr>
<tr>
<td>vim</td>
<td>vim-behave</td>
<td>vim plugin: Port of vim-cucumber to Python behave.</td>
</tr>
</tbody>
</table>

Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cucutags</td>
<td>Generate <strong>ctags</strong>-like information (cross-reference index) for Gherkin feature files and behave step definitions.</td>
</tr>
</tbody>
</table>

1.14.7 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 2

Indices and tables

• genindex
• modindex
• search
Symbols
- capture
  command line option, 25
- capture-stderr
  command line option, 25
- color
  command line option, 24
- junit
  command line option, 24
- junit-directory
  command line option, 24
- lang
  command line option, 26
- lang-help
  command line option, 26
- lang-list
  command line option, 26
- logcapture
  command line option, 25
- logging-clear-handlers
  command line option, 25
- logging-datetime
  command line option, 25
- logging-filter
  command line option, 25
- logging-format
  command line option, 25
- logging-level
  command line option, 25
- multiline
  command line option, 25
- no-capture
  command line option, 25
- no-capture-stderr
  command line option, 25
- no-junit
  command line option, 24
- no-logcapture
  command line option, 25
- no-snippets
  command line option, 24
- no-summary
  command line option, 25
- show-skipped
  command line option, 24
- show-source
  command line option, 26
- show-timings
  command line option, 26
- snippets
  command line option, 24
- stage
  command line option, 26
- steps-catalog
  command line option, 24
- stop
  command line option, 26
- summary
  command line option, 25
- tags-help
  command line option, 26
- version
  command line option, 26
- D, -define
  command line option, 24
- T, -no-timings
  command line option, 24
- c, -no-color
  command line option, 24
- d, -dry-run
  command line option, 24
- e, -exclude
  command line option, 24
- f, -format
  command line option, 24
- i, -include
  command line option, 24
- k, -no-skipped
  command line option, 24
- m, -no-multiline
  command line option, 25
- n, -name
  command line option, 25
- o, -outfile
  command line option, 25
- q, -quiet
  command line option, 25
-s, -no-source
command line option, 26
-t, -tags
command line option, 26
-v, -verbose
command line option, 26
-w, -wip
command line option, 26
-x, -expand
command line option, 26
@active.with_{category}={value}
active tag schema (dialect 1), 70
@not.with_{category}={value}
active tag schema (dialect 2), 70
@not_active.with_{category}={value}
active tag schema (dialect 1), 70
@only.with_{category}={value}
active tag schema (dialect 2), 70
@use.with_{category}={value}
active tag schema (dialect 2), 70
A
abandon() (behave.log_capture.LoggingCapture method), 45
aborted (behave.runner.Context attribute), 35
Active Tag Logic, 70
Active Tag Schema, 70
active tag schema (dialect 1)
@active.with_{category}={value}, 70
@not_active.with_{category}={value}, 70
active tag schema (dialect 2)
@not.with_{category}={value}, 70
@only.with_{category}={value}, 70
@use.with_{category}={value}, 70
Active Tags, 70
active_outline (behave.runner.Context attribute), 35
add_cleanup() (behave.runner.Context method), 35
any_errors() (behave.log_capture.LoggingCapture method), 45
Argument (class in behave.model_core), 32
arguments (behave.matchers.Match attribute), 32
B
background (behave.model.Feature attribute), 40
Background (class in behave.model), 40
buffer (behave.log_capture.LoggingCapture attribute), 45
C
capture() (in module behave.log_capture), 45
cells (behave.model.Row attribute), 44
check_match() (behave.matchers.Matcher method), 32
command line option