behave Documentation

Release 1.2.6.dev0

Benno Rice, Richard Jones and Jens Engel

October 03, 2016

Contents

behave is behaviour-driven development, Python style.

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

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

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

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

There is also a comparison with the other tools available.

Contents

1.1 Installation

1.1.1 Using pip (or ...)

Category Stable version

Precondition pip (or setuptools) is installed

Execute the following command to install behave with pip:

pip install behave

To update an already installed behave version, use:

pip install -U behave

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 Bleading edge

Precondition pip is installed

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

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

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

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

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

1.2 Tutorial

First, install behave.

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

```
Feature: showing off behave

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

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

```
from behave import *

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

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

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

Run behave:

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

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

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

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

1.2.1 Features

behave operates on directories containing:

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

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

The minimum requirement for a features directory is:

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

A more complex directory might look like:

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

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

1.2.2 Feature Files

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

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

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

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

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

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

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

Then we *observe outcomes*.

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

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

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

Scenario Outlines

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

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

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

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

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

Step Data

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

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

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

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

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

```
Scenario: some scenario
  Given a set of specific users
     | name      | department  |
     | Barry     | Beer Cans   |
     | Pudey     | Silly Walks |
     | Two-Lumps | Silly Walks |
 When we count the number of people in each department
 Then we will find two people in "Silly Walks"
  But we will find one person in "Beer Cans"
```

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

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

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

1.2.3 Python Step Implementations

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

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

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

Given a Scenario:

```
Scenario: Search for an account
   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).

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:

```
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 a 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`):

```
@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 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 "(?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:

```
@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:

```
import threading
from wsgiref import simple_server
from selenium import webdriver
from my_application import model
from my_application import web_app

def before_all(context):
    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_all(context):
    context.server.shutdown()
    context.thread.join()
    context.browser.quit()

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

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 to 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
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()
```

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

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

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

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

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

BEHAVE_DEBUG_ON_ERROR = False

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

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

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

1.3 Behavior Driven Development

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

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

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

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

1.3.1 BDD practices

The practices of BDD include:

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

1.3.2 Outside–in

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

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

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

1.3.3 The Gherkin language

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

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

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

Each scenario is an 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:

```
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:

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

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

Features

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

```
Feature: feature name

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

In all its glory it could look like:

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

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

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

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

  Scenario: ...
```

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

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

Background

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

- logging into a web browser or
- setting up a database

with test data used by the scenarios.

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

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

It contains *steps* as described below.

Good practices for using Background

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

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

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

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

Scenarios

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

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

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

There may be additional conditions imposed on the scenario, and these would take the form of "when" steps following the initial "given" condition. If necessary, additional "and" or "but" steps may also follow the "given", "when" and "then" steps if more 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 practise to have a scenario test only one behaviour or desired outcome.

Scenarios contain *steps* as described below.

Scenario Outlines

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

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

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

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

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

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

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

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

Steps

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

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

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

```
Given a browser client is used to load the URL "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:

```
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:

```
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 preceeds 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:

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

For example:

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

or:

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

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

Controlling Your Test Run With Tags

Given a feature file with:

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

  @slow
  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`.

`-c, --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.

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

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

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

`-i, --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.

`-f, --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

`-k, --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:

```
--tags @dev,@wip
```

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" or "setup.cfg" (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 config.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`
Alias for –no-snippets –no-source.

`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:

```
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:

```
@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":

```
@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 "(?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).

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:

```python
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)

# -- REGISTER TYPE-CONVERTER: With behave
register_type(Number=parse_number)

# -- STEP DEFINITIONS: Use type converter.
@given('{amount:Number} vehicles')
def step_impl(context, amount):
    assert isinstance(amount, int)
```

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*, *string*, *step_type=None*)

Pull parameters out of step names.

string

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).

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:

```
@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.

from behave import *

The import statement:

```
from behave import *
```

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

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

See also the description in *step parameters*.

1.6.2 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:

```
    # -- 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:

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

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

Capture Logging in Hooks

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

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

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

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

Detecting that user code overwrites behave Context attributes

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

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

Hold contextual information during the running of tests.

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

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

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

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

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

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

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

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

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

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

`config`
The configuration of *behave* as determined by configuration files and command-line options. The attributes of this object are the same as the configuration file settion 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 overwite 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.

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 fail.

ValueError will be raised if this is invoked outside a feature context.

Returns boolean False if the steps are not parseable, True otherwise.

use_with_user_mode()
Provides a context manager for using the context in USER mode.

class behave.runner.**ContextMaskWarning**
Raised if a context variable is being overwritten in some situations.

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

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

1.6.3 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_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_feature
after_all
```

1.6.4 Model Objects

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

keyword “Feature”, “Scenario”, “Given”, etc.

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

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

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

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

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

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

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

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

A feature parsed from a *feature file*.

The attributes are:

`keyword`

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

`name`

The name of the feature (the text after “Feature”.)

`description`

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

`background`
The `Background` for this feature, if any.

`scenarios`
A list of `Scenario` making up this feature.

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

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

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

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

"passed" The feature was tested successfully.

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

`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:

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

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

"passed" The scenario was tested successfully.

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

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:

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

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

"passed" The scenario was tested successfully.

"failed" One or more scenarios of this outline failed.

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.

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*.

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:

"skipped" This step was passed over during testing.

"passed" The step was tested successfully.

"failed" The step failed.

`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.

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

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

Tables may be associated with either Examples or Steps:

class `behave.model.`**`Table`** (*headings*, *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. Headings and row data are compared.

class `behave.model.`**Row**(*headings*, *cells*, *line=None*, *comments=None*)
One row of a table parsed from a *feature file*.

Row data is accessible using a number of methods:

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

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

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

The attributes are:

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

headings
The headings of the table as a list of strings.

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

And Text may be associated with Steps:

class `behave.model.`**Text**
Store multiline text from a Step definition.

The attributes are:

value
The actual text parsed from the *feature file*.

content_type
Currently only "text/plain".

1.6.5 Logging Capture

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

class `behave.log_capture.`**LoggingCapture**(*config*, *level=None*)
Capture logging events in a memory buffer for later display or query.

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

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

By default the format of the messages will be:

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

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

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

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

abandon()
Turn off logging capture.

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

any_errors()
Search through the buffer for any ERROR or CRITICAL events.

Returns boolean indicating whether a match was found.

findEvent(*pattern*)
Search through the buffer for a message that matches the given regular expression.

Returns boolean indicating whether a match was found.

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:

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

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

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

1.7 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.

1.7.1 Manual Integration

Alternatively, you can integrate Django using the following boilerplate code in your `environment.py` file:

```python
# -- FILE: features/environment.py
import os
import django
from django.test.runner import DiscoverRunner
from django.test.testcases import LiveServerTestCase

os.environ["DJANGO_SETTINGS_MODULE"] = "test_project.settings"

def before_all(context):
    django.setup()
    context.test_runner = DiscoverRunner()
    context.test_runner.setup_test_environment()
    context.old_db_config = context.test_runner.setup_databases()

def before_scenario(context, scenario):
    context.test_case = LiveServerTestCase
    context.test_case.setUpClass()

def after_scenario(context, scenario):
    context.test_case.tearDownClass()
    del context.test_case

def after_all(context):
    context.test_runner.teardown_databases(context.old_db_config)
    context.test_runner.teardown_test_environment()
```

Taken from Andrey Zarubin's blog post "BDD. PyCharm + Python & Django".

1.7.2 Automation Libraries

With *behave* you can test anything on the Django stack: front-end behavior, REST 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 Django application by using the web interface (see *A Note on Testing* below 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 selenium.webdriver import Firefox

def before_all(context):
    context.browser = Firefox()

def after_all(context):
    context.browser.quit()
    context.browser = None
```

In your step implementations you can use the `context.browser` object to access Selenium features. See the Selenium docs (`remote.webdriver`) for details. Example using behave-django:

```
# -- FILE: features/steps/browser_steps.py
from behave import given, when, then

@when(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:

```
# -- FILE: features/environment.py
# CONTAINS: Browser fixture setup and teardown
from splinter.browser import Browser

def before_all(context):
    context.browser = Browser()

def after_all(context):
    context.browser.quit()
    context.browser = None
```

In your step implementations you can use the `context.browser` object to access Splinter features. See the Splinter docs for details. Example using *behave-django*:

```
# -- FILE: features/steps/browser_steps.py
from behave import given, when, then

@when(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.

A Note on Testing

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.
```

1.8 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.

1.8.1 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.

1.8.2 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.

1.8.3 Freshen

freshen is a plugin for nose that implements a Gherkin-style language with Python step definitions. The main differences with behave are:

- Operates as a plugin for nose, and is thus tied to the nose runner and its output model.
- Has some additions to its Gherkin syntax allowing it to specify specific step definition modules for each feature.
- Has separate context objects for various levels: `glc`, `ftc` and `scc`. These relate to global, feature and scenario levels respectively.

The issues we had with Freshen that stopped us using it were:

- The integration with the nose runner made it quite hard to properly debug how and why tests were failing. Quite often you'd get a rather cryptic message with the actual exception having been swallowed.
- The feature-specific step includes could lead to specific sets of step definitions for each feature despite them warning against doing that.
- The output being handled by nose meant that you couldn't do cucumber-style output without the addition of more plugins.
- The context variable names are cryptic and moving context data from one level to another takes a certain amount of work finding and renaming. The behave *context* variable is much more flexible.
- Step functions must have unique names, even though they're decorated to match different strings.
- As with Lettuce, Freshen uses import hackery so .pyc files are left around and the module namespace is polluted.
- Only Before and no contextual before/after control, thus requiring use of atexit for teardown operations and no fine-grained control.

1.9 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.

1.9.1 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 asynchronuous frameworks or protocols (`asyncio` based)

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:

```
# -- FILE: features/tagged_examples.feature
Feature:
  Scenario Outline: Wow
    Given an employee "<name>"

    @develop
    Examples: Araxas
      | name  | birthyear |
      | Alice |  1985     |
      | Bob   |  1975     |

    @integration
    Examples:
      | name   | birthyear |
      | Charly |  1995     |
```

Note: The generated scenarios from a ScenarioOutline inherit the tags from the ScenarioOutline and its Examples section:

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

To run only the first `Examples` section, you use:

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

```
Scenario Outline: Wow -- @1.1 Araxas  # features/tagged_examples.feature:7
  Given an employee "Alice"

Scenario Outline: Wow -- @1.2 Araxas  # features/tagged_examples.feature:8
  Given an employee "Bob"
```

Tagged Examples with Active Tags and Userdata

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

```
# -- FILE: features/tagged_examples2.feature
# VARIANT 2: With active tags and userdata.
Feature:
  Scenario Outline: Wow
```

```
    Given an employee "<name>"

    @use.with_stage=develop
    Examples: Araxas
      | name  | birthyear |
      | Alice |  1985     |
      | Bob   |  1975     |

    @use.with_stage=integration
    Examples:
      | name   | birthyear |
      | Charly |  1995     |
```

Select the `Examples` section now by using:

```
# -- 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)
```

Testing asyncio Frameworks

Since behave 1.2.6.dev0

The following support was added to simplify testing asynchronuous framework and protocols that are based on `asyncio` module (since Python 3.4).

There are basically two use cases:

- async-steps (with event_loop.run_until_complete() semantics)
- async-dispatch step(s) with async-collect step(s) later on

Async-steps

It is now possible to use `async-steps` in `behave`. An async-step is basically a coroutine as step-implementation for behave. The async-step is wrapped into an `event_loop.run_until_complete()` call by using the `@async_run_until_complete` step-decorator.

This avoids another layer of indirection that would otherwise be necessary, to use the coroutine.

A simple example for the implementation of the async-steps is shown for:

- Python 3.5 with new `async`/`await` keywords
- Python 3.4 with `@asyncio.coroutine` decorator and `yield from` keyword

```python
# -- 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)
```

```python
# -- FILE: features/steps/async_steps34.py
# -- REQUIRES: Python >= 3.4
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
@asyncio.coroutine
def step_async_step_waits_seconds_py34(context, duration):
    yield from asyncio.sleep(duration)
```

When you use the async-step from above in a feature file and run it with behave:

```
# -- TEST-RUN OUTPUT:
$ 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 `èvent_loop.run_until_complete()` call. As the timings show, it actually needs approximatly 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 0.206s

1 feature passed, 0 failed, 0 skipped
1 scenario passed, 0 failed, 0 skipped
3 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.208s
```

Note: The final async-collect step needs approx. 0.2 seconds until the two dispatched async-tasks have finished. In contrast, the async-dispatch steps basically need no time at all.

An `AsyncContext` object is used on the context, to hold the event loop information and the async-tasks that are of interest.

The implementation of the steps from above:

```
# -- FILE: features/steps/async_dispatch_steps.py
# REQUIRES: Python 3.4 or newer
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())
```

1.9.2 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:
      | name   | birthyear |
      | Charly |  1995     |         # line 12
```

Up to now, the following scenarios were generated from the scenario outline:

```
Scenario Outline: Wow          # features/xxx.feature:2
  Given an employee "Alice"

Scenario Outline: Wow          # features/xxx.feature:2
  Given an employee "Bob"

Scenario Outline: Wow          # features/xxx.feature:2
  Given an employee "Charly"
```

Note that all generated scenarios had the:

- same name (scenario_outline.name)

- same file location (scenario_outline.file_location)

From now on, the generated scenarios better represent the example/row combination within a scenario outline:

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

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

Scenario Outline: Wow -- @2.1         # features/xxx.feature:12
  Given an employee "Charly"
```

Note that:

- scenario name is now unique for any examples/row combination
- scenario name optionally contains the examples (group) name (if one exists)
- each scenario has a unique file location, based on the row's file location

Therefore, each generated scenario from a scenario outline can be selected via its file location (and run on its own). In addition, if one fails, it is now possible to rerun only the failing example/row combination(s).

The name annoations schema for the generated scenarios from above provides the new default name annotation schema. It can be adapted/overwritten in "behave.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.

Placeholder	Description
examples.name	Refers name of the example group, may be an empty string.
examples.index	Index of the example group (range=1..N).
row.index	Index of the current row within an example group (range=1..R).
row.id	Shortcut for schema: "<examples.index>.<row.index>"

Name may contain Placeholders

Since behave 1.2.5a1

A scenario outline can now use placeholders from example/rows in its name or its examples name. When the scenarios a generated, these placeholders will be replaced with the values of the example/row.

Up to now this behavior did only apply to steps of a scenario outline.

EXAMPLE:

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

    Examples:
      | name  | birthyear |
      | Alice |  1985      |          # line 7
      | Bob   |  1975      |          # line 8

    Examples: Benares-<ID>
      | name    | birthyear | ID |
      | Charly |  1995      | 42 |    # line 12
```

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

```
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:

```
# -- 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: Araxas
      | name  | birthyear |
      | Alice |  1985      |          # line 11
      | Bob   |  1975      |          # line 12

    Examples: Benares
      | name    | birthyear | ID |
      | Charly |  1995      | 42 |    # line 16
```

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

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

```

```
@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
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.
  +-- *.feature
```

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

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

Note: Command-line definitions override userdata definitions in the configuration file.

If the command-line contains no value part, like in `-D NEEDS_CLEANUP`, its value is `"true"`.

The userdata settings can be accessed as dictionary in hooks and steps by using the `context.config.userdata` dictionary.

```
# -- FILE: features/environment.py
def before_all(context):
    browser = context.config.userdata.get("browser", "chrome")
    setup_browser(browser)
```

```
# -- FILE: features/steps/userdata_example_steps.py
@given('I setup the system with the user-specified server"')
def step_setup_system_with_userdata_server(context):
    server_host = context.config.userdata.get("server", "beatrix")
    context.xxx_client = xxx_protocol.connect(server_host)
```

```
# -- ADAPT TEST-RUN: With user-specific data settings.
# SHELL:
behave -D server=obelix features/
behave --define server=obelix features/
```

Other examples for user-specific data are:

- Passing a URL to an external resource that should be used in the tests
- Turning off cleanup mechanisms implemented in environment hooks, for debugging purposes.

Type Converters

The userdata object provides basic support for "type conversion on demand", similar to the `configparser` module. The following type conversion methods are provided:

- `Userdata.getint(name, default=0)`
- `Userdata.getfloat(name, default=0.0)`
- `Userdata.getbool(name, default=False)`
- `Userdata.getas(convert_func, name, default=None, ...)`

Type conversion may raise a `ValueError` exception if the conversion fails.

The following example shows how the type converter functions for integers are used:

```
# -- 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.

```
# -- 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:

```
{
    "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
    ...
```

```
# -- FILE: features/environment.py
from behave.tag_matcher import ActiveTagMatcher
import sys

# -- MATCHES ANY TAGS: @use.with_{category}={value}
# NOTE: active_tag_value_provider provides category values for active tags.
active_tag_value_provider = {
    "browser": os.environ.get("BEHAVE_BROWSER", "chrome"),
    "os":      sys.platform,
}
active_tag_matcher = ActiveTagMatcher(active_tag_value_provider)

# -- BETTER USE: from behave.tag_matcher import setup_active_tag_values
def setup_active_tag_values(active_tag_values, data):
    for category in active_tag_values.keys():
        if category in data:
            active_tag_values[category] = data[category]

def before_all(context):
    # -- SETUP ACTIVE-TAG MATCHER (with userdata):
    # USE: behave -D browser=safari ...
    setup_active_tag_values(active_tag_value_provider, context.config.userdata)

def before_feature(context, feature):
    if active_tag_matcher.should_exclude_with(feature.tags):
        feature.skip(reason="DISABLED ACTIVE-TAG")

def before_scenario(context, scenario):
    if active_tag_matcher.should_exclude_with(scenario.effective_tags):
        scenario.skip("DISABLED ACTIVE-TAG")
```

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

```
# -- 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)
- `my.module::MyClass` (alternative; with double colon as separator)

User-defined Formatter on Command-line

Just use the formatter class (as "scoped class name") on the command-line as value for the `-format` option (short option: `-f`):

```
behave -f my.own_module:SimpleFormatter ...
behave -f behave.formatter.plain:PlainFormatter ...
```

```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 ...
```

1.9.3 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)* .

1.10 More Information about Behave

1.10.1 Tutorials

For new users, that want to read, understand and explore the concepts in Gherkin and behave (after reading the behave documentation):

- "Behave by Example" (on github)

The following small tutorials provide an introduction how you use behave in a specific testing domain:

- Phillip Johnson, Getting Started with Behavior Testing in Python with Behave
- Bdd with Python, Behave and WebDriver
- Wayne Witzel III, Using Behave with Pyramid, 2014-01-10.

Warning: A word of caution if you are new to **"behaviour-driven development" (BDD)**. In general, you want to avoid "user interface" (UI) details in your scenarios, because they describe **how something is implemented** (in this case the UI itself), like:

- `press this button`
- then `enter this text into the text field`
- ...

In **BDD** (or testing in general), you should describe **what should be done** (meaning the intention). This will make your scenarios much more robust and stable because you can change the underlying implementation of:

- the "system under test" (SUT) or
- the test automation layer, that interacts with the SUT.

without changing the scenarios.

1.10.2 Books

Behave is covered in the following books:

1.10.3 Presentation Videos

- Benno Rice: Making Your Application Behave (30min), 2012-08-12, PyCon Australia.
- Selenium: First behave python tuorial with selenium (8min), 2015-01-28, http://www.seleniumframework.com/python-basic/first-behave-gherkin/

- Jessica Ingrasselino: Automation with Python and Behave (67min), 2015-12-16
- Selenium Python Webdriver Tutorial - Behave (BDD) (14min), 2016-01-21

1.10.4 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

1.10.5 Find more Information

See also:

- google:python-behave examples
- google:python-behave tutorials
- google:python-behave videos

1.11 Appendix

Contents:

1.11.1 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.

Formatters

The following formatters are currently supported:

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

Note: You can use more than one formatter during a test run. But in general you have only one formatter that writes to `stdout`.

The "Mode" column indicates if a formatter is intended to be used in dry-run (`--dry-run` command-line option) or normal mode.

Reporters

The following reporters are currently supported:

Name	Description
junit	Provides JUnit XML-like output.
summary	Provides a summary of the test run.

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

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

User Attributes

A user can assign (or modify) own attributes to the context object. But these attributes will be removed again from the context object depending where these attributes are defined.

Kind	Assign Location	Lifecycle Layer (Scope)
Hook	`before_all()`	test run
Hook	`after_all()`	test run
Hook	`before_tags()`	feature or scenario
Hook	`after_tags()`	feature or scenario
Hook	`before_feature()`	feature
Hook	`after_feature()`	feature
Hook	`before_scenario()`	scenario
Hook	`after_scenario()`	scenario
Hook	`before_step()`	scenario
Hook	`after_step()`	scenario
Step	Step definition	scenario

1.11.3 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*:

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

1.11.4 Regular Expressions

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

Special Characters	Description
`.`	Matches any character (dot).
`^`	"^...", matches start-of-string (caret).
`$`	"...$", matches end-of-string (dollar sign).
`|`	"A|B", matches "A" or "B".
`\`	Escape character.
`\.`	EXAMPLE: Matches character '.' (dot).
`\\`	EXAMPLE: Matches character '\' (backslash).

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

Character sets	Description
`[...]`	Define a character set, like `[A-Za-z]`.
`\d`	Matches digit character: [0-9]
`\D`	Matches non-digit character.
`\s`	Matches whitespace character: `[ \t\n\r\f\v]`
`\S`	Matches non-whitespace character
`\w`	Matches alphanumeric character: `[a-zA-Z0-9_]`
`\W`	Matches non-alphanumeric character.

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

Grouping	Description
`(...)`	Group a regular expression pattern (anonymous group).
`\number`	Matches text of earlier group by index, like: "`\1`".
`(?P<name>...)`	Matches pattern and stores it in parameter "name".
`(?P=name)`	Match whatever text was matched by earlier group "name".
`(?:...)`	Matches pattern, but does non capture any text.
`(?#...)`	Comment (is ignored), describes pattern details.

If a *group*, *character* or *character set* should be repeated several times, it is necessary to specify the cardinality of the regular expression pattern.

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

1.11.5 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:

Testing Domain	Name	Description
Command-line	behave4cmd	Test command-line tools, like behave, etc. (coming soon).
Web Apps	behave-django	Test Django Web apps with behave (solution 1).
Web Apps	django-behave	Test Django Web apps with behave (solution 2).
Web, SMS, ...	behaving	Test Web Apps, Email, SMS, Personas (step library).

Step Usage Examples

This examples show how you can use behave for testing a specific problem domain. This examples are normally not a full-blown step library (that can be reused), but give you an example (or prototype), how the problem can be solved.

Testing Domain	Name	Description
GUI	Squish test	Use Squish and Behave for GUI testing (cross-platform).
Robot Control	behave4poppy	Use behave to control a robot via pypot.
Web	pyramid_behave	Use behave to test pyramid.
Web	pycabara-tutorial	Use pycabara (with behave and Selenium).

See also:

- google-search: behave python example

1.11.6 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)

IDE Plugins

IDE	Plugin	Description
PyCharm	PyCharm BDD	PyCharm 4 (Professional edition) has **built-in support** for behave.
PyCharm	Gherkin	PyCharm/IDEA editor support for Gherkin.
Eclipse	Cucumber-Eclipse	Plugin contains editor support for Gherkin.
VisualStudio	cuke4vs	VisualStudio plugin with editor support for Gherkin.

Editors and Editor Plugins

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

Tools

Tool	Description
cucu-tags	Generate ctags-like information (cross-reference index) for Gherkin feature files and behave step definitions.

1.11.7 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

Bibliography

[TDD-Python] Harry Percival, Test-Driven Web Development with Python, O'Reilly, June 2014, Appendix E: BDD (covers behave)