

Feature Testing Setup

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.

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`:

```
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.