# Introduction

behave is Behaviour-Driven Development framework, **written in Python and used for Python**, to describe the behavior of software in natural language format.

If you know about BDD and Gherkin, you might already know about Cucumber. While Cucumber supports many programming languages, it does not support Python. And behave is born to resolve this issue. In fact, **Cucumber and behave shares a lot of similarities**. If you know about Cucumber before, you can learn behave very fast.

**Following concepts in behave are the same as in Cucumber:**

* **Gherkin**
* **Features**
* **Step, step argument, step definitions, hooks (exception is Python of course)**
* **Execution flow, test results**
* **Tags**
* **Test reporting**

In this tutorial, we only focus on concepts which are different between behave and Cucumber. Mostly, they're about Python.

# Installation

<https://behave.readthedocs.io/en/latest/install/>

# File Structures

features        # must be the root, cannot change to another name.

├───featureA

│   │   featureA1.feature

│   │   featureA2.feature

│   └───steps   # can be any name as long as it has .py files

│           featureA1.py

│           featureA2.py

│

└───featureB

    │   featureB.feature

    └───steps   # can be any name as long as it has .py files

            featureB.py

# Step Definition

## Step Functions

Given a scenario with following steps:

Scenario: Search for an account

   Given I search for a valid account

    Then I will see the account details

And And I can go back to search other accounts

Step definition for the above scenario will look like:

@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)

@then('I can go back to search other accounts')

def step\_impl(context):

    ...

You can see that steps beginning with and or but in the feature file are renamed to take the name of their preceding keyword. In above example, we used @then, not @and.

**Note**: Step function names do not need to have a unique name.

## Step Parameters

### Example

Given the feature:

-- FILE: features/example\_step\_parameters.feature

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 can define a Python step definition that handles both cases by using step parameters . In this case, the Then step verifies the context.response parameter that was stored in the context by the Given step:

# -- FILE: features/steps/example\_steps\_with\_step\_parameters.py

# HINT: Step-matcher "parse" is the DEFAULT step-matcher class.

from behave import then

@then('the result page will include "{text}"')

def step\_impl(context, text):

    if text not in context.response:

        fail('%r not in %r' % (text, context.response))

### Step Matchers

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

* parse (**default** step-matcher class, based on: [parse](https://pypi.org/project/parse))
* cfparse (extending of parse, requires: [parse\_type](https://pypi.org/project/parse_type))
* re (step-matcher class is based on regular expressions)

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

# -- FILE: features/steps/example\_use\_step\_matcher\_in\_steps.py

# HINTS:

#   \* "parse" in the DEFAULT step-matcher

#   \* Use "use\_step\_matcher(...)" in "features/environment.py" file

#     to define your own own default step-matcher.

from behave import given, when, use\_step\_matcher

use\_step\_matcher("cfparse")

@given('some event named "{event\_name}" happens')

def step\_given\_some\_event\_named\_happens(context, event\_name):

    pass    # ... DETAILS LEFT OUT HERE.

use\_step\_matcher("re")

@when('a person named "(?P<name>...)" enters the room')

def step\_when\_person\_enters\_room(context, name):

    pass    # ... DETAILS LEFT OUT HERE.

More details: <https://behave.readthedocs.io/en/latest/tutorial/#step-matchers>

## Step Data

**Example 1:**

Given the feature:

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 Context.table attribute:

@given('a set of specific users')

def step\_impl(context):

    for row in context.table:

        model.add\_user(name=row['name'], department=row['department'])

**Example 2:**

Given the feature:

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 Context.text attribute:

## 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().

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

@when(u'I do the same thing as before with the {color:w} button')

def step\_impl(context, color):

    context.execute\_steps(u'''

        When I press the big {color} button

         And I duck

    '''.format(color=color))

# Hooks (or Environment Controls)

## Summary Table

|  |  |  |  |
| --- | --- | --- | --- |
| **Hook** | **Description** | **Syntax** | **Note** |
| before\_scenario | Run before the first step of each scenario. | def before\_scenario(context, scenario):      # do something | Whatever happens in a Before hook is invisible to people who only read the features.  Consider using a Background in feature file if the setup should be readable by non-technical people. And only use a Before hook for low-level logic such as starting a browser or deleting data from a database. |
| after\_scenario | Run after the last step of each scenario, even when the step result is failed, undefined, pending, or skipped. | def after\_scenario(context, scenario):      # do something |  |
| before\_step | Run before a step. | def before\_step(context, step):      # do something |  |
| after\_step | Run after a step. | def after\_step(context, step):      # do something |  |
| before\_feature | Run before a feature. | def before\_feature(context, feature):      # do something |  |
| after\_feature | Run after a feature. | def after\_feature(context, feature):      # do something |  |
| before\_tag | Run before a section tagged with the given name. | def before\_tag(context, tag):      # do something |  |
| after\_tag | Run after a section tagged with the given name. | def after\_tag(context, tag):      # do something |  |
| before\_rule | Run before a rule. | def before\_rule(context, rule):      # do something |  |
| after\_rule | Run before a rule. | def after\_rule(context, rule):      # do something |  |
| before\_all | Run before the whole shooting match. | def before\_all(context):      # do something |  |
| after\_all | Run before the whole shooting match. | def after\_all(context):      # do something |  |

**Example:** Using before\_tag:

# -- ASSUMING: tags @browser.chrome or @browser.any are used.

def before\_tag(context, tag):

    if tag.startswith("browser."):

        browser\_type = tag.replace("browser.", "", 1)

        if browser\_type == "chrome":

            context.browser = webdriver.Chrome()

        else:

            context.browser = webdriver.PlainVanilla()

## Some Useful Hooks Ideas

### 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):

    ...

# Fixture

Fixtures helps **simplify setup/teardown** task in behave. Instead of calling hooks, you can use fixtures.

## Example

In your python file:

# -- FILE: my\_project/fixtures.py  (or in: features/environment.py)

from behave import fixture

from somewhere.browser.firefox import FirefoxBrowser

# -- FIXTURE-VARIANT 1: Use generator-function

@fixture

def browser\_firefox(context, timeout=30, \*\*kwargs):

    # -- SETUP-FIXTURE PART:

    context.browser = FirefoxBrowser(timeout, \*\*kwargs)

    yield context.browser

    # -- CLEANUP-FIXTURE PART:

    context.browser.shutdown()

Or:

# -- FIXTURE-VARIANT 2: Use normal function

from somewhere.browser.firefox import FirefoxxBrowser

@fixture

def browser\_firefox(context, timeout=30, \*\*kwargs):

    # -- SETUP-FIXTURE PART: And register as context-cleanup task.

    browser = FirefoxxBrowser(timeout, \*\*kwargs)

    context.browser = browser

    context.add\_cleanup(browser.shutdown)

    return browser

    # -- CLEANUP-FIXTURE PART: browser.shutdown()

    # Fixture-cleanup is called when current context-layer is removed.

In your feature file, add a @fixture tag:

# -- FILE: features/use\_fixture1.feature

**Feature**: Use Fixture on Scenario Level

    @fixture.browser.firefox

**Scenario**: Use Web Browser Firefox

        Given I load web page "https://somewhere.web"

        ...

    # -- AFTER-SCENARIO: Cleanup fixture.browser.firefox

In your step definition file, call the fixture using the use\_fixture() function:

# -- FILE: features/environment.py

from behave import use\_fixture

from my\_project.fixtures import browser\_firefox

def before\_tag(context, tag):

    if tag == "fixture.browser.firefox":

        use\_fixture(browser\_firefox, context, timeout=10)

The use\_fixture() call performs the SETUP-FIXTURE part and returns the setup result. In addition, it ensures that CLEANUP-FIXTURE part is called later-on when the current context-layer is removed. Therefore, any manual cleanup handling in the after\_tag() hook is not necessary.

## Fixture Setup/Cleanup Semantics

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

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

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

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

# API Reference

<https://behave.readthedocs.io/en/latest/api/>

# Command Line

<https://behave.readthedocs.io/en/latest/behave/>

# Configuration File

<https://behave.readthedocs.io/en/latest/behave/#configuration-parameters>

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

    ''')

# VS Code Extension Supports

[**Behave VSC**](https://marketplace.visualstudio.com/items?itemName=jimasp.behave-vsc)

Execute and debug test individually from VS Code.

# Python Plugin Supports

## [pylint-behave](https://pypi.org/project/pylint-behave/)

Improve code analysis for Pylint when analyzing a Behave project.