

Behavior-Driven Python

Andrew Knight
Friday, May 11th @ PyCon 2018

Who is Andy Knight?

- Software engineer
- Expert in testing, automation, and BDD
- Lives in Raleigh, NC
- Works at PrecisionLender
- RIT Class of 2010
- Avid Pythoneer!

Twitter: **@AutomationPanda**

Blog: AutomationPanda.com

GitHub: [AndyLPK247](https://github.com/AndyLPK247)



Agenda

Behavior-Driven
Development using
Python's **behave**
framework!

behavior bdd tags passed driven gherkin outline given specify feature testing
scenario step behave examples steps wip failed design setup

Behavior-Driven Development

What is a “Behavior”?

be·hav·ior

the way in which one acts or conducts oneself

In software, a **behavior** is how a feature operates. A behavior is defined as a scenario of inputs, actions, and outcomes. A product or feature exhibits countless behaviors.

- Submitting forms on a website
- Searching for desired results
- Saving a document
- Making REST API calls
- Running CLI commands

Behavior-Driven Development

BDD is a quality-centric software development process that puts product behaviors first. It complements existing process like Agile.

Behaviors are identified early in development using **specification by example**: *plain-language* descriptions (Gherkin) that tell *what* more than *how*. Behavior specs become requirements, acceptance criteria, and acceptance tests all in one.

Test frameworks can directly automate specs as well: every step in a behavior scenario can be “glued” to code to run it. Behavior specs are the units of coverage.



Python has many BDD test frameworks. This talk will focus on **behave**.

Gherkin Behavior Spec Example

Scenario: Basic DuckDuckGo Search

Given the DuckDuckGo home page is displayed

When the user searches for "panda"

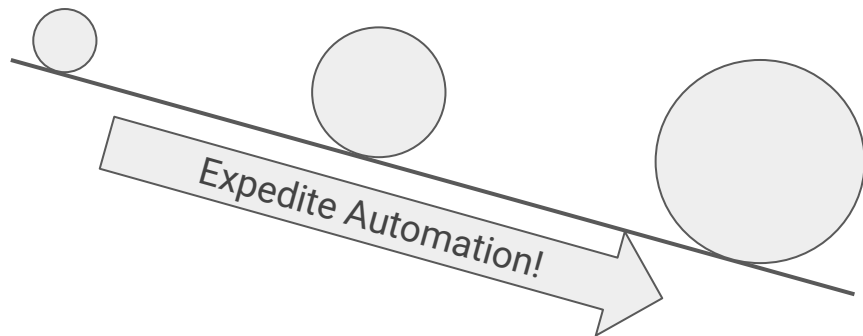
Then results are shown for "panda"

- ✓ Concise
- ✓ Focused
- ✓ Meaningful
- ✓ Declarative

Benefits of BDD

The main benefits of BDD are better **collaboration** and **automation**.

- Everyone can contribute to development, not just programmers.
- Expected behavior is well defined and understood from the beginning.
- Tests can be automated together with the features they cover.
- Each test covers a singular unique behavior, avoiding duplication.
- Steps can be reused by behavior specs, creating a snowball effect.



The *behave* Framework

How *behave* Works

The **behave** framework essentially automates behavior specs using Python for the purpose of testing. There are two main layers:

1. Behavior specs written in **Gherkin** “.feature” files for *test focus*.
2. Step definitions and hooks written in **Python** for *automation focus*.

Gherkin scenarios use plain language steps:

1. *Given* some initial state
2. *When* an action is taken
3. *Then* verify the outcome

Each step is “glued” by decorator to a Python function when **behave** runs tests.

Installing *behave*

Only one package is needed:

```
pip install behave
```

Other packages may also be useful, such as:

```
pip install requests      # for REST API calls
```

```
pip install selenium      # for Web browser interactions
```

✓ Try using **pipenv**! It combines pip, Pipfile, and virtualenv.

Gherkin Features

Example Feature

Feature: Cucumber Basket

As a gardener,
I want to carry many cucumbers in a basket,
So that I don't drop them all.

@cucumber-basket

Scenario: Add and remove cucumbers

Given the basket is empty
When "4" cucumbers are added to the basket
And "6" more cucumbers are added to the basket
But "3" cucumbers are removed from the basket
Then the basket contains "7" cucumbers

The Background Section

Feature: Cucumber Basket

Background:

Given the basket contains "5" cucumbers

Scenario: Add cucumbers

When "4" cucumbers are added to the basket

Then the basket contains "9" cucumbers

Scenario: Remove cucumbers

When "3" cucumbers are removed from the basket

Then the basket contains "2" cucumbers

A Scenario Outline

Feature: Cucumber Basket

Scenario Outline: Add cucumbers

Given the basket contains "<initial>" cucumbers

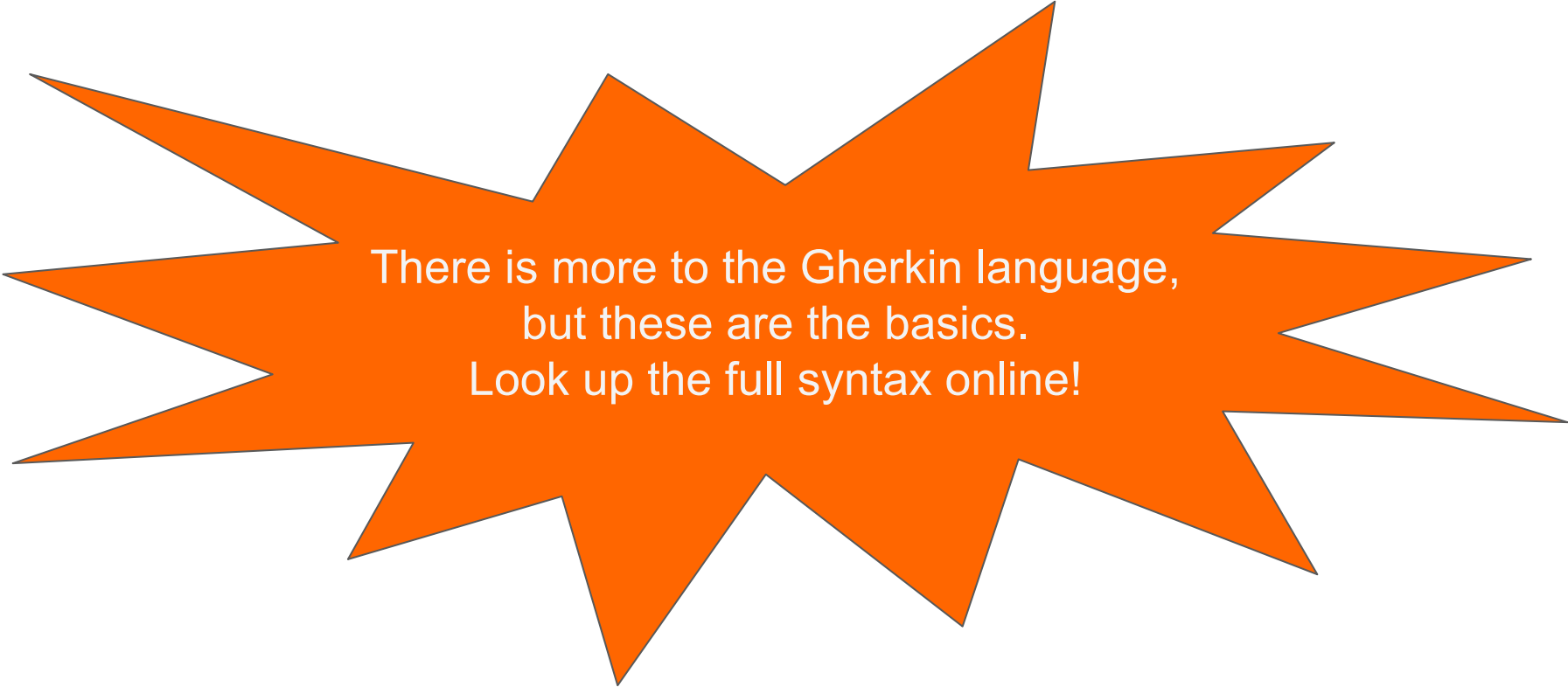
When "<more>" cucumbers are added to the basket

Then the basket contains "<total>" cucumbers

Examples: Cucumber Counts

initial	more	total	
0	1	1	
1	2	3	
5	4	9	

So Much Gherkin



There is more to the Gherkin language,
but these are the basics.
Look up the full syntax online!

Python Mechanics

Step Definitions

Step definitions are Python functions that provide implementations for Gherkin steps. Each function has a step type decorator with the matching string. It also receives a shared *context* and any step parameters.

Three step matchers are available: (1) *parse*, (2) *cfparse*, and (3) *re*. The default and simplest is *parse* (shown to the right).

✓ Pro tip:
Surround parameters with “ ”!

```
from behave import *
from cucumbers.basket import CucumberBasket

@given('the basket has "{initial:d}" cucumbers')
def step_impl(context, initial):
    context.basket = CucumberBasket(initial_count=initial)

@when('"{some:d}" cucumbers are added to the basket')
def step_impl(context, some):
    context.basket.add(some)

@then('the basket contains "{total:d}" cucumbers')
def step_impl(context, total):
    assert context.basket.count == total
```

Gluing Steps to Definitions

Feature: Cucumber Basket

Scenario: Add and remove cucumbers

Given the basket has "4" cucumbers

When "3" cucumbers are added to the basket

Then the basket contains "7" cucumbers

```
from behave import *
from cucumbers.basket import CucumberBasket

@given('the basket has "{initial:d}" cucumbers')
def step_impl(context, initial):
    context.basket = CucumberBasket(initial_count=initial)

@when('"{some:d}" cucumbers are added to the basket')
def step_impl(context, some):
    context.basket.add(some)

@then('the basket contains "{total:d}" cucumbers')
def step_impl(context, total):
    assert context.basket.count == total
```

Scenario Context

Notice the **context** variable passed into each step def. It holds data specific to the currently-running scenario. This is the proper way to pass data between steps!

Standard fields include *feature*, *scenario*, and *tags*. The *text* and *table* fields get step data. Custom fields may be added, too.

⊗ Warning:

Do *not* use globals to share data!

```
from behave import *
from cucumbers.basket import CucumberBasket

@given('the basket has "{initial:d}" cucumbers')
def step_impl(context, initial):
    context.basket = CucumberBasket(initial_count=initial)

@when('"{some:d}" cucumbers are added to the basket')
def step_impl(context, some):
    context.basket.add(some)

@then('the basket contains "{total:d}" cucumbers')
def step_impl(context, total):
    assert context.basket.count == total
```

Hooks

Steps focus on behavior, but automation often has extra needs. For example, Web tests must set up and tear down a WebDriver instance.

Hooks in the **environment.py** file can add instructions before and after steps, scenarios, features, tags, and the whole test run. This is similar to *Aspect-Oriented Programming*.

✓ Use tags to selectively apply hooks!

⊗ Don't put cleanup in steps!

```
from selenium import webdriver

def before_scenario(context, scenario):
    if 'web' in context.tags:
        context.browser = webdriver.Firefox()
        context.browser.implicitly_wait(10)

def after_scenario(context, scenario):
    if 'web' in context.tags:
        context.browser.quit()
```

Fixtures

While hooks can insert any sort of logic, **fixtures** specifically handle setup and cleanup with more advanced options.

For example, any scenario tagged with the fixture to the right will have a Firefox WebDriver instance automatically created before the scenario and cleaned up after the scenario.

Check **behave**'s doc for examples of *composite* fixtures and *registries*.

```
from behave import fixture
from selenium import webdriver

# The tag to use for the following fixture is:
# @fixture.browser.firefox

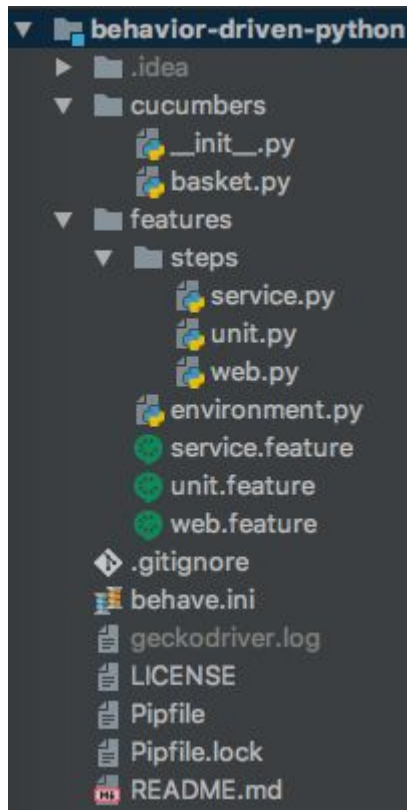
@fixture
def browser_firefox(context):
    # -- SETUP-FIXTURE PART:
    context.browser = webdriver.Firefox()
    context.browser.implicitly_wait(10)
    yield context.browser
    # -- CLEANUP-FIXTURE PART:
    context.browser.quit()
```

Directory Layout

behave is picky about the directory layout. By default, all feature files and the environment module must be under a directory named “features/”, and all step definition modules must be under “features/steps/”.

Feature file paths may be overridden by the *paths* config option. It’s popular to put everything under “tests/”.

- ✓ Feature files can use step definitions from *any* module under “steps”.
Files don’t need to have the same name.



Support Classes

Any Python packages or custom modules can be used with **behave**. Use them to build a better framework! Major packages include *logging*, *requests*, and *selenium*.

Be sure to employ **good design patterns** as well. For example, use the Page Object Model or the Screenplay Pattern instead of raw WebDriver calls for Web tests. Step def code should be concise!



Running Tests

Running Tests

Use the **behave** command from the project root directory to run features as tests:

run all tests

behave

run the scenarios in a feature file

behave features/web.feature

filter tests by tag

behave --tags-help

behave --tags @duckduckgo

behave --tags ~@unit

behave --tags @basket --tags @add,@remove

⚠ Use “pipenv run behave”
if using **pipenv**.

Config Files

Options may be provided in **config files**, too. Config files may be named “.beaverc”, “behave.ini”, “setup.cfg”, or “tox.ini”. They follow Windows INI format and use the “[behave]” label. The best place to put the file is under the project root.

Run “behave --help” to see all available options. Popular ones include:

- Custom feature file paths
- JUnit reporting
- Logging options



Command line options override config file options.

Testing Considerations


1. Using **virtual environments** for package isolation is a Python best practice, especially for test automation. Using a tool like **pipenv** is even better. Packages used only for testing (like **behave**) can be dev dependencies.
2. The **behave** framework does not support **parallel execution** out of the box. Offshoot projects make it happen, but it is still an open issue for the main project. Alternatively, scripts could be written to partition tests across separate processes and then aggregate results into one report.
3. **Build server integration** is easy: just recreate the virtual environment and run the commands. JUnit reports are widely compatible (for example, Jenkins).

Final Remarks

Other Python BDD Frameworks

<u>pytest-bdd</u>	<ul style="list-style-type: none">● Gherkin-style BDD plugin for pytest● Benefits from all pytest features and plugins● Joint execution and filtering with non-Gherkin tests● More flexible fixtures and directory layout
<u>radish</u>	<ul style="list-style-type: none">● BDD framework compatible with Cucumber Gherkin● Adds Scenario Loops and Preconditions to Gherkin● Rich command line options
<u>lettuce</u>	<ul style="list-style-type: none">● Another Gherkin-style BDD framework● Small differences in framework mechanics● Little recent GitHub activity

Beyond Python



Python test frameworks can be used for
any black box testing, even for
non-Python products!

Links and Resources

behave

- GitHub project: <https://github.com/behave/behave>
- Official docs: <http://behave.readthedocs.io/en/latest/index.html>
- Andy's examples: <https://github.com/AndyLPK247/behavior-driven-python>

Automation Panda

- Twitter: **@AutomationPanda**
- More on BDD: <https://automationpanda.com/bdd/>
- More on Testing: <https://automationpanda.com/testing/>
- More on Python: <https://automationpanda.com/python/>

Thank you!