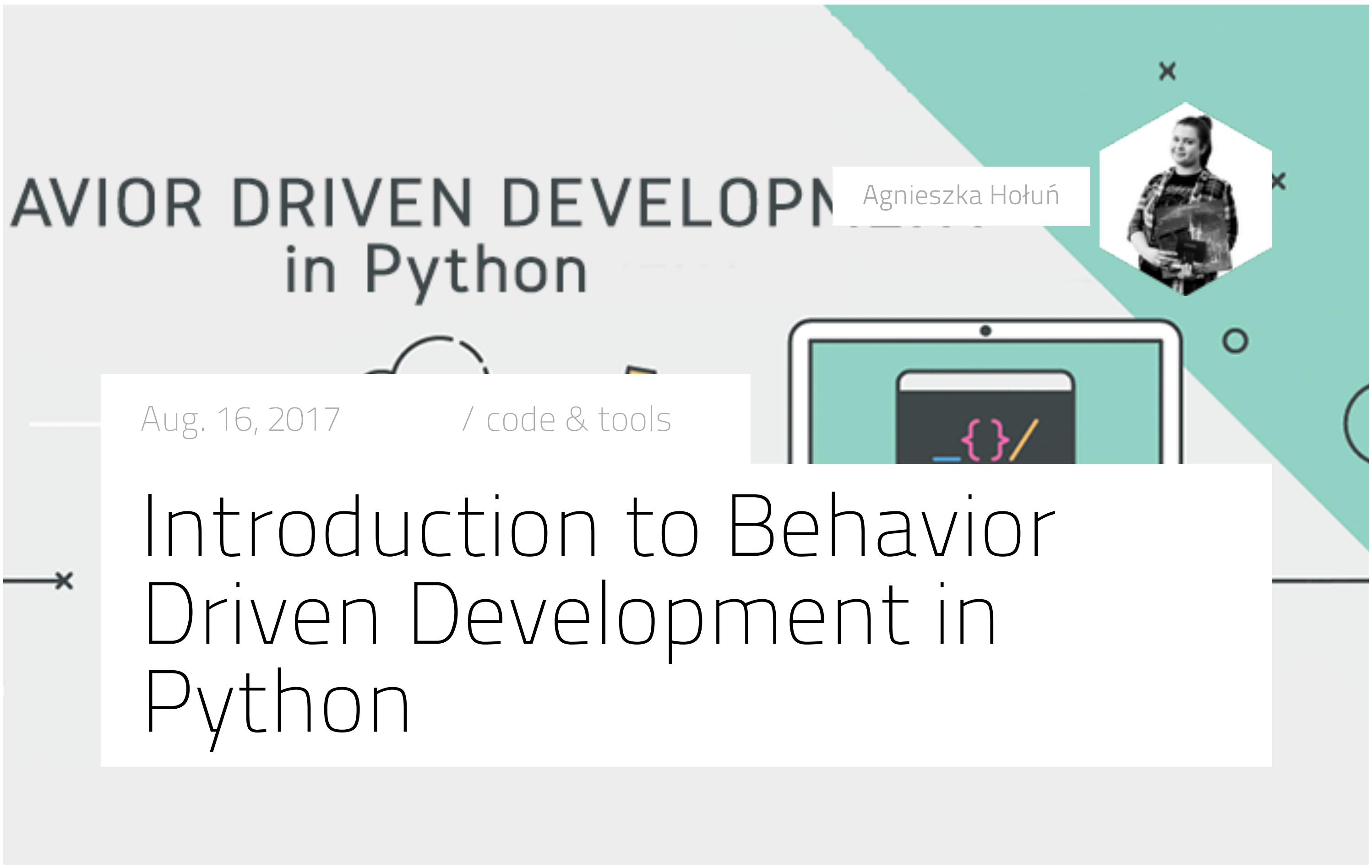
PL



Agnieszka Hołuń

Aug. 16, 2017 / code & tools

Introduction to Behavior Driven Development in Python

Automated testing is still neglected, pushed aside, or even avoided in many IT projects. The well-written unit, integration, or acceptance tests can help detect bugs and problems at very early stage of the

/ join our newsletter

First Name

development. But how to maintain many test cases without wasting time on writing lots of documentation that is not really business readable? Well, that's where BDD comes in handy!

If you're not familiar with the concept of **Behavior Driven Development**, be sure first to check out Dorota's article covering this topic. In my article, I will focus specifically on **BDD in Python with the use of** behave **framework**.

What is behave?

First BDD framework, called JBehave, was developed by Dan North, the father of BDD. It was written for Java and was soon followed by RBehave, a Ruby version Dan helped create (later integrated with RSpec). **RSpec was then replaced by Cucumber**, which nowadays has implementations in many different programming languages, including Python.

Email Address

Subscribe

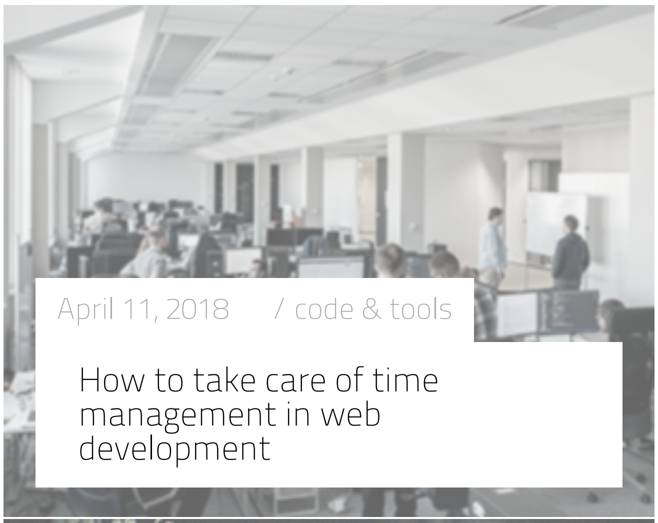
REVIEWED ON

34 REVIEWS

BLOG FIRMOWY ROKU 2015

BLOG PROFESJONALNY

read also



April 11, 2018 / code & tools

How to take care of time management in web development

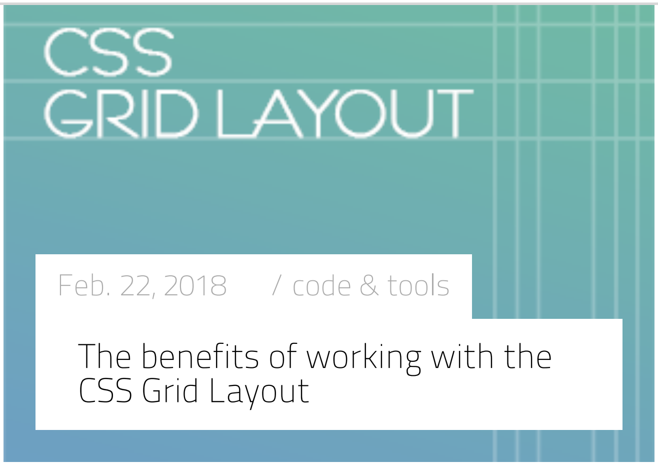
Right now, the most popular Python BDD frameworks are **behave** and **Lettuce**. There are also other alternatives, such as **radish**

March 28, 2018 / code & tools

Learning web development without breaking the bank

↑ back to top

for pytest). However, **behave seems to have the biggest community**, therefore you can find many examples and help online. With that in mind, I decided to stick with it.



Feb. 22, 2018 / code & tools

The benefits of working with the CSS Grid Layout

See our portfolio

Creating and installing the project

Requirements

- Python 3,
- code editor and console.

I'd recommend using a **virtualenv** – if you're not familiar with it, you can find an instruction how to create and use one here.

To install behave we simply have to use pip.

```
pip install behave
```

Then, we can create a directory for our project.

```
mkdir behave-example
```

And that's it, we're ready to write our first test!

Test scenario

Before we write any Python code, we need to **prepare our test case first**. Although there are no formal requirements regarding how exactly scenarios should be written, Gherkin is the most popular syntax for preparing them as it's ubiquitous and has many useful features.

Gherkin test scenarios are placed in a .feature file, so **remember to save the file with this extension.** They consist of a few keywords that are necessary to run the tests – **Feature**, **Scenario**, **Scenario Outline**, **Background**, **Examples**, and steps keywords – **Given**, **When**, **Then**, **And**.

We start with **Feature**, where we state the title of our test case suite. **It should be clear, explicit, and shortly describe the feature that is going to be tested.** Optionally, under Feature we can provide the additional description that is usually written in a form of a user story. Each Feature can consist of multiple test scenarios, which will be executed one after another.

Scenario is where we write the test case, starting with a short description what our test is supposed to check. After that, we write test step starting with proper keywords.

Here's an example of a .feature file I wrote:

```
Feature: Calculator

        As an author of
this article

        I want to
demonstrate

        How to write a
simple test using behave

                with a
calculator as an example



        Scenario: Add
two numbers

                Given I
```

```
have entered 2 into the
calculator

                And I
have also entered 7 into
the calculator

                When I
press add

                Then the
sum should be 9
```

As you can see, **this is a simple test case for adding two numbers in a calculator**. We've got two preconditions that start with **Given** and **And**, then an action that is taken with When, and lastly, after Then, an outcome resulting from action in the previous step.

Now, that's a good test case to check if our calculator adds correctly. However, we could use some more examples just to be sure. That would mean writing more scenarios with different sets of numbers, what would be quite time-consuming. We can use **Scenario Outline** and **Examples** for that. Let's take a look:

```
    Scenario Outline:
Add any two numbers

        Given I have
```

```
entered <number1> into
the calculator

        And I have also
entered <number2> into
the calculator

        When I press add

        Then the sum
should be <result>



        Examples:

            |  number1|
 number2|   result|

            |        5|
       2|        7|

            |        4|
       8|       12|

            |      100|
     200|      300|
```

Examples are written in the form of a table, which has as many columns as parameters that we want to change in each scenario run. The first row contains names of the parameters that have to be the same as the ones used in steps.

Step definition

Now that we have our scenarios written, we can proceed and **write an implementation of the test steps**. Steps are written in a Python file and have to be placed in a separate directory /steps, so let's make one and create our file there (make sure you are in your project directory first):

```
mkdir steps

cd steps
```

For steps to be properly interpreted, we have to **import keywords from behave**:

```
from behave import
given, when, then
```

Now we can proceed with step implementation. **Here's my step definition for the scenario I presented above:**

```
from behave import
given, when, then

from calculator import
Calculator
```

```
@given('I have entered
{number1:d} into the
calculator')

def
enter_number1(context,
number1):

    context.number1 =
number1



@given('I have also
entered {number2:d} into
the calculator')

def
enter_number2(context,
number2):

    context.number2 =
number2



@when('I press add')

def press_add(context):

    context.calculator =
Calculator()

    context.result =
context.calculator.add(…
 context.number2)
```

```
@then('the sum should be
{result:d}')

def
check_result(context,
result):

    assert
context.result ==
result
```

Each step implementation starts with a **decorator named with a keyword that was used in the test scenario and that takes the rest of the step as an argument**. If test step contains a parameter, you should, optionally, **tell the parser what type your parameter is** (otherwise it will be interpreted as string). Syntax for step parameters is {param:Type}, so in case of a number you should write {parameter_name:d}. You can check the full list of parameter types here.

The decorated function expects to receive context as an argument and, additionally, parameters that are parsed from the test step. In case of my first step, number1 is a parameter, so I expect behave to pass it, together with context, as an argument to the function. You can name the function as you like but it should clearly describe what that step is supposed to do.

Context allows you to **share values between steps**, so if I set number1 as context attribute, I can use it in another step and add it to number2.

Test execution

Once we have test scenario and step definition, we can finally run our test. The most basic way of executing tests is just running behave command in the console (in your project directory). **Behave will find all the test scenarios in your project and map them with proper step definitions.** If scenario has Examples, each of them will be run as a separate scenario with a different set of values, so in my case I would have three scenarios executed.

What’s next?

Features that I've described here are not even a half of what you can do with behave. If you’d like to know more, **stay tuned for the second part of my article, where I’ll describe how to use behave**

together with Selenium WebDriver. And, obviously, you can always check behave’s documentation.

Agnieszka Hołuń

14
Like

Tweet

G+

Comments Community

1 Login

Recommend 1 Share

Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Mirosław Zalewski
• 8 months ago

For whatever reason Python script is missing indentation - it won't run if copypasted. It doesn't seem to be a problem with blog itself, as gherkin examples have indented lines.

On a sidenote, this example also shows how extremely verbose gherkin is. I am not convinced this is a good thing.

• Reply • Share ›

Agnieszka Hołuń Mirosław Zalewski • 8 months ago

It seems that formatting for that particular part didn't want to cooperate - it should be fixed by now. Thanks for letting us know! :)

• Reply • Share ›

Privacy