Behavior-Driven Python with pytest-bdd

Andrew Knight
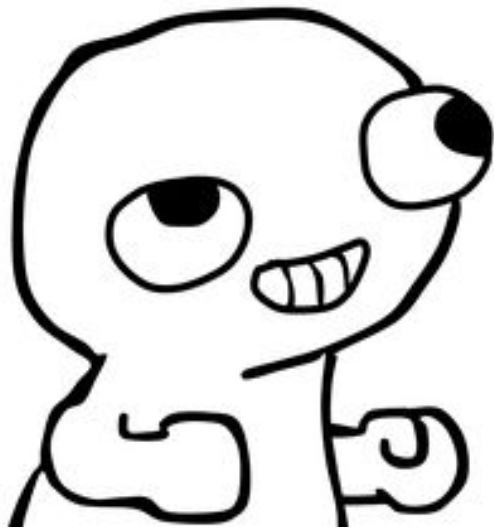Sunday, November 11th @ PyCon Canada 2018

Who is Andy Knight?

- Software engineer and consultant
- Testing, automation, and BDD
- Lives in Raleigh, NC, USA
- Avid Pythoneer!

Twitter: **@AutomationPanda**
Blog: AutomationPanda.com

Software Testing Attitude

VS

Software Testing Impetus

Software Testing Problems

1. Tests take too long to automate
2. Tests are complicated and unreadable
3. Tests always crash instead of passing or finding bugs
4. Tests quickly fall out of sync when features change

Tests Specs Should Be:

✓ Concise

✓ Focused

✓ Meaningful

✓ Declarative

✓

Transparent

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.

Common Practices
✓ Three Amigos Meetings
✓ Example Mapping
✓ Specification by Example

The Goals of BDD
✓ Collaboration
✓ Automation

Test Automation

Behaviors are identified early in development using *plain-language* descriptions (Gherkin) that tell *what* more than *how*.

A BDD test framework separates test *cases* from test *code*. It will "glue" each step to a Python function to run it like a script.

⚠ This talk will focus on automation with **pytest-bdd**.

pytest-bdd

Start with pytest

pytest: helps you write better programs

The `pytest` framework makes it easy to write small tests, yet scales to support complex functional testing for applications and libraries.
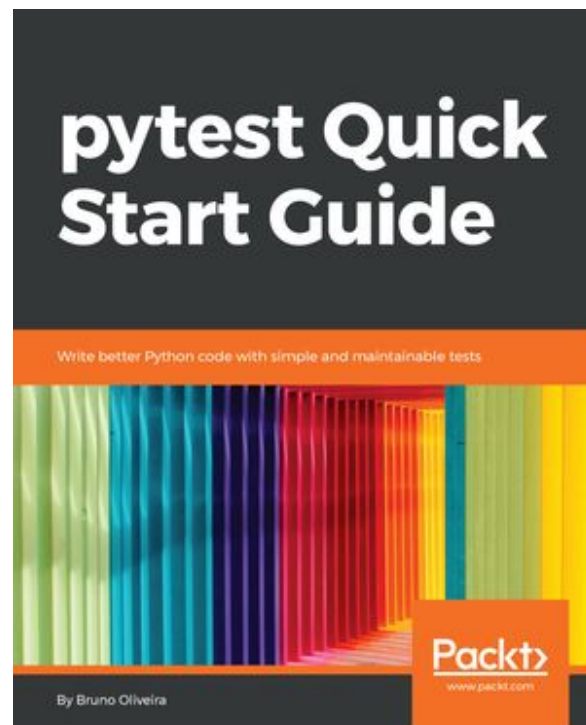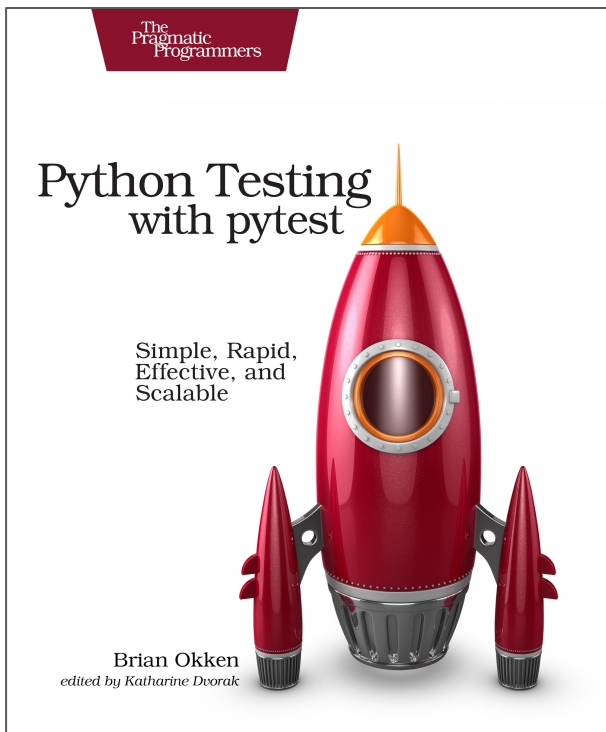
An example of a simple test:

```python
# content of test_sample.py
def inc(x):
    return x + 1


def test_answer():
    assert inc(3) == 5
```

About pytest

pytest is a mature full-featured Python testing tool that helps you write better programs.

Screenshot taken from the pytest.org home page

Recommended pytest Books

Installing `pytest-bdd`

Two packages are needed:

```
pip install pytest
pip install pytest-bdd
```

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.

Our First Gherkin Feature

@basket

Feature: Cucumber Basket

  As a gardener,

  I want to carry many cucumbers in a basket,

  So that I don't drop them all.

  @add

  Scenario: Add cucumbers to a basket

   Given the basket has 2 cucumbers   # GIVEN an initial state

   When 4 cucumbers are added to the basket# WHEN action is taken

   Then the basket contains 6 cucumbers   # THEN verify outcomes

Our First Step Definitions

```python
from pytest_bdd import scenarios, given, when, then

scenarios('../features/cuke_basket.feature')          # Specifies feature files for these steps


@given('the basket has 2 cucumbers')
def basket():
    return CucumberBasket(initial_count=2)             # Givens act like fixtures


@when('4 cucumbers are added to the basket')
def add_cucumbers(basket):                             # Get the given's value as an argument
    basket.add(4)                                      # Steps can access and modify it


@then('the basket contains 6 cucumbers')
def basket_has_total(basket):                          # It's the same basket here
    assert basket.count == 6                           # Use the same assertions as pytest
```

Recommended Directory Layout

<ins>my-project</ins>

```
|-- [product code]
|-- tests
|    |-- features
|    |    `-- cuke_basket.feature
|    `-- step_defs
|         |-- conftest.py
|         `-- test_cuke_basket.py
`-- [pytest.ini|tox.ini|setup.cfg]
```

Our Feature, Again

@basket

Feature: Cucumber Basket

  As a gardener,

  I want to carry many cucumbers in a basket,

  So that I don't drop them all.

  @add

  Scenario: Add cucumbers to a basket

    Given the basket has 2 cucumbers

    When 4 cucumbers are added to the basket

    Then the basket contains 6 cucumbers

Our Feature, Parametrized!

@basket

Feature: Cucumber Basket

  As a gardener,

  I want to carry many cucumbers in a basket,

  So that I don't drop them all.

  @add

  Scenario: Add cucumbers to a basket

    Given the basket has "2" cucumbers

    When "4" cucumbers are added to the basket

    Then the basket contains "6" cucumbers

Parametrized Step Definitions

```python
from pytest_bdd import *

scenarios('../features/cuke_basket.feature')

@given(parsers.parse('the basket has "{initial:d}" cucumbers'))    # Parse as an int
def basket(initial):                                               # Pass it by name
    return CucumberBasket(initial_count=initial)

@when(parsers.parse('"{some:d}" cucumbers are added to the basket'))
def add_cucumbers(basket, some):                                   # OK with fixtures
    basket.add(some)

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

Scenario Outlines

Feature: Cucumber Basket

  Scenario Outline: Add cucumbers to a basket
    Given the basket contains "<initial>" cucumbers
    When "<some>" cucumbers are added to the basket
    Then the basket contains "<total>" cucumbers

    Examples: Cucumber Counts
      | initial | some | total |
      | 0 | 3 | 3 |
      | 2 | 4 | 6 |
      | 5 | 5 | 10 |

Extra Code for Scenario Outlines

```python
from pytest_bdd import *

CONVERTERS = dict(initial=int, some=int, total=int)
scenarios('../features/cuke_basket.feature', example_converters=CONVERTERS)

@given('the basket has "<initial>" cucumbers')
@given(parsers.parse('the basket has "{initial:d}" cucumbers'))
def basket(initial):
    return CucumberBasket(initial_count=initial)

@when('"<some>" cucumbers are added to the basket')
@when(parsers.parse('"{some:d}" cucumbers are added to the basket'))
def add_cucumbers(basket, some):
    basket.add(some)

@then('the basket contains "<total>" cucumbers')
@then(parsers.parse('the basket contains "{total:d}" cucumbers'))
def basket_has_total(basket, total):
    assert basket.count == total
```

A Service Call Feature

Feature: DuckDuckGo Instant Answer API
  As an application developer,
  I want to get instant answers for search terms via a REST API,
  So that my app can get answers anywhere.

  Scenario: Basic DuckDuckGo API Query
    Given the DuckDuckGo API is queried with "panda" using "json" format
    Then the response status code is "200"
    And the response contains results for "panda"

Service Call Steps using requests

```python
import requests
from pytest_bdd import *


scenarios('../features/service.feature')


@given(parsers.parse('the DuckDuckGo API is queried with "{phrase}" using "{fmt}" format'))
def ddg_response(phrase, fmt):
    return requests.get('https://api.duckduckgo.com/', params={'q': phrase, 'format': fmt})


@then(parsers.parse('the response status code is "{code:d}"'))
def ddg_response_code(ddg_response, code):
    assert ddg_response.status_code == code


@then(parsers.parse('the response contains results for "{phrase}"'))
def ddg_response_contents(ddg_response, phrase):
    assert phrase.lower() == ddg_response.json()['Heading'].lower()
```

A Web Feature

Feature: DuckDuckGo Web Browsing
  As a web surfer,
  I want to find information online,
  So I can learn new things and get tasks done.

Background:
    Given the DuckDuckGo home page is displayed

Scenario: Basic DuckDuckGo Search
    When the user searches for "panda"
    Then results are shown for "panda"

Web Steps using Selenium WebDriver

```python
import pytest
from pytest_bdd import *
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

scenarios('../features/web.feature')

@pytest.fixture
def browser():
    b = webdriver.Firefox()
    b.implicitly_wait(10)
    yield b
    b.quit()
```

```python
@given('the DuckDuckGo home page is displayed')
def ddg_home(browser):
    browser.get('https://duckduckgo.com/')


@when(parsers.parse('the user searches for "{phrase}"'))
def search_phrase(browser, phrase):
    search_input = browser.find_element_by_name('q')
    search_input.send_keys(phrase + Keys.RETURN)


@then(parsers.parse('results are shown for "{phrase}"'))
def search_results(browser, phrase):
    links_div = browser.find_element_by_id('links')
    assert len(links_div.find_elements_by_xpath('//div')) > 0
    search_input = browser.find_element_by_name('q')
    assert search_input.get_attribute('value') == phrase
```

Support Classes

Any Python packages or custom modules can be used with **pytest-bdd**. 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

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

```
# run all tests
pytest


# run a test module by path
pytest tests/step_defs/cuke_basket.py


# filter tests by tag
pytest -k "basket"
pytest -k "service or web"
pytest -k "add and not remove"
```

The Power of pytest

Since **pytest-bdd** is a plugin, it can do everything **pytest** can do!

- BDD-style tests can be filtered and executed together with traditional tests
- Any fixtures can be used by step functions
- Scenario outlines can be parametrized using **@pytest.mark.parametrize**

Other plugins can work together with **pytest-bdd**:

- Use **pytest-cov** for code coverage
- Use **pytest-xdist** for parallel execution
- Use **pytest-django** for Django integration

The Big Picture

Why Do Behavior-Driven Development?

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.
- Each test covers a singular unique behavior, avoiding duplication.
- Steps can be reused by behavior specs, creating a snowball effect.

Objections!

What people say:

- BDD adds a lot of overhead to the development process.
- I don't need Gherkin because Python is already readable and elegant.
- I don't need to write descriptive steps because nobody else reads my tests.
- Feature files just get in the way of coding.

My rebuttals:

1. How do you write tests for features? By describing them in plain language.
2. How do you structure your automation? For maximum reusability.

Some Advice

1. Write short scenarios.
2. One scenario should focus on one behavior.
3. Reuse steps as much as possible.
4. Put shared steps in *conftest.py*.
5. Put common execution options in *pytest.ini*.
6. Don't use BDD frameworks for unit testing.

<u>The Golden Gherkin Rule</u>
Treat other readers as you would want to be treated.
Write Gherkin so that people will intuitively understand it.

What about behave?

behave is another popular Python BDD test framework.

Advantages of **behave**:

- Simpler step function implementation
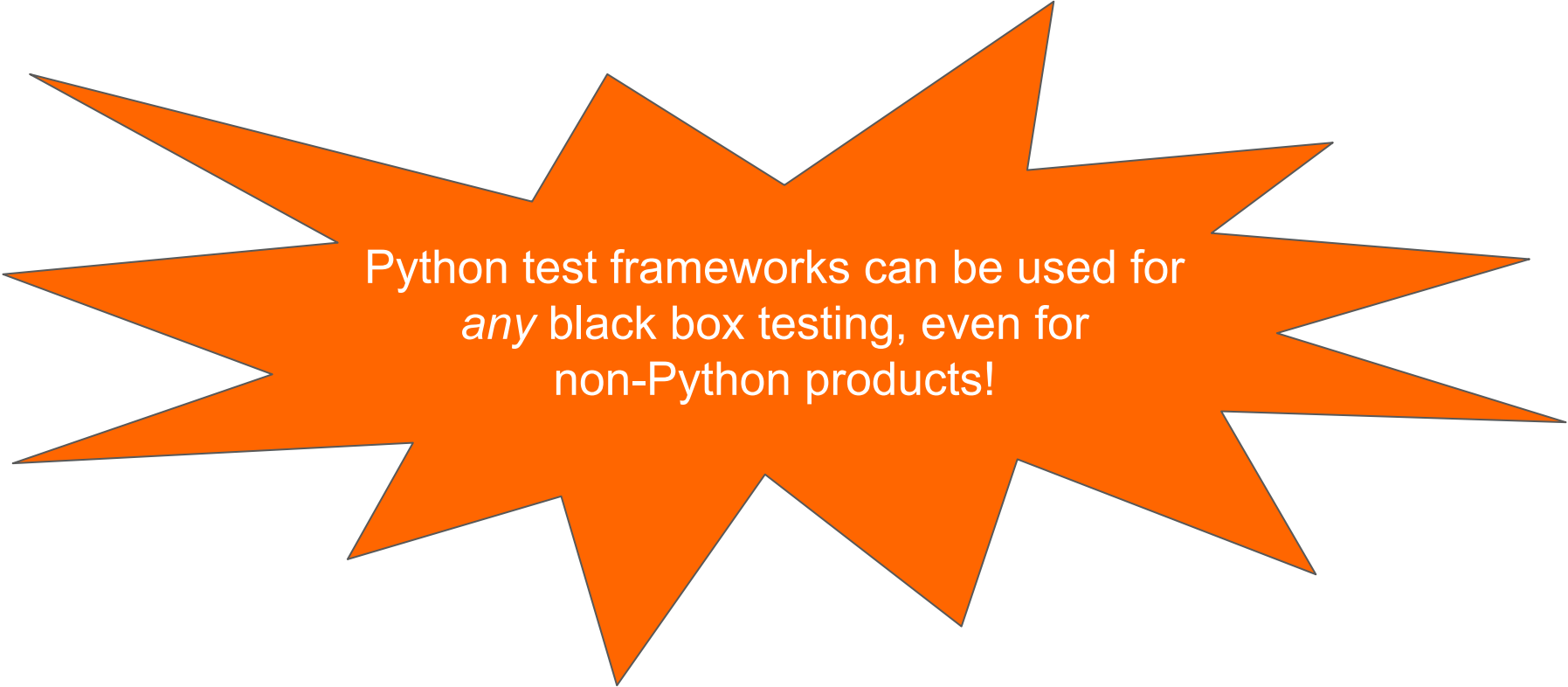- Feature files are more like Cucumber

Advantages of **pytest-bdd**:

- Can leverage the full strength of **pytest**
- Uses **pytest** fixtures

My current preference is pytest-bdd.

Beyond Python

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

Resources

Twitter: **@AutomationPanda**
Blog: AutomationPanda.com

Check out the *Python*, *Testing*, and *BDD* pages on my blog.

Look for the "Python Testing 101: pytest-bdd" article.

See example code at https://github.com/AndyLPK247/behavior-driven-python.

Reach out through those channels to ask me questions.