Introduction to BDD in Python

PyLadies Workshop 2025-03-31

Agenda

Part 0: Setup

- Clone the Repo and create virtual environment

Part I: What is BDD?

- Introduction of Frameworks: BDD vs TDD vs TLD
- Introduction of Given-When-Then Statements => Gherkin
- Benefits and Common Pitfalls of BDD

Exercise 1: Introduction to Gherkin

- Good vs. Bad Given-When-Then statements
- Write Given-When-Then statements

Agenda

Part II: Frameworks

- Overview over BDD frameworks in Python
- Implementing Given-When-Then statements in Behave

Exercise 2:

- Implement Given-When-Then statements in either Behave or pytest-bdd
- Run tests and discuss output

Agenda

Part III: Practical Gherkin Features

- Scenario Outlines
- Tables
- Parameterization
- Backgrounds

Exercise: Advanced Syntax

- Use Parameterization, Scenario Outlines, Data Tables and Backgrounds

The Goal of This Workshop

At the end of the workshop, you should be able to...

1. ... understand the benefits of BDD.

2. ... use BDD to specify the behaviour of your code in a structured way *before* writing it.

3. ... assert that your code exhibits the expected behaviour *while* writing it.

4. ... advocate for the use of BDD in your organisation should you chose to do so.

Part 0 - Setup

Setup

1. Go to the GitHub Repo for this project
2. Clone the repository with `git clone git@github.com:pyladiesams/bdd-with-python-mar2025.git`
3. Install the dependencies in your preferred way, example:
   a. `python3.8 -m venv .venv`
   b. `source .venv/bin/activate`
   c. `pip install -r requirements.txt`
4. Verify that everything went well, by running:
   a. `behave -qo test.txt solutions/features && rm test.txt`

Part I - What is BDD?

What is Behavior-Driven Development?

- Evolved from Test-Driven Development (TDD)
- Focuses on collaboration between devs, testers, and non-technical stakeholders
- Describes system behavior in *plain language*

🧠 *BDD = testing + collaboration + clarity*

How is BDD Different?

Feature	TDD	ATDD	BDD
Written by	Developers	Devs + Testers	Devs + Testers + Biz
Format	Code (Unit Tests)	Natural Language	Natural Language
Focus	Code correctness (?)	Acceptance Criteria	Behavior (User Perspective)
Tooling	E.g. pytest	E.g. Cucumber, Behave	E.g. Cucumber, Behave

Alternative: Test-Last-Development

“*Any idiot could implement a behaviour that's already designed and specified in detail by a clear, readable set of tests. I got into programming because I like a challenge, and there's no greater challenge than trying to write a function while simultaneously trying to work out what it should do.*”

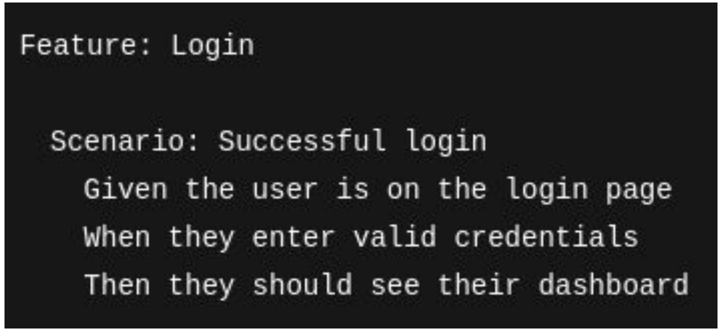
- John Arundel for Bitfield Consulting

Gherkin: A Common Language

- Plain-text, domain-specific language

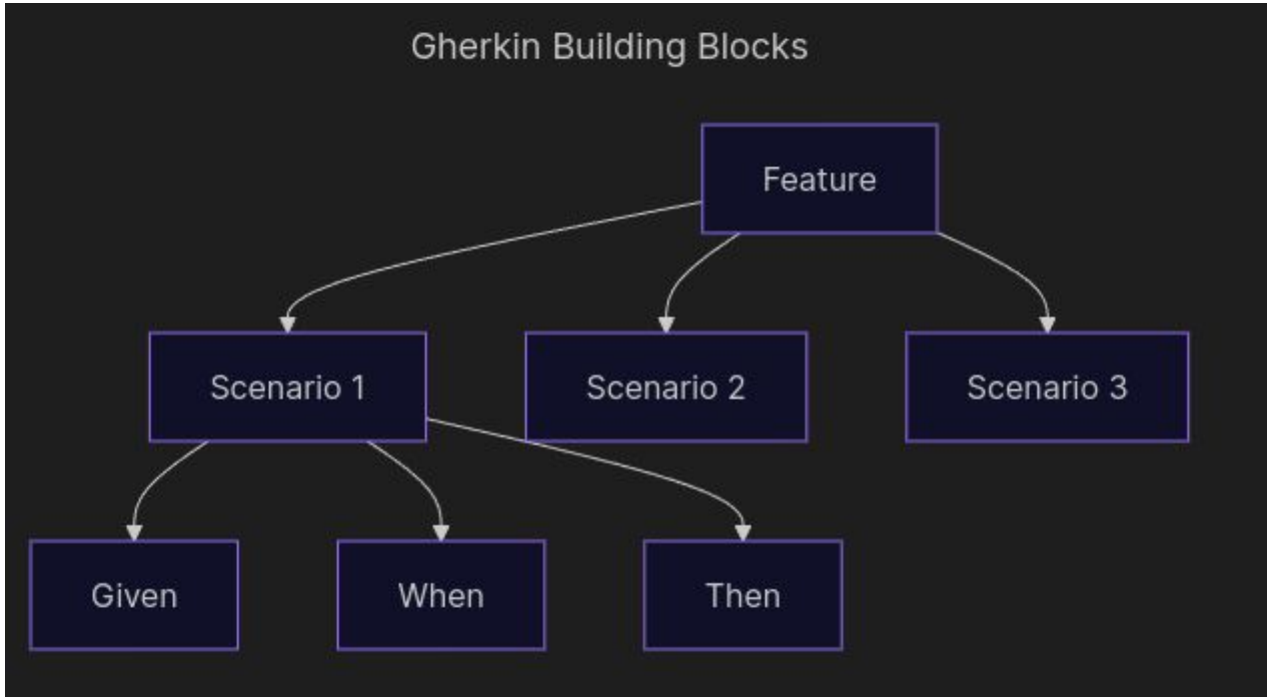
- Used by non-devs and devs alike

- Keywords: **Feature**, **Scenario**, **Given**, **When**, **Then**



```
Feature: Login

  Scenario: Successful login
    Given the user is on the login page
    When they enter valid credentials
    Then they should see their dashboard
```

Building Blocks of Gherkin



Why Use BDD?

✅ Improves collaboration

✅ Clarifies expectations

✅ Produces living documentation

✅ Tests are readable by everyone

✅ Encourages focusing on value, not just implementation

Common Pitfalls in BDD

⚠️ Writing tests, not behavior

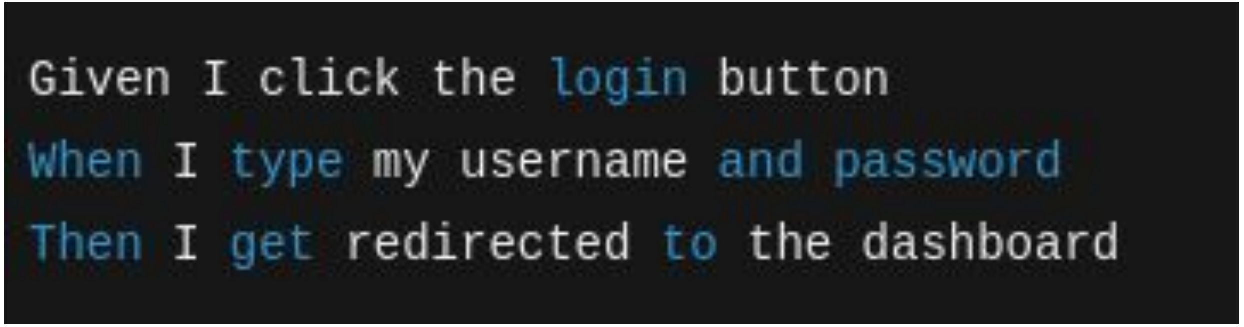
⚠️ Over-detailed steps ("click this button" vs "submit form")

⚠️ Not involving business stakeholders

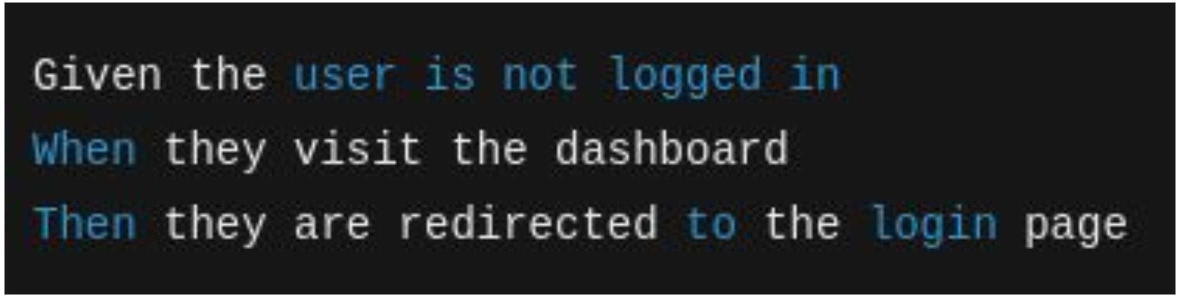
⚠️ Duplicate or unmaintainable scenarios

⚠️ Using Gherkin for everything (not every test should be a feature test)

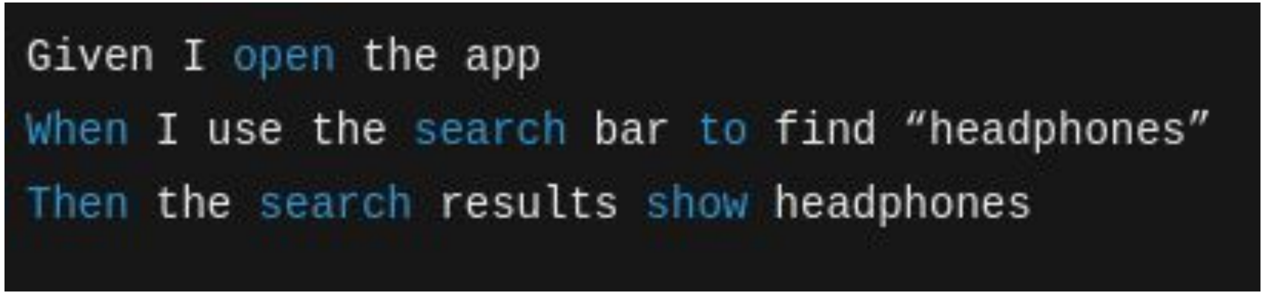
Exercise 1a: Good and Bad Gherkin



```
Given I click the login button
When I type my username and password
Then I get redirected to the dashboard
```



```
Given the user is not logged in
When they visit the dashboard
Then they are redirected to the login page
```



```
Given I open the app
When I use the search bar to find “headphones”
Then the search results show headphones
```

Exercise 1b: Write your own Gherkin from Requirement Files

Exercise 1b: Defining Scenarios with Gherkin

1. Open the workshop repository

2. Navigate to workshop/features/basket.feature. The file contains already a header as well as an example scenario to get you started, but feel free to delete the file, if you would like to start from scratch.

3. Try adding 2-4 new scenarios based on the requirements on the following slides

Your PM gives you the following requirements…

1. A logged in customer can add items to their basket.

2. A logged in customer should be able to see all items currently in their basket.

3. If the same item is added multiple times to the basket, the quantity of the item should increase instead of the item being duplicated.

4. The basket should show the total price for all items currently in it.

5. Customers can remove items from their basket.

6. Customers cannot add out of stock items to their basket.

7. If a customer is not logged in, their basket should still be stored temporarily.

8. If a customer with a temporary basket logs in, their temporary basket should merge with their user basket.

Part II: Frameworks

BDD Frameworks in Python

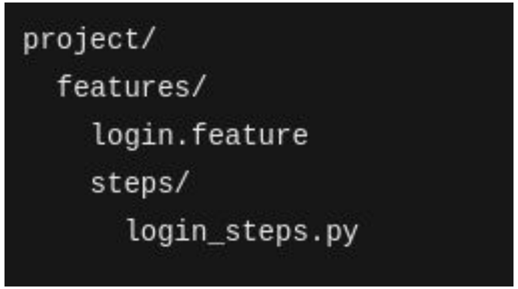
Framework	**Description**
Behave	Most widely used BDD tool in Python. Follows Cucumber style.
pytest-bdd	Integrates BDD into the pytest ecosystem.
Radish	More flexible, supports advanced Gherkin dialects.

Behave: How It Works

- .feature files define scenarios (written in Gherkin).

- Step implementations live in Python modules.

- “behave” command runs everything.



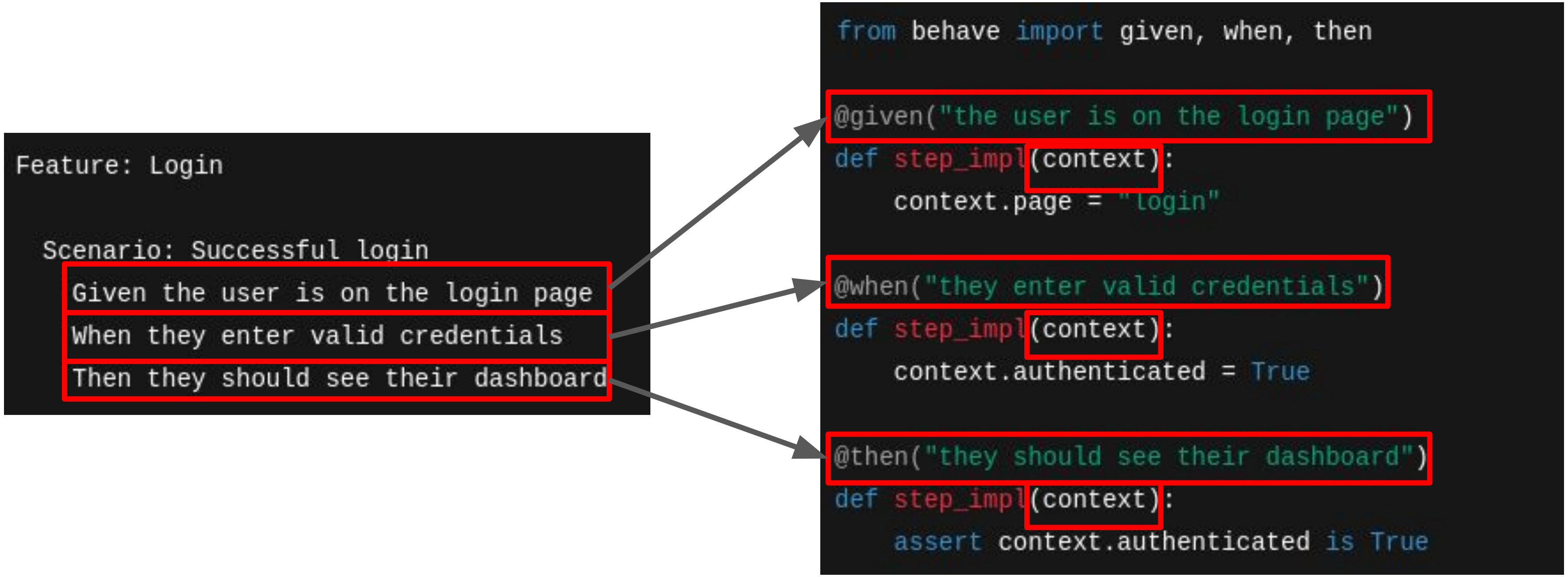
```
project/
  features/
    login.feature
    steps/
      login_steps.py
```

Implementing Steps in Behave

```
Feature: Login

  Scenario: Successful login
    Given the user is on the login page
    When they enter valid credentials
    Then they should see their dashboard
```

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

@given("the user is on the login page")
def step_impl(context):
    context.page = "login"

@when("they enter valid credentials")
def step_impl(context):
    context.authenticated = True

@then("they should see their dashboard")
def step_impl(context):
    assert context.authenticated is True
```

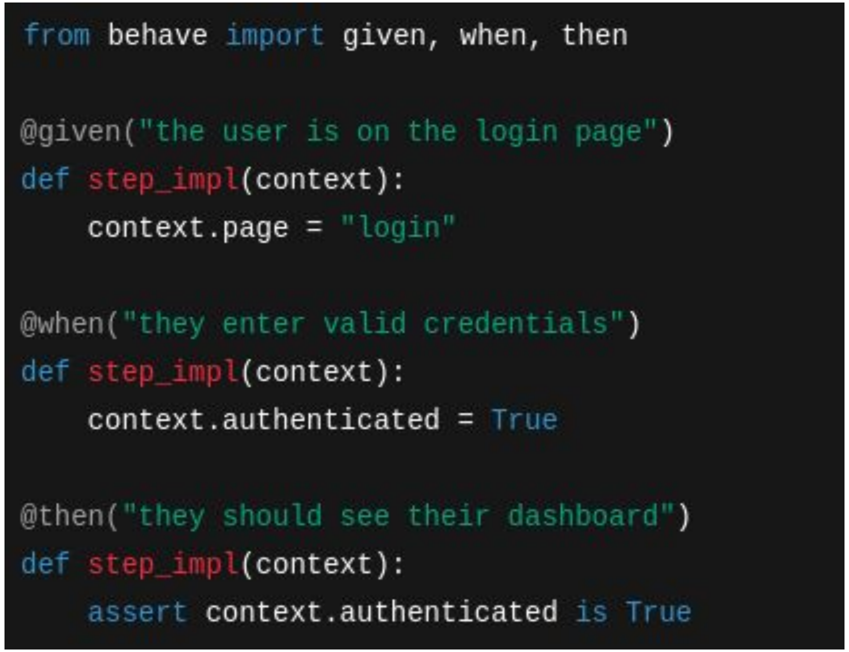
Implementing Steps in Behave



```
Feature: Login

  Scenario: Successful login
    Given the user is on the login page
    When they enter valid credentials
    Then they should see their dashboard
```

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

@given("the user is on the login page")
def step_impl(context):
    context.page = "login"

@when("they enter valid credentials")
def step_impl(context):
    context.authenticated = True

@then("they should see their dashboard")
def step_impl(context):
    assert context.authenticated is True
```

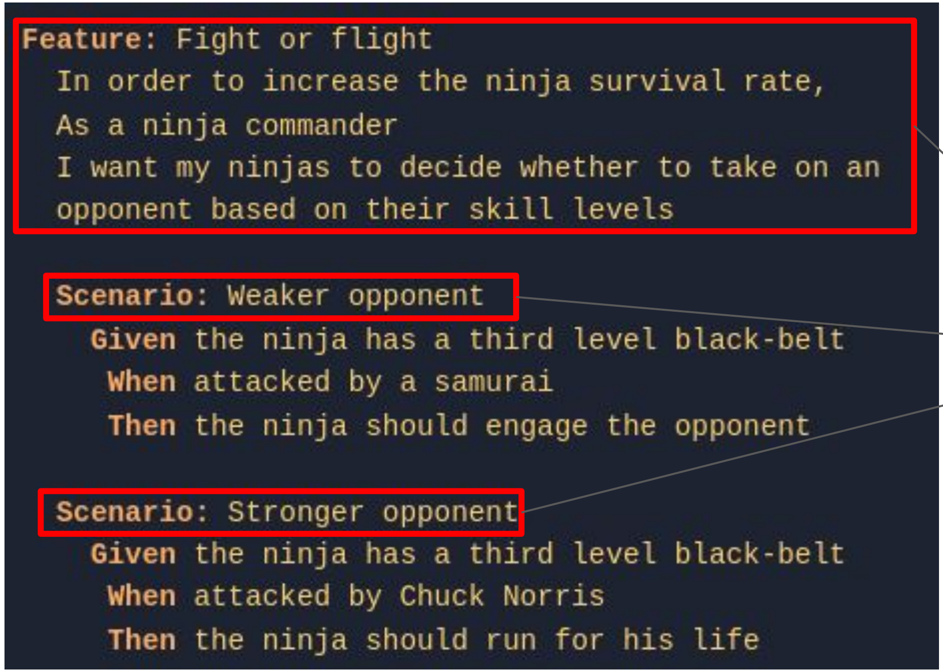
Implementing Steps in Python



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

@given("the user is on the login page")
def step_impl(context):
    context.page = "login"

@when("they enter valid credentials")
def step_impl(context):
    context.authenticated = True

@then("they should see their dashboard")
def step_impl(context):
    assert context.authenticated is True
```

🛠️ Behave uses decorators like `@given`, `@when`, and `@then`.
🧠 Store shared state in the `context` object

User Stories can be written into feature files



```
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

  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
```

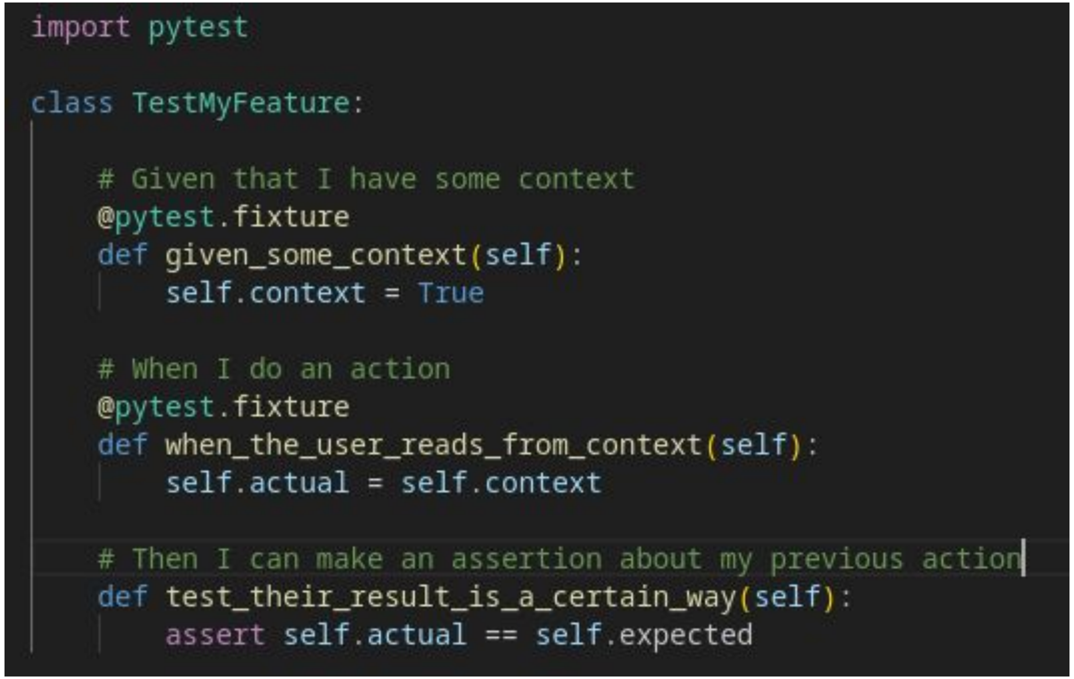
Feature and Scenario names show behave how to group tests together

Feature descriptions are only for the reader

Do You Need a BDD Framework?

❌ **No, you don't strictly need one:**

- Then implement tests using standard tools like `pytest`.
- **Given/When** steps could be represented with `pytest` fixtures.
- **Then** steps could be normal assertions in test functions.



```python
import pytest

class TestMyFeature:

    # Given that I have some context
    @pytest.fixture
    def given_some_context(self):
        self.context = True

    # When I do an action
    @pytest.fixture
    def when_the_user_reads_from_context(self):
        self.actual = self.context

    # Then I can make an assertion about my previous action
    def test_their_result_is_a_certain_way(self):
        assert self.actual == self.expected
```

Do You Need a BDD Framework?

✅ **But using a framework (like Behave) is beneficial:**

- Keeps **feature files and step definitions tightly coupled**.
- Encourages focus on **user behavior**, not implementation details.
- Makes scenarios **easier to read**, share, and maintain.
- Prevents features and tests from **diverging**

Exercise 2: Implement Your First BDD Test

Exercise 2: Hands-On Implementation

Instructions:

1. Navigate to workshop/features/steps/basket_steps.py
2. The file contains already a few step implementations to get you started, but feel free to start from scratch.
3. Try to implement the steps for (some of) the scenarios you have written in Exercise 1b.
4. When you are ready, run “behave workshop/features/basket.feature”.
5. Iterate on the code in workshop/src/models.py, until your tests succeed

Part III: Practical Gherkin Features

From Functional Requirements to Scenarios

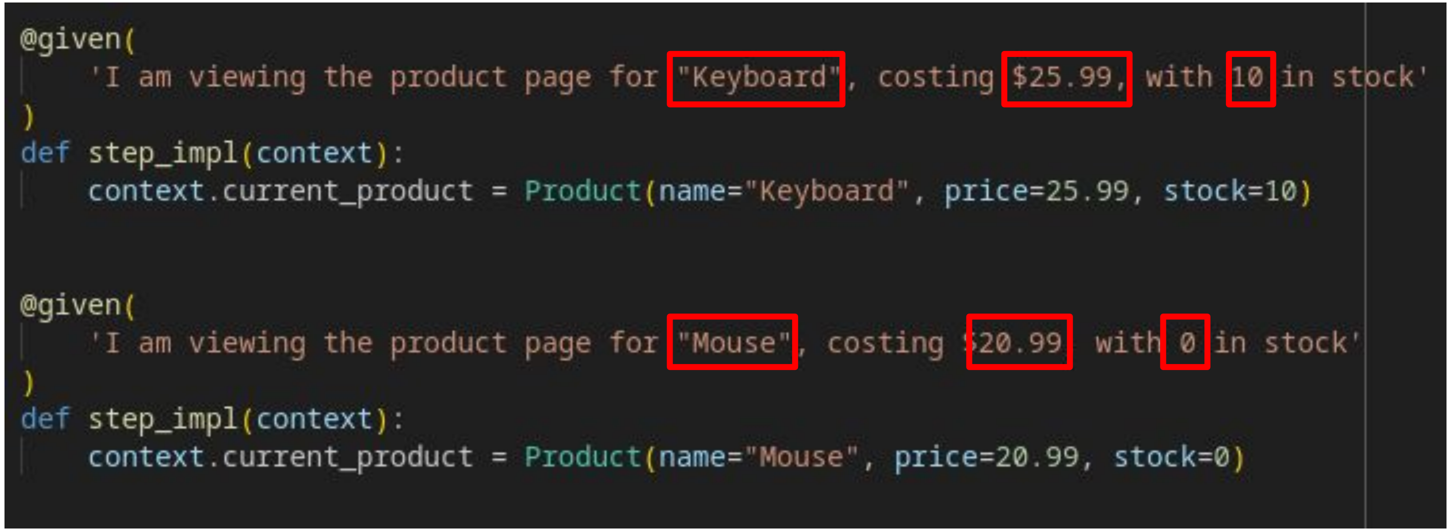
- Real systems = multiple states, validations, and edge cases
- Functional specs often describe business logic in if-then or table form
- BDD helps us turn that into executable documentation

Example Requirement: Tiered Discounts

💬 "Apply different discounts depending on customer type and total purchase amount."

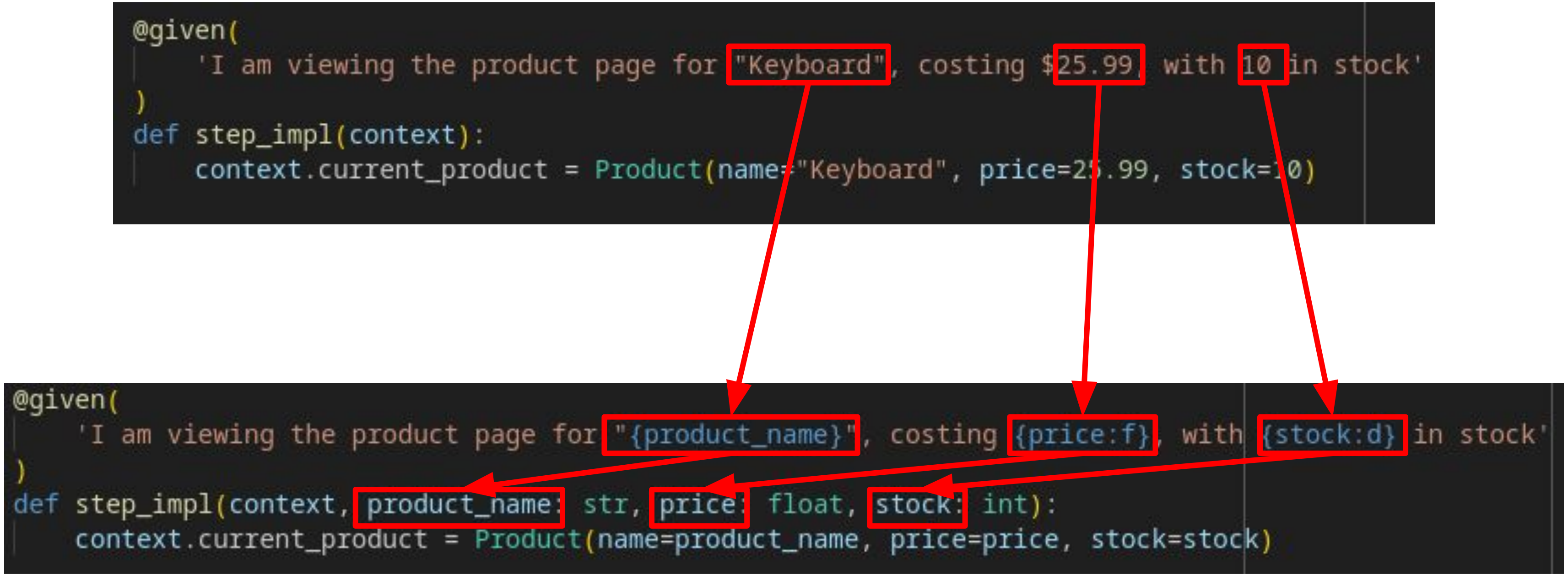
Customer Type	Total >= 100$	Discount
Regular	No	0%
Regular	Yes	5%
VIP	No	10%
VIP	Yes	20%

Parametrization of Steps



```
@given(
    'I am viewing the product page for "Keyboard", costing $25.99, with 10 in stock'
)
def step_impl(context):
    context.current_product = Product(name="Keyboard", price=25.99, stock=10)


@given(
    'I am viewing the product page for "Mouse", costing $20.99, with 0 in stock'
)
def step_impl(context):
    context.current_product = Product(name="Mouse", price=20.99, stock=0)
```

Parametrization of Steps

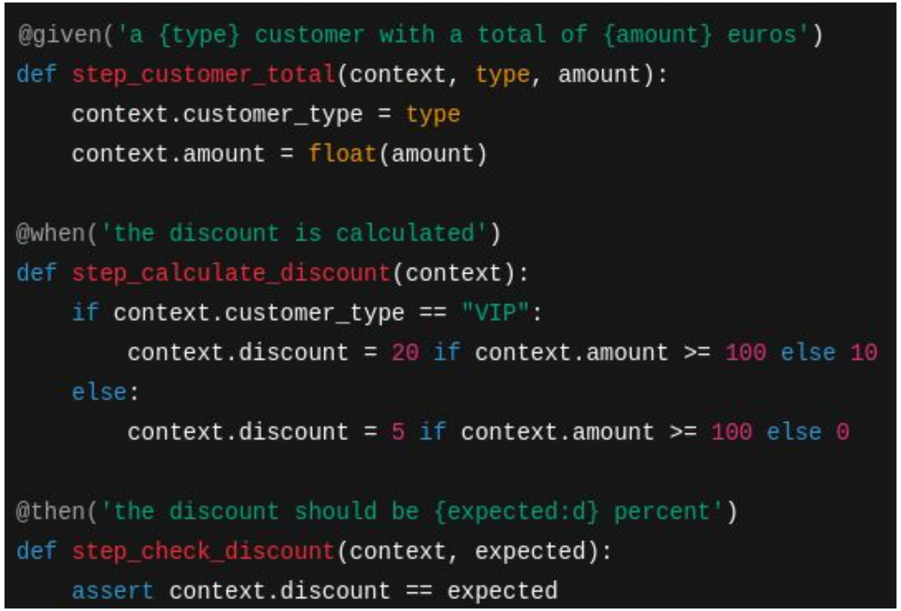


```
@given(
    'I am viewing the product page for "Keyboard", costing $25.99, with 10 in stock'
)
def step_impl(context):
    context.current_product = Product(name="Keyboard", price=25.99, stock=10)
```

```
@given(
    'I am viewing the product page for "{product_name}", costing {price:f}, with {stock:d} in stock'
)
def step_impl(context, product_name: str, price: float, stock: int):
    context.current_product = Product(name=product_name, price=price, stock=stock)
```

Note: Typehints are added for convenience and readability, they do not influence behave's behaviour!

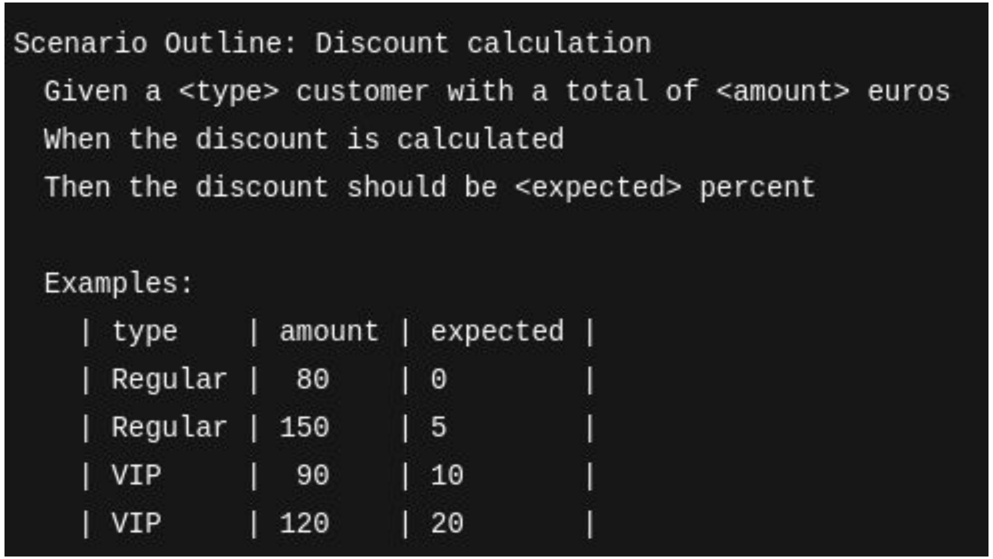
Step Definitions with Parameters



```
@given('a {type} customer with a total of {amount} euros')
def step_customer_total(context, type, amount):
    context.customer_type = type
    context.amount = float(amount)

@when('the discount is calculated')
def step_calculate_discount(context):
    if context.customer_type == "VIP":
        context.discount = 20 if context.amount >= 100 else 10
    else:
        context.discount = 5 if context.amount >= 100 else 0

@then('the discount should be {expected:d} percent')
def step_check_discount(context, expected):
    assert context.discount == expected
```

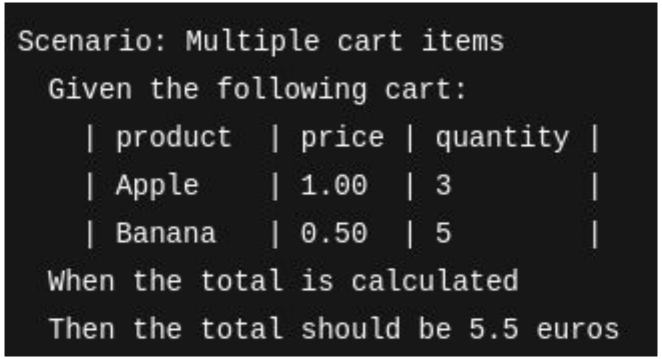
✅ Use placeholders like `{amount}` or `{expected:d}` for data injection

Parametrization of Scenarios -> Scenario Outlines

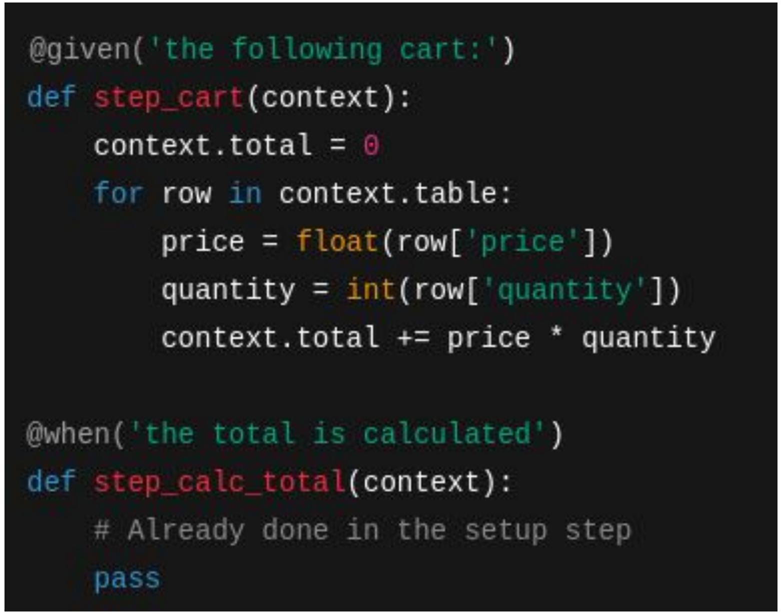


```
Scenario Outline: Discount calculation
  Given a <type> customer with a total of <amount> euros
  When the discount is calculated
  Then the discount should be <expected> percent

  Examples:
    | type    | amount | expected |
    | Regular |  80    | 0        |
    | Regular | 150    | 5        |
    | VIP     |  90    | 10       |
    | VIP     | 120    | 20       |
```

Alternative: Using Tables in regular Scenarios



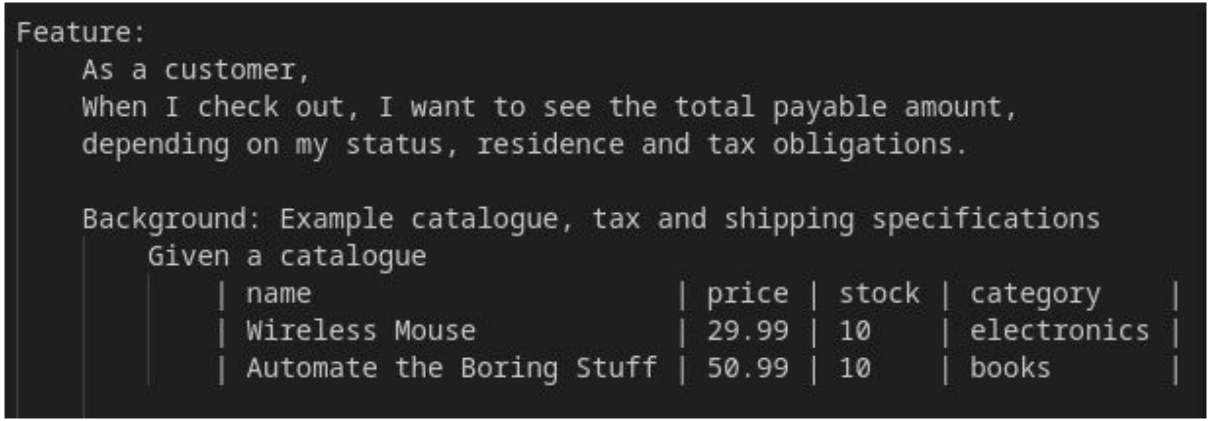
```
Scenario: Multiple cart items
  Given the following cart:
    | product  | price | quantity |
    | Apple    | 1.00  | 3        |
    | Banana   | 0.50  | 5        |
  When the total is calculated
  Then the total should be 5.5 euros
```



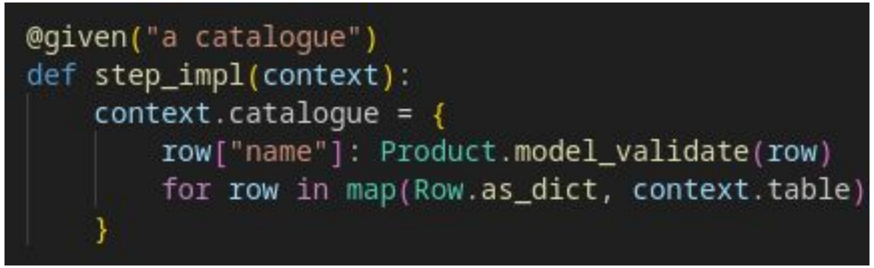
```python
@given('the following cart:')
def step_cart(context):
    context.total = 0
    for row in context.table:
        price = float(row['price'])
        quantity = int(row['quantity'])
        context.total += price * quantity

@when('the total is calculated')
def step_calc_total(context):
    # Already done in the setup step
    pass
```

Backgrounds

- Backgrounds are used to set up larger testing contexts, e.g. setting up browsers or databases
- Backgrounds consist of Given Statements and And statements
- Conceptually, they are Given statements that apply to all scenarios
- The associated steps are implemented in the same way as other Given statements



```
Feature:
    As a customer,
    When I check out, I want to see the total payable amount,
    depending on my status, residence and tax obligations.

    Background: Example catalogue, tax and shipping specifications
        Given a catalogue
            | name                    | price | stock | category    |
            | Wireless Mouse          | 29.99 | 10    | electronics |
            | Automate the Boring Stuff | 50.99 | 10    | books       |
```



```
@given("a catalogue")
def step_impl(context):
    context.catalogue = {
        row["name"]: Product.model_validate(row)
        for row in map(Row.as_dict, context.table)
    }
```

Note: Execution is done for each associated Scenario individually!

Exercise 3: Advanced Syntax

Exercise 3: Advanced Syntax

Instructions (Easier):

1. Modify your step implementations from Exercise 2 to include:
   a. Parametrization
   b. Scenario Outlines
   c. Background
   d. Data Tables

Instructions (Harder):

1. Navigate to “workshop/features/checkout.feature”
2. Write a scenario outline based on the requirement in the file header. Make sure to use the Background Data provided.
3. Implement your Scenario Outline in “workshop/features/steps/checkout_steps.py”
4. Can you make the feature succeed?

Exercise Input: Order Fulfillment Rules

💬 “Orders can be placed only if the user is authenticated and all items are in stock.
If an item is out of stock, the order should be rejected with a specific message.”

💬 “Shipping cost depends on region:

- EU: €5 flat
- US: €10 flat
- Rest of world: €20”