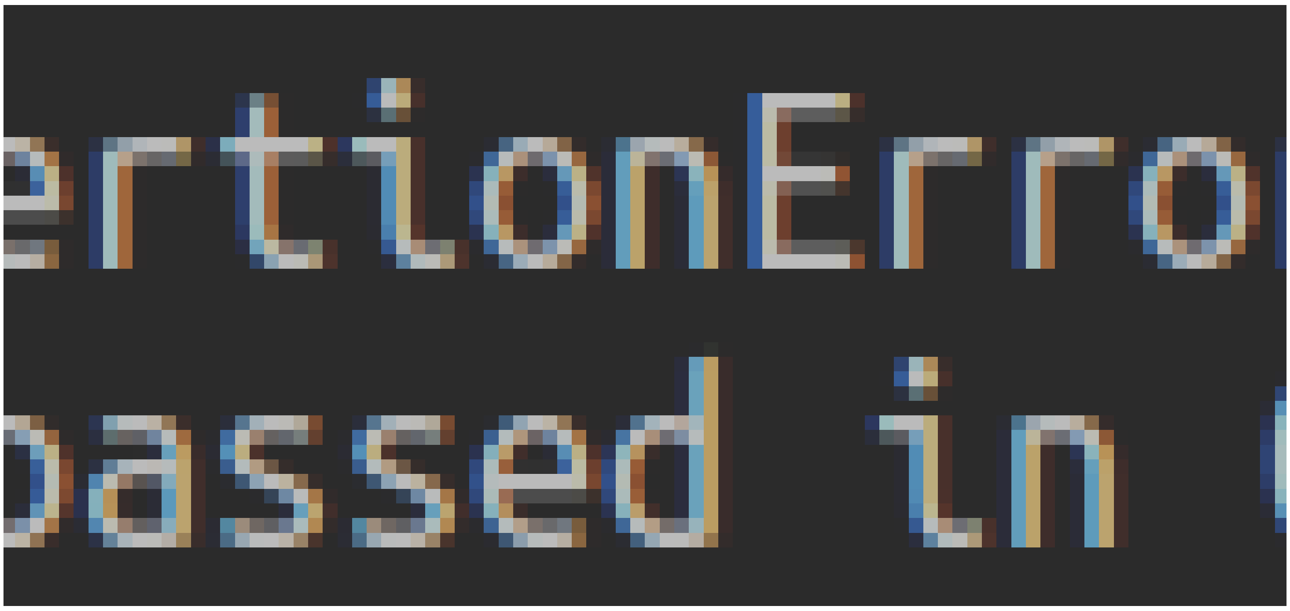
Articles > **Schedule**

Oi! Posso indicar os melhores artigos para tirar suas dúvidas!

VIEW PLANS



Yuri Matheus
04/12/2018

SHARE

This article is part of the
Training

Check out this article:

- Copied and pasted, copied and pasted, isolated
- Getting to know the fixtures
- Understanding fixtures
- To know more

I am testing an online store application. The tests ensure that the shopping cart works.

VIEW PLANS

```
from src.compras import CarrinhoDeCompras, ItemDoCarrinho, U

class TestCarrinhoDeCompras:
    def test_deve_retornar_subtotal_dos_itens_no_carrinho(se
        usuario = Usuario('Matheus')
        carrinho = CarrinhoDeCompras(usuario)
        celular = ItemDoCarrinho('Celular', 2100.0, 1)
        notebook = ItemDoCarrinho('Notebook', 4500.0, 1)
        caneta = ItemDoCarrinho('Caneta', 3.00, 5)

        carrinho.adiciona(celular)
        carrinho.adiciona(notebook)
        carrinho.adiciona(caneta)

        valor_esperado = 6615.0
        assert valor_esperado == carrinho.subtotal

    def test_deve_retornar_total_dos_itens_no_carrinho_quand
        usuario = Usuario('Matheus')
        carrinho = CarrinhoDeCompras(usuario)
        celular = ItemDoCarrinho('Celular', 2100.0, 1)
        notebook = ItemDoCarrinho('Notebook', 4500.0, 1)
        caneta = ItemDoCarrinho('Caneta', 3.00, 5)

        carrinho.adiciona(celular)
        carrinho.adiciona(notebook)
        carrinho.adiciona(caneta)

        valor_esperado = 6615.0
        assert valor_esperado == carrinho.total

    def test_deve_aplicar_desconto_ao_subtotal_dos_itens_no_
```

```
        notebook = ItemDoCarrinho('Notebook', 4500.0, 1)
        caneta = ItemDoCarrinho('Caneta', 3.00, 5)

        carrinho.adiciona(celular)
        carrinho.adiciona(notebook)
        carrinho.adiciona(caneta)
        carrinho.aplica_desconto(500)

        valor_esperado = 6115.0
        assert valor_esperado == carrinho.total
```

The tests are passing, but what's strange about this class? We have a lot of repeated code!

Copied and pasted, copied and pasted, isolated

Here at Caelum, I have a friend who always says: Copy and paste, copy and paste, isolate! The same piece of code spread throughout the system hinders code maintenance.

If the way to create a shopping cart changes, or the way to create a user changes, we will need to change all the tests that create the user. We know that isolating repeated sections is a way to avoid this repetition:

```
class TestCarrinhoDeCompras:

    def usuario(self):
        return Usuario('Matheus')
```

VIEW PLANS

This way, we can call this function every time we need a user:

```python
# restante do código omitido

def test_deve_retornar_subtotal_dos_itens_no_carrinho(self):
    usuario = self.usuario()
    carrinho = CarrinhoDeCompras(usuario)
```

The tests continue to pass, but for those who are used to testing, or have used other testing frameworks, they know that there is another way to create scenarios for testing.

Getting to know the fixtures

We can basically divide the tests into three parts: the scenario - what the test needs to be executed -, the use case execution part and the assertion - the validation of the execution result and the expected result.

In many tests, the scenario is the same! In other words, most of the tests use the same scenario to run. In our case, the `CarrinhoDeCompras`, the `Usuario` and the items are used in all tests.

method, which is inherited from the class `TestCase` , allows us to create test scenarios.

In this case, we are using a `pytest` and we do not inherit from any class, that is, we do not have a method to override. What can we do then?

Let's break this problem down into parts. The first thing we want to do is isolate the creation of an object, for example, a user:

```
class TestCarrinhoDeCompras:

    def usuario(self):
        return Usuario('Matheus')
```

Nice! We already have the function that creates a user for testing. How do I `pytest` invoke this function to run the tests?

A test scenario is also known as `fixture` a . Therefore, we need to say that this piece of code is a fixture of the `pytest`

```
class TestCarrinhoDeCompras:

    @pytest.fixture
    def usuario(self):
        return Usuario('Matheus')
```

Cool! Now we just need to say which tests need this object, we do this by passing the function name as a parameter to the test method:

VIEW PLANS

```
    @pytest.fixture
    def usuario(self):
        return Usuario('Matheus')

    def test_deve_retornar_subtotal_dos_itens_no_carrinho(self
        carrinho = CarrinhoDeCompras(usuario)
        celular = ItemDoCarrinho('Celular', 2100.0, 1)
        notebook = ItemDoCarrinho('Notebook', 4500.0, 1)
        caneta = ItemDoCarrinho('Caneta', 3.00, 5)

        carrinho.adiciona(celular)
        carrinho.adiciona(notebook)
        carrinho.adiciona(caneta)

        valor_esperado = 6615.0
        assert valor_esperado == carrinho.subtotal
```

The tests continue to pass. Let's create them `fixtures` for the other objects as well:

```
# restante do código omitido

@pytest.fixture
def usuario(self):
    return Usuario('Matheus')

@pytest.fixture
def carrinho(self, usuario):
    return CarrinhoDeCompras(usuario)

@pytest.fixture
```

VIEW PLANS

```python
@pytest.fixture
def notebook(self):
    return ItemDoCarrinho('Notebook', 4500.0, 1)

@pytest.fixture
def caneta_qtd5(self):
    return ItemDoCarrinho('Caneta', 3.00, 5)

# restante do código omitido
```

Now, we just need to receive as a parameter of the methods:

```python
ens_no_carrinho(self, usuario, carrinho, celular, notebook, ca
```

The tests keep passing, but what exactly is going on?

Understanding fixtures

When we decorate a function, or method, with `@pytest.fixture` , by default, this function is executed before each test method that needs it. In other words, each time the test is run, a new object is instantiated in memory and is used by that test.

This behavior is declared by the parameter `scope` that by default receives the value `’function’` , indicating that before each test function, this fixure is executed. Therefore, decorating the method with `@pytest.fixture` is the same as `@pytest.fixture(scope=’function’)` :

```
        return Usuario('Matheus')
```

In some cases, most of them actually, it is nice to have fixtures run before each test. This way, we have a clean scenario, without side effects from other tests.

However, sometimes creating fixtures can be costly for the system. Let's imagine that we have a connection to the database. Opening the connection before each test is costly. In this case, what we can do is open the connection at the beginning of the test module and only close it at the end.

Connection to a database, an email service, an external service, or even an object that has an immutable value. These are examples of objects that we can create once and reuse in tests.

For example, the object `usuario` does not change state, it is just passed to the shopping cart constructor. In other words, if we want, we can change its scope, but which scope should we put?

There are several scopes that make it `pytest` available for us to use. In this case, we only have one test class in the module, which is quite common, so we can make this object be instantiated only once in the class, we can do this by changing the scope to `class` :

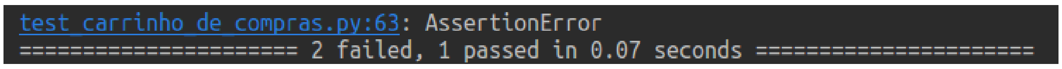
```
class TestCarrinhoDeCompras:

    @pytest.fixture(scope='class')
    def usuario(self):
        return Usuario('Matheus')
```

This way, the object is instantiated only once, as soon as the test class is instantiated, and its instance is shared with the methods.

It is important to note that since the instance is shared between methods, it is important **to be careful about side effects** . For example, if we set the scope

VIEW PLANS



```
test_carrinho_de_compras.py:63: AssertionError
===================== 2 failed, 1 passed in 0.07 seconds ======================
```

In other words, when we mess with the scope, we must always be careful and ensure that there are no side effects.

To know more

In addition to function (‘function’) and class ( ) scopes, there are also module ( ), session ( ) and package ( ) ‘class’ scopes - the latter, as of the date of writing this post, is considered experimental. ’module’ ’session’ ’package’

Each of them loads the object at some stage. The module scope instantiates an object at the beginning of the test module, that is, in the test file.

The session one is responsible for instantiating the object at the beginning of the test session (more than one test module, for example), while the package one instantiates the object when the test package - the directory -, or a subpackage, is loaded.

In addition to creating objects, when working with scopes we often have to worry about how these objects are destroyed. When we open a connection to a database, or to an email service, we have to make sure that we close it. This is called in `tear_down` the testing world.

We can do this in `pytest` the following way. Let's take some code that makes a connection to an email service:

```
@pytest.fixture(scope="module")
def conexao_email():
    conexao = smtplib.SMTP("smtp.dominio.com", 587, timeout=
    return conexao
```

VIEW PLANS

```
@pytest.fixture(scope="module")
def conexao_email():
    conexao = smtplib.SMTP("smtp.dominio.com", 587, timeout=
    yield conexao
```

After generating this object, we can tell the function to close the connection:

```
@pytest.fixture(scope="module")
def conexao_email():
    conexao = smtplib.SMTP("smtp.dominio.com", 587, timeout=

    yield conexao

    conexao.close
```

Fixtures are one of the most important features in `pytest` . For each situation, we can use a different strategy. Therefore, one thing that is always good for us to do is **to take a look at the documentation** to see how the library can best serve us.

If you want **to learn more about Python, here at Alura we have several courses on the language** . From the basics, to **advanced object-oriented features, web systems with Django or Flask, design patterns and much more** .

VIEW PLANS

Yuri Matheus

Yuri is a developer and instructor. He is a student of Information Systems at FIAP and graduated as a Computer Technician at Senac SP. His focus is on Java and Python platforms and other areas such as Software Architecture and Machine Learning. Yuri also works as a content editor on the Alura blog, where he writes mainly about Networks, Docker, Linux, Java and Python.

VIEW PLANS

Read also:

- How to compare non-Python objects?
- Python: Learn what iterators are
- Sorting lists in Python
- Python: Currency Formatting and Internationalization
- Python datetime: How do I set date and time in Python?

See other articles about
Schedule

VIEW PLANS

Our networks and apps

Institutional

About us

Alura Careers

For Companies

For Your School

Privacy Policy

Commitment to Integrity

Terms of Use

Institutional Documents

Status

The Alura

Trainings

How it Works

All courses

Testimonials

Instructors

Dev em <T>

Luri, Alura's artificial intelligence

IA Conference 2024

VIEW PLANS

Immersions

Frequently Asked Questions

Articles

Podcasts

Corporate Education Articles

News and Releases

Email*

TO SEND

COURSES

Programming Courses
Logic | Python | PHP | Java | .NET | Node JS | C | Computing | Games | IoT

Front-end Courses
HTML, CSS | React | Angular | JavaScript | jQuery

Data Science Courses
Data Science | WITH A | SQL and Database | Excel | Machine Learning | NoSQL | Statistic

Artificial Intelligence Courses
AI for Programming | AI for Data

DevOps Courses
AWS | Azure | Docker | Security | IaC | Linux

UX & Design Courses
Usability and UX | Video and Motion | 3D

Mobile Courses
Flutter | iOS e Swift | Android, Kotlin | Games

VIEW PLANS