**Montando cenários de testes com o Pytest**

* 4 de dezembro de 2018
* /
* [Programação](https://blog.alura.com.br/category/programacao/) [Python](https://blog.alura.com.br/category/programacao/python/)
* /
* [0 Comments](https://blog.alura.com.br/montando-cenarios-de-testes-com-o-pytest/#disqus_thread)

***(Last Updated On: 22 de novembro de 2018)***

Estou realizando os testes de uma aplicação de loja virtual. Os testes, garantem o funcionamento do carrinho de compras

Os métodos de testes estão dentro da classe TestCarrinhoDeCompras e cada método testa um caso das regras de negócios relacionadas às compras. A classe está disposta dessa forma:

from src.compras import CarrinhoDeCompras, ItemDoCarrinho, Usuario

class TestCarrinhoDeCompras:

def test\_deve\_retornar\_subtotal\_dos\_itens\_no\_carrinho(self):

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\_quando\_este\_nao\_tiver\_desconto(self):

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\_carrinho\_quando\_este\_nao\_tiver\_desconto(self):

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)

carrinho.aplica\_desconto(500)

valor\_esperado = 6115.0

assert valor\_esperado == carrinho.total

Copy

Os testes estão passando, mas o que tem de estranho nessa classe? Temos muito código repetido!

**Copiou e colou, copiou e colou, isolou**

Aqui na Caelum, tenho uma amiga que sempre fala: Copiou e colou, copiou e colou, isolou! O mesmo trecho de código espalhado pelo sistema prejudica a manutenção do código.

Caso o maneira de criar um carrinho de compras mude, ou então a maneira de se criar usuário, precisaremos alterar todos os testes que criam o usuário. Sabemos que isolar os trechos repetidos é uma forma de evitar essa repetição:

class TestCarrinhoDeCompras:

def usuario(self):

return Usuario('Matheus')

# restante do código omitido

Dessa forma, podemos invocar essa função cada vez que precisarmos de um usuário:

# restante do código omitido

def test\_deve\_retornar\_subtotal\_dos\_itens\_no\_carrinho(self):

usuario = self.usuario()

carrinho = CarrinhoDeCompras(usuario)

Copy

Os testes continuam passando, mas, para quem está acostumado com testes, ou já usou outros frameworks de testes, sabe que existe outra forma de criar os cenários para os testes.

**Conhecendo as fixtures**

Podemos dividir os testes em, basicamente, três partes: o cenário – aquilo que o teste precisa para ser executado -, a parte da execução do caso de uso e a asserção – a validação do resultado da execução e do resultado esperado.

Em muitos testes, o cenário é o mesmo! Ou seja, grande parte dos testes utilizam o mesmo cenário para rodar. No nosso caso, o CarrinhoDeCompras, o Usuario e os itens são utilizados em todos os testes.

Podemos isolar a criação desses objetos. Dessa forma, reutilizamos eles em outros pontos da aplicação.

Na biblioteca unittest, já deve conhecer o método setUp(). Este método, que é herdado da classe TestCase, permite que criemos cenários de testes.

No caso, estamos utilizando a pytest e não herdamos de nenhuma classe, ou seja, não temos um método para sobrescrever. O que podemos fazer então?

Vamos quebrar esse problema em partes. A primeira coisa que queremos é isolar a criação de um objeto, por exemplo, de um usuário:

class TestCarrinhoDeCompras:

def usuario(self):

return Usuario('Matheus')

Copy

Bacana! Já temos a função que cria um usuário para os testes. Como fazer a pytest invocar essa função para rodar os testes?

O cenário de um testes, também é conhecido como fixture. Logo, precisamos falar que esse trecho de código é uma fixture da pytest

class TestCarrinhoDeCompras:

@pytest.fixture

def usuario(self):

return Usuario('Matheus')

Copy

Legal! Agora só precisamos falar quais os testes que precisa desse objeto, fazemos isso passando o nome da função como parâmetro do método de teste:

class TestCarrinhoDeCompras:

@pytest.fixture

def usuario(self):

return Usuario('Matheus')

def test\_deve\_retornar\_subtotal\_dos\_itens\_no\_carrinho(self, usuario):

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

Copy

Os testes continuam passando. Vamos criar as fixtures para os outros objetos também:

# restante do código omitido

@pytest.fixture

def usuario(self):

return Usuario('Matheus')

@pytest.fixture

def carrinho(self, usuario):

return CarrinhoDeCompras(usuario)

@pytest.fixture

def celular(self):

return ItemDoCarrinho('Celular', 2100.0, 1)

@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

Copy

Agora, só precisamos receber como parâmetro dos métodos:

def test\_deve\_retornar\_subtotal\_dos\_itens\_no\_carrinho(self, usuario, carrinho, celular, notebook, caneta\_qtd5):

# restante do código omitido

Copy

Os testes continuam passando, mas o que exatamente está acontecendo?

**Entendendo as fixtures**

Quando decoramos uma função, ou método, com @pytest.fixture, por padrão, essa função é executada antes de cada método de teste que precise dela. Ou seja, para cada vez que o teste é rodado, um novo objeto é instanciado na memória e é utilizado por aquele teste.

Esse comportamento é declarado pelo parâmetro scope que por padrão recebe o valor ’function’, indicando que antes de cada função de teste, essa fixure é executada. Logo, decorar o método com @pytest.fixture é a mesma coisa de @pytest.fixture(scope=’function’):

@pytest.fixture(scope='function')

def usuario(self):

return Usuario('Matheus')

Copy

Em alguns casos, na maioria deles na verdade, é legal que as fixtures sejam executadas antes de cada teste. Assim, temos um cenário limpo, sem efeitos colaterais de outros testes.

Porém, algumas vezes criar as fixtures podem ser custosas para o sistema. Vamos imaginar que temos uma conexão com o banco de dados. Abrir a conexão antes de cada teste é custoso. Neste caso, o que podemos fazer é abrir a conexão no começo do módulo de testes e só fechá-la ao final.

Conexão com o banco de dados, com um serviço de e-mails, com um serviço externo, ou até mesmo, um objeto que tem um valor imutável. São exemplos de objetos que podemos criar uma única vez e ir reutilizando nos testes.

Por exemplo, o objeto usuario não altera estado, ele é apenas passado no construtor do carrinho de compras. Ou seja, se quisermos, podemos alterar seu escopo, mas qual escopo colocar?

Existem diversos escopos que a pytest disponibiliza para nós utilizarmos. Nesse caso, temos apenas uma classe de teste no módulo, o que é bem comum, logo, podemos fazer que esse objeto seja instanciado apenas uma vez na classe, podemos fazer isso, alterando o escopo para class:

class TestCarrinhoDeCompras:

@pytest.fixture(scope='class')

def usuario(self):

return Usuario('Matheus')

Copy

Dessa forma, o objeto é instanciado uma única vez, logo que a classe de testes é instanciada, e sua instância é compartilhada com os métodos.

É importante notar que como a instância é compartilhada entre os métodos, é importante **tomar cuidado com efeitos colaterais**. Por exemplo, se colocarmos o escopo do carrinho como class, apenas o primeiro teste que é executado passa, já que o carrinho está vazio, enquanto os outros testes falham.

[https://blog.alura.com.br/wp-content/uploads/2018/11/image_0.png](https://blog.alura.com.br/wp-content/uploads/2018/11/image_0.png)

Ou seja, quando mexemos com o escopo, devemos sempre tomar cuidado e garantir que não haja efeitos colaterais.

**Para saber mais**

Além do escopo de função (‘function’) e de classe (‘class’), existem também os escopos de módulo (’module’), sessão (’session’) e pacote (’package’) – este último, até a data de escrita desse post, é considerado experimental.

Cada um deles carrega o objeto em alguma fase. O escopo de módulo instancia um objeto no começo do módulo de testes, ou seja, no arquivo de testes.

O de sessão se encarrega de instanciar o objeto no começo da sessão de testes (mais de um módulo de testes, por exemplo), já o de pacote, instancia o objeto quando o pacote de testes – o diretório -, ou um subpacote, é carregado.

Além de criar objetos, quando trabalhamos com escopos é comum termos que nos preocupar sobre como esses objetos são destruídos. Quando abrimos a conexão com o banco, ou com um serviço de e-mail, temos que assegurar que a fecharemos. Isso é chamado de tear\_down no mundo de testes.

Conseguimos realizar isso com a pytest da seguinte forma. Vamos pegar um código que realiza uma conexão com um serviço de e-mail:

@pytest.fixture(scope="module")

def conexao\_email():

conexao = smtplib.SMTP("smtp.dominio.com", 587, timeout=5)

return conexao

Copy

Podemos falar para o Python que não queremos retornar a conexão imediatamente, mas só quando ela for realmente necessária, ou seja, podemos transformar essa função em uma [função geradora](https://docs.python.org/3/howto/functional.html#generators).

@pytest.fixture(scope="module")

def conexao\_email():

conexao = smtplib.SMTP("smtp.dominio.com", 587, timeout=5)

yield conexao

Copy

Após gerar esse objeto, podemos falar para a função fechar a conexão:

@pytest.fixture(scope="module")

def conexao\_email():

conexao = smtplib.SMTP("smtp.dominio.com", 587, timeout=5)

yield conexao

conexao.close

Copy

As fixtures são um dos recursos mais importantes que existem na pytest. Para cada situação, podemos utilizar uma estratégia diferente. Por isso, uma coisa que é sempre bacana a gente fazer é [dar uma olhada na documentação](https://docs.pytest.org/en/latest/fixture.html) para ver como a biblioteca pode melhor nos atender.